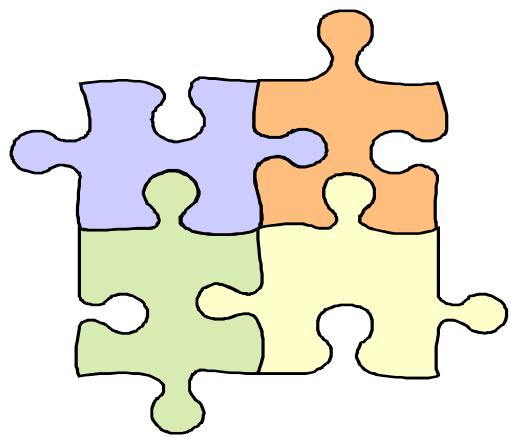


DESIGN PATTERNS

Design Patterns



Student workbook

Design Patterns

by Marilyn McCord

(DBA **Associated Consultants**)

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

Marilyn McCord lives in the San Juan Mountains northeast of Durango, Colorado, on the edge of the Weminuche Wilderness where the hiking is great. Besides courseware she has written two books: *A Practical Guide to Neural Nets*, Addison Wesley, 1990; and *Parallel Flights: A Father-Daughter Memoir*, 1stBooks, 2003. During the decade of the 1980s she worked for Texas Instruments in Dallas.



Contents

Chapter 1 – Course Introduction	7
Course Objectives	9
Course Overview	11
Suggested References	13
Chapter 2 –Design Pattern Overview	19
Objectives in Software Design/Module Design.....	25
Overview of Patterns	27
Qualities of a Pattern	29
Pattern Systems	31
Heuristics vs. Patterns	33
Chapter 3 – Principles of Object-Oriented Design	37
Overview of Principles	41
Single-Responsibility Principle (SRP).....	43
Open-Closed Principle (OCP)	45
Tell vs. Ask	47
Command/Query Separation (CQS).....	49
Composed Method	51
Combined Method	53
Liskov Substitution Principle (LSP).....	55
Dependency Inversion Principle (DIP).....	57
Interface Segregation Principle (ISP)	59
Law of Demeter	61
Review of Life Cycle Process	63
<i>Exercises</i>	64
Chapter 4 – Principles of Package Architecture	69
Package Cohesion Principles	73
Package Coupling Principles	75
Martin Package Metrics	83
<i>Exercises</i>	86
Chapter 5 – Basic Object-Oriented Design Patterns	87
Delegation vs. Inheritance	91
Interface	93
Immutable.....	95

Null Object	97
Marker Interface.....	99
General Responsibility Assignment Software Patterns	101
<i>Exercises</i>	102
Chapter 6 –Catalog of GoF Patterns.....	105
Overview of GoF Patterns.....	109
Factory Method.....	111
Abstract Factory.....	113
Builder.....	115
Prototype	119
Singleton.....	121
<i>Exercises</i>	122
Adapter	127
Decorator	129
Proxy.....	131
<i>Exercises</i>	132
Facade.....	139
<i>Exercises</i>	141
Composite.....	143
Flyweight.....	145
Bridge	149
Chain of Responsibility	151
<i>Exercises</i>	153
Strategy	155
<i>Exercises</i>	157
Iterator	161
Template Method.....	163
<i>Exercises</i>	164
Mediator.....	167
Observer	169
<i>Exercises</i>	171
Memento.....	173
<i>Snapshot</i>	175
Command	177
<i>Exercises</i>	179
State	181
<i>Exercises</i>	183
Visitor.....	185
Interpreter	189
<i>Exercises – Summary of GoF Case Study</i>	191
Chapter 7 –Other Micro-Architecture Patterns.....	193

Object Pool.....	197
Dynamic Linkage	199
Cache Management	201
Type Object	203
Extension Object	205
Smart Pointer (C++)	207
<i>Exercises</i>	209
Chapter 8 –Concurrency Patterns	211
Single Threaded Execution	217
Guarded Suspension.....	219
Balking.....	221
Scheduler	221
Read/Write Lock	225
Producer/Consumer	227
Two-Phase Termination	229
Double-Checked Locking	231
Chapter 9 – Patterns-Oriented Software Architecture	233
Architectural Patterns	239
Layers Architecture.....	241
Pipes & Filters Architecture	245
Blackboard Architecture	249
Broker.....	253
Model-View-Controller	257
Presentation-Abstraction-Control.....	261
Reflection	265
Microkernel.....	269
Summary: How to Select an Architecture	273
<i>Exercises</i>	274
Chapter 10 – Catalog of J2EE Patterns.....	277
J2EE Pattern Relationships	283
Intercepting Filter.....	285
Front Controller.....	289
View Helper	293
Composite View.....	295
Service to Worker	297
Dispatcher View.....	299
Business Delegate.....	301
Value Object.....	303
Session Facade.....	305
Composite Entity	307

Value Object Assembler	309
Value List Handler.....	311
Service Locator.....	313
Data Access Object	315
Service Activator.....	317
Chapter 11 – Selected Process Patterns (from PLoP).....	319
The Selfish Class	325
Patterns for Evolving Frameworks	327
Patterns for Designing in Teams.....	329
Patterns for System Testing.....	331
Chapter 12 – Selected Anti-Patterns.....	333
Stovepipe System.....	339
Stovepipe Enterprise	341
Reinvent the Wheel.....	343
Golden Hammer.....	345
Death by Planning.....	347
Death March Projects	349
Additional Management Anti-Patterns.....	351
Chapter 13 – Patterns Summary.....	353
Appendix A: UML Review	A-1
Appendix B: C# Code Examples for GoF.....	B-1
Appendix C: Possible Solutions for Exercises	C-1
Index	

Chapter 1 – Course Introduction

Notes

Course Objectives

- Build **Design Pattern Vocabulary**
 - Learn basic object design principles
 - Learn names and intent of all 23 GoF design patterns
 - Learn basic object-oriented architectural patterns
 - (Optional) Learn names and intent of all 15 J2EE design patterns
 - (Optional) Learn other relevant object-oriented design patterns
- Be able to discuss **trade-offs** in applying various design patterns
- Gain **concepts and tools** for writing better object-oriented code
- Gain concepts for better **documenting object-oriented code**
- Build a **framework for reading** and using the GoF book, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, et al.
- Review **UML notation**

Notes

Course Overview

- **Audience:** Analysts, designers, and programmers who want to learn more about object-oriented design patterns
- **Prerequisites:** Basic understanding of Object-Oriented Analysis and Design and the Unified Modeling Language
- **Student Materials:**
 - Student workbook
 - Text: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, et al.
- **Classroom Environment:**
 - Lecture format with board space, flip chart, etc.
 - *Optional:* Networked PCs or workstations (instructor machine display unit) and Java installed, to run a Java example of Observer Pattern “ChatServer.”

Notes

Suggested References

Alur, Crupi, Malks, “*Core J2EE Patterns*”, Sun Microsystems Press, 2001, ISBN: 0-13-0648846-1.

An excellent presentation of best practices in J2EE; easy to read and follow.

Beck, Kent, “*Test Driven Development: By Example*”, Addison-Wesley, 2002, ISBN: 0-321-14653-0.

Test-driven development (TDD) provides a set of techniques that encourage simple designs and test suites. The emphasis is on writing the tests first before you code, then making the code work – and continue working after each change. If you are interested in this approach, it is worth doing the examples.

Binder, “*Testing Object-Oriented Systems*”, Addison-Wesley, 2000, ISBN: 0-201-80938-9.

This reference book has 4 main sections: (1) Preliminaries, an introduction to testing issues, (2) Models, state machines and model-based testing, (3) Patterns, the “how-to” of OO design, a test design handbook, and (4) Tools, a test implementation handbook.

Booch, Grady, et. al, “*The Unified Modeling Language User Guide*”, Addison-Wesley, 1999, ISBN: 0-201-57168-4.

The “bible” on UML.

Brown, William J., et. al, “*AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*”, Wiley & Sons, 1998, ISBN:

A fun book – yet bittersweet because too often “you’ve been there.” Clarifies negative patterns that cause development roadblocks due to poor management, lack of architectural control, or personality clashes. Shows how to detect and defuse 40 of the most common AntiPatterns, providing a refactored solution for both the symptoms and the cause(s).

Buschmann, et. al, “*Pattern-Oriented Software Architecture - A System of Patterns*”, Wiley & Sons Ltd., 1996, ISBN: 0-471-95869-7

As the title says, Architectural Patterns. Represents the progressions and evolution of the pattern approach into a system of patterns capable of describing and documenting large-scale applications. Worth reading.

Cockburn, Alistair, “*Surviving Object-Oriented Projects: a Manager’s Guide*”, Addison-Wesley, 1998, ISBN: 0-201-49834-0 (paper).

¶ This is probably my favorite book for managers and project leads on object-oriented projects. Takes eleven real-life projects and examines things that went right/wrong. Many of Cockburn’s ideas (“patterns” for OO development and managing) show up in other articles and compendiums of “how-to-do-it” OO books.

Douglass, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns", Addison-Wesley, 1999, ISBN: 0-201-49837-5.

Real-time/embedded systems have been sort of the "step-child" when it comes to OO books – but the demand for such is growing. Topics include schedulability, behavioral patterns, real-time frameworks, timing and performance of systems. Code is C++. Includes CD-ROM with demonstration version of Rhapsody by I-Logix, PowerPoint demonstration version of TimeWiz by TimeSys, and Rhapsody sample models related to the text.

D'Souza, Desmond F., and Wills, Alan Cameron, "Objects, Components, and Frameworks with UML: The Catalysis Approach", Addison-Wesley, 1999, ISBN: 0-201-31012-0 (paper).

Catalysis is an emerging UML-based method for object and component-based development and combines the concepts of objects, frameworks, and component technologies. Part III, Factoring Models and Designs, contains a section on Process Patterns for Refinement, and Part IV, Architecture, includes several architectural patterns.

Fowler, Martin, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997, ISBN: 0-201-89542-0.

Fowler has analysis experience in a number of domains. This is a book of practical patterns and ideas you can use right away – look for chapters that match the kind of problem you are working on now.

Fowler, Martin, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2002, ISBN: 0-321-12742-0.

Distills over 40 recurring solutions into patterns; contains a reference section describing domain logic patterns. Covers layering, organizing domain logic, mapping to relational databases, Web presentation, concurrency, session state, and distribution strategies.

Fowler, Martin, "Refactoring", Addison-Wesley, 1999, ISBN: 0-201-48567-2.

I particularly liked chapter 3, entitled "Bad Smells in Code", which has the flavor of "anti-patterns" in identifying opportunities for refactoring. Uses UML, Java code.

Fowler, Martin, "UML Distilled", Second Edition, Addison-Wesley, 2000, ISBN: 0-201-65783-X.

♪ An easy read with lots of good stuff. Chapter 2 is one of the best concise descriptions of the OO development process I have seen. I have often used this book with OOA&D classes – it is an excellent basic introduction to UML.

Gamma, Erich, et. al, "Design Patterns: elements of Reusable Object-Oriented Software", Addison-Wesley, 1994, ISBN: 0-201-63361-2.

♪ **The Classic.** Started the whole Patterns thing off. Describes simple and elegant solutions to specific problems in object-oriented software design. Probably every other

book on patterns references this one.

Grand, Mark, "*Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*", John Wiley & Sons, Inc., 1998, ISBN: 0-471-25839-3.

CD-ROM contains Java source code for the 41 design patterns and trial versions of Together/J, Rational Rose 98, System Architect, and Optimizelt.

Hartman, Robert, "Building on Patterns," *Application Development Trends* magazine, May 2001, p.19-26.

Jacobson, Ivar, et. al, "*The Unified Software Development Process*", Addison-Wesley, 1999, ISBN: 0-201-57169-2.

Focuses on the process, but there is lots of stuff on patterns along the way, such as "analysis stereotypes" of boundary, control and entity, some pages on layered architecture, etc.

Kassem, and the Enterprise Team, "*Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition*", Sun Microsystems Press, 2000, ISBN: 0-201-70277-0.

Describes the principles and technologies employed in building J2EE applications and the specific approach adopted by a sample application for an e-commerce Web site.

Larman, Craig, "*Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design*", Prentice Hall, 1998, ISBN: 0-13-748880-7.

I often tell an OOA&D class: "The good thing about this book is that it shows every little step. The bad thing about this book is, it shows every little step." Depends on whether you want that level of detail or not! Takes a Point Of Sales Terminal (POST) example and does the A & D work. Has quite a bit on patterns, and emphasizes the GRASP (General Responsibility Assignment Software Patterns) patterns, most of which are just good common sense in OO Design, but brings them to the top of your consciousness. Good chapter on persistence patterns.

Lea, Doug, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley, 1996, ISBN: 0-201-69581-2.

⌘ Doug Lea is probably the best when it comes to concurrent programming tips.

Martin, "*Agile Software Development: Principles, Patterns, and Practices*," Prentice-Hall, 2003, ISBN: 0-13-597444-5.

Foreword by Erich Gamma: "This book is crammed with sensible advice for software development." Contains a lot of Java and C++ code. Clear impact of Extreme Programming (XP), expansions on GoF patterns in case studies, other stuff.

Martin, Riehle, Buschmann, "*Pattern Languages of Program Design 3*", Addison-Wesley, 1998, ISBN: 0-201-31011-2 (paper).

Third in a series. A collection of the current best practices and trends in the patterns community, including international essay submissions.

Riel, Arthur J., “*Object-Oriented Design Heuristics*”, Addison-Wesley, 1996, ISBN: 0-201-63385-X.

Lots you probably already know, but worth checking.

Vlissides, John, “*Pattern Hatching: Design Patterns Applied*,” Addison-Wesley, 1998, ISBN: 0-201-43293-5 (paper)

Presents themes and variations on several established patterns, but with new insights. Puts patterns into the broader context of basic object-oriented design principles. Note: Vlissides is one of the GoF authors.

WEB SITES:

(Note that web sites can and do change often...)

<http://www.ootips.org>

Tons of good stuff here, plus links to other sites

<http://hillside.net/patterns/books/>

Lists lots of patterns books – 9 pages with links

Includes Brad Appleton's Patterns Intro

<http://www.cs.wustl.edu/~schmidt/>

Doug Schmidt's Home Page

<http://www.bell-labs.com/user/cope>

Jim Coplien's Home Page

<http://gee.cs.oswego.edu/dl>

Doug Iea's Home Page, Patterns FAQs

Patterns for Concurrent Programming in Java

<http://members.aol.com/acockburn/riskcata/risktoc.htm>

Alistair Cockburn, Surviving Object-Oriented Projects

<http://www.objectmentor.com>

Robert Martin's company

Just do a search on what you want – there is more information than you ever dreamed.

Notes

Chapter 2 – Design Pattern Overview

Notes

Chapter Objectives

- Build **rationale** for using design patterns
- Discuss **software “rot”**
- Define “**pattern**”
- Build **template** for what we want to see in a pattern description
- **Contrast patterns and heuristics**

Notes

What is a Design Pattern?

Why should I use patterns?

***Why bother writing patterns
that just boil down to advice
my grandmother would give me?***

Notes

Good habits:

1. Always use Source Control system.
2. Automate development process with makefiles, batch, etc.
3. First version of code:
 - gets cosmetics right – comments, formatting, naming
 - handles abnormal cases
 - uses assertions
 - builds in tests
4. Always maintain a working system
5. Compile often
6. When discovering an error, ask yourself:
 - Is this mistake also somewhere else?
 - What other bugs might be hidden behind this one?
 - How can I prevent bugs like this in the future?
7. Scrap poor code. Refactor. Document choices/discards.
8. “Live for today.” Suppose you were called away from the project suddenly and permanently?

Objectives in Software Design/Module Design



We *want* to write “good” code that lasts a long time, is mobile, accommodates change, is easy to understand and easy to extend. S-I-M-P-L-E.

So, why don’t we achieve this?

What makes some software designs rot?

What can we do to help prevent software rot?

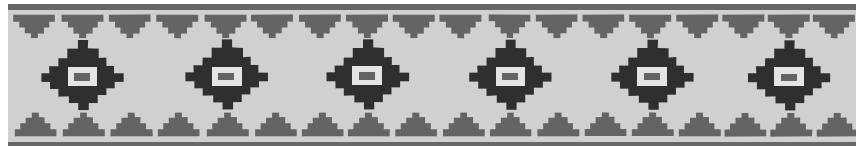
What is refactoring?

Notes

- Some GoF patterns exist simply because of C++ limitations:
Abstract Factory, Factory Method, Singleton, Prototype, Command, and Visitor
- GoF book points out C++ pitfalls, e.g.
 - ◆ combination of interface and implementation inheritance
 - ◆ use of inheritance instead of delegation or forwarding
 - ◆ safety/flexibility tradeoff
 - ◆ confusion between lexical - dynamic scoping in methods
 - ◆ flat class namespace.
- If you were creating your own template for patterns, what would you include?
(Check our list against GoF, Grand, etc.)
 - 1.
 - 2.
 - 3.
 - 4.
 - 5.
 - 6.

Overview of Patterns

It helps to be able to think at a design level rather than a code level – this has been described as the instinct to know what can be implemented without actually implementing it.



How concisely can you describe your last software design?

- Patterns seldom used in isolation - it helps to learn some before others
- Some patterns simple, some complex

What is your definition of a software pattern?

- Template Brainstorm:

Notes

Patterns Encapsulate and Abstract

- well-defined problem/solution in a particular domain
- crisp, clear boundaries -- parceling into lattices of distinct, interconnected fragments
- abstractions that embody domain knowledge and experience
- may occur at varying hierarchical levels of conceptual granularity within domain

Patterns Exhibit Openness and Variability

- open for extension or parameterization by other patterns
- capable of many different implementations
- usable in isolation or with other patterns

Patterns Have Generativity and Composability

- one pattern provides context for application of next pattern
- subsequent patterns applied to progress further toward final goal
- not linear in nature -- more like fractals

Patterns realize an Equilibrium

- balance among pattern forces and constraints
- invariants typify principles/philosophy for domain
 to minimize conflict within solution space
- a rationale for each step/rule in the pattern.



Qualities of a Pattern

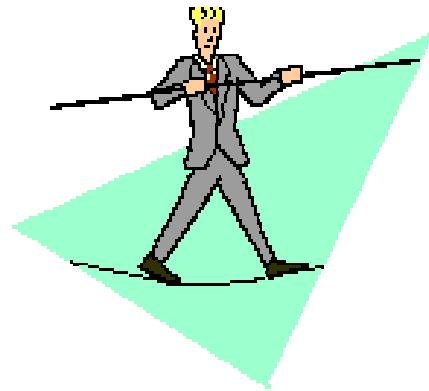
- ⇒ Well-written patterns describe a whole greater than the sum of parts
- ⇒ Elements work together to satisfy varying demands

- **Patterns Encapsulate and Abstract**

- **Patterns Exhibit Openness and Variability**

- **Patterns Have Generativity and Composability**

- **Patterns realize an Equilibrium**



► *What do these bullets really mean?*

Notes

Simply using objects to model an application is not sufficient to create robust, maintainable and reusable designs. Other attributes of a design are required. These attributes are based on a pattern of interdependencies between the subsystems of the design to:

- support communications within the design,
- isolate reusable elements from non-reusable elements,
- block the propagation of change due to maintenance.

"It would not be reasonable to design a house without knowing the lay of the land, or a skyscraper without knowing the materials that could be used. The idea that we can treat concepts such as threading and distribution as mere coding details is a sure fire way to waste a lot of energy (and time, money, etc.) in big, up-front design only to discover that the difference between theory and practice is bigger in practice than in theory."¹

Compare with language:

vocabulary = patterns
grammar = rules, heuristics

"... the best software patterns are also geometric... When a system encounters stress, it loses symmetry."²

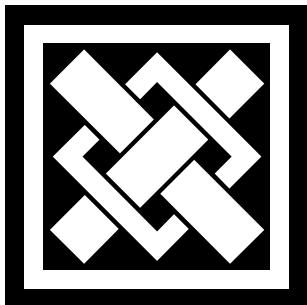
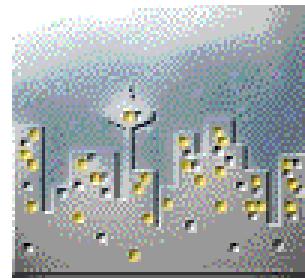
¹ Henney, Kevlin, "A Tale of Two Patterns," *Java Report*, December 2000, p. 84.

² Coplien and Zhao, "Symmetry and Symmetry Breaking in Software Patterns," Oct. 2000.

Pattern Systems

Pattern Levels:

- (1) **highest level:** architectural patterns define overall shape
- (2) specific architecture related to purpose of application
- (3) architecture of modules and their interconnections
(the domain of design patterns)
- (4) principles are like **firewalls** to help prevent improper dependencies
(main reason software rots)



*The real **art** is **interweaving** patterns – design patterns are not used in isolation. Not so easy, not straightforward. And, you have to know the pieces to weave them together well.*

“A **pattern system** for software architecture is a **collection of patterns** for software architecture, together **with guidelines** for their implementation, combination and practical use in software development.”³

³ Buschmann, et al, *Pattern-Oriented Software Architecture: A System of Patterns*, © Wiley, 1996, p. 361,

Notes

Heuristics from Riel⁴ (note that some are C++ specific):

Classes and Objects

- All data should be hidden within its class.
- Users of a class should be dependent on its public interface; classes should not be dependent on its users.
- Minimize the number of messages in the protocol of a class.
- Implement a minimal public interface that all classes understand.
- Do not put implementation details such as common-code private functions into the public interface of a class.
- Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
- Classes should only exhibit nil or export coupling with other classes.
- A class should capture one and only one key abstraction.
- Keep related data and behavior in one place.
- Spin off non-related information into another class.
- Be sure abstractions that you model are classes and not simply roles objects play.

Object-Oriented Applications

- Distribute system intelligence horizontally as uniformly as possible.
- Do not create god classes/objects in your system.
- Beware of classes that have many accessor methods defined in public interface. Having many implies that related data and behavior are not being kept in one place.
- In applications that consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface.
- Model the real world whenever possible.
- Eliminate irrelevant classes from your design.
- Do not turn an operation into a class.
- Agent classes are often placed in the analysis model of an application.

Relationships between Classes and Objects

- Minimize the number of classes with which another class collaborates.
- Minimize the number of message sends between a class and its collaborator.
- Minimize the amount of collaboration between a class and its collaborator.
- Minimize fan-out in a class.
- If a class contains objects of another class, then the containing class should be sending messages to the contained objects.
- Most of the methods defined on a class should be using most of the data members most of the time.
- Classes should not contain more objects than a developer can fit in his or her short-term memory.
- Distribute system intelligence vertically down narrow and deep containment hierarchies.

⁴ Riel, Arthur J., *Object-Oriented Design Heuristics*, © Addison Wesley, 1996.

Heuristics vs. Patterns

Heuristics can be stated in a sentence or two; Patterns take more space. Heuristics can provide the **glue** to help designers know when to select a particular pattern or how to combine them in useful ways during the design process.

Heuristics categories deal with...



Classes and Objects

- Encapsulation
- Cohesion
- Targeted abstractions



Object-Oriented Applications

- Group abstractions in analysis:
 - ✓ Boundary
 - ✓ Control
 - ✓ Entity

Relationships between Classes and Objects

- Minimize coupling
- Use of polymorphism

Notes

- When implementing semantic constraints, it is best to implement them in terms of the class definition.
- When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as the domain allows.
 - The semantic information on which a constraint is based is best placed in a central, third-party object when that information is volatile.
 - The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.
 - A class must know what it contains, but it should never know who contains it.
 - Objects which share lexical scope should not have uses relationships between them.

The Inheritance Relationship

- Inheritance should be used only to model a specialization hierarchy.
- Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.
 - All data in a base class should be private.
 - In theory, inheritance hierarchies should be deep.
 - In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory.
 - All abstract classes must be base classes.
 - All base classes should be abstract classes.
 - Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.
 - If two or more classes share only common data (no common behavior), then that common data should be placed in a class which will be contained by each sharing class.
 - If two or more classes have common data and behavior (i.e. methods), then those classes should each inherit from a common base class which captures those data and methods.
 - If two or more classes share only common interface (i.e. messages, not methods), then they should inherit from a common base class only if they will be used polymorphically.
 - Explicit case analysis on the value of an attribute is often an error.
 - Do not model the dynamic semantics of a class through the use of the inheritance relationship.
 - Do not turn objects of a class into derived classes of the class.
 - If you think you need to create new classes at runtime, take a step back and realize that what you are trying to create are objects.
 - It should be illegal for a derived class to override a base class method with a NOP method.
 - Do not confuse optional containment with the need for inheritance. Modeling optional containment with inheritance will lead to a proliferation of classes.
 - When building an inheritance hierarchy, try to construct reusable frameworks rather than reusable components.

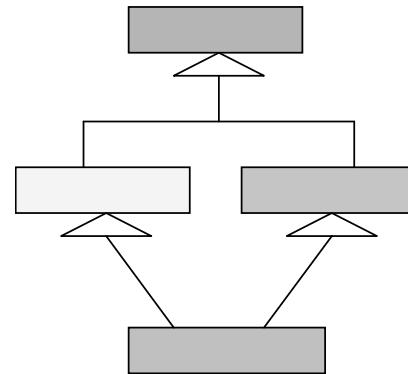
Heuristics vs. Patterns, continued

The Inheritance Relationship

- With inheritance, ask yourself two questions:
 - Am I a special type of the thing from which I'm inheriting?
 - Is the thing from which I'm inheriting part of me?

Multiple Inheritance

- Assume you have made a mistake and prove otherwise.



The Association Relationship

- Prefer containment to simple association if appropriate

Class-Specific Data and Behavior

- Use of class variables and methods

Physical Object-Oriented Design

- Do not allow physical design criteria to corrupt logical designs
- Do not change state of an object without going through its public interface.

Notes

Chapter 3 – Principles of Object-Oriented Design

Notes

Chapter Objectives

- Review **symptoms** of poor **design** choices
- Learn **basic principles** which address many OO software problems
- Be able to **write better code**
- Build a **foundation** for many of the GoF Patterns in Chapter 6 (and others)
- Accumulate Pattern **Vocabulary**

Notes

Symptoms of Poor Design:

- Rigidity** – the design is hard to change
- Fragility** – the design is easy to break
- Immobility** – the design is hard to reuse
- Viscosity** – it is hard to do the right thing
- Needless Complexity** – Overdesign
- Needless Repetition** – Mouse abuse
- Opacity** – Disorganized expression.

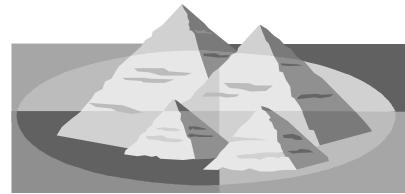
The Principles of Object-Oriented Design help eliminate the above problems (“smells”).

1. **SRP:** A class should have only one reason to change.
2. **OCP:** Software entities (classes, modules, etc) should be open for extension, but closed for modification.
3. **LSP:** Derived classes must be usable through the base class interface without the need for the user to know the difference.
4. **DIP:** Details should depend upon abstractions. Abstractions should not depend upon details.
5. **ISP:** Many client specific interfaces are better than one general purpose interface.
6. **REP:** The granule of reuse is the same as the granule of release. Only components released through a tracking system can be effectively reused.
7. **CCP:** Classes that change together belong together.
8. **CRP:** Classes that aren't reused together should not be grouped together.
9. **ADP:** The dependency structure for released components must be a directed acyclic graph. There can be no cycles.
10. **SDP** Dependencies between released categories must run in the direction of stability. The dependee must be more stable than the depender.
11. **SAP** The more stable a class category is, the more it must consist of abstract classes. A completely stable category should consist of nothing but abstract classes.

Overview of Principles¹ (Some Thumbnails)



1. **The Single-Responsibility Principle, SRP**
2. **The Open/Closed Principle, OCP**
3. **The Liskov Substitution Principle, LSP**
4. **The Dependency Inversion Principle, DIP**



5. **The Interface Segregation Principle, ISP**
6. **The Reuse/Release Equivalency Principle, REP**
7. **The Common Closure Principle, CCP**
8. **The Common Reuse Principle, CRP**



9. **The Acyclic Dependencies Principle , ADP**
10. **The Stable Dependencies Principle. SDP**
11. **The Stable Abstractions Principle. SAP**

► *Of the Symptoms of Poor Design (facing page), which most catches your attention?*

¹ These are from Robert C. Martin's *Agile Software Development*, © 2003, ISBN: 0-13-597444-5. See the book for more information and examples.

Notes

Comments: Tom DeMarco and Meilir Page-Jones called this principle “cohesion.” They talked about the functional relatedness of the elements of a module. Here we can talk about the forces that cause a class to change.

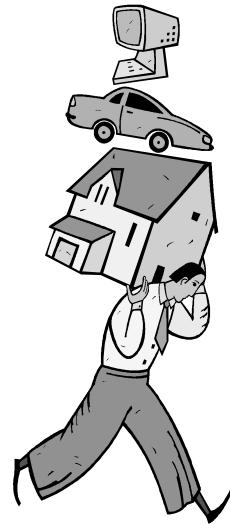
Example:

```
Modem.java – SRP Violation          // How many responsibilities here?  
  
Interface Modem {  
    public void dial (String pno);      // Two: Connection management (dial/hangup),  
    public void hangup();  
    public void send(char c);           // and data communication (send/recv).  
    public char recv();  
}
```

Should these two responsibilities be separated? It depends on how the application is changing. There is no need to separate them unless the 2 responsibilities change at different times and cause one set to be needlessly recompiled and redeployed.

Single-Responsibility Principle, SRP

Pattern Thumbnail: A class should have **only one reason to change**. Each responsibility in a class provides an axis of change. When requirements change, that ripples through as a change in responsibility among the classes. If a class has more than one responsibility it has more than one reason to change.



Forces:

- Designs become fragile and hard to maintain when responsibilities become coupled.
- Changes to coupled responsibilities cause otherwise unnecessary recompilation and redeployment of classes.

Question: Should you still apply SRP (or any principle) if there is no symptom? In other words, is there an axis of change if no changes actually occur?

Usage: Java has tons of small, highly targeted classes.

Solution: Create classes with a single responsibility – and combine them later as needed by implementing multiple interfaces, etc. Refactor by using Façade or Proxy patterns to separate responsibilities.

Consequences: Simplified, improved maintenance.

Notes

Comments: The idea is that once a class has been approved for use having gone through code reviews, unit tests and other qualifying procedures, you don't want to change the class very much, just extend it. In practice the open-closed principle simply means making good use of abstraction and polymorphism.

Anti-Example:

```
class ResourceAllocator {
    public boolean allocate(int resourceType) {
        int resourceId;
        switch (resourceType) {
            case TIME_SLOT: resourceId = findFreeTimeSlot();
                markTimeSlotBusy(resourceId);
                break;
            case SPACE_SLOT: resourceId = findFreeSpaceSlot();
                markSpaceSlotBusy(resourceId);
                break;
            default: System.err.println("invalid resource");
        }
    ...
}
```



- ▶ What's the potential problem here?

Any module that manipulates solely abstractions will never need to change since the abstractions are fixed -- closed. However, deriving a new class from the abstractions can change the behavior of the module – open for extension.

Open-Closed Principle, OCP

Pattern Thumbnail: A principle used in OOPL which states that a class must be open and closed, where open means it has the ability to be extended and closed means it cannot be modified other than by extension. Change should be ADDITIVE, not INVASIVE. In other words, add new functionality by adding new code, not by editing old code.

Forces: Change happens. Plan for it; manage it.

Question: How can an immutable module exhibit anything but fixed behavior?



Usage: The cornerstone principle of OOD, laid by Bertrand Meyer.

Solution: The answer is abstraction. An abstract base class or an interface.

Consequences: Reusability and maintainability.

Old code does not have to be unit tested due to new changes; developers adding code (such as a new resource type in the above example) don't need to understand inner workings for the old code.

Further abstractions can be developed to group together similar algorithms (such as a base class for all resource pools implemented with free/busy queue, base class for all resource pools implemented as timebound bookings (like movie tickets), etc.

Notes

Comments: Procedural code gets information then makes decisions. Object-oriented code tells objects to do things. To ask is a query; to tell is a command.

It's very easy to get lulled into examining some referenced object and then calling different methods based on the results. The biggest danger is that by asking for data from an object, you are only getting data; you're not getting an object in the large sense. Even if the thing you received from a query is an object structurally (e.g., a String) it is no longer an object semantically. An object knows what the string contents "RED" mean (Color of the car? Current condition of the account? Owners last name?)— data does not. Instead, tell the object what you want. Let it figure out how to do it. Think declaratively instead of procedurally.

Example: Suppose you have a container object. You could expose iterators for the objects held in this container (as many of the JDK core routines do). Or, you could provide a method that would run some function over all members in the collections for you: "Run this function over all contained items, I don't care how."

Declarations:

```
public interface Applicable {
    public void each( Object anObject);
}

public class SomeClass {
    void apply(Applicable);
}
```

Calls:

```
SomeClass foo;
...
foo.apply(new Applicable() {
    public void each(Object anObject) {
        // do what you want to anObject
    }
});
```

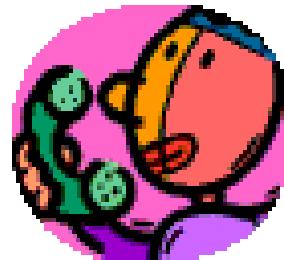
Participants: Client, Server

Tell vs. Ask

Style Thumbnail: Keep your orders (commands) and your facts separate.

Forces: As the caller you should not be making decisions based on the state of the called object that result in you then changing the state of the object.

The logic you are implementing is probably the called object's responsibility, not yours. For you to make decisions outside the object violates its encapsulation.



Question: How does our procedural background lay a trap for us when we switch paradigms?

Consequences: It is easier to stay out of this trap if you start by designing classes based on their responsibilities. You can then progress naturally to specifying commands that the class may execute, as opposed to queries that inform you as to the state of the object.

Ensure a correct division of responsibility by placing the right functionality in the right class without causing excess coupling to other classes. Expose the minimum amount of state necessary.

- So, what does this say about methods?

Notes

Comments: The combined modifier-query style is common in C, C++, and Java, but this does not automatically make it desirable.

Example: In a database application, `fetchRow()` vs. `currentRow()` illustrates command vs. query.

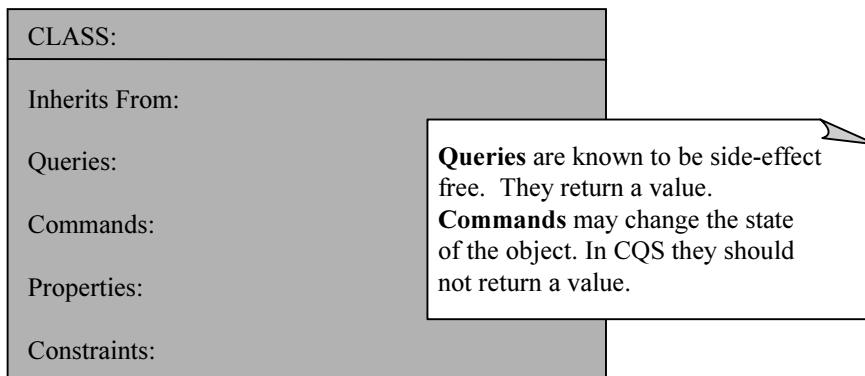
It is not possible to query an Iterator object as to its current value without causing it to advance to the next value.

```
public interface Iterator {
    boolean hasNext(); //pure query
    Object next();    //both
    void remove();    //pure command
}
```

The result of combining query and command responsibility in the same method leads to a mismatch with the structure of a `for` loop where initialization, continuation, access, and advancement are separated for clarity, defining the loop signature. This can obstruct assertion-based programming and typically requires additional temp variables:

```
for(Iterator iterator=collection.iterator(); iterator.hasNext(); ) {
    Object current=iterator.next();
    // use current several times
}
```

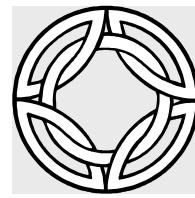
Diagram: Thinking of each class as follows allows you to think in terms of Command/Query Separation.



Command/Query Separation (CQS)

Style Thumbnail: Ensure that a method is clearly *either* a command *or* a query.

Question: How could you redefine the Iterator interface to achieve CQS?



Usage: Eiffel requires that you use only side-effect free methods within an assertion.

Consequences:

Positive: While this may seem rather draconian, it leads to simple, clear, well-specified class interfaces. If you can assume that evaluation of a query is free of any side effects, then you can:

- use queries from within a debugger without affecting the test
- create built-in, automatic regression tests
- evaluate class invariants, pre-and post-conditions.

Negative: This style can encourage property-style programming and must be held in tension with Combined Method. This is a stylistic problem that becomes a practical problem in the presence of threads, requiring explicit external synchronization by the object user (when the object designer should solve the problem).

Notes

Comments: Methods should perform one identifiable task and should be only a few lines long. A method's operations should be at the same level of abstraction. Kent Beck (author of *Smalltalk Best Practice Patterns*) once said that some of his best patterns are those that he thought someone would laugh at him for writing.

Example:

(“extract method” ---- to compose ---- this new method)

```
void printOwing(double amount) {  
    printBanner();  
  
    // print details  
    System.out.println("name: " + name);  
    System.out.println("amount: " + amount);  
}  
  
void printOwing(double amount) {  
    printBanner();  
    printDetails(amount);  
}  
  
void printDetails(double amount) {  
    System.out.println("name: " + name);  
    System.out.println("amount: " + amount);  
}
```

Composed Method

Pattern Thumbnail: A small, simple method that is easy to understand and whose name reveals the intent.

Forces: If a method seems awkward, it is. If a method requires a comment, it isn't simple enough or appropriately named.

Implementation:

Top-Down

Invoke several smaller methods

The smaller methods need not exist yet

Bottom-Up

Factor common code into a single method

Put long loop bodies into separate method

If lines of code need a comment, place the code in a method with intention-revealing name

Consequences: Minimizes code copying in subclasses and number of methods needing changes in subclass. Two or more messages to another object is suspicious.



Note: The entirety of Chapter 6 is on Composing Methods in Martin Fowler's *Refactoring: Improving the Design of Existing Code*

Notes

Comments: Interfaces that offer predominantly fine-grained methods can appear minimal and cohesive—both desirable. However, some interfaces are primitive without being cohesive, simplistic rather than simple. Class users (rather than class designers) are forced to work harder to achieve common tasks and navigate subtle ordering dependencies (temporal coupling) that exist between methods. This is both tedious and error-prone, leading to code duplication. The overall effect of Combined Method is to support a more transaction-like style of method design.

Example: Acquiring a resource if it is available, and continuing with something else if it is not. Assume in the following interface that the acquire method blocks until the resource becomes available.

```
interface Resource {
    boolean isAcquired();
    void acquire();
    void release();
    ...
}

class ResourceExample {
    ...
    public void example() {
        boolean acquired = true;
        synchronized(resource) {
            if (!resource.isAcquired())
                resource.acquire();
            else
                acquired=false;
        }
        if(!acquired) ...
    }
    private Resource resource;
}
```

//This design is an unsuitable application of CQS.
//It fails if you introduce a Proxy arrangement:
// class ActualResource implements Resource { ... }
// class ResourceProxy implements Resource { ... }
//
//The synchronized block fails because the proxy rather
//than the target is synchronized. This can have a rather
//fundamental impact on the correctness of a system.
//
//Because the use of proxy is transparent behind an
//interface ... (given an interface to an object, how can you
//be sure that you are talking to the target directly?) there
//is little the caller can do to guarantee good behavior.

A combined Method resolves the issues, making the presence of concurrency and indirection more transparent.

// The following is simpler as well as correct:

```
interface Resource {
    ...
    boolean tryAcquire();
    ...
}
```

```
class ResourceExample {
    ...
    public void example() {
        if (!resource.tryAcquire() ) ...
    }
    private Resource resource;
}
```

Combined Method



Pattern Thumbnail: Combine methods that are commonly used together to guarantee correctness and improve efficiency in threaded and distributed environments.

Forces:

- If two actions commonly performed together must follow commit-or-rollback semantics—they must both be completed successfully or the effect of the first one must be rolled back if the second fails—and both are expressed as separate methods, clean up falls to the class user rather than the class supplier.
 - Exceptions, threading and distributed environments make it even more critical to guarantee the desired outcome. Combine methods that must be executed together coherently.

Consequences:

- CQS and Combined Method can reinforce each other; other times they are in tension.
 - Combined Method is driven by correctness in concurrent and distributed environments and desire to encapsulate rather than leak nontransparent details.

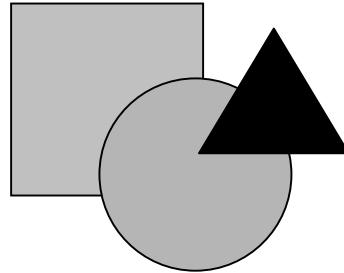
Notes

Comments: First written by Barbara Liskov as follows: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

If a function does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class. Such a function violates the open-closed principle because it must be modified whenever a new derivative of the base class is created.

Anti-Example: Using C++ RTTI to select a function based upon the type of an object is a violation of this principle. The drawShape() function is badly formed; it must know about every possible derivative of Shape.

```
void drawShape( const Shape& s)
{ if (typeid(s) == typeid(Square))
    drawSquare(static_cast<Square&>(s));
  else if (typeid(s) == typeid(Circle))
    drawCircle(static_cast<Circle&>(s));
}
```



Liskov Substitution Principle, LSP (aka Design by Contract)

Pattern Thumbnail: An instance of a subclass should be substitutable wherever a base class is expected.

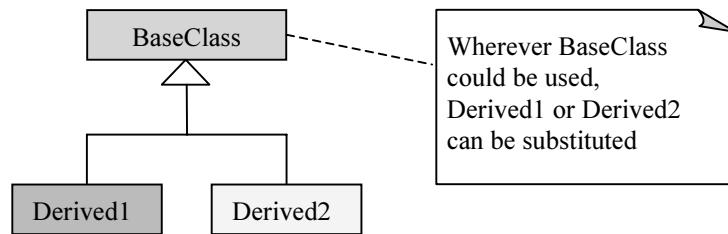


Forces: If you make changes to a derived class or add new derived classes, you should not have to change the base class. Well-designed code should be able to be extended without modification to old, already working code.

From Meyer: "When redefining a method in a subclass, you may only replace its precondition by a weaker one, and its postcondition by a stronger one."

⇒ **Question: Explain Meyer's statement (above)...**

Diagram:



Question: What are the consequences of violating this principle? How is this "Design by Contract?"

Suppose Square inherits from Rectangle which inherits from base class Shape? What could go wrong?

Consequences:

- More robust, maintainable, reusable applications.
- Guidance for the use of public inheritance. Otherwise there is a need to check type, a violation of the Open-Closed Principle.

Notes

Comments: The word “inversion” is used because more traditional software development methods tend to create software structures in which high level modules do depend on low level modules, and in which abstractions depend upon details. Thus, the dependency structure of a well designed object-oriented program is inverted.

Example: The Copy Program – Listings 1 – is NOT reusable in any context not involving keyboard/printer. Listing 2 adds new interdependencies to the system and will eventually become rigid and fragile.

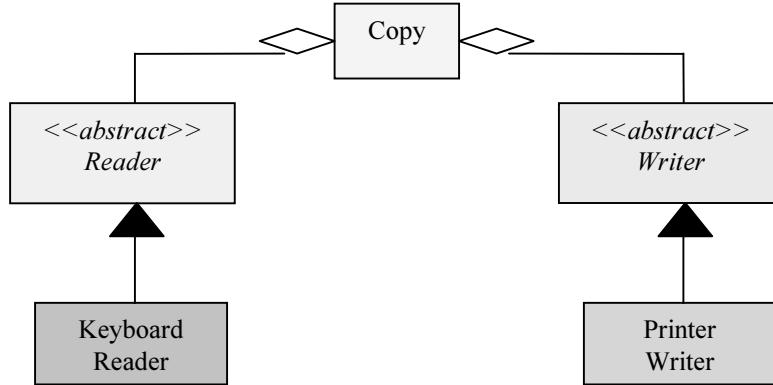
Listing 1.

```
void copy() {
    int c;
    while ((c = readKeyboard()) != EOF)
        writePrinter(c);
}
```

Listing 2.

```
enum OutputDevice {printer, disk};
void copy(OutputDevice dev) {
    int c;
    while ((c = readKeyboard()) != EOF)
        if (dev == printer)
            writePrinter(c);
        else
            writeDisk(c);
}
```

Diagram:



We could reuse copy freely if we could make it independent of the details and be able to copy from any input device to any output device.

Dependency Inversion Principle, DIP

Pattern Thumbnail: High level modules should not depend upon low level modules.
Both should depend upon abstractions.

Abstractions should not depend upon details; details should depend upon abstractions.

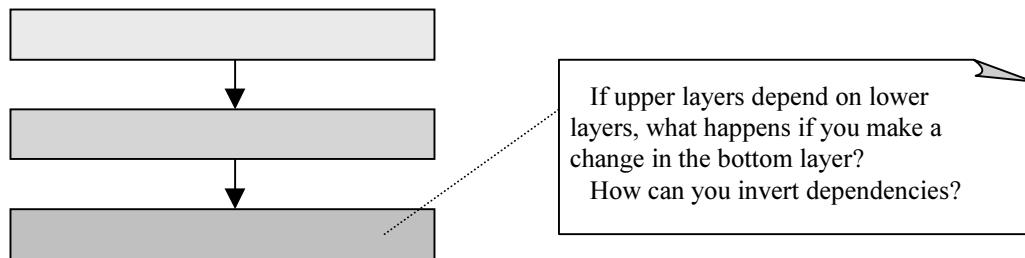
Forces: The module containing the high level policy, the copy() module, is dependent upon low level detailed modules that it controls.

Question: How would you rewrite the example code so that Copy is independent of the Reader and Writer devices and copy() can then be used in many different contexts? (For C programmers: Is there device independence within the stdio.h library with getchar and putchar? Is this an example of dependency inversion?)

Usage: This principle is at the heart of framework design.

Consequences:

- High level modules contain policy decisions and business models of an application.
- If they depend upon lower level modules, then changes to the lower level modules can have direct effects on them and force them to change – an absurd predicament.
- High level modules should be reusable and not depend upon low level modules in any way.
- Dependency Inversion can be applied wherever one class sends a message to another.



Notes

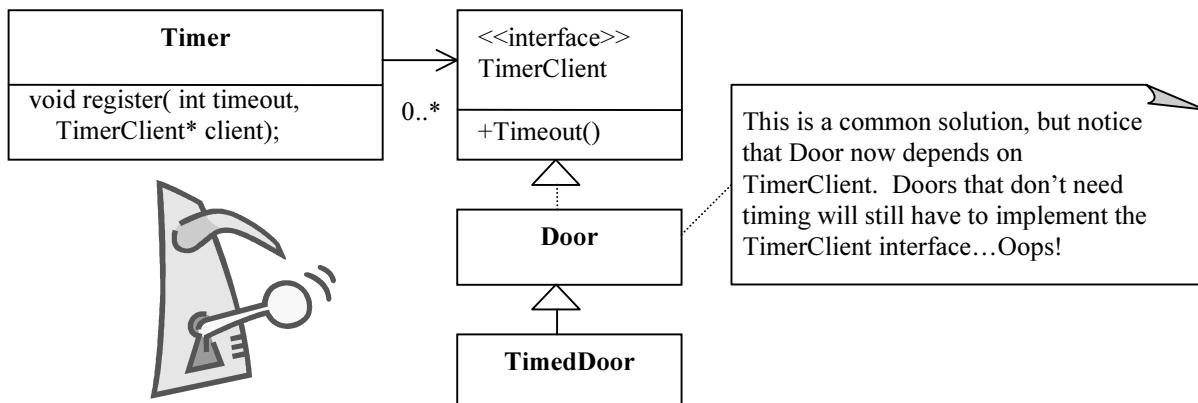
Comments: This principle deals with the disadvantages of large interfaces. Classes with “fat” interfaces are NOT cohesive; some clients will use a portion of the interfaces, other clients will use other portions.

The ISP recognizes that some objects may require noncohesive interfaces but suggests clients should not know about them as a single class. Clients should instead know about abstract base classes that have cohesive interfaces.

Example of Interface Pollution: Consider a security system with Door objects that can be locked/unlocked, and which know whether they are open or closed.

```
class Door {
public:
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual bool isOpen() = 0;
};
```

Now suppose we have a TimedDoor which needs to sound an alarm when the door has been left open for too long. The TimedDoor object communicates with another object, Timer. We could structure a solution like this:



A better solution is to let **TimedDoor** inherit from both **Door** and **TimerClient**. Although clients of both the class and the interface can make use of **TimedDoor**, neither actually depends on the **TimedDoor** class.

Note: You could also create a **DoorTimerAdapter** object if translation is needed. It would inherit from **TimerClient** and delegate to **TimedDoor**, but needs a new object created every time you register a time-out.

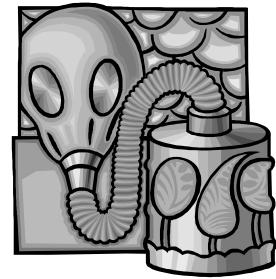
Interface Segregation Principle, ISP

Pattern Thumbnail: Clients should not be forced to depend upon methods that they do not use.

Interfaces belong to clients, not to hierarchies.

Forces: The interfaces to existing classes and components often change; these changes can have a huge impact and force the recompilation and redeployment of a very large part of the system.

Fat classes cause bizarre and harmful couplings between their clients – when one client forces a change on the fat class, all other clients are affected.



Question: Suppose you accommodate new derivatives of a hierarchy by implementing do-nothing methods in the base interface, which you can then override for classes which need to actually do something with these methods? Discuss this approach in light of LSP and potential maintenance and reusability.

How does ISP relate to SRP? To DIP?

Consequences:

Positive: Breaks dependence of clients on methods they don't use and allows clients to be independent of each other. Fat classes can inherit all the client-specific interfaces and implement them.

Negative: There has to be a balance; a class with hundreds of different interfaces, some segregated by client and others segregated by version, would be a nightmare.



Notes

Comments: You can play with yourself.

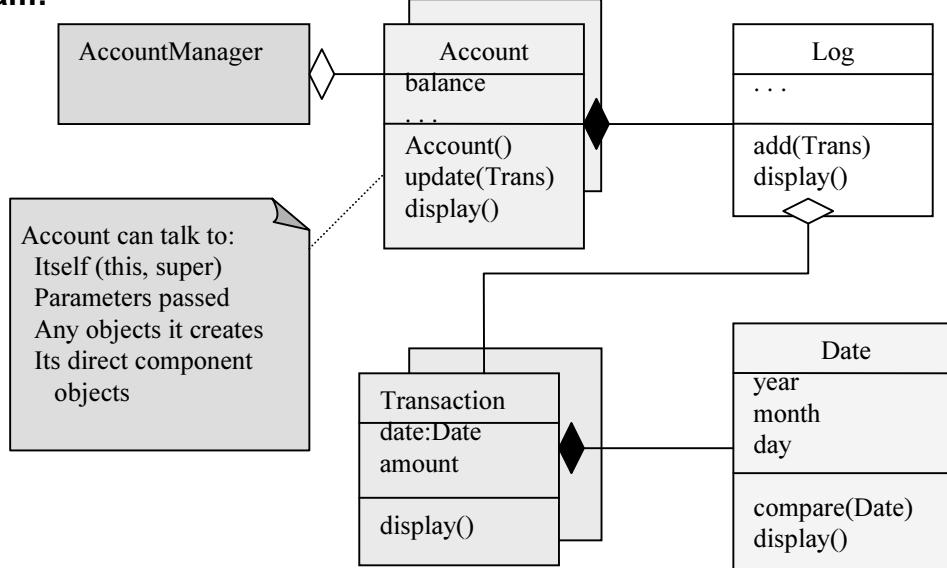
You can play with your own toys (but you can't take them apart).

You can play with toys that were given to you.

You can play with toys you've made yourself. (from a Usenet posting)

Example: If you have a method that takes a “BigMac” as an argument and needs to access the seeds on the pickles, you don’t have to know about the lettuce, onions, and sesame seed bun, nor care if their order is changed.

Diagram:



Usage: Jet Propulsion Laboratory: “I have been using LoD pervasively since about 1990, and it has taken firm hold in many areas of the JPL. Major systems which have used LoD extensively include the Telemetry Delivery System (a real-time database begun in 1990), the Flight System Testbed, and Mars Pathfinder flight software (both begun in 1993). We are going to use LoD as a foundational software engineering principle for the X2000 Europa orbiter mission. I also used it within a couple of commercial systems for Siemens in 91-93, including a Lotus Notes like system, and an email system.” – David E. Smyth

Demeter home page: <http://www.ccs.neu.edu/home/lieber/LoD.html>

Law of Demeter (aka “Don’t Talk to Strangers”) *a style guideline...*

Style Thumbnail: “Only talk to your immediate friends.”

This “law:”

- minimizes logical coupling, maximizes physical coupling
- assumes “least structural knowledge”
- where possible, avoids middlemen

The idea is to assume as little as possible about the structure and properties of instances and their subparts. It is okay to request a service of an object instance but if you reach into that object to access another sub-object and request a service of that sub-object, you are assuming knowledge of the deeper structure of the original object that was made available. Never call a method on an object you got from another call or on a global object.



Question: On facing page, suppose Account could “talk to” Date directly?

Consequences:

- Adhering to the Law of Demeter (LoD) reduces coupling.
- Violations of the Law of Demeter are probably indications your code needs refactoring.
 - LoD may increase maintainability and adaptiveness of your software, but you also end up having to write lots of little wrapper methods to propagate methods calls to its components, which can add noticeable time and space overhead.
 - LoD makes unit testing easier.

Notes

Example: Consider building software for an ATM system.

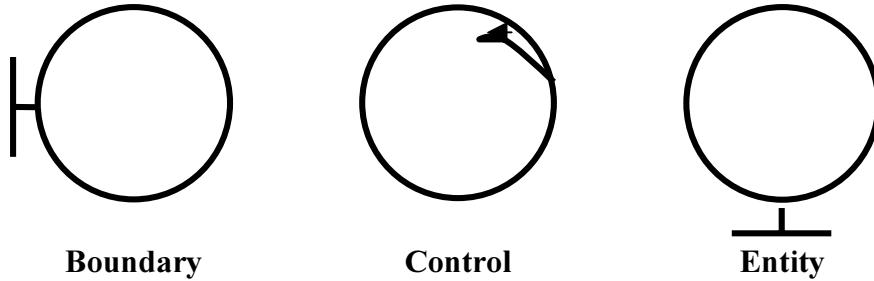
Name the **boundary** elements (interface devices):

Name the potential **entity** elements:

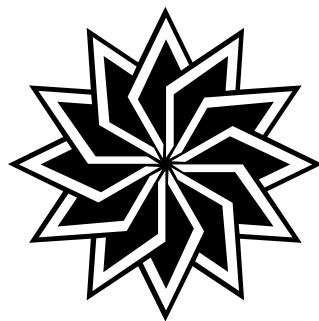
So what else is needed? (**Controller** elements). What do they do? How reusable are they?

Review of the Life Cycle Process (suggestions only)

1. Start with a textual Use Case.
2. Use a Glossary – document vocabulary as it evolves.
3. From a Use Case, make a 1st pass at Dynamic Class model:



4. Trace message flows (rough collaboration or sequence diagrams).
5. Start a static Class Model.
6. Implement a slice of the model, end-to-end, touching all aspects (executable architecture).



Then **ITERATE ITERATE ITERATE**

Exercises

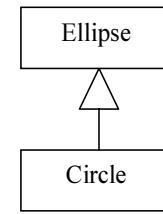
1. Open-Closed Principle:

Rewrite the ResourceAllocator anti-example code (see workbook p. 44) so that it follows the Open-Closed Principle and is completely transparent to the actual resource types being supported. Consider an abstract class or interface ResourcePool from which TimeSlotPool and SpaceSlotPool would both inherit; and a ResourceAllocator class with methods allocate() and free().

2. Circle/Ellipse Dilemma:

Suppose you have the inheritance hierarchy at right, where Ellipse has attributes (focusA, focusB, and major-axis) and methods such as setFoci(Point a, Point b).

What are the problems for Circle? What “fixes” would be needed?



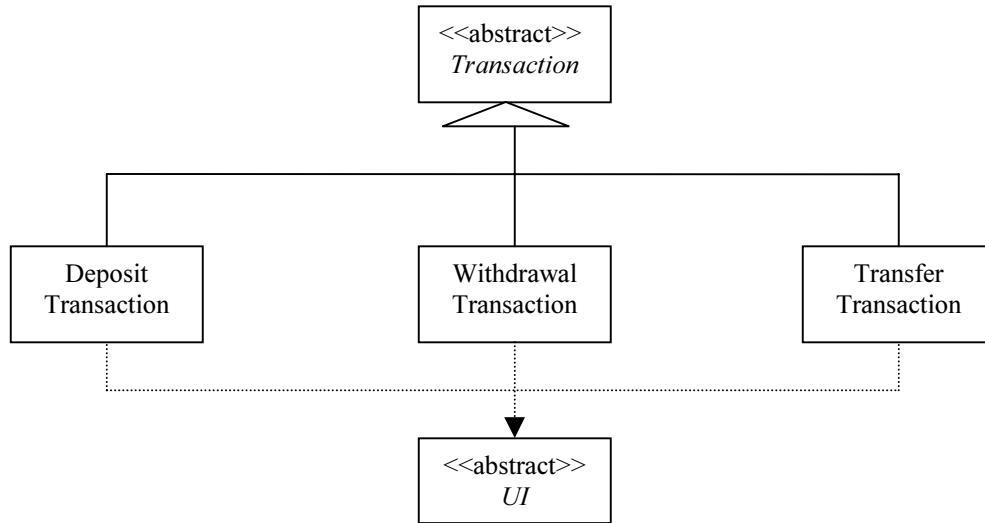
Suppose a client of this hierarchy writes a method similar to the following:

```
void f(Ellipse e) {  
    Point a = new Point(-1,0);  
    Point b = new Point(1,0);  
    e.setFoci(a, b);  
    e.setMajorAxis(3);  
    assert(e.getFocusA() == a);  
    assert(e.getFocusB() == b);  
    assert(e.getMajorAxis() == 3);  
}
```

Then suppose method f() is called, but instead of passing an Ellipse, a Circle is passed?

What principle is violated, and what are the repercussions?

3. ISP Exercise: ATM Transaction Hierarchy



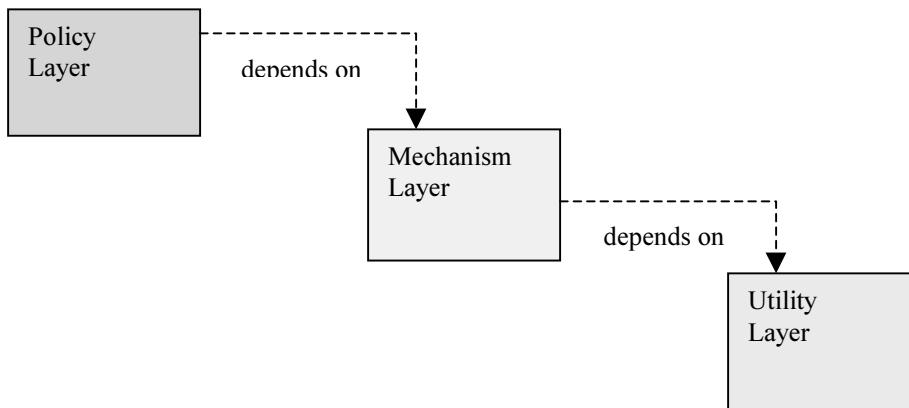
Identify some of the methods required by the UI interface to support behavior needed by the various transactions. What “smells” about this? (Consider *Rigidity* and *Fragility*.)

What would be the impact on the UI interface of adding new subclasses for the various transaction classes (e.g. PayGasBillTransaction, TransferWithCharges, TimeDependentTransfer, InternationalTransfer, etc)?

How could you restructure to better support ISP?

Draw the UML for both the multiple inheritance solution and the DoorTimerAdapter solution in the example on workbook page 58.

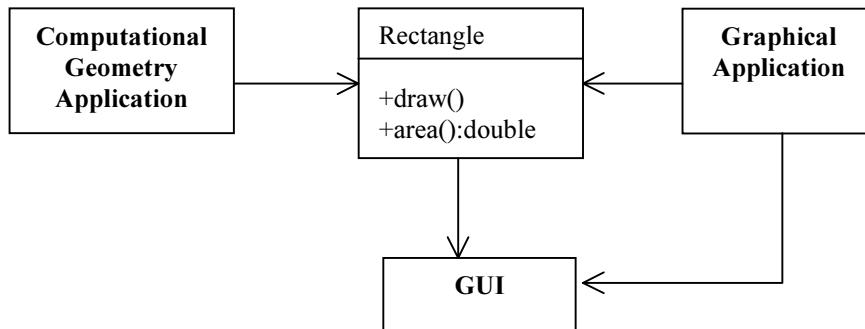
4. **DIP Exercise:** A naïve interpretation of layering might result in the diagram below.



What happens if you make changes in the Utility Layer?

How can you create a more appropriate model? Draw it below!

5. **SRP Exercise:** Consider the Rectangle class below. Rectangle has two methods shown, one for computing area and one for drawing on the screen. Thus, two responsibilities. The Computational Geometry Application needs Rectangle to help with mathematics of geometrical shapes, but never draws on the screen. This is a violation of SRP.



What are the potential problems? Consider especially the GUI issues, and what could happen if the Graphical Application requires some changes in Rectangle.

Redraw the UML to separate the responsibilities.

Chapter 4 – Principles of Package Architecture

Notes

Chapter Objectives

- Understand some of the **dependency issues** in creating packages
- Learn how to better **organize classes** to reduce dependencies
- Be able to apply **Martin's Metrics** to package organization

Notes

Release Reuse Equivalency Principle, REP:

Users will be unwilling to use the element if they have to upgrade every time it changes. Packages are the unit of release; they are also the unit of reuse. Group reusable classes together.

Common Closure Principle, CCP:

Group classes that change together into the same package to minimize package impact from release to release.

Common Reuse Principle, CRP:

Classes not reused together should not be grouped together. When a package changes, all clients of that package must verify that they work with the new package – even if nothing they used within the package actually changed.

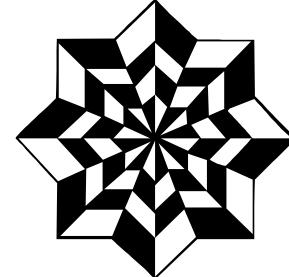
Note: These are mutually exclusive principles; they cannot be simultaneously satisfied. REP and CRP address reuse; CCP addresses maintenance. CCP strives to make packages as large as possible; CRP tries to make packages very small.

Principles of Package Architecture

Package Cohesion Principles

How to choose which classes belong in which packages?

- **Release Reuse Equivalency Principle, REP**
- **Common Closure Principle, CCP**
- **Common Reuse Principle, CRP**



Packages are not fixed in stone. As architecture stabilizes, refactoring may be necessary to maximize REP and CRP for external users.

Notes

Draw in the lines on the facing page to carry out the “what-if” in the note below:

2. Now suppose CommError displays a message on the screen. CommError becomes dependent on GUI. Now what happens when there is a new release of the Protocol package? (Draw in the line...) This would be cyclic and disastrous! Instead, add a new package, MessageManager, under CommError. Needed classes from GUI are moved there; both CommError and GUI depend on it.

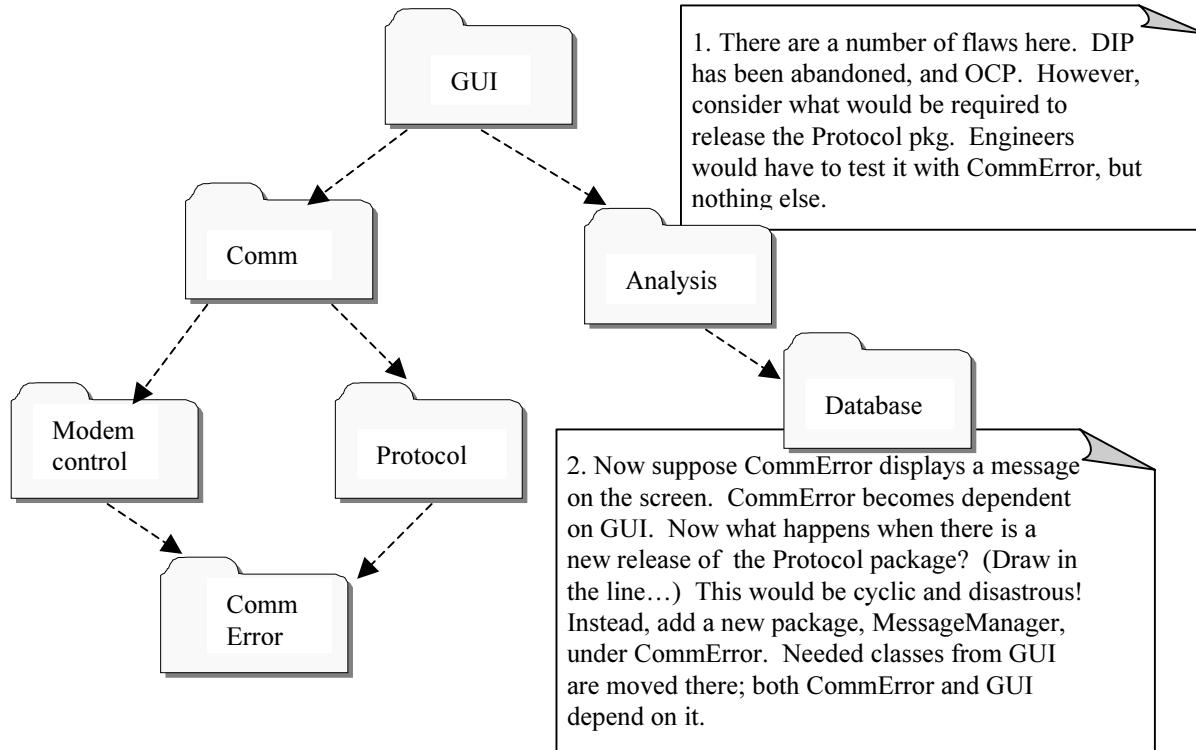
Now draw the picture which shows breaking the cycle.

Principles of Package Architecture, continued

Package Coupling Principles

How to decide interrelationships between packages?

Acyclic Dependencies Principle, ADP:



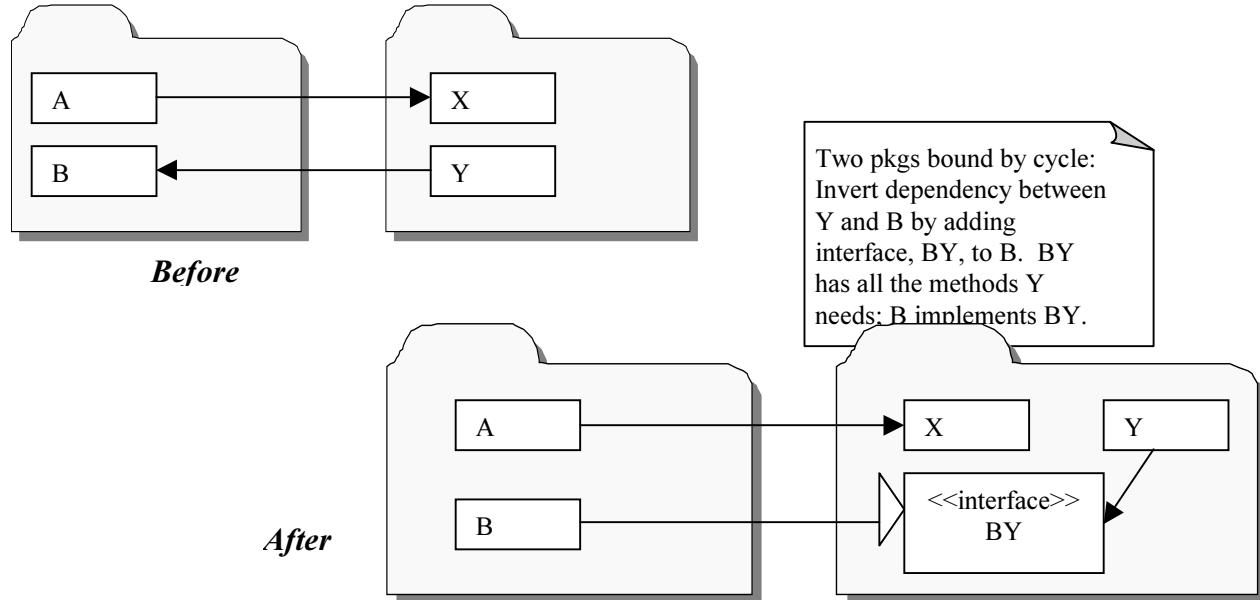
Notes

The **After** diagram on the facing page creates the interface needed by package B and puts it into Y's package. Notice how this avoids a cyclic dependency.

Principles of Package Architecture, continued

Package Coupling Principles

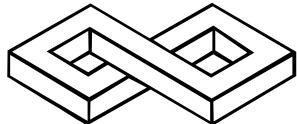
Another technique for breaking cycles:



- Who “owns” an interface, the client or the server?

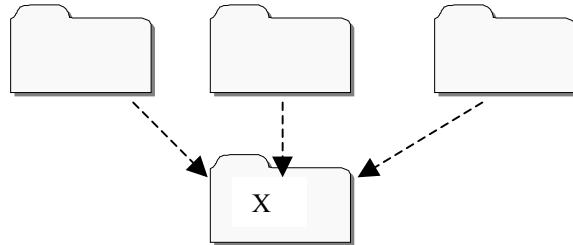
Notes

Principles of Package Architecture, continued

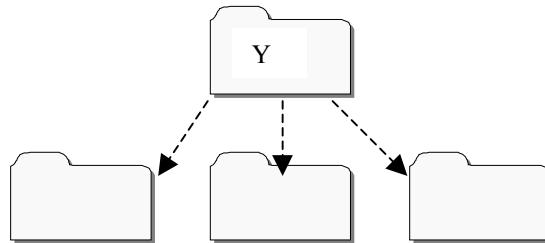


Package Coupling Principles, cont.

Stable Dependencies Principle, SDP:



Package X (above) is a stable package.
It has 3 packages depending on it – good reason not to change.



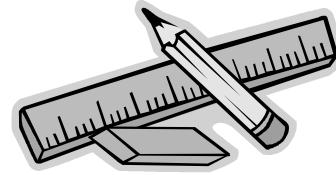
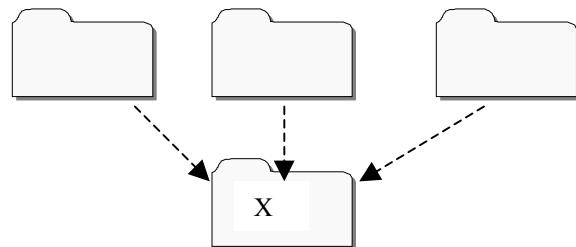
Package Y (above) is dependent on 3 packages.
Changes may come from 3 external sources.
Y has no other packages depending upon it (so is irresponsible).

Notes

Note: SAP is simply a restatement of DIP.

Principles of Package Architecture, continued

Stable Abstractions Principle, SAP:



Consider Package X in the top diagram above. If the stable packages at the bottom are also highly abstract, then they can be easily extended. Therefore, compose applications from instable packages that are easy to change, and stable packages that are easy to extend. This is goodness.

So, how to measure stability and abstractness? Coming next...

Notes

Compute:

If $C_e = 0$, Instability = ?

If $C_a = 0$, Instability = ?

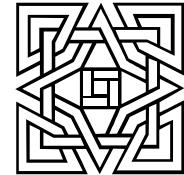
Martin¹ Package Metrics

Measure Instability

Ca: Afferent Coupling – number of classes outside package that depend on classes inside package
(incoming dependencies, like package X on previous pages)

Ce: Efferent Coupling – number classes outside package that the package depends on
(outgoing dependencies like package Y on previous pages)

$$I: \text{Instability} - I = \frac{Ce}{Ca + Ce} \quad \text{range } [0,1]$$



- SDP rephrase: depend on packages whose I-metric is lower (more stable) than yours.

¹ “Martin” is Robert C. Martin, @ www.objectmentor.com

Notes

Describe the situation where **Na = 0**.

Describe the situation if **Na = Nc**.

D' = 0 means what?

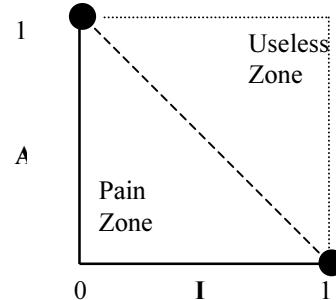
D' = 1 means what?

Measure Abstractness

Nc: number of classes in package

Na: number of abstract classes in package

$$\mathbf{A: Abstractness - } A = \frac{Na}{Nc} \quad \text{range } [0,1]$$



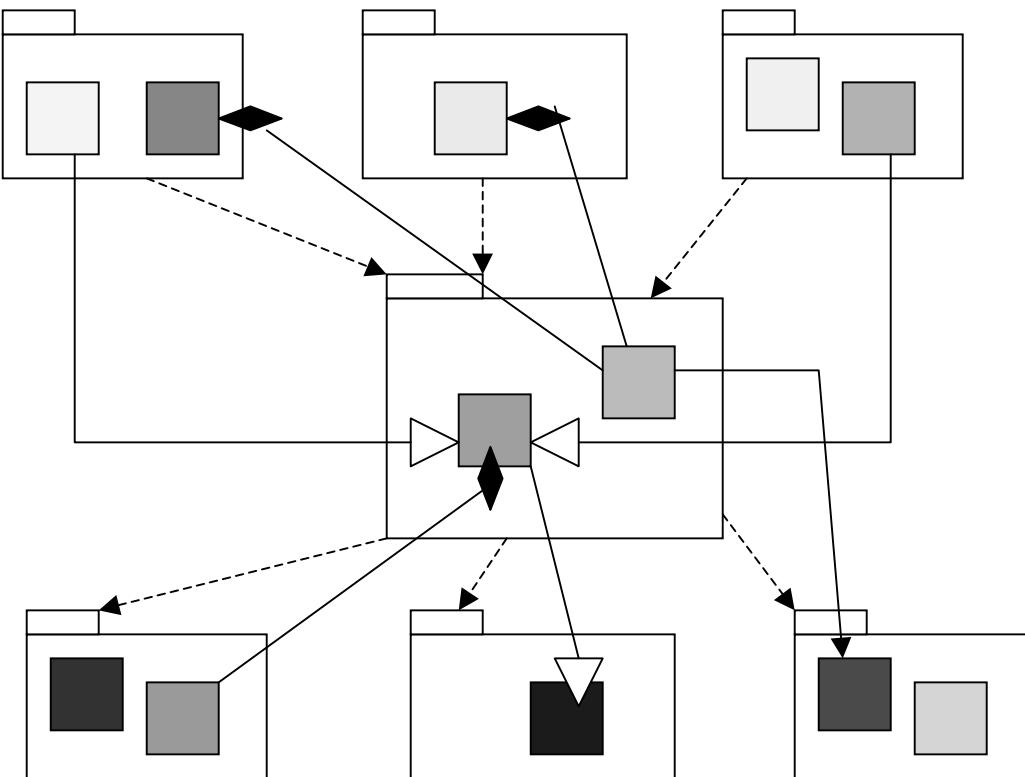
Distance Metrics

$$\mathbf{D: Distance - } (\text{from line}) \quad D = \frac{|A + I - 1|}{\sqrt{2}} \quad \text{range } [0, \sim 0.707]$$

$$\mathbf{D': Normalized Distance - } |A + I - 1| \quad \text{range } [0, 1]$$

Exercises

1. Martin Package Metrics Exercise:



Determine: Ca

Ce

I

[Remember to count # of classes, not # of packages...]

Chapter 5 – Basic Object-Oriented Design Patterns

Notes

Chapter Objectives

- Become more familiar with some **basic patterns**:
 - **Delegation vs. Inheritance**
 - Programming to an **Interface**
 - Creating an **Immutable Interface**
 - Use of **Null Object** pattern
 - Use of **Marker Interface**
- Learn the **GRASP** patterns for general responsibility assignment

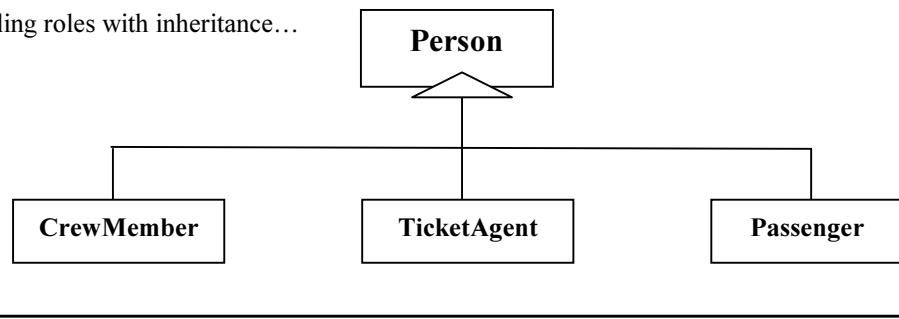
Notes

Comments: Delegation is a way to extend and reuse the functionality of a class by creating an additional class with added functionality that uses instances of the original class to provide the original functionality. You can *always* replace inheritance with object composition as a mechanism for code reuse. Many other patterns use delegation – notice how in Interface, Decorator, Proxy, State, Strategy, Visitor... and others. The GoF authors state: “Favor object composition over class inheritance.”

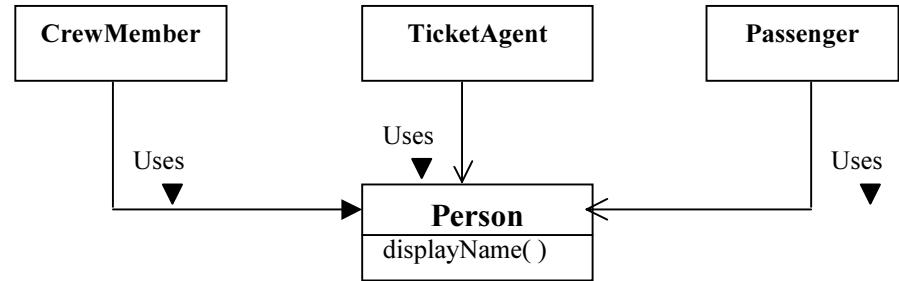
Example: The same person can play different roles at different times. Inheritance is static. Suppose a TicketAgent in the first diagram below is also a passenger? How would you model combinations? Java has no multiple inheritance (MI) and MI makes for difficult designs so the objective is to avoid it.

Diagram:

Modeling roles with inheritance...



Modeling roles with delegation...



Participants: Delegator, Delegate. (Delegator uses Delegate)

► **Which class is the Delegator above? The Delegate?**

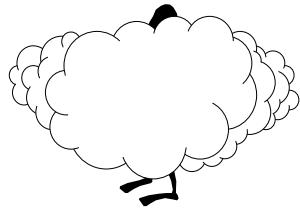
Delegation vs. Inheritance (GoF p. 20)

Pattern Thumbnail: Inheritance is a common way to extend and reuse functionality of classes yet delegation is a more general and flexible way to accomplish the reuse. In delegation, a class calls another class's methods instead of inheriting them.

Forces: If an object needs to be a different subclass of a class at different times, don't use inheritance – use delegation. If a subclass needs to hide a method or variable inherited from a super class, don't use inheritance. If your class really is modeling a role played, don't use inheritance – model IS-A-ROLE-PLAYED-BY with delegation. (see facing page).

Usage: The Java API is full of delegation; it is the basis for Java's event model, where an event source (such as aButton) sends the event context object (anEventObject) to an event listener object (an object with the code for actionPerformed()).

Consequences: Delegation is less obvious than inheritance and therefore potentially harder to understand.



There are also some runtime inefficiencies.

- (1) Use consistent naming schemes,
- (2) clarify the purpose of delegation with comments,
- (3) follow the “Don’t Talk To Strangers” pattern, and
- (4) use well-known design & coding patterns (no obfuscated code!)

Notes

Comments: Any applicant (server) who implements that interface can potentially do the job needed for the client – the client does not have to depend on any particular applicant. Interface pattern is often used with Delegation, and many other patterns use the Interface pattern – it is basic.

Example:

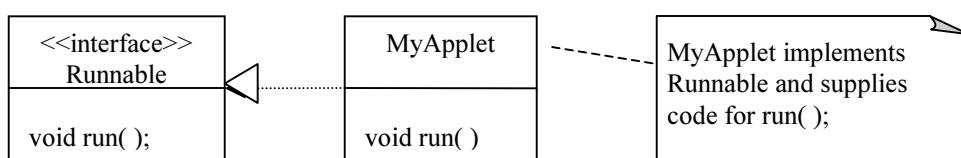
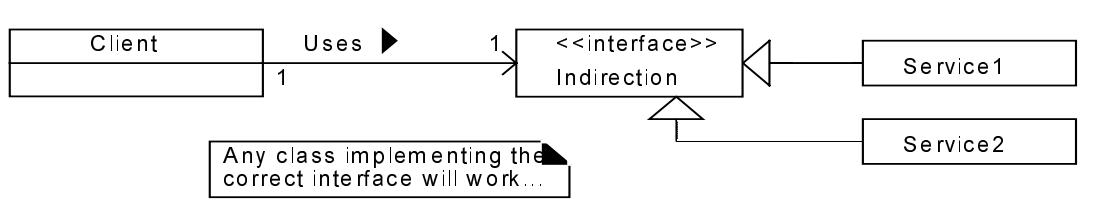


Diagram:



Participants: Client, IndirectionIF, Service.

Usage: Java API defines the interface “java.io.Filenamefilter” which declares a method to allow you to decide if a named file should be included in a collection or not. Lots of IO operations are accomplished through interfaces. RMI depends on interfaces. So do many other operations.

Interface (GoF p. 13)



An INTERFACE is like a job description – and any class meeting those qualifications can apply for the job.

Pattern Thumbnail: Keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface.

Forces:

1. If instances of a class must use another object and that object is assumed to belong to a particular class, the reusability of the class is compromised.
2. Suppose instances of a class use other objects but only rely on certain methods. Limit the dependency by defining an interface that has just those methods and design your class to use the interface.

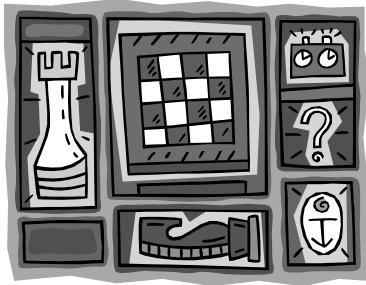
Consequences:

- Promotes loose coupling since a class can use services from any class that meets the necessary interface.
- Like any other form of indirection, this can make a program more difficult to understand.

This technique is basic for almost all of the GoF Patterns in the next chapter!

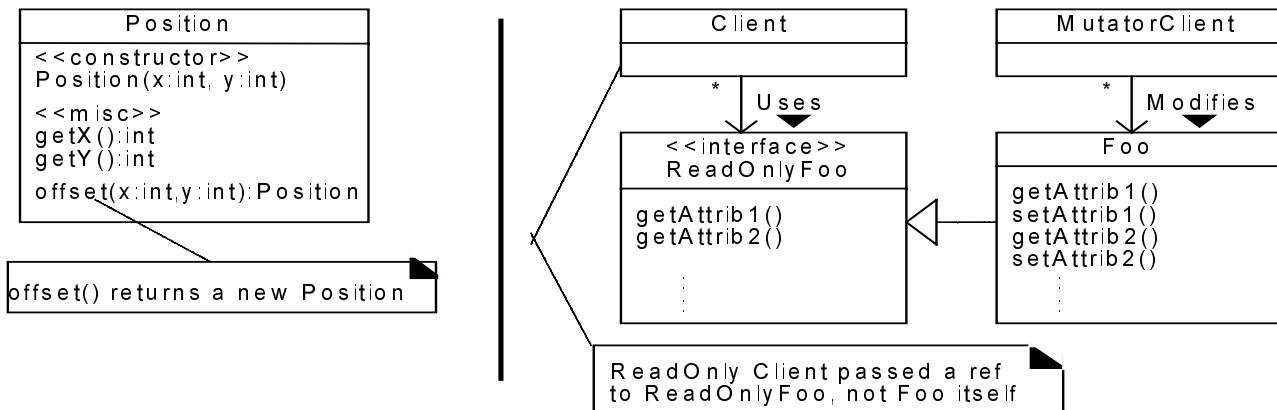
Notes

Comments: One variation is a “read-only” class interface which would allow some classes to modify but others could only read state information if their access is through an immutable interface.



Example: A game program such as chess or checkers which might use immutable objects to represent the pieces as positions on a game board or field. Whenever a position changes, a new Position instance is created.

Diagram:



Immutable (and Immutable Interface)

Pattern Thumbnail: Disallow any changes to an object after its construction. This could reduce the overhead of concurrent access to an object being shared and avoid the need to synchronize multiple threads which share an object.

Forces:

- (1) Instances of a class are passive and can be used by multiple objects,
- (2) for shared objects that do need to be changed, state changes can be difficult and error-prone – this scheme helps to coordinate changes,
- (3) the overhead of synchronized threads can add unacceptable additional overhead if they need to access state information in a shared object.

Usage: In the Java API, the **String class** is an example – any String methods which look like they are modifying the String instance are actually creating a new instance of String.

(Also see example on facing page)



Consequences:

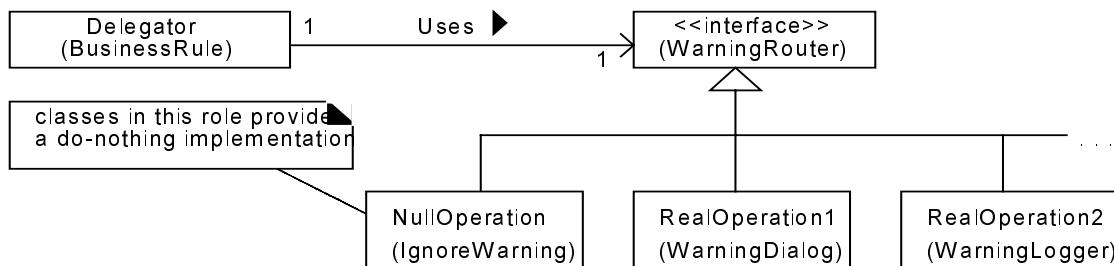
- An object's state never changes, so mutator methods can be eliminated, or should be modified to now create a new object.
- There is no need to synchronize threads.
- Increase robustness of objects that share references to the same object and reduce the overhead of concurrent access to an object.

Notes

Comments: NullObject is often used with the Strategy pattern. If instances of a NullObject class contain no instance-specific information, implement that NullObject as a Singleton class. Where an Adapter class would wrap an object, a null adapter would pretend to wrap another class without actually providing any behavior for the operations. There is also a NullIterator, which doesn't iterate over anything.

Example: You are writing classes to encapsulate business rules for an enterprise. These classes will be used in a variety of environments, so they must be able to send warning messages to a dialog box, a log file, other destinations, or nowhere at all. Define an interface called *WarningRouter* and have the classes you write delegate the sending of warnings to objects that implement that interface. Depending on the specific business-rule class, some messages don't need to be sent anywhere. Putting a test for *null* in the code means there is room for error in forgetting all the necessary places. So, create a class that does nothing with a warning message – no test for null or any other special logic is needed; simply send the message to an instance of the *IgnoreWarning* (do nothing) class.

Diagram:



Participants: Delegator, AbstractOperation, NullOperation, RealOperation

Consequences Expanded: The Null Object pattern relieves a class that delegates an operation to another class of the responsibility for implementing the do-nothing version of that operation. Code is simpler (and usually more reliable) if you don't have to test for null. If there is one consistent do-nothing behavior that works for several Delegator classes, the class in the NullOperation role is reusable.

If clients expect different “do-nothing” behavior, the pattern forces uniformity and a NullObject does not transform into a RealObject.

Null Object

Pattern Thumbnail: A Null Object is a “do-nothing” surrogate for another object that shares the same interface.

Using this pattern is an alternative to using “null” to show the absence of an object to delegate an operation to. Instead of using/testing for null before each call to the other object’s methods, use a reference to a null object which encapsulates and hides the implementation decisions of how to do nothing.



Forces:

- Suppose an object requires a collaborator for an existing collaboration, but some collaborator instances should do nothing.
- Clients should not have to be concerned with differences between collaborators that implement behavior and those that do nothing.

Usage: There are many examples. Smalltalk VisualWorks has the NoController class in the Controller hierarchy, NullDragMode in the DragMode hierarchy, NullInputManager in the InputManager hierarchy, etc. Null Lock is a type of lock mode in the VERSANT Object DBMS which does not block other locks and cannot be blocked by other locks; it is not really a lock but acts like one for operations that require a lock.

Consequences: Uses polymorphic classes, simplifies client code, and encapsulates do-nothing behavior and even makes it reusable.

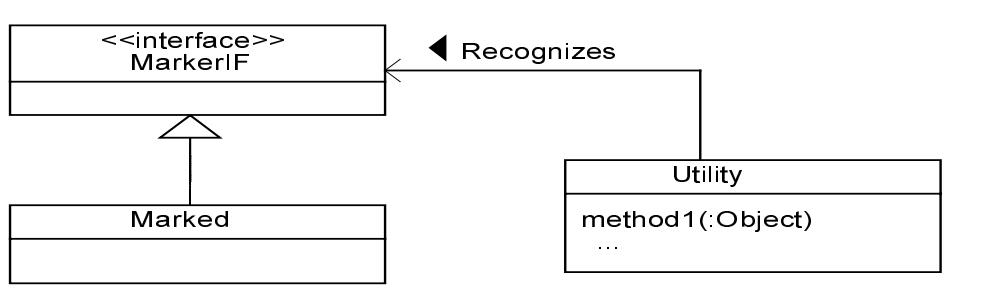
However, the total number of classes becomes greater, and there is a **tradeoff** between the number of classes and the simplification of code as the potential number of classes explodes.

Notes

Comments: Placing a “boolean” type attribute on a class; used by the Snapshot pattern.

Example: Java’s Object class defines the equals() method which takes an argument that can be a reference to any object. All classes inherit the equals() method, and may override it if they wish. When this method is called, for example in searching a Vector to find an object equal to a given object, it would be much more efficient to use the == operator to determine if two objects are the same object when the target object does NOT override the equals() method. But, there is no way to determine if an arbitrary object’s class does the override. So, define an interface such as EqualByIdentity; it needs no methods, but is simply a marker. Any class which implements this interface can be handled differently than one which does provide the equals() override.

Diagram:



Participants: Utility object, Marked object

Marker Interface

Pattern Thumbnail: Use an interface with NO methods or variables to indicate semantic attributes of a class.

Forces:

- Utility classes may need to know something about how a class is to be used but you don't want to be tied to an instance of a particular class.
- Classes can implement as many interfaces as needed (avoiding multiple inheritance) and you can determine if an object's class implements any particular interface of interest.



Usage: Java API - Serializable interface. If a class implements Serializable, instances of that class can be written out as a stream of bytes. Otherwise, read/write operations are not allowed.

Consequences:

- Clients can make inferences about objects passed to their methods without depending on the objects to be instances of any particular class.
- The relationship between the utility class and the marker interface is transparent to all other classes except for those implementing the interface.

Notes

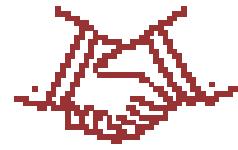
Pattern	Description
Expert	<i>Who is responsible?</i> Who has the necessary information?
Creator	<i>Who creates?</i> (Note that Factory is a common solution.) Assign B as creator if: B contains A, B aggregates A, B has init data for A, B records A, B closely uses A
Controller	<i>Who handles a system event?</i> Assign responsibility to a class which represents: 1. overall system, device, or subsystem (façade) 2. a use case scenario within which system event occurs (Note: NOT the Presentation Layer)
Low Coupling (evaluative)	<i>How to support low dependency and increased reuse?</i> Assign responsibilities so coupling remains low
High Cohesion (evaluative)	<i>How to keep complexity manageable?</i> Assign responsibilities so cohesion remains high
Polymorphism	When <i>related behaviors</i> vary by class, assign responsibility using polymorphic operations
Pure Fabrication	To keep coupling/cohesion values, create <i>artificial class</i> even if not modeling anything
Indirection	Assign responsibility to <i>intermediate object</i> to mediate other components or services and avoid direct coupling
Protected Variations	Identify points of predicted variation or instability; assign responsibilities to create a stable “interface” around them.
Don’t Talk to Strangers	Within a method, messages are <i>only</i> sent to: this (self), a parameter of method, attribute of self, element of collection which is attribute of self, object created within the method

Fundamental Principles in Assigning Responsibilities to Objects¹

¹ Larman, Craig, *Applying UML and Patterns*, inside front cover.

General Responsibility Assignment Software Patterns (GRASP)

► ***Discuss chart on facing page***



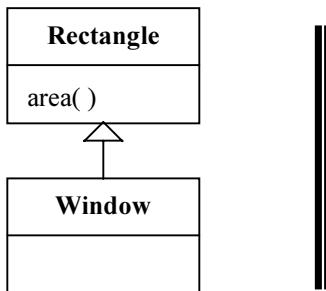
- **Skillful assignment of responsibilities** is important in design
- **Interaction Diagrams** are key in making decisions
- **Patterns** codify good advice and principles

☒ "The first thing to know about software laws is that they were meant to be broken."

Exercises

1. Composition and delegation.

Change the inheritance below to composition and delegation so that Window HAS-A Shape and draw the corresponding UML. Make Shape an abstract class with several different subclasses, including Rectangle. Write the pseudo-code for an area method of Window.



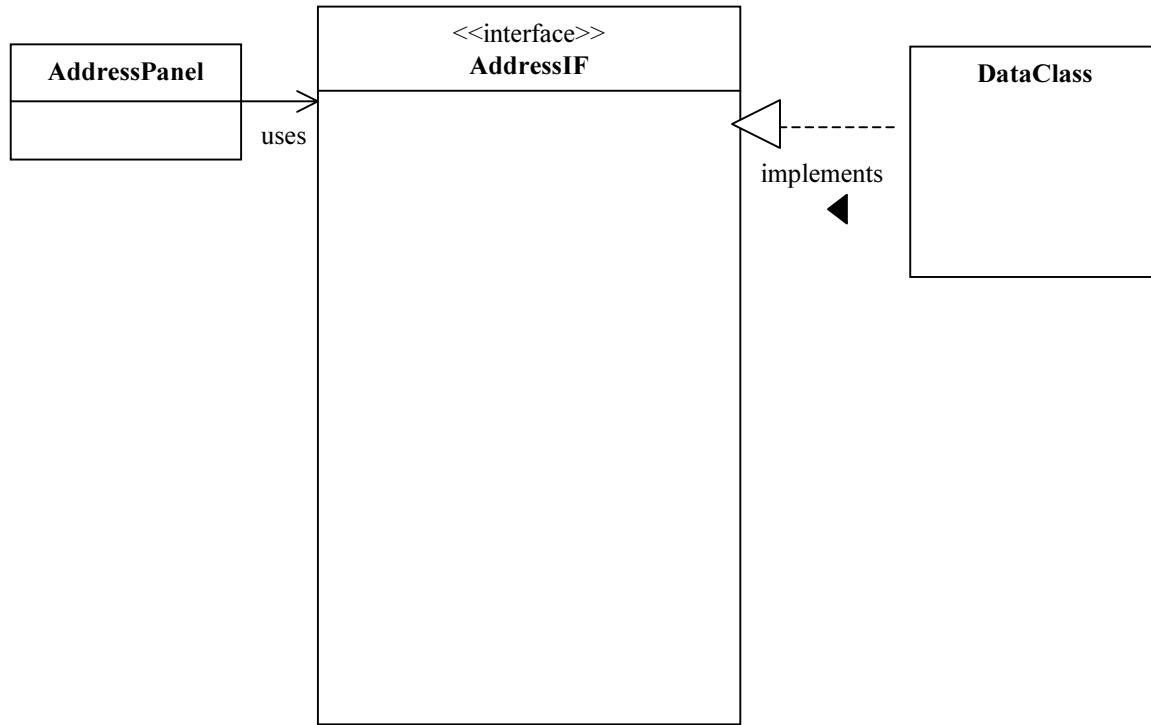
```

Window w = new Window();
float area = w.area();
// Window inherits area()
    
```

2. Interface.

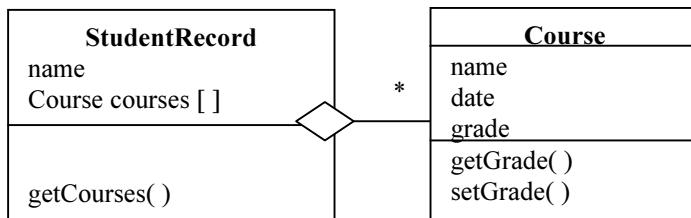
- What interfaces have you used in your programming? (In C++ look for a class with *only* pure virtual functions and no attribute fields.)

- b. Create an interface called AddressIF and list possible methods.



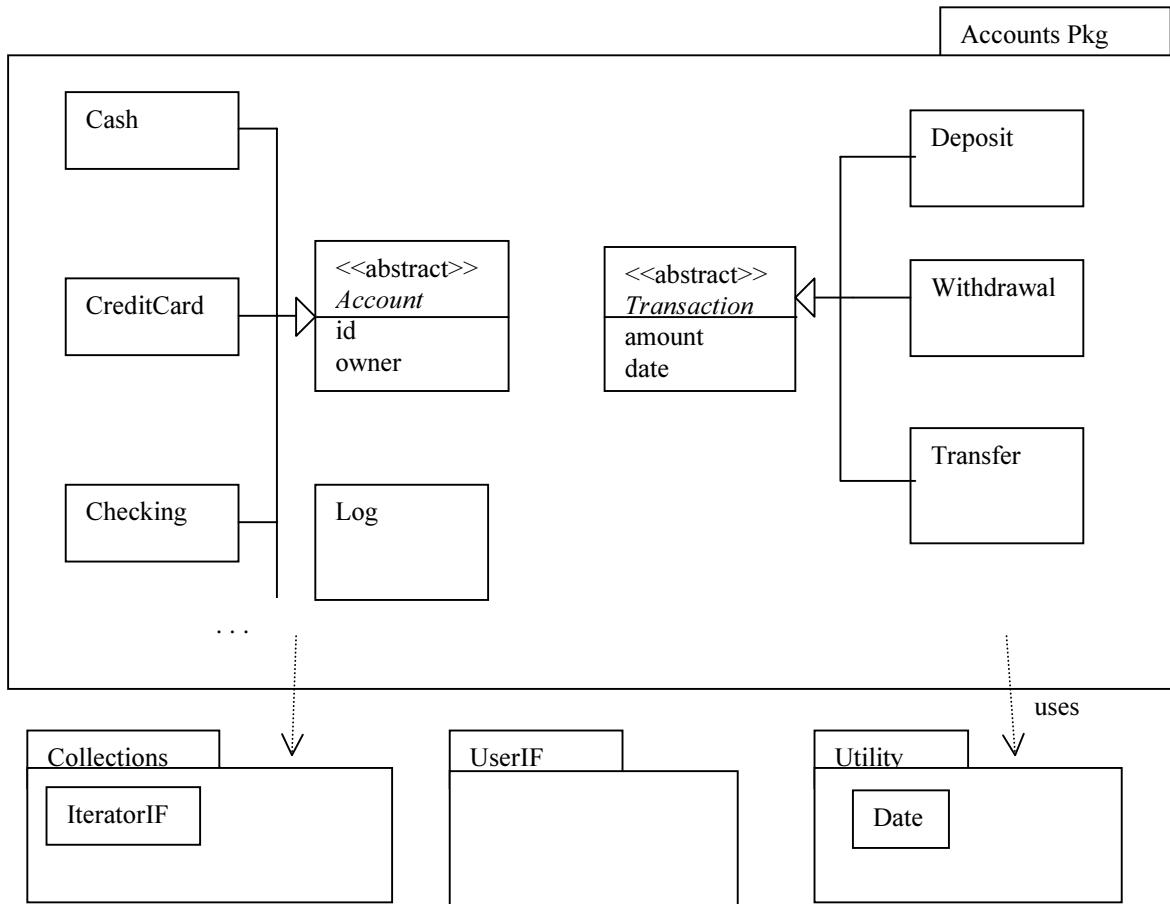
3. Immutable.

A university records system keeps info on **StudentRecord**. The registrar needs to be able to enter/change grades. How could you use an immutable interface to give a student access to their info but *not* with edit capability? Elaborate on the possible model below or roll your own.



4. GRASP

Suppose you are creating a software package (like Quicken) to keep your accounts. You want multiple accounts, the ability to view or print a log of transactions sorted by date (date1 – date2) for each account, etc. Suppose your initial class diagram resembles the one below. Add classes to it and insert links. Which class is “in charge” of system events? (i.e., who handles a new transaction; who opens a new account, etc.) Which class objects create other objects? How can you apply polymorphism? Can you improve cohesion or reduce coupling? Do you have any examples of indirection? (Be prepared to explain your suggestions and walk through what would happen with opening a new account, entering a new transaction, or printing a log from date1 to date2.)



Chapter 6 – Catalog of GoF Patterns

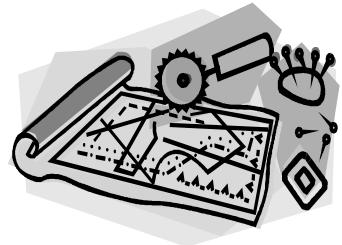
Notes

Chapter Objectives

- Become familiar with all 23 **GoF Design Patterns**
 - 5 **Creational Patterns**
 - 7 **Structural Patterns**
 - 11 **Behavioral Patterns**
- Be able to read the diagrams and **create UML** for the patterns
- Build **vocabulary**
- Improve ability to **read** and **use** the GoF book
 - Understand the **book structure**
 - Examine some **sample code**
 - Compare/contrast some of the **examples**, such as Maze
 - Develop a **reading outline**
 - Begin to understand the **interplay of various patterns** put together in a design

Notes

Overview of GoF Patterns



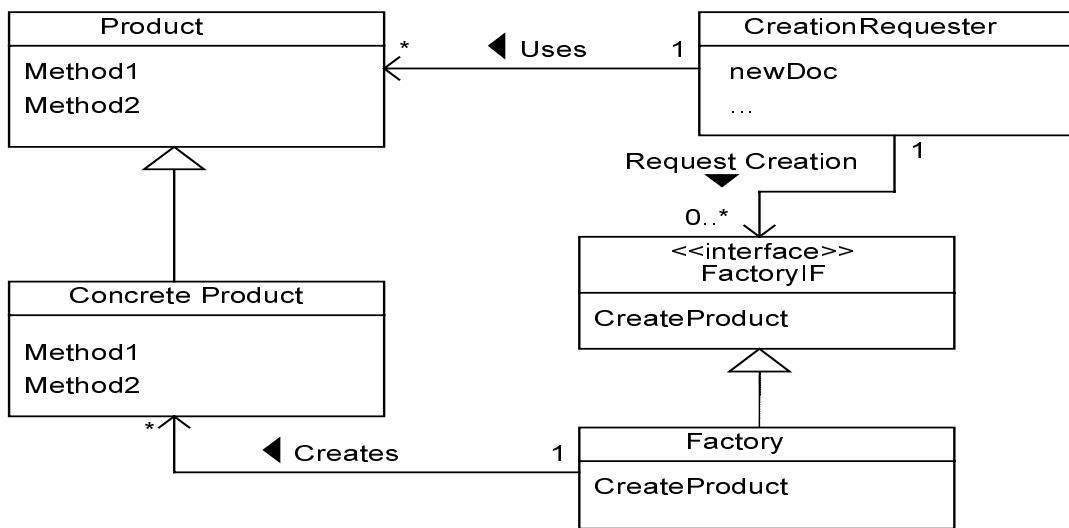
Design aspects that design patterns let you vary:

Purpose	Design Pattern	Aspect(s) That Can Vary
<i>Creational</i>	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
<i>Structural</i>	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Façade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location
<i>Behavioral</i>	Chain of Resp.	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object; how dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Notes

Comments: Discussed first since used by a number of other patterns. Sort of a method parameterized by another class (an artifact of classes not being values in C++).

Diagram:



Participants: Product, ConcreteProduct, Creator (application), ConcreteCreator (factory) which overrides the factory method to return an instance of a ConcreteProduct.

Usage: Toolkits, frameworks. e.g. In Java API, **URLConnection** object (Creation Requester) is associated with each URL object. The **getContext** method returns content in the appropriate subobject: if a .gif file, returns an **Image** object. **ContentHandler** is the abstract class; **URLConnection** gets a concrete **ContentHandler** through **ContentHandlerFactory**, the factory interface.

Solution: See Code, Appendix page B-16.

1. Factory Method (aka: Virtual Constructor) GoF p. 107



Pattern Thumbnail: A class is to be reused with arbitrary data types. You want it to be able to instantiate objects yet remain independent of its subclasses by delegating the choice of which subclass to instantiate to another object.

Example: Designing an editor for different files such as .java, .c, .cpp ...

Forces: It should be possible to organize a class so it can create objects that inherit from a given class or interface. However, to be reusable, the class must be able to create objects without knowing what appropriate classes are available. Instead, tie that use-specific knowledge to a separate use-specific class.

Question: How does Factory Method promote loosely coupled code?

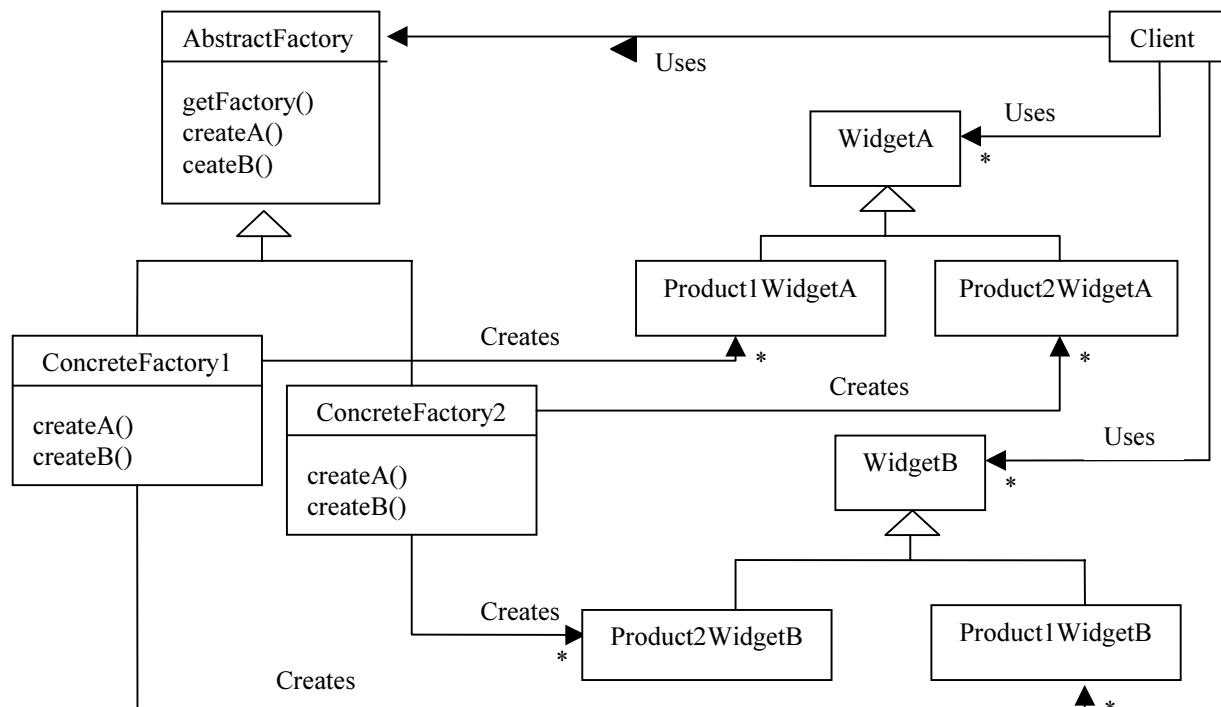
Consequences:

1. hooks for subclasses,
2. parallel class hierarchies

Notes

Comments: A class parameterized by another class; an artifact of classes not being values in C++. Often implemented with factory methods, but can also use Prototype. The AbstractFactory is often a Singleton.

Diagram:



Participants: Client, AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct

Consider the MazeFactory example in GoF. The MazeFactory contains a method called `MakeRoom`, which takes as a parameter one integer, representing a room number. What happens if you would also like to specify the room's color and size? Would this mean that you would need to create a new Factory Method for your MazeFactory, allowing you to pass in room number, color and size to a second `MakeRoom` method?

Of course, nothing would prevent you from setting the color and size of the room object after it has been instantiated, but this could also clutter your code, especially if you are creating and configuring many objects. How could you retain the MazeFactory and keep only one `MakeRoom` method but also accommodate different numbers of parameters used by `MakeRoom` to both create and configure Room objects?

Usage: `java.awt.Toolkit`, with `getDefaultToolkit` static method which returns a singleton `ConcreteFactory` object. Plaf (pluggable look and feel) widgets.

Solution: See code, Appendix page B-3.

2. Abstract Factory (aka: Kit or Toolkit) GoF p.87

Pattern Thumbnail: Interface for creating families of related or dependent objects without specifying a concrete class.



Example: Creating widgets for different windowing systems.

Forces:

- A system that works with multiple products should be independent of any specific product.
- You want to configure the system to work with multiple members of a family of products.
- Instances of classes intended to interface with a product should be used together and only with that product (constraint enforced).
- Want an extensible system to work with additional products by adding sets of classes with minimum change.

Question: In the implementation section of this pattern, the authors discuss the idea of *defining extensible factories*.

Since an Abstract Factory is composed of Factory Methods, and each Factory Method has only one signature, does this mean that the Factory Method can only create an object in one way?



Button and Textfield.

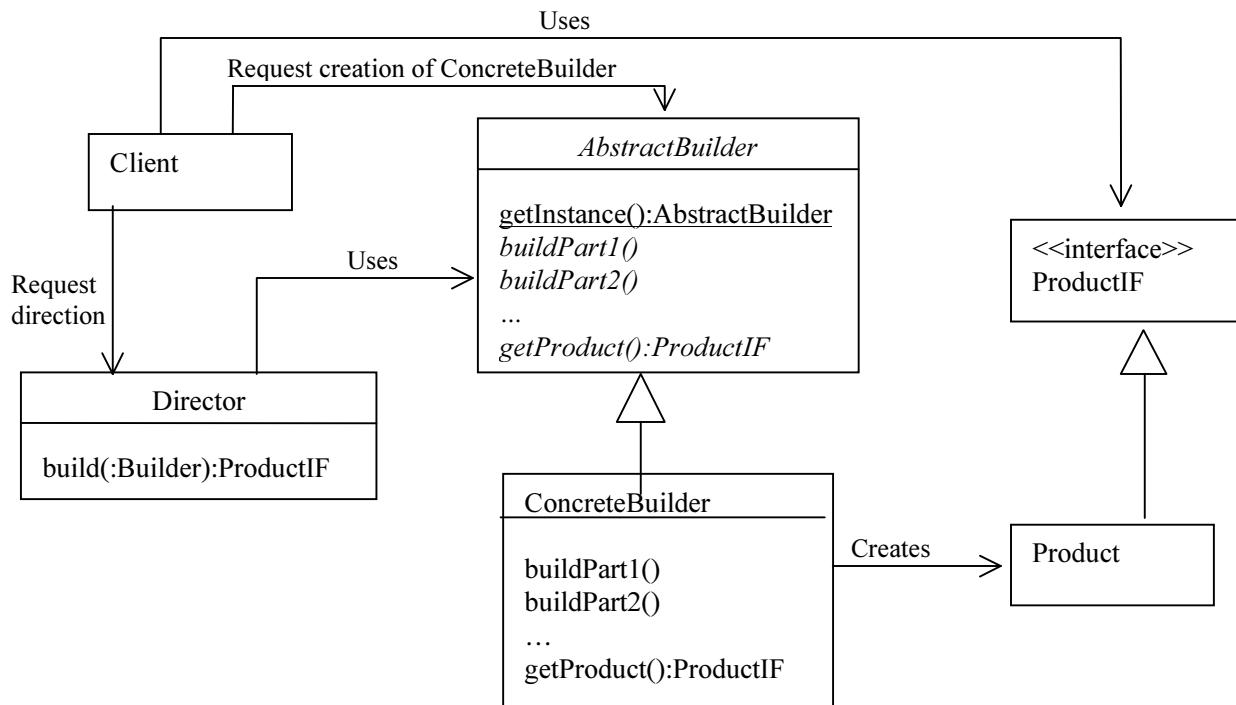
Consequences:

- Concrete widget classes are independent of clients using them because AbstractFactory encapsulates the process of creating widget objects.
- Adding new products is easy and the pattern ensures a consistent set of products.
- However, if you have to interface with a new product, it's a lot of work to write classes to do this.
 - Client classes may have to organize products into hierarchies for their own use.
 - AbstractFactory can be used with Bridge to create hierarchies of product-independent widget classes such as

Notes

Comments: Allows a client object to construct a complex object step-by-step by specifying only its type and context.

Diagram:



Participants: Abstract Builder, Concrete Builder, Director, ProductIF, Product, Client

3. Builder GoF p. 97



Pattern Thumbnail: A client creates a Director object and configures it with the appropriate Builder object.

Example: Email gateway program receives MIME¹ messages and forwards them in various formats for different kinds of email systems. Organize the program with an object that parses MIME messages. For each message to parse, the message is paired with a builder object the parser uses to build a message in the desired format. As the parser recognizes each header field and message body part, it can call the corresponding method of the current builder object it is working with.

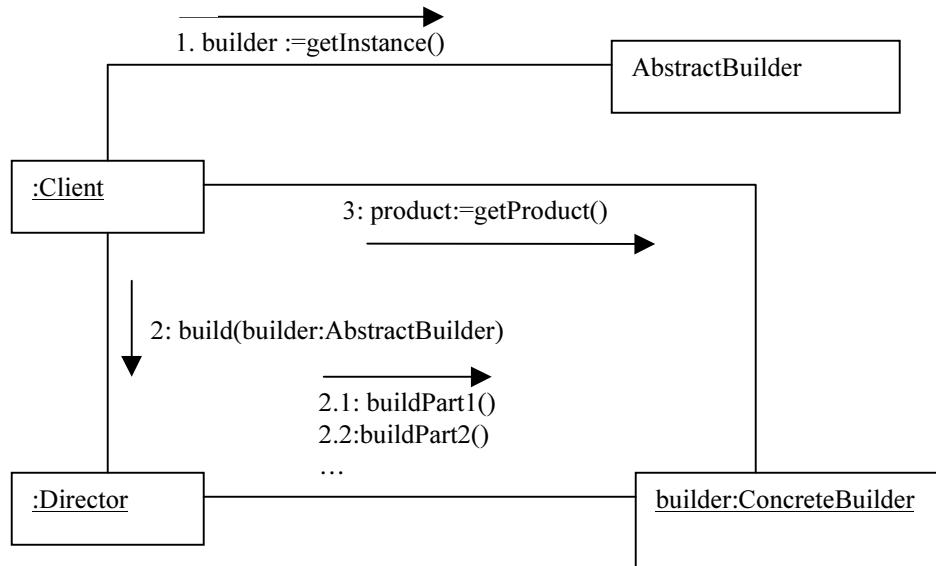
Forces:

- A program needs to produce multiple external representations of the same data.
- The class(es) responsible for providing content should be independent of any external data representation and the classes that do the building. (If content-providing classes have no such dependencies then mods to external data representation classes won't require maintenance to the content classes.)
- Likewise, the classes responsible for the building should also be independent of the content classes and not have to know anything about the content-providing objects.

¹ MIME is an acronym for Multipurpose Internet Mail Extensions – the standard for most e-mail messages on the net.

Notes

Builder Collaboration:



Usage: A common pattern in Smalltalk. Useful for parsers.

Solution: See Code, Appendix page B-7.

Builder, continued



Question: Like the Abstract Factory pattern, the Builder pattern requires that you define an interface, which will be used by clients to create complex objects in pieces.

In the MazeBuilder example, there are BuildMaze(), BuildRoom() and BuildDoor() methods, along with a GetMaze() method.

How does the Builder pattern allow one to add new methods to the Builder's interface, without having to change each and every subclass of the Builder?

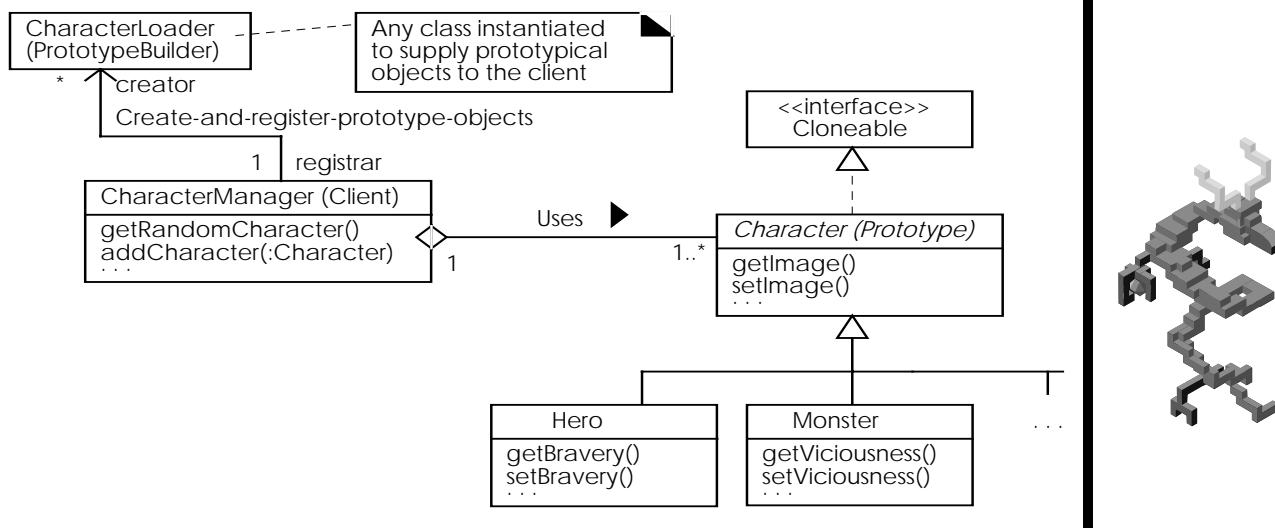
Consequences:

- Builder provides finer control over construction than other patterns such as Factory Method.
- The Director has step-by-step control over the creation of the entire product whereas other patterns create objects in a single step.
- Pattern hides details from clients and allows the internal representation to vary for good encapsulation.

Notes

Comments: Structured instantiation. Often used with Composite or Decorator. PrototypeBuilder may use AbstractFactory to create a set of prototypical objects.

Diagram:



Participants: Prototype, ConcretePrototype, Client, PrototypeBuilder

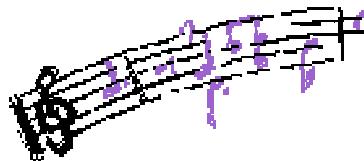
Usage: Java API. JavaBeans! Beans are instances of classes that conform to specialized naming conventions which allow a bean builder tool information on how to customize. Customized beans are saved to a file to be loaded later by other applications – a time-delayed way of cloning objects.

Solution: See Code, Appendix page B-26.

4. Prototype GoF p. 117

Pattern Thumbnail: Create new objects by copying (cloning) a prototype instance which specifies the kinds of objects to create.

Example: Music editor (as in GoF),
interactive role-playing game (see diagram facing).



Forces:

1. Be able to create objects without knowing their exact class, how they are created, or what data they represent.
2. Classes to be instantiated not known until runtime when they are acquired on the fly.
3. Other approaches that allow creation of a large variety of objects are undesirable for the following reasons: (a) the creator classes create objects directly, which makes them aware of and dependent on a large number of other classes, (b) a factory method may be very large and difficult to maintain, (c) an abstract factory approach would have to have a large variety of concrete factory classes in a hierarchy that parallels the classes to be instantiated.

Question:

Part 1 - when should this creational pattern be used over the other creational patterns?

Part 2 - Explain the difference between deep vs. shallow copy.

Part 3 – Suppose you are using existing classes (and do not have the source) and there is no clone() method – what to do?

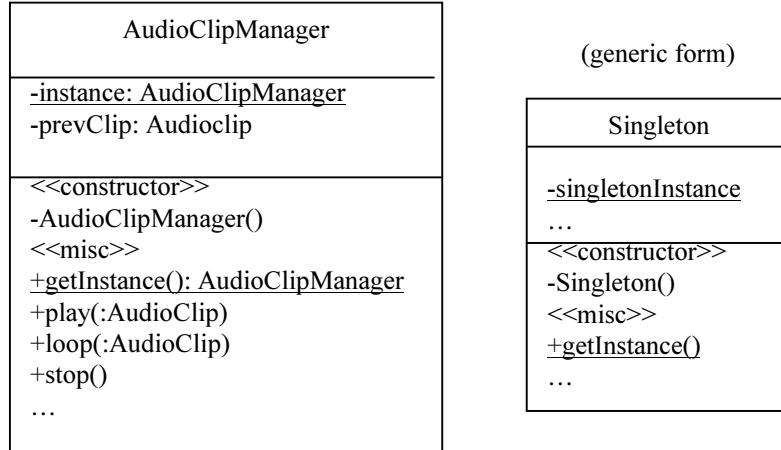
Consequences: Same benefits as AbstractFactory. Also:

1. Can add/remove products at runtime,
2. Specify new objects by varying values,
3. Specify new objects by varying structure,
4. Reduce subclassing,
5. Configure an application with classes dynamically.

Notes

Comments: Another artifact of the object/class distinction in C++. An Abstract Factory is often a Singleton. (In multithreaded environments be sure to check the Double-Checked Locking pattern, and issues surrounding that.)

Diagram:



Participants: Singleton

Usage: `java.lang.Runtime` is a singleton class with exactly one instance. It has no public constructors; to get a reference to the instance clients call its static method `getRuntime()`.

Solution: See Code, Appendix page B-28.

Note: Consider looking at the Double Checked Locking Pattern (Chapter 8, p. 231) in conjunction with Singleton.

5. Singleton (aka: Oddball) GoF p.127

Pattern Thumbnail: Ensures only one instance of a class gets created. All objects that use that class share the same instance.

Example: You want to write a class an applet can use to make sure no more than one audio clip is played at a time. If an applet contains two sections of code that independently play audio clips, then it would be possible for both to play clips at the same time. Results would depend on the platform and range from confusing (hearing both) to terrible (sound mechanisms cannot cope with situation). Your class should stop one clip before starting another.

Forces: You desire exactly one instance of a class which must be accessible to all clients.

Question: The Singleton pattern is often paired with the Abstract Factory pattern. What other creational or non-creational patterns would you use with the Singleton pattern?



Consequences:

- Controlled access, reduced namespace.
- Subclassing a singleton class is awkward and results in imperfectly encapsulated classes since the constructor can no longer be private.
- If the subclass is also a singleton, in Java you cannot override static methods, such as `getInstance()`.

Exercises

1. Scenario: Your company has a POS (point of sale) application to use the IBM Java drivers if IBM hardware is used, NCR drivers if NCR hardware is used, etc. The JavaPOS implementations will be purchased from the manufacturers. Suppose, for example, the following were available:

```
//IBM drivers  
com.ibm.pos.jpos.CashDrawer (implements jposCashDrawer)  
com.ibm.pos.jpos.CoinDispenser (implements jpos.CoinDispenser)  
...  
  
//NCR drivers  
com.ncr.posdrivers.CashDrawer (implements jposCashDrawer)  
com.ncr.posdrivers.CoinDispenser (implements jpos.CoinDispenser)  
...
```

- a. Define in UML a factory interface (the abstract factory).
- b. Define in UML a concrete factory class for the two above families of things to create (i.e., IBMJavaPOSDevicesFactory and NCRJavaPOSDevicesFactory – feel free to abbreviate)
- c. Include in your UML an interface for CashDrawer with methods such as DrawerOpened() and show the relationship with the concrete classes which implement this interface.
- d. Show the UML to insert a **Singleton** pattern between #1 (the interface) and #2 (the concrete factory class) above to insure only one instance of the concrete factory gets created. Your new abstract factory class will have a class variable called instance and a class method getInstance() which returns the instance of type defined in the interface in #1.
- e. Finally, how does your application know which factory to instantiate?

2. a. List the benefits of the **Singleton** pattern (see p. 128).

b. What has to be changed if you decide later you want multiple copies?

c. Why should you consider at least one private constructor in a **Singleton** class?

d. GoF talks about a registry of singletons (p. 130). Explain how this would work and why it adds flexibility to a design.

e. How could garbage collection (as in Java) be a problem? (Consider a Singleton class which keeps performance statistics – example code is in part f.)

f. Optional (for Java Programmers): Explain how the java code below² prevents garbage collection from occurring on a class instance (which could be a Singleton) by adding comments. This could be relevant if classes are dynamically loaded and a Singleton needs to stick around – you don't want a new instance reloaded.

```

public class ObjectPreserver implements Runnable {
    //
    //
    //
    private static ObjectPreserver lifeline
        = new ObjectPreserver();
    //
    //
    //
    private static HashSet protectedSet = new HashSet();

    private ObjectPreserver() {
        new Thread(this).start();
    } // constructor

    public void run() {
        try {
            wait();
        } catch (InterruptedException e) { }
    }

    /**
     *
     */
    public static void preserveObject( Object o ) {
        protectedSet.add( o );
    }

    /**
     *
     */
    public static void unpreserveObject( Object o ) {
        protectedSet.remove( o );
    }
} // class ObjectPreserver

```

² Grand, “Patterns in Java”, p. 130-131.

3. Use the sample code on GoF p. 101-104 to explain the **Builder** pattern in the style of the collaboration diagram on p. 99 (or the version in the notes) – sort of a code review session. (Be prepared!)
4. Assume a class called CharacterLoader, elaborate on the diagram in **Prototype** pattern and write code or pseudo-code for a prototype Character class (with image field, etc.), Hero, Monster, and another subclass of your own, and the CharacterManager class.

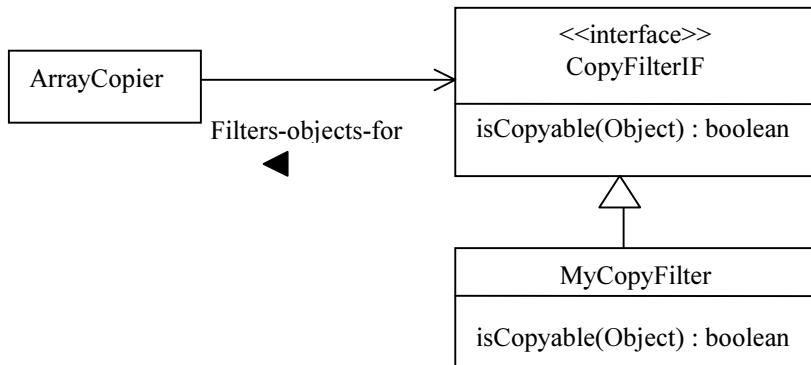
For example:

```
class Hero extends Character {  
    private int bravery = 10;  
  
    public void setBravery(int b) { bravery = b; }  
    public int getBravery() { return bravery; }  
  
}
```

Notes

Comments: An intermediary which changes the interface of an object.

Diagram: (also see GoF p. 141)



Participants: Client, TargetIF, Adapter, Adaptee

Usage: `java.awt.event.WindowAdapter` – intended for subclassing.

Solution: See Code, Appendix page B-5.



6. Adapter (aka: Wrapper) GoF p.139

Pattern Thumbnail: An object adapter implements an interface known to clients; the Adapter provides the functionality of the interface without the client having to know what class it represents.

Example: You need to write a method that copies an array of objects, filtering out objects that don't meet certain criteria. Make the method independent of the actual filtering criteria used by defining an interface that declares a method the array copier calls to find out if it should include a particular object in the new array.

Forces:

- You want to use a class that calls a method through an interface, but you want to use it with a class that does not implement that interface.
- Modifying the class to implement the interface is not an option (i.e., don't have the source, or the class is general-purpose and shouldn't implement an interface for a specialized purpose).
- Or, you want to determine dynamically which methods of another object a given object calls, without the client object having to know the class of the called object.

Question: Would you ever create an Adapter that has the same interface as the object which it adapts? Would your Adapter then be a Proxy?

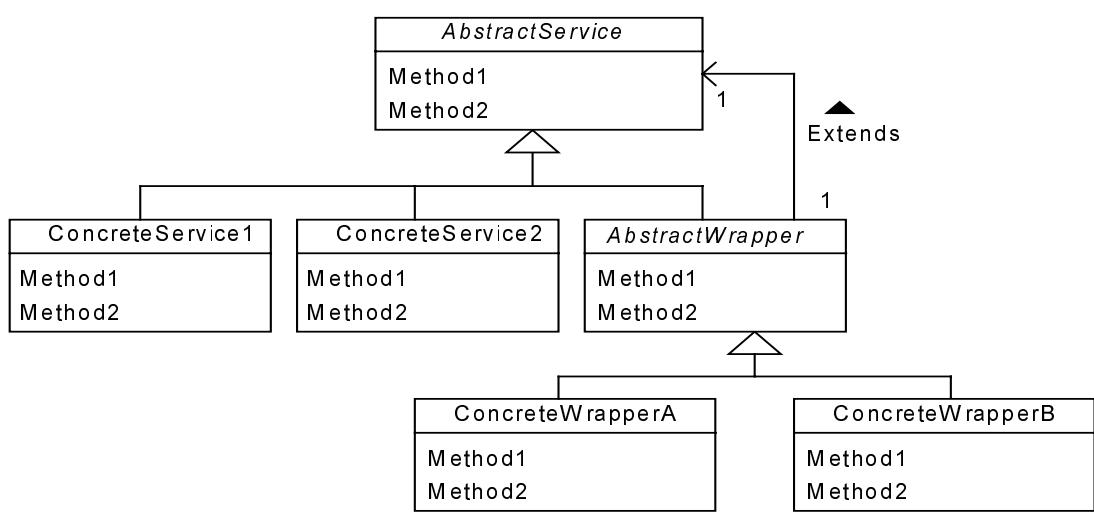
How is Adapter different from Decorator?

Consequences: Independence of client and adaptee classes. Indirection generally adds to difficulty of understanding a program.

Notes

Comments: A structured way of delegating behavior; an intermediary which attaches behavior to, but does not change the interface of, an object. The Filter pattern is a specialized decorator to manipulate a data stream. Decorator is good for arranging before/after events of other methods. Decorator changes the skin of an object, Strategy lets you change the guts; if you need behavior in the middle of a method call, consider Strategy, or Template Method.

Diagram:



Participants: AbstractServices, ConcreteServices, AbstractWrapper (decorator), ConcreteWrapperA, ConcreteWrapperB, etc.

Usage: OO user interface toolkits use decorators to add graphical embellishments to widgets. Text has example of adding responsibilities to Stream classes.

Solution: See Code, Appendix page B-13.

7. Decorator (aka: Wrapper) GoF p. 175

Pattern Thumbnail: Extends functionality of an object in a way transparent to clients by using subclass of original that delegates operations to original.



Example: Controller for physical access to building. Need to integrate a surveillance system. You discover an inheritance hierarchy for 3 kinds of door controllers, and another hierarchy for 2 kinds of surveillance monitors. Six subclasses is a lot to write and could even grow, so Decorate the Door Controller interface with 2 new concrete classes which include the surveillance options.

Forces:

- Need to extend functionality of a class, but there are reasons not to use inheritance.
- Or, need to dynamically extend (or withdraw) functionality of an object.

Question: In the implementation section of the Decorator Pattern, the authors write: "A decorator object's interface must conform to the interface of the component it decorates." Now consider an object A, that is decorated with an object B. Since object B "decorates" object A, object B shares an interface with object A. If some client is then passed an instance of this decorated object, and that method attempts to call a method in B that is not part of A's interface, does this mean that the object is no longer a Decorator, in the strict sense of the pattern?

Furthermore, why is it important that a decorator object's interface conforms to the interface of the component it decorates?

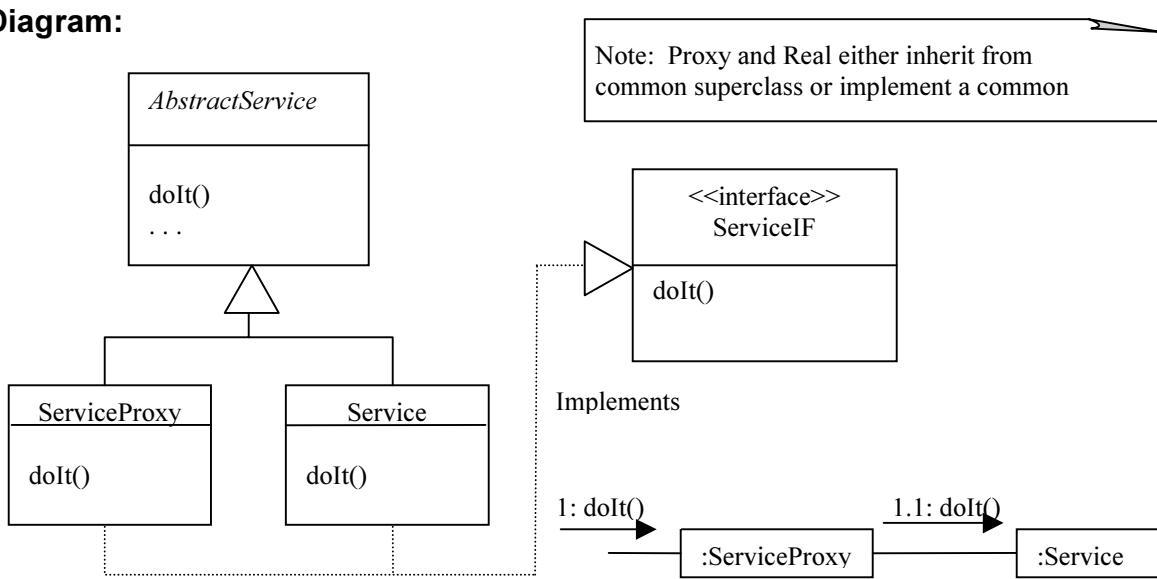
Consequences:

1. More flexibility than static inheritance – can dynamically add/remove wrappers,
2. Avoids feature-heavy classes high in hierarchy,
3. Decorator not equal to Composite,
4. Often fewer classes, but lots of little objects – can complicate debugging,
5. Harder to use object identity since service objects hidden behind wrapper objects.

Notes

Comments: A stub which bridges to another machine, device, database, etc.

Diagram:



Participants: Proxy Subject, Real Subject, Common Interface

Usage: Java's RMI uses skeleton and stub interfaces as proxy objects for the server and remote object. CORBA uses proxies to achieve a degree of transparency for remote object communication across languages.

Solution: See Code, Appendix page B-27.

8. Proxy (aka: Surrogate, Ambassador, Virtual Object) GoF p. 207



Pattern Thumbnail: A Proxy object receives method calls on behalf of another object.

Example: ImageProxy used in GoF (p. 209) by the DocumentEditor – it takes a long time to load the graphic image and the editing may be confined to text anyway, so defer the cost of loading the real image.

Forces:

- (1) It isn't possible for some object to provide its service at a convenient time or place;
- (2) gaining visibility to an object is nontrivial and you want to hide the complexity;
- (3) access to an object needs to be controlled without adding complexity or coupling to the access control policy;
- (4) management of an object's services needs to be provided in a way transparent to the clients of the service.

Question: If a Proxy is used to instantiate an object only when it is absolutely needed, does the Proxy simplify code?

Variations: Access Proxy, Broker, Device Proxy, Façade, Remote Proxy, Virtual Proxy. (Related: Adapter, Decorator.)

Consequences:

Unless proxies introduce new failure modes, code in client classes is unchanged.

Possibilities: hide that an object is remote, perform optimizations (such as copy on write – see example in Exercises), allow additional tasks.

Exercises

1. Discuss the differences in purposes of **Adapter**, **Decorator**, and **Proxy**.
2. What are the differences in adapting a class vs. adapting an object? (GoF p. 142)
3. What are the issues when designing an **adapter**? (GoF p. 142).
4. Mark Grand says “the **Proxy** pattern is not very useful unless it implements some particular service management policy.”³ What does this mean?
5. The following code defines a subclass of `java.util.Hashtable` that is functionally equivalent to `Hashtable`. The difference is in the `clone()` method. `Hashtable`’s `clone` returns a new `Hashtable` that is in all respects identical to the original. Cloning a `Hashtable` is an expensive operation, though it does avoid the need to synchronize access with locks. If the `Hashtable` is unmodified the effort and space of cloning is wasted.

Here’s the question: suppose a client is using an instance of the class, `LargeHashTable`. Another client reads the same instance. What happens? Then one client calls the `put(someKey, someValue)` method – what happens?

³ Grand, “Patterns in Java”, p. 80.

(Do a code walk-through...)

```

public class LargeHashtable extends Hashtable {
    // The ReferenceCountedHashTable that this is a proxy for.
    private ReferenceCountedHashTable theHashTable;
    . .
    public LargeHashtable() {
        theHashTable = new ReferenceCountedHashTable();
    } // constructor()
    . .
    /**
     * Return the number of key-value pairs in this hashtable.
     */
    public int size() {
        return theHashTable.size();
    } // size()
    . .
    /**
     * Return the value associated with the specified key in this Hashtable.
     */
    public synchronized Object get(Object key) {
        return theHashTable.get(key);
    } // get(key)
    /**
     * Add the given key-value pair to this Hashtable.
     */
    public synchronized Object put(Object key, Object value) {
        copyOnWrite();
        return theHashTable.put(key, value);
    } // put(key, value)
    . .
    /**
     * Return a copy of this proxy that accesses the same Hashtable as this
     * proxy. The first attempt for either to modify the contents of the
     * Hashtable results in that proxy accessing a modified clone of the
     * original Hashtable.
     */
    public synchronized Object clone() {
        Object copy = super.clone();
        theHashTable.addProxy();
        return copy;
    } // clone()
    /**
     * This method is called before modifying the underlying Hashtable. If it
     * is being shared then this method clones it.
     */
    private void copyOnWrite() {
        if (theHashTable.getProxyCount() > 1) {
            // Synchronize on the original Hashtable to allow consistent
            // recovery on error.
            synchronized (theHashTable) {
                theHashTable.removeProxy();
                try {
                    theHashTable
                        = (ReferenceCountedHashTable)theHashTable.clone();
                } catch (Throwable e) {
                    theHashTable.addProxy();
                } // try
            } // synchronized
        }
    }
}

```

```
        } // if proxyCount
    } // copyOnWrite()

. . .

private class ReferenceCountedHashTable extends Hashtable {
    private int proxyCount = 1;

    public ReferenceCountedHashTable() {
        super();
    } // constructor()

    /**
     * Return a copy of this object with proxyCount set back to 1.
     */
    public synchronized Object clone() {
        ReferenceCountedHashTable copy;
        copy = (ReferenceCountedHashTable)super.clone();
        copy.proxyCount = 1;
        return copy;
    } // clone()

    /**
     * Return the number of proxies using this object.
     */
    synchronized int getProxyCount() {
        return proxyCount;
    } // getProxyCount()

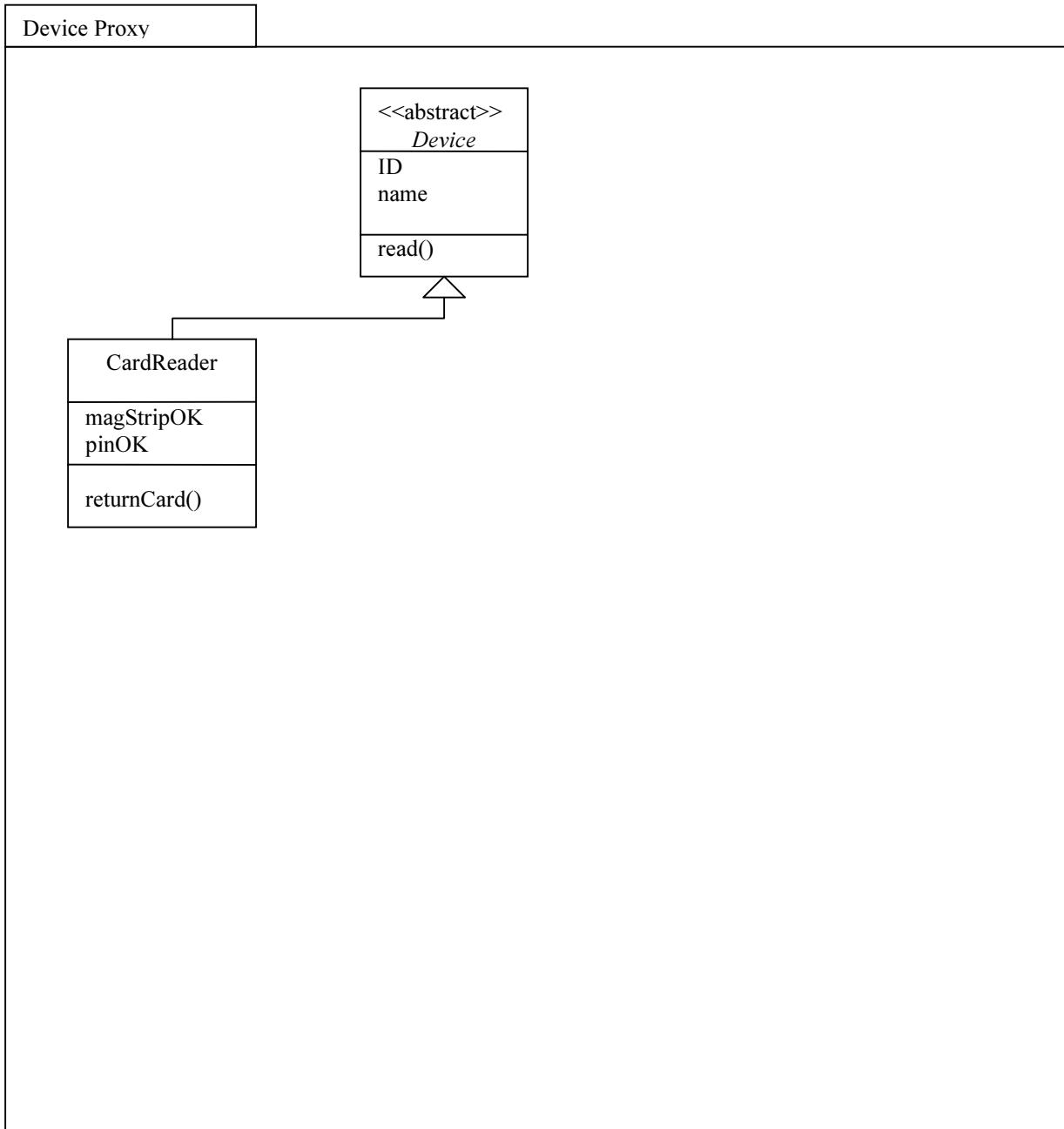
    /**
     * Increment the number of proxies using this object by one.
     */
    synchronized void addProxy() {
        proxyCount++;
    } // addProxy()

    /**
     * Decrement the number of proxies using this object by one.
     */
    synchronized void removeProxy() {

        proxyCount--;
    } // removeProxy()
} // class ReferenceCountedHashTable
} // class LargeHashtable
```

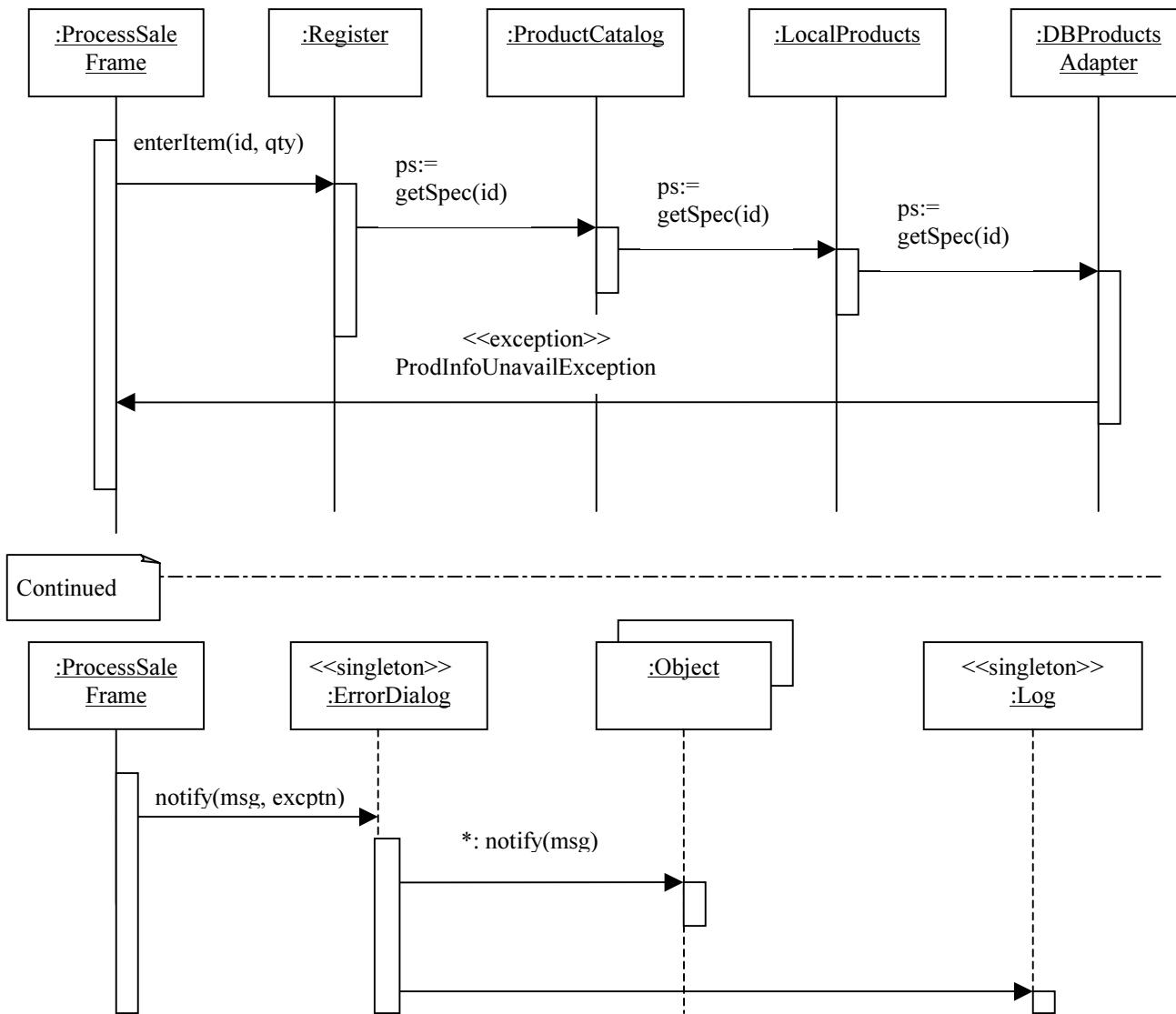
6. **Modeling physical devices:** At a low level a physical device has a device driver for the underlying operating system. A Java class (for example, one that implements `jpos.CashDrawer`) generally uses JNI (Java Native Interface) to make calls out to these device drivers. Such classes adapt the low-level device driver to the JavaPOS⁴ interfaces and can thus be characterized as **Adapter** objects or **Device Proxy** objects – local proxies that control or enhance access to the physical devices. [It is not uncommon to be able to classify a design in terms of multiple patterns.]

If you were designing ATM software, you might create a software proxy/adapter for each external device. Begin a UML package that shows the classes and interfaces. Include a few attributes and methods...



⁴ There are standards for POS devices. Buy rather than build. See www.nrf-arts.org and www.javapos.com.

7. **Remote Proxy Example** -- failover to a local service for product information in the diagram below.⁵
 Sometimes called a Redirection Proxy or a Failover Proxy.



Explain how this works. Where are there identifiable patterns? Should you try the external service before the local? Where must there be common interfaces implemented?

Larman (ref below) further talks about using a Singleton-accessed central error logging object as a pattern and error dialog pattern (a singleton-accessed, application independent, non-UI object – both from Renzel97) to notify users of errors. In the diagram above, what are examples of objects (represented by `:Object`) that error dialog could notify?

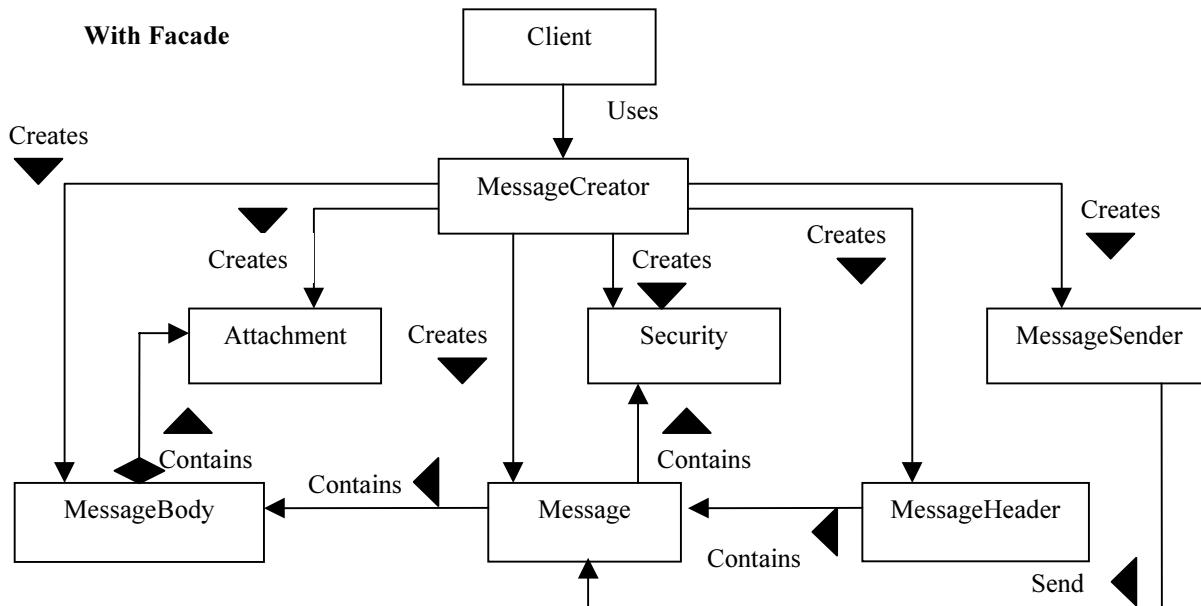
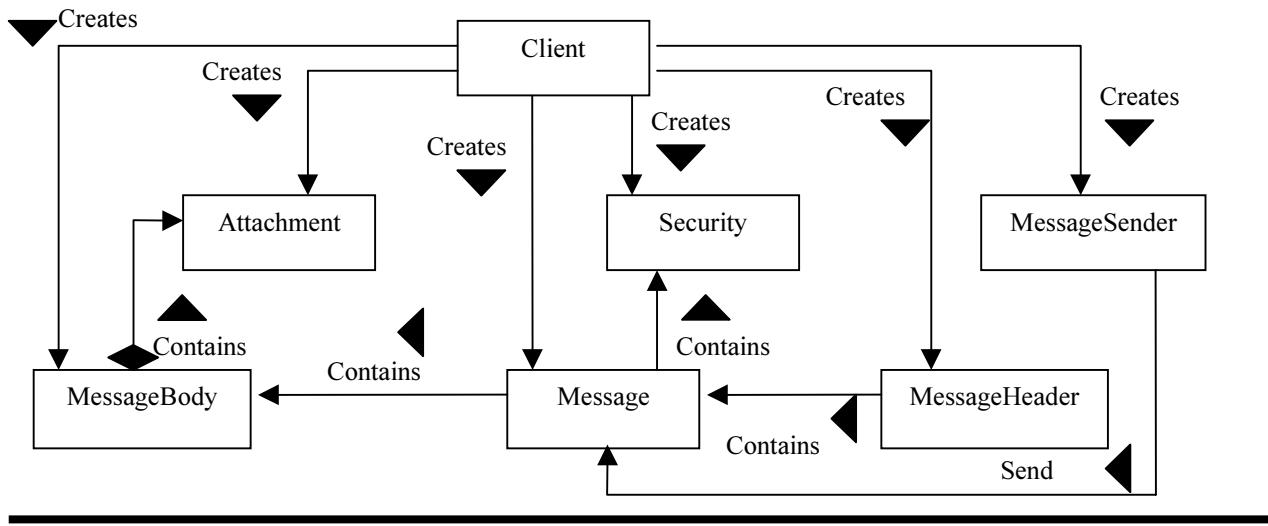
⁵ Larman, “Applying UML and Patterns”, Second Edition, p. 520.

8. Draw the UML diagram for the Door Controller surveillance system example in the **Decorator Pattern** explanation. Consider a DoorController interface decorated with two concrete classes for surveillance options (i.e., TV Monitor, Camera) and 3 kinds of concrete door controllers (simply call them “DoorControllerA”, “DoorControllerB”, “DoorControllerC” if you wish.

Notes

Comments: Stylizes the use of other classes, without adding functionality. Allows simple clients without requiring them. Provides unified interface to a set of interfaces in a subsystem. Often a Singleton. Related: AbstractFactory, Mediator.

Example and diagram: Set of classes to support creating/sending email.



Participants: Façade, subsystem classes

Usage: `java.net.URL` provides access to contents of URLs; many classes operate behind the façade provided. Compilers delegate tasks to scanner, parser, program node, etc. CHOICES operating system (GoF p. 192).

Solution: See Code, Appendix page B-15.

9. Facade (aka: convenience routines) GoF p. 185

Pattern Thumbnail: Simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.

Forces:

- There are many dependencies between classes that implement an abstraction and their client classes.
 - You want to simplify client classes and move dependencies from clients to a façade.
 - You want to isolate communication with a set of classes which is likely to change.
 - You want to provide a default way to access the functionality of a set of subclasses.
 - You want to shield clients of a subsystem from the complexity of the interfaces of all the classes in the subsystem.

Question:

Part 1 - How complex must a sub-system be in order to justify using a facade?

Part 2 - What are the additional used of a facade with respect to an organization of designers and developers with varying abilities? What are the political ramifications?

Part 3 – (Java) What about making classes private inner classes of façade? How does this relate to “don’t talk to strangers?”

Consequences: Clients don't need to know about any classes behind the façade. This reduces coupling and facilitates change of classes behind the facade. Yet, clients needing direct access to abstraction implementing objects can still access those objects directly.



Notes

Exercises

1. . Discuss the use of a **Façade** class to interface to various subsystems such as a printer subsystem (actually any I/O subsystem) or a database interface subsystem. What advantages are derived by using this pattern?

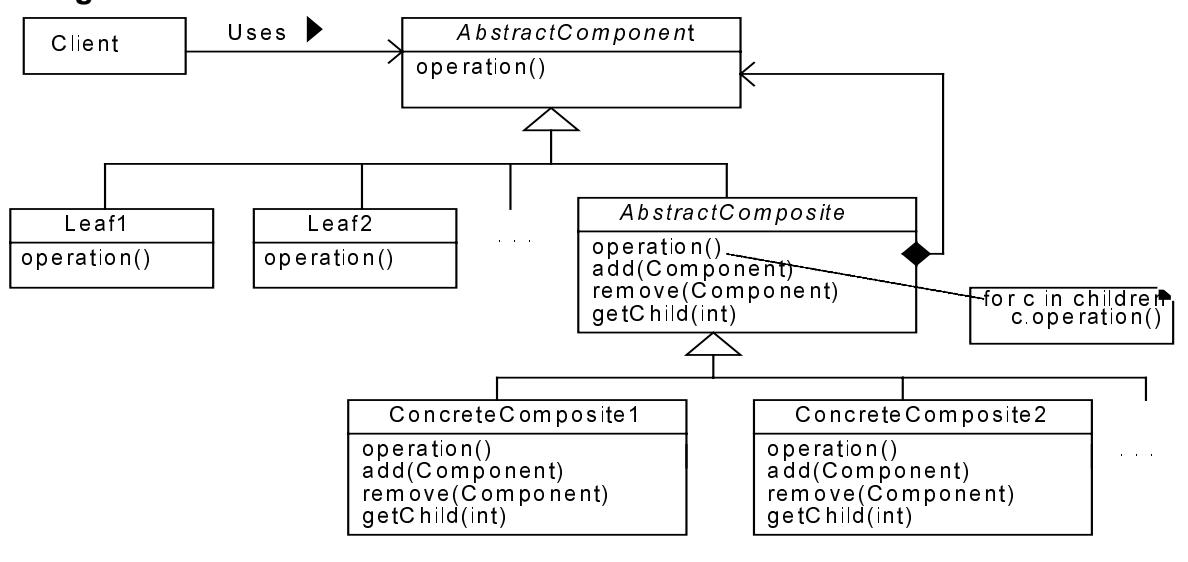
What disadvantages?

2. How is **Façade** different from **Proxy**?

Notes

Comments: The key here is having one abstract class that represents *both* the primitives (leaf classes) and the containers (composite classes). Specialized methods should appear only in classes that need them – putting a specialized method in a general-purpose class is contrary to high cohesion, which this pattern encourages. Chain of Responsibility pattern can be combined with Composite by adding child-to-parent links. Visitor pattern can be used to encapsulate operations in a single class.

Diagram:



Participants: Component, Leaf, Composite, Client.

Usage: Java API swing classes – **JComponent** is an abstract Component, **JContainer** is an abstract Composite; examples of concrete Composite are **JPanel**, **JFrame**, and **JDialog**. Leaf examples include **JLabel**, **JTextfield**, **JButton**.

Solution: See Code, Appendix page B-12.

Issues:

1. Explicit parent references – how to ensure consistency?
2. Sharing Components -- often useful, but may not work when Components have one parent
3. Maximizing the Component interface – methods for Composite may not make sense for Leaf
4. Declaring the child management operations – which level to put add and remove? Safety vs. Transparency
5. Should Component implement a list of Components? Child refs in base class take space
6. Child ordering -- front to back? Iterator pattern may help here
7. Caching to improve performance – then need interface to tell Composites when cache is invalid
8. Who should delete components? (in languages without garbage collection)
9. What's the best data structure for storing components? Linked list, array, hash table, etc.

10. Composite (aka: Recursive Composition) GoF p. 124



Pattern Thumbnail: This pattern allows you to build complex objects by recursively composing similar objects in a tree-like manner. The objects in the tree can also be manipulated consistently if all objects in the tree have a common superclass or interface.

Example: Building a document which has subcomponents of Page, Columns, LineOfText, Character; additionally the document has Frame, Image. (Apply these classes to the diagram below.) Also Dir and File.

Forces:

1. You have a complex object you want to decompose into a whole-part hierarchy;
2. You want to minimize complexity in the hierarchy by minimizing the number of different kinds of child objects and be able to treat objects similarly.

Question:

Part 1 - How does composite help to consolidate system-wide conditional logic?

Part 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?

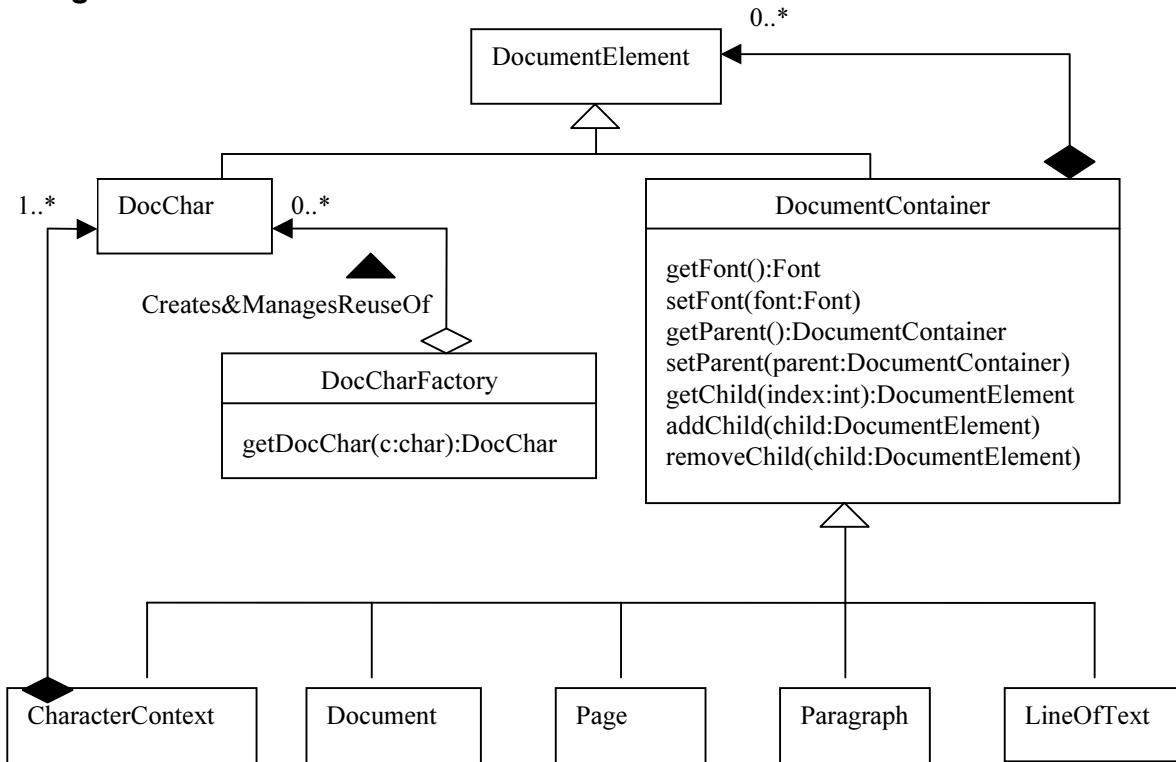
Consequences:

1. You can define class hierarchies consisting of both primitive and composite objects where more complex composite objects can be composed of either primitives or composites,
2. Make the client simple by creating a tree-structured composite object that treats all objects as instances of AbstractComponent without having to be aware of subclasses,
3. Easy to add new kinds of Components,
4. Design can end up being too general since it is hard to restrict the components in a composite – you may have to use runtime checks since the type system cannot enforce constraints.

Notes

Comments: Omit state from an object's representation to promote sharing; state is not stored but rather passed as a method argument. Shared flyweight objects are often immutable. Factory method is generally used to create flyweight objects. Often combined with Composite to represent hierarchical structure or graph with shared leaf nodes. (Note: leaf cannot then contain ptr to parent, ptr to parent must be passed.)

Diagram:



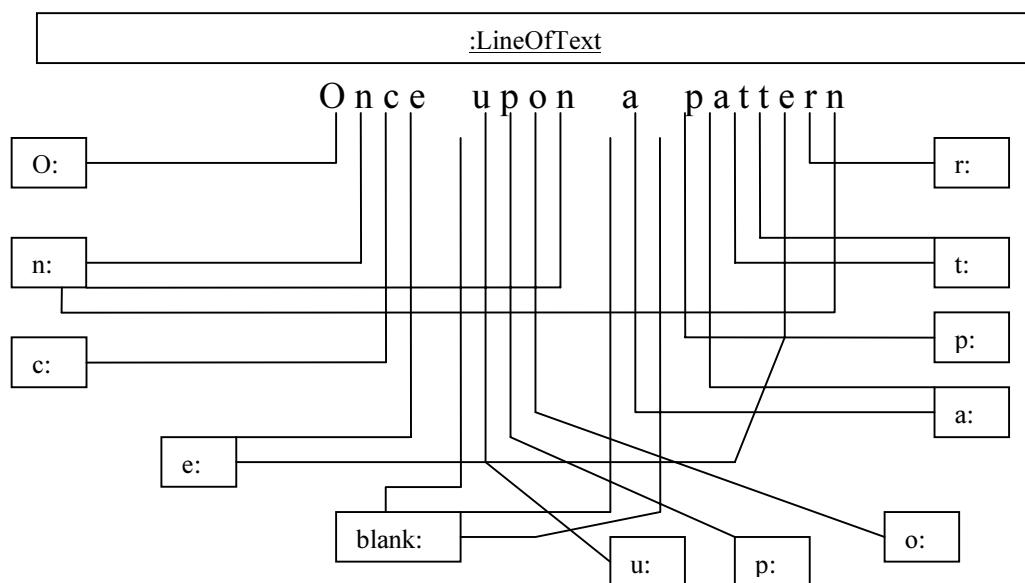
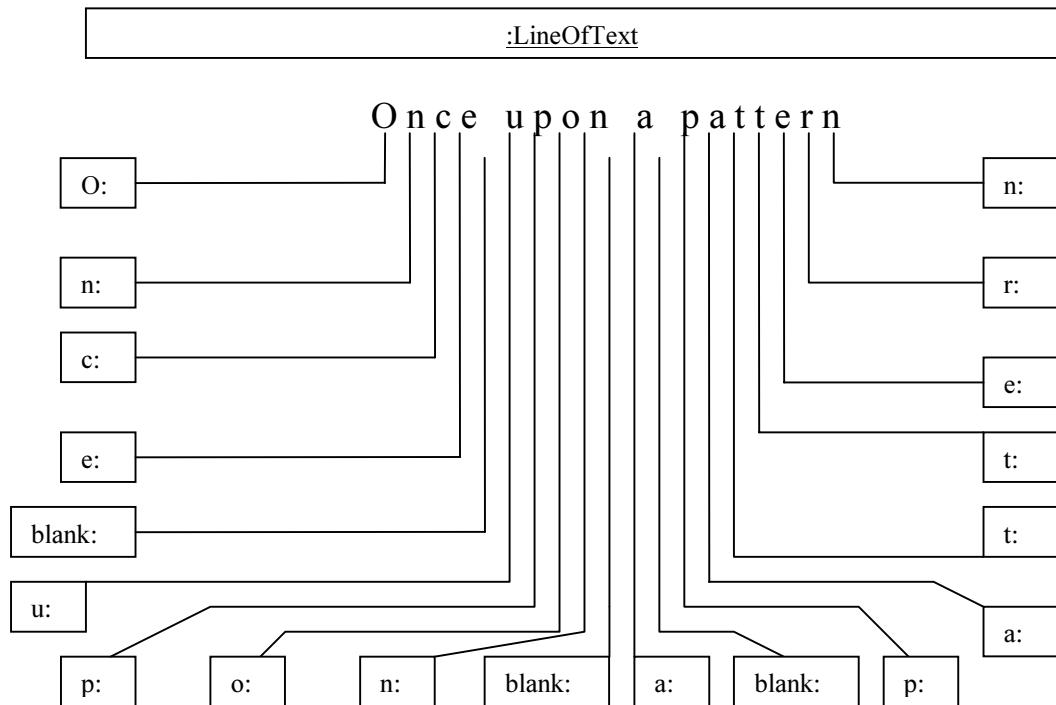
Note: **CharacterContext** class instances store extrinsic attributes for a range of characters.



11. Flyweight GoF p. 195

Pattern Thumbnail: Efficiently support sharing of many fine-grained objects and avoid the expense of multiple instances of the same information.

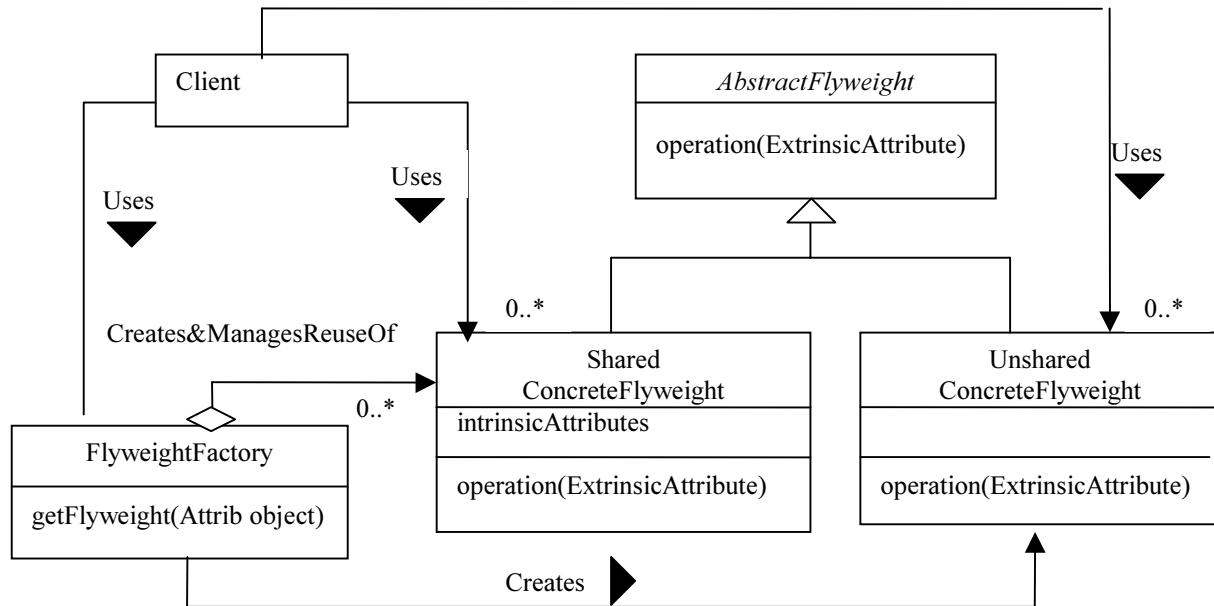
Example: Characters in a document.



► **How much storage reduction for this simple example?**

Notes

Diagram:



Participants: AbstractFlyweight (superclass), SharedConcreteFlyweight, UnsharedConcreteFlyweight, FlyweightFactory, Client.

Usage: Java's management of String literal objects in a pool; the String class' intern method is responsible for managing the String objects used. LayoutManagers are strategies implemented as flyweights.

Solution: See Code, Appendix page B-17.

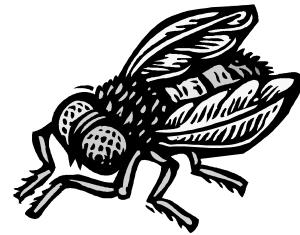
Flyweight, continued

Forces: Need to optimize similar fine-grained instances by sharing, and it is possible; most object state can be made extrinsic.

Question:

Part 1 - What is a non-GUI example of a flyweight?

Part 2 - What is the minimum configuration for using flyweight? Do you need to be working with thousands of objects, hundreds, tens?



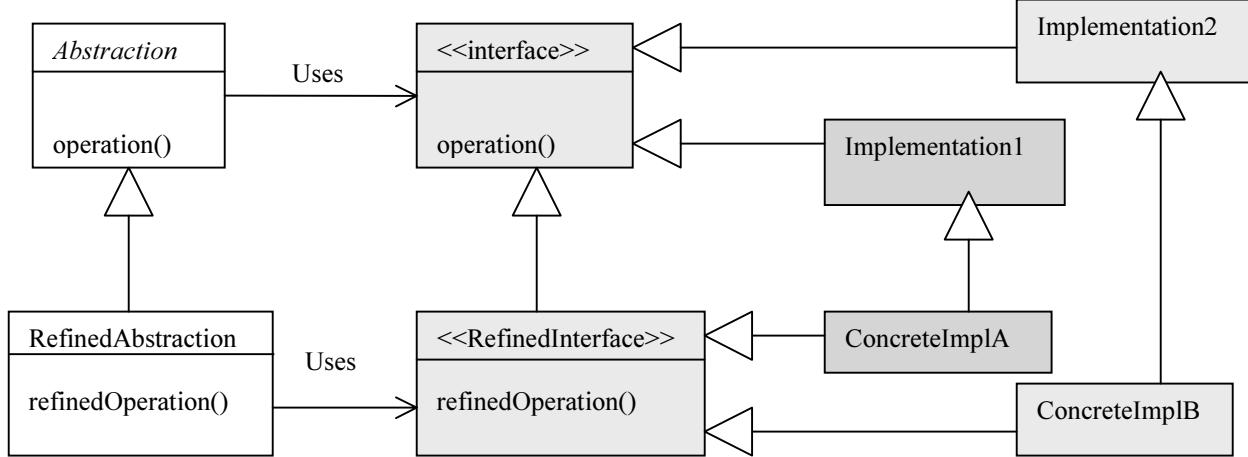
Consequences: Memory vs. Speed issues.

- Reduces the number of objects in memory.
Makes program more complex by having to provide extrinsic state and managing the reuse of flyweight objects.
- Can increase runtime of a program because it requires more effort for objects to access extrinsic state than intrinsic state.
- Therefore, consider for optimization only after rest of design is worked out.

Notes

Comments: Related to the Layered architecture pattern and the Abstract Factory pattern.

Diagram:



Participants: Abstraction, RefinedAbstraction, Interface implemented by Abstraction classes, RefinedInterface (to match RefinedAbstraction), Implementation1 & 2, ConcreteImplementationA & B.

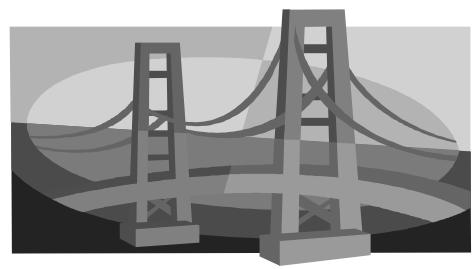
Usage: java.awt contains the abstract Component class which encapsulates the common logic for all GUI components. Subclasses (such as Button, List, TextField) encapsulate the logic for those components and are platform independent. Abstract Factory pattern is used to create/change concrete component styles on the fly for pluggable look and feel (PLAF).

Solution: See Code, Appendix page B-6.

12. Bridge (aka: Handle/Body) GoF p. 151

Pattern Thumbnail: A stylized kind of Strategy pattern, useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations.

Bridge implements abstractions and implementations as independent classes that can be combined dynamically.



Example: Suppose you need to create classes to model devices such as scales, speed-measuring devices, and location-sensing devices. In common, these devices perform a physical measurement and produce a number at the request of a computer. They differ, among other things, in the type of measurements they produce. If there are different manufacturers, communication with the sensors will also likely vary. The challenge is to produce a hierarchy of sensors in a way that does not tie to any particular implementation, so you also create a parallel hierarchy of implementation interfaces. Common logic goes in a manufacturer-independent class; logic specific to a manufacturer goes in a manufacturer-specific class.

Forces:

- If you combine hierarchies of abstractions and hierarchies of their implementations into a single class hierarchy, client classes become tied to a specific implementation of the abstraction – bad idea. Changing the implementation used for an abstraction should not require changes to the clients using that abstraction.
- You want to reuse logic common to different implementations of an abstraction; encapsulating it in a separate class would work.
- You want to create a new implementation of an abstraction without having to reimplement the common logic of the abstraction.
- You want to extend the common logic by writing one new class rather than writing a new class for each combination of the base abstraction and its implementation. When appropriate, multiple abstractions should share the same implementation.

Question: How does a Bridge differ from a Strategy and a Strategy's Context?

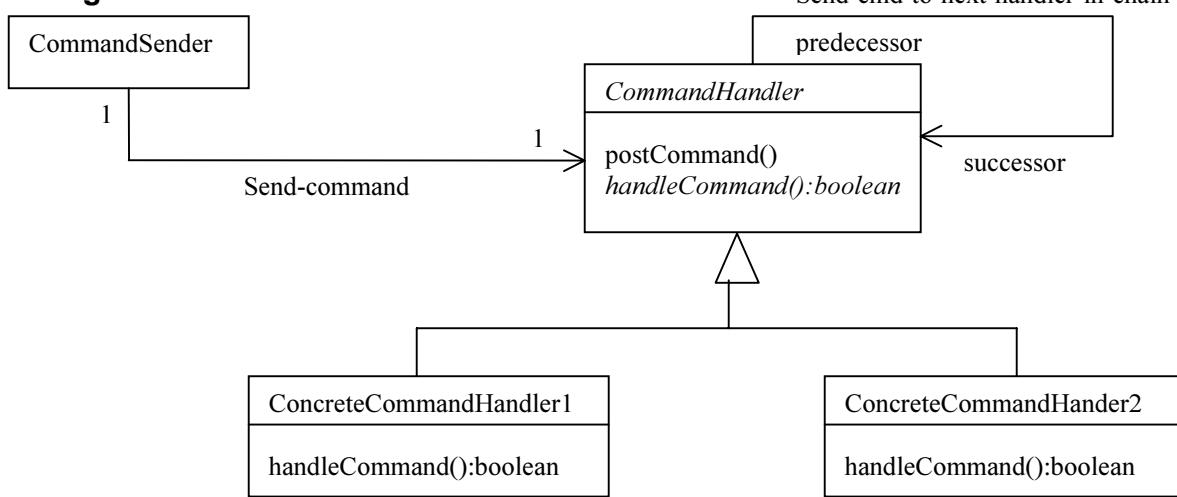
Consequences:

- Keeps the classes representing the abstraction independent of the classes that supply an implementation for the abstraction.
- You end up with separate class hierarchies which can each be extended independently.
- You can have multiple implementation classes for an abstraction class, or, multiple abstraction classes using the same implementation class. A mix-n-match strategy.
- Clients using an abstraction class have no knowledge of the implementation classes, so implementation can change without any impact on the clients.

Notes

Comments: Mediator which only forwards messages. Related to Command (bundle the info passed), Composite (parent can act as successor in chain), TemplateMethod.

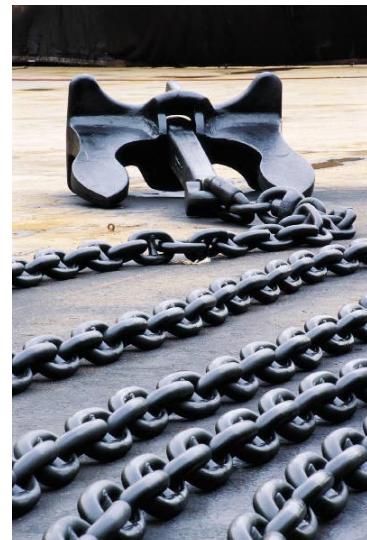
Diagram:



Participants: CommandSender, CommandHandler (abstract superclass), ConcreteCommandHandler1, ConcreteCommandHandler2

Usage: Java 1.0 event model used this – there it was a BAD idea. An event posted to a button, for example, would either handle the event or post it to its container. A mouse move, for example, generated lots of events, kept being posted out the containment hierarchy, and perhaps no object really cared. Inefficient, inflexible.

Solution: See Code, Appendix page B-9.



13. Chain of Responsibility (aka: Merge) GoF p. 223



Pattern Thumbnail: An object can send a command without knowing what objects will receive it. The command passes to a chain of potential handlers that is typically part of some larger structure. Each object in the chain may handle the command, pass it on, or both. Objects in the chain could be assembled/ordered dynamically.

Example: You have to program a security system with a number of sensors: motion, temperature, smoke, carbon monoxide, etc. A computer receives the sensor info over a network, logging all status information and maintaining a display, transmitting alarms when appropriate, etc. You want to keep the set-up scalable. Instantiate an object per sensor (Device Proxy) where objects related to the sensor don't assume any particular environment except that they are at the bottom of a hierarchy and simply pass their information up.

Other objects correspond to rooms, areas, floors, buildings. For example, temp > 30° in the freezer room should sound an alarm; temp > 150° in a warehouse might turn on the sprinkler system there, etc. The actions to be taken are delegated to objects with more contextual information than is available to the lowly sensors.



Forces:

- You want an object to be able to send a command to another object without specifying the receiver; the sending object doesn't care who handles the command, only the an object will receive and handle the command.
- More than one object may be able to receive and handle a command, so there needs to be a way to prioritize among the receivers without the sending object knowing anything about them.

Question:

Part 1 - How does the Chain of Responsibility pattern differ from the Decorator pattern or from a linked list?

Part 2 - Is it helpful to look at patterns from a structural perspective? In other words, if you see how a set of patterns are the same in terms of how they are programmed, does that help you to understand when to apply them to a design?

Consequences:

- Reduces coupling between sending and receiving objects.
- Allows flexibility in deciding how to handle commands and a strategy can be changed by re-ordering the chain.
- Commands not handled are ignored, so receipt is not guaranteed.
- Could be inefficient for large chains.

Notes

Exercises

1. **Chain of Responsibility:** Explain this pattern in your own words.
2. Draw UML for the device modeling example in the **Bridge** pattern. You can use the following suggested classes and interfaces or make up your own set:

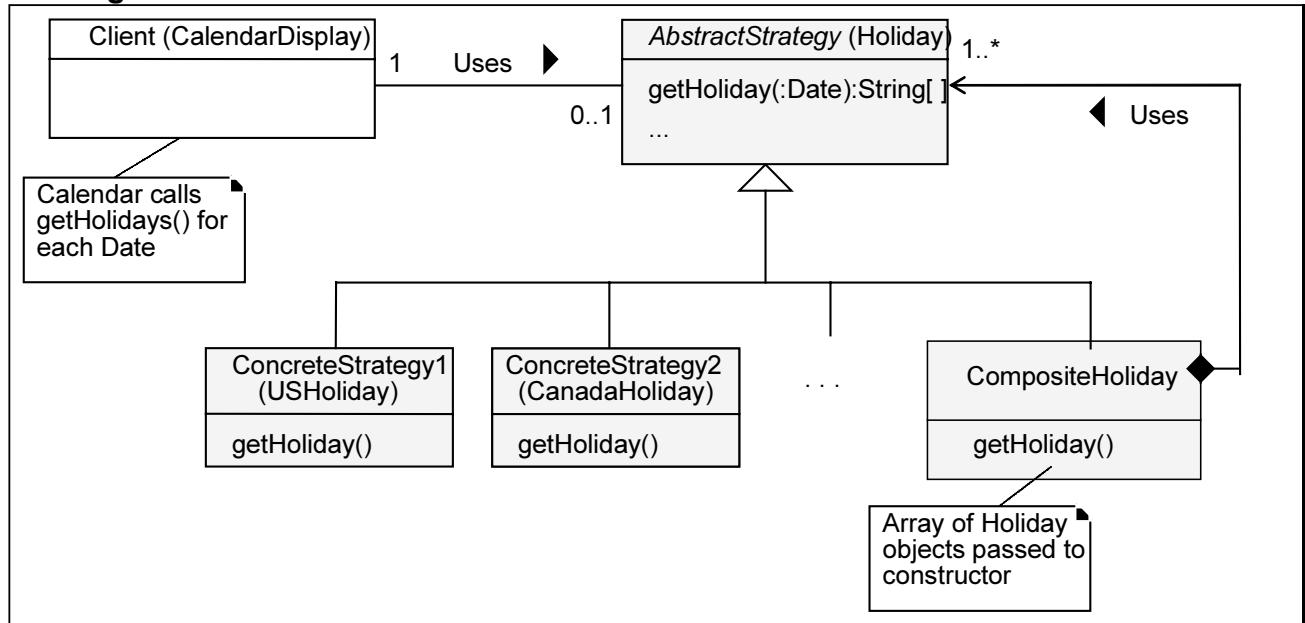
Interfaces: SimpleSensor with getValue() method; AveragingSensor with startAveraging(); StreamingSensor with setSamplingFrequency()

Classes: SimpleSensor and subclasses AveragingSensor and StreamingSensor which implement the corresponding interfaces;
FalconSimpleSensor, FalconAveragingSensor, FalconStreamingSensor;
OspreySimpleSensor, OspreyAveragingSensor, OspreyStreamingSensor.
(You get the idea... the trick is to draw the diagram.)

Notes

Comments: Delegation for changing behavior; generalizes higher-order functions. Similar to Adapter – difference is the intent. Strategy often used with Null Object. Template Method achieves alternative behavior through subclassing rather than through delegation. If there are many client objects, consider using Flyweight.

Diagram:



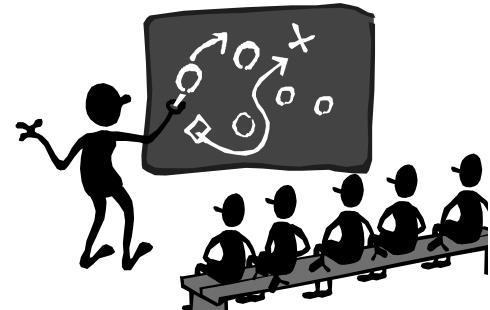
Participants: Strategy, ConcreteStrategy, Context

Usage: Java API: `CheckedInputStream`, `CheckedOutputStream` both use this pattern to compute checksums on byte streams. Constructors for both clients take a `Checksum` argument. Although this example does not include Composite, you could easily combine Strategy with Composite.

Solution: See Code, Appendix page B-30.

14. Strategy (aka: Policy) p. GoF 315

Pattern Thumbnail: Encapsulates related algorithms in classes that are subclasses of a common superclass. This allows selection of algorithm to vary by object and also allows it to vary over time.



Example: Calendar display: must be able to display holidays celebrated by different nations and different religious groups. User specifies which sets of holidays to display.

Forces: A program needs to provide multiple variations of some behavior. You want a consistent way to access behaviors, so encapsulate the behavior in separate classes. But, want clients to be unaware of implementation details, so use a common superclass or interface for access.

Question:

Part 1 -What happens when a system has an explosion of Strategy objects? Is there some way to better manage these strategies?

Part 2 - In the implementation section of this pattern, the authors describe two ways in which a strategy can get the information it needs to do its job. One way describes how a 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?

Consequences:

- Behavior of client objects is determined dynamically on a per-object basis.
- Client objects are simplified – do not have responsibility for selecting/implementing alternate behaviors.

Notes

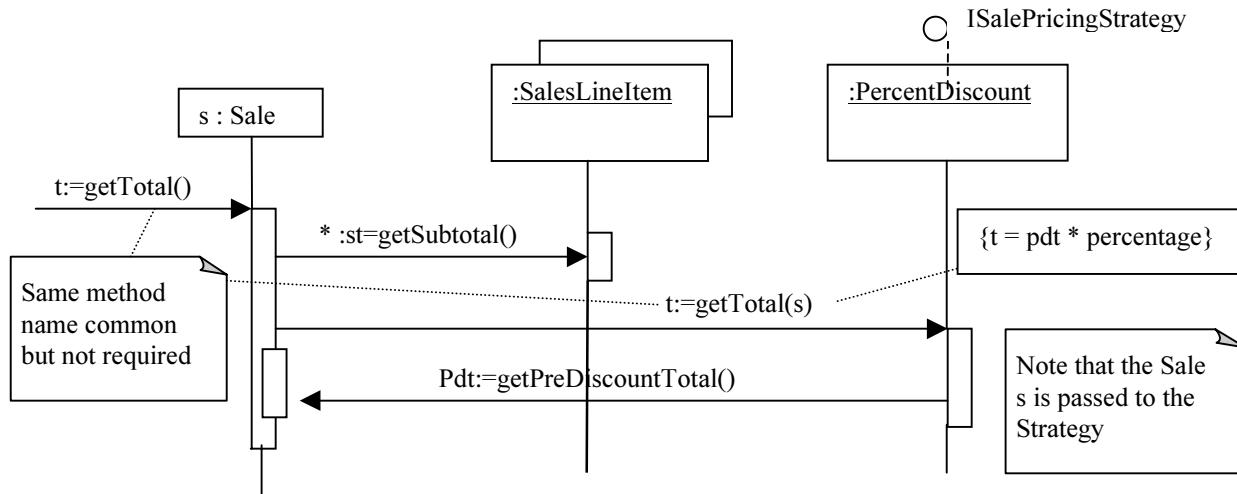
Exercises

1. How does **Strategy** differ from **Adapter**?
 2. How is the macro command (from **Command** pattern, GoF p. 235, 241) an example of a **Composite**? Where is the **Composite** in the middle of the **Strategy** calendar example?
(Don't worry that we haven't discussed Command yet – just note the Composite aspects.)
 3. List the implementation issues for **Composite** (p. 166-170).
 4. How would you use the **Null Object** pattern with the Holiday example in **Strategy**?
 5. List the consequences of using the **Strategy** pattern (p. 317).

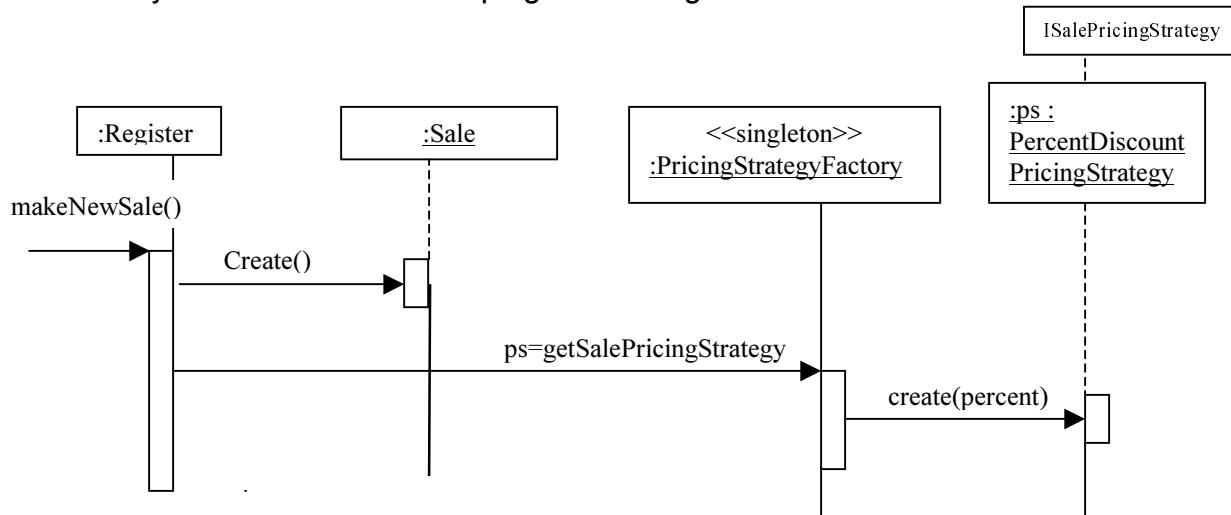
6. Suppose you need to design a complex pricing policy ...

which includes such variations as discounts on particular items, a store-wide discount for the day (e.g., 20% on Mondays, 10% on Tuesdays), an absolute discount amount over some threshold (\$50 on sales => \$500), a preferred customer discount, a senior citizen discount (15%), etc. You also want the flexibility to change any of these policies or add new variations (**Strategy Pattern**).

- b. Create a UML diagram which shows an interface called `ISalePricingStrategy` which has a method `getTotal(Sale)` which returns `Money`. Assume a `Sale` object has an attribute called "amount" of type `Money`.
- c. Add concrete classes which implement that interface: `PercentDiscount`, `AbsoluteDiscount`, `SeniorDiscount` (possibly others) with code or pseudo-code for the `getTotal` method.
- d. Identify participants in the collaboration diagram below. Note that `Sale` delegates some of its work to its `SalePricingStrategy` object.



7. Your UML diagram could include a PricingStrategyFactory (a **Singleton**) to create the instances of the various Strategy classes. When a Sale instance is created it can ask the **factory** for the pricing strategies as in the collaboration diagram below. Why is this better than keeping the strategies as attributes of Sale?

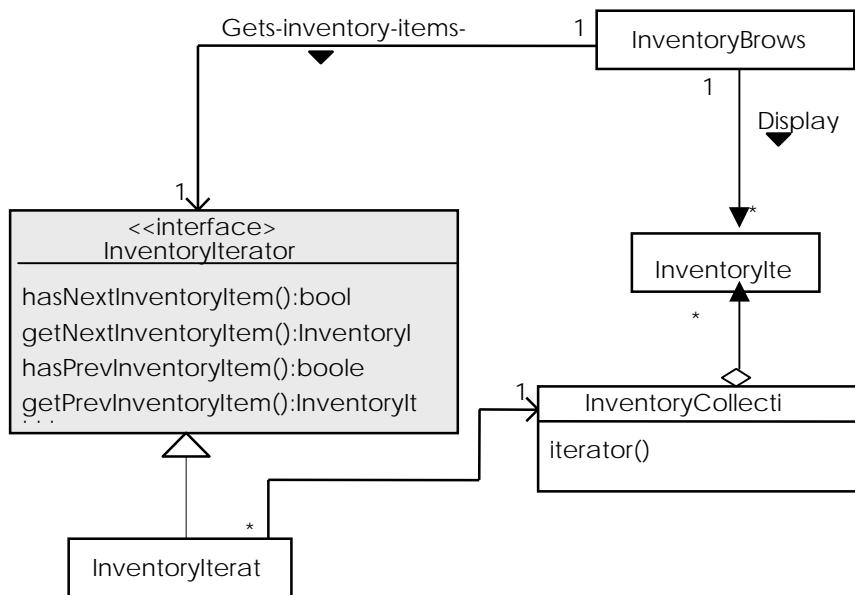


8. Now, how to handle multiple, conflicting pricing strategies? Suppose a senior who is also a preferred customer buys an item with a discount and total sales exceed the threshold for the day? Combine with **Composite** by adding an abstract CompositePricingStrategy which also implements the interface in #1 as well as having an add(ISalePricingStrategy) method. Design two conflict resolution subclasses: CompositeBestForCustomer and CompositeBestForStore. Add to your UML diagram and redesign the collaboration diagram in part 3 above to include CompositePricingStrategy. Optional: write sample code for CompositePricingStrategy and CompositeBestForCustomer classes.

Notes

Comments: Often used with Composite. A stylized kind of adapter. Collection classes may use the Factory Method pattern to determine what kind of iterator to instantiate. Null iterators are sometimes used, implementing the Null Object pattern.

Diagram:



Participants: Iterator, Concreteliterator, Aggregate, ConcreteAggregate

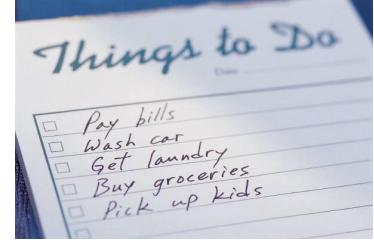
Usage: `java.util` collection classes follow the Iterator pattern and use the Iterator interface; classes that implement the Collection interface define private inner classes that implement Iterator.

Solution: See Code, Appendix page B-20.

15. Iterator (aka: Cursor) GoF p. 257

Pattern Thumbnail: A way to access an aggregate object sequentially without having to know the underlying structure. A robust iterator ensures insertions/deletions won't mess up the traversal.

Example: Browsing the inventory of a warehouse. A user interface allows a user to see things like description, quantity on hand, location, etc., about each inventory item. You don't want to be tied to the actual class that provides the collections of inventory items (loose coupling).



Forces: A class needs access to the contents of a collection without becoming dependent on the class used to implement the collection. You want to have a consistent way to access the contents of collections independent of the specific type of collection.

Question: Consider a composite that contains loan objects. The loan object interface contains a method called "AmountOfLoan()" which returns the current market value of a loan. Given a requirement to extract all loans above, below or in between a certain amount, would you write or use an Iterator to do this?

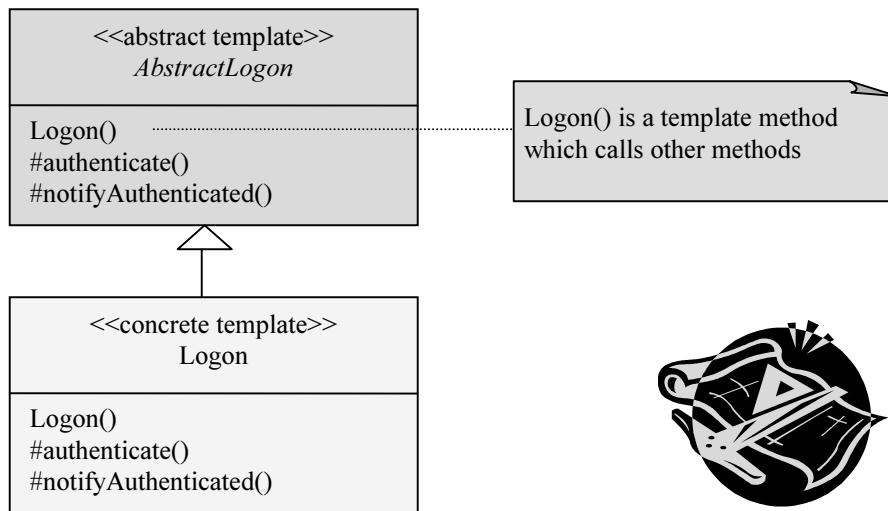
Consequences:

- A uniform interface for traversing aggregate structures.
- Separation of the iterator from the list itself; the possibility of multiple iterators on a list, and the possibility of different kinds of iterator objects that traverse the list in different ways. For example, if the list is a dictionary type (key, value), there could be different methods for creating iterators that traverse just the key objects and ones that traverse just the value objects.

Notes

Comments: A framework – combination of facade and strategy – “don’t call us, we’ll call you.” An abstract class that contains only part of the logic needed, coded so the concrete methods call abstract methods for the missing logic. Subclass methods override the abstract methods to provide the behavior which needs to vary.

Diagram:



Participants: AbstractTemplate, ConcreteTemplate

Solution: See code, Appendix page B-31.

16. Template Method GoF p. 327

Pattern Thumbnail: An abstract class that contains only part of the logic needed, coded so concrete methods call an abstract method for the missing logic. Subclass methods override the abstract methods.

Example: You need a reusable class for logging on users. It should (1) prompt for ID and password, (2) authenticate, with the results encapsulated in an object, (3) show some “progress” display, and (4) notify the application and supply the authentication object when complete.

Numbers 1 and 3 are application independent; for 2 and 4, each application must provide its own logic.



Forces:

1. You need to design a class for use in multiple programs. The overall responsibility remains basically the same, but portions of the behavior need to vary from program to program.
2. You have an existing design with multiple classes that do similar things and want to minimize code duplication by factoring differences into discrete methods with common names.

Question: 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?

Consequences: Programmer is forced to override abstract methods to complete a superclass. If well-designed, provides guidance of how to do this. Factoring out common behavior is a fundamental technique for code reuse.

Exercises

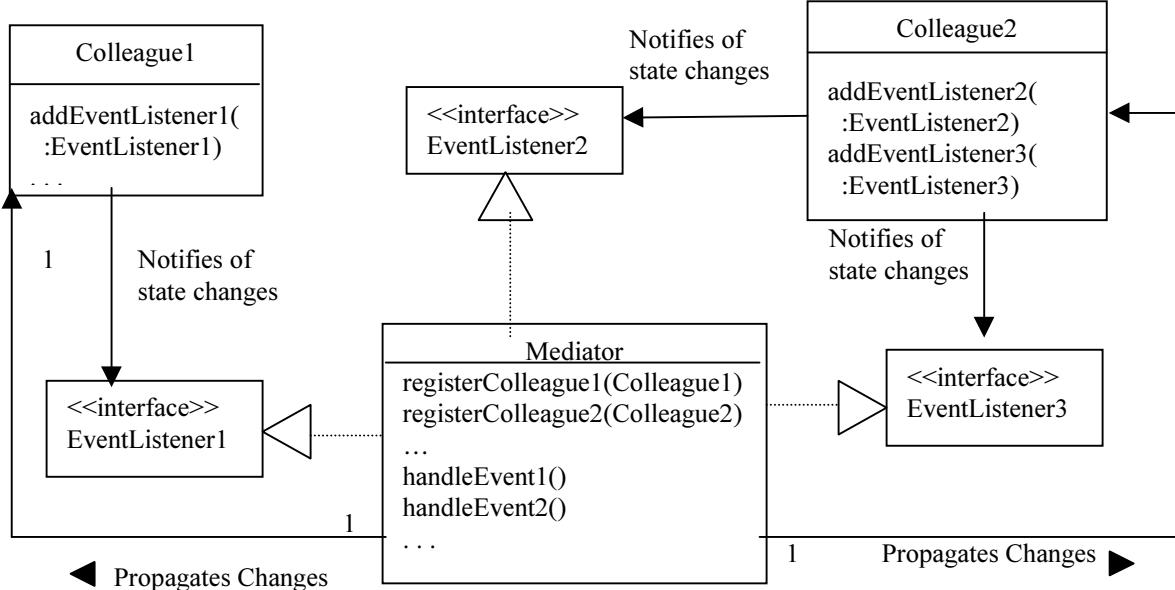
1. a. Write code or pseudo-code for the four **Iterator** interface methods `first()`, `next()`, `current()`, `done()` for both an array and a singly linked list. Write pseudo-code for a client to traverse each of the two concrete data structures. Note that an interface (in Java) has no variable such as “index” yet you will need an index for an array. Where will that variable come from?
- b. What changes would be needed in the client code to traverse a binary tree? To have two traversals going at the same time on the array?
- c. How could **Factory Method** be used to determine what kind of **Iterator** to instantiate?

2. Explain how **Iterator** is a specialized version of the **Adapter** pattern.
 3.
 - a. Explain The “Hollywood principle” – “Don’t call us, we’ll call you” (GoF p. 327) as it relates to the **Template** pattern.
 - b. Describe the kinds of operations **Template** methods tend to call.
 - c. Discuss at least one of the implementation issues (p. 328).

Notes

Comments: Reduces the number of interconnections between senders and candidate receivers by defining an object that encapsulated information on how a set of objects interact; can do other things besides forward messages. Forms classic bond with Observer

Diagram:



Participants: MediatorIF, ConcreteMediator, Colleagues

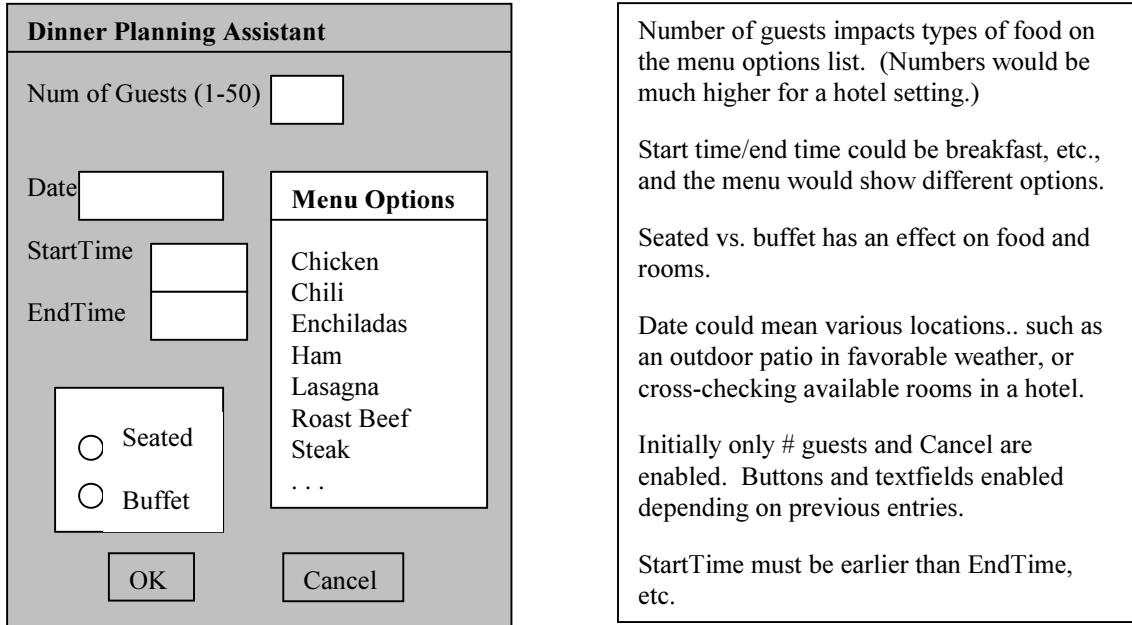
Usage: In `java.awt.swing`, `JComponent` objects use an instance of `FocusManager` as a mediator. The `FocusManager` recognizes keystrokes that should cause a different `JComponent` to receive the focus, and changes the focus accordingly. This is a bit different than the pattern: (1) only passes very low level `KeyEvents` – your implementations generally have to handle multiple events, and (2) `JComponents` talk to `FocusManager` object directly, not through an interface.

Solution: See Code, Appendix page B-21.

17. Mediator GoF p. 273

Pattern Thumbnail: Uses one object to coordinate state changes between other objects. Logic is in one place instead of distributed among a set of objects, thereby adding cohesion in the logic and providing looser coupling among the other objects.

Example: A dinner planning aide (from your personal to Hotel Banquet Planning):



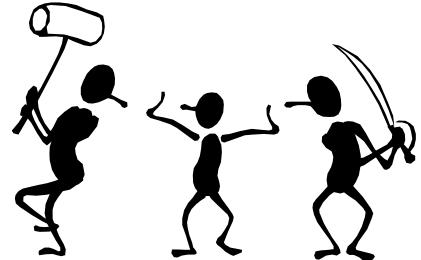
Forces:

- You have a set of related objects involved in multiple dependencies.
- You don't want to end up creating lots of dependent subclasses so the individual objects can implement the various dependency relationships.
- Classes are hard to reuse due to all the complex relationships.

Question: Since a Mediator becomes a repository for logic, can the code that implements this logic begin to get overly complex, possibly resembling spaghetti code? How could this potential problem be solved?

There are two approaches when implementing: (1) Mediator object maintains its own internal model of state of Colleague objects, and (2) Mediator object fetches state of each Colleague when it needs state information. What are the tradeoffs in approaches?

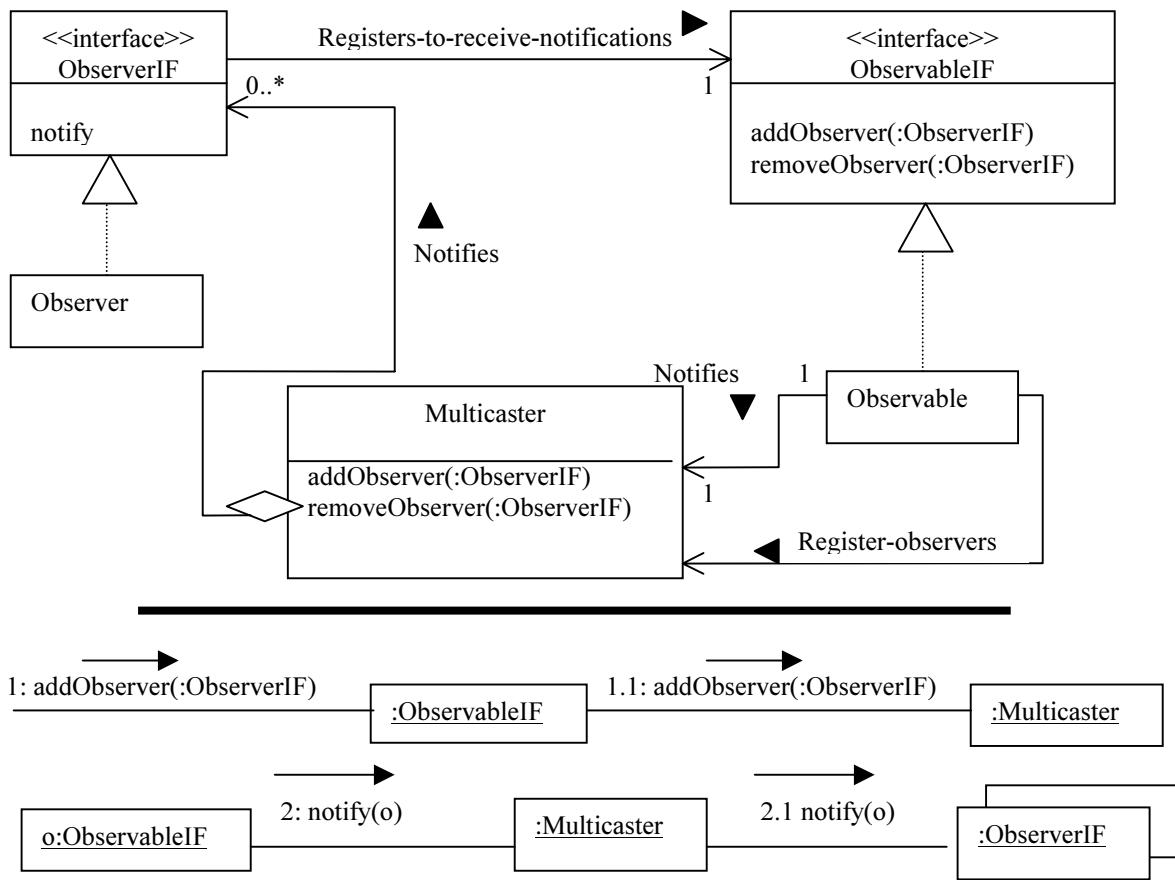
Consequences: Using the pattern limits subclassing needed, decouples colleagues thereby making them more reusable, simplifies object protocols, abstracts the interactions, centralizes control by shifting the complexity to the Mediator object (which is not generally reusable).



Notes

Comments: Maintains a list of dependents to notify of changes; part of MVC pattern. Often used with Mediator.

Diagram:



Participants: `ObserverInterface`, `ConcreteObserver`, `ObservableInterface (Subject)`, `ConcreteObservable (ConcreteSubject)`

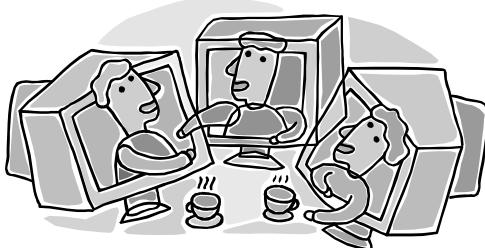
Usage: Java's delegation event model is a specialized form of the Observer pattern and includes a `Multicaster` class (as in the diagram).

Solution: See Code, Appendix page B-24.

18. Observer (aka: Publish-Subscribe, Dependents) GoF p. 293

Pattern Thumbnail: Allows objects to register as a dependent of another object, with a callback.

Example: ChatServer with multiple clients who look like they know each other and are communicating directly, but all communication is through the Server and the clients are totally separate.



Forces: You want to implement two classes which except for some dependency are otherwise independent. An instance of one needs to notify one or more other objects when its state changes, but you want to keep the objects independent of each other.

Question:

Part 1 - the classic Model-View-Controller design is explained in Implementation note #8: *Encapsulating complex update semantics*. Would it ever make sense for an Observer (or View) to talk directly to the Subject (or Model)?

Part 2 - What are the properties of a system that uses the Observer pattern extensively? How would you approach the task of debugging code in such a system?

Part 3 - Is it clear to you how you would handle concurrency problems with this pattern? Consider an Unregister() message being sent to a subject, just before the subject sends a Notify() message to the ChangeManager (or Controller).

Consequences:

- An object can deliver notifications to other objects without either sending or receiving objects being aware of each other.
- Delivering notifications can take a long time if an object has lots of notifications; notifications can be cascaded via indirect observers.
- Cyclical dependencies can cause serious problems and throw StackOverflowError, so internal flags may need to be set to avoid recursive notification. Multi-threading may need to ensure consistency of objects and avoid blocking threads.



Notes

Exercises

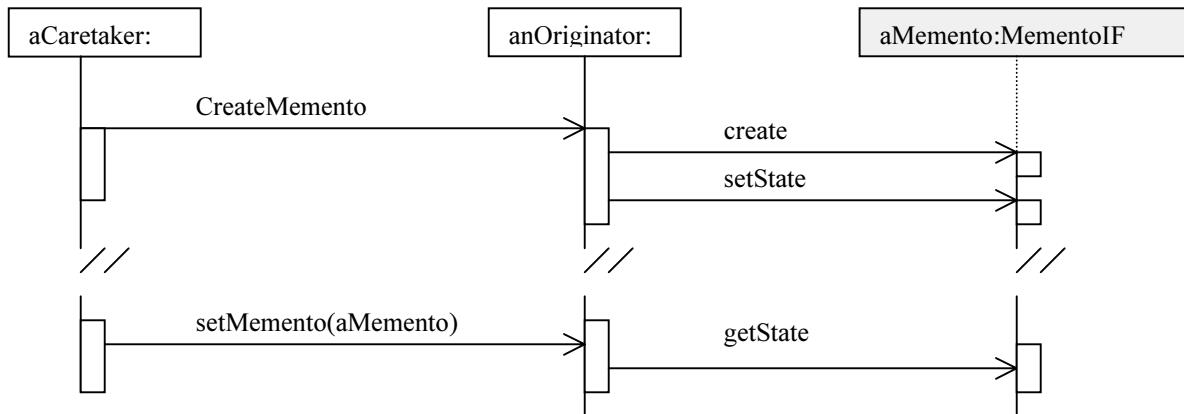
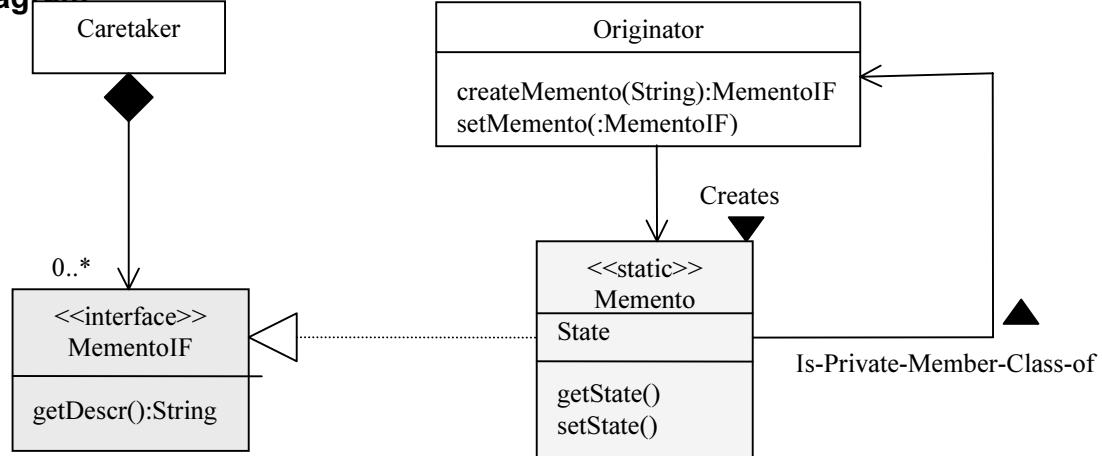
1. There are at least 8 Colleague objects in the **Mediator** dinner planning example (feel free to add others). Draw two UML dependency diagrams: one with the objects decentralized, and a second with MealDialogMediator in the center and the various instances of colleague objects around it. (Possibly combine two diagrams into one, using color overlay for the 2nd)

2. If networked computers are available, run the ChatServer and Client example java code. Otherwise, discuss the application first. Then look at the code handout and add your comments on how ChatServer uses the **Observer** pattern in a multithreaded environment. What would happen in this example if multithreading was not used?

Notes

Comments: Opaque object for holding state; can be used for iteration.

Diagram:



Participants: Memento, MementoInterface (wide for originator, narrow for other objects), Originator, Caretaker

Solution: See Code, Appendix page B-22.

19. Memento (aka: Token, Snapshot) GoF p. 283

Pattern Thumbnail: Without violating encapsulation, you want to capture and externalize the current internal state of some object for possible later restoration.



Example: An “adventure game” which takes several days to play should have a way to save current state to a file. (This is actually the Snapshot Pattern, a much larger-grained variety of Memento.) Or, perhaps you want the ability to return to some previous state of the game (not the beginning) when some dire circumstance occurs (such as your character being killed by the dwarf...).

Forces:

- Want a mechanism to save and restore internal state, but it should be independent of the internal structure of the object, which could change.
- Changes shouldn't affect the save/restore mechanism.

Question: The authors write that the “Caretaker” participant *never operates on or examines the contents of a memento*. Can you consider a case where a Caretaker would in fact need to know the identity of a memento and thus need the ability to examine or query the contents of that memento? Would this break something in the pattern?



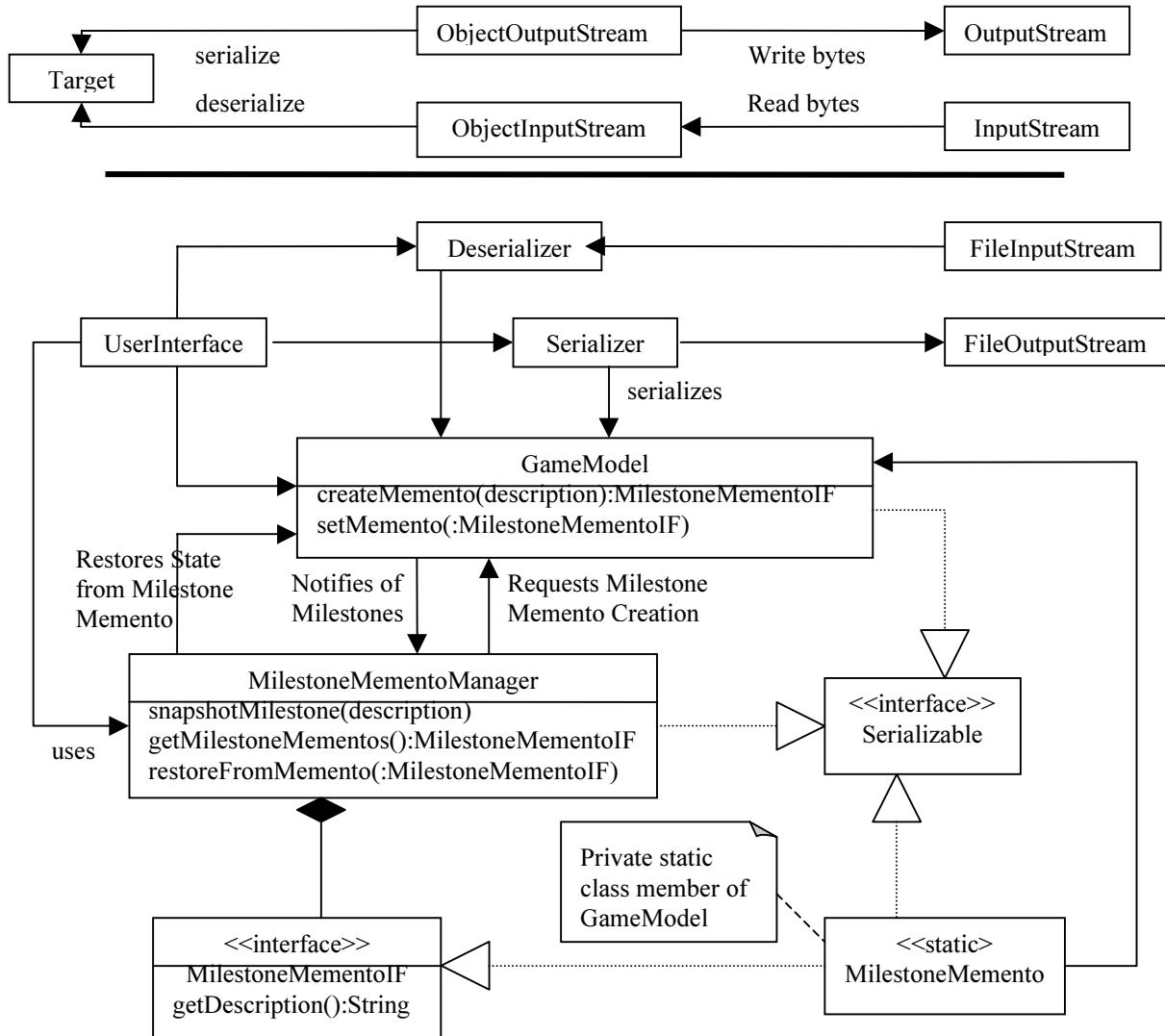
Consequences:

- You preserve encapsulation boundaries, simplify the originator class.
- The complexity of save/restore of an object is taken out of its class.
- The potential costs are lots of overhead; if undoing a fine-grained sequence of commands, for example, the Command pattern is probably more efficient.

Notes

Comments: Memento, but with Serialization. There are important differences.

Diagram: (see previous example for the simple Memento diagram)



Participants: Originator, Memento, MementoIF, Caretaker, Target, ObjectOutputStream, OutputStream, ObjectInputStream, InputStream.

Usage: Adventure Games; Rollback situations.

Solution: Another solution besides using Memento objects to store state information is to use serialization.

19a. Snapshot, a Memento variation

Pattern Thumbnail: Create a snapshot of an object's state for possible later restoration. The object initiating the capture or restoration of the state does not need to know anything about the state information; only that the object it is working with implements a particular interface. Save to and restore from a file.



Example: Saving the state of a game (such as Adventure) to allow the game to be played over a series of intervals or to allow a player to revert to some previously saved milestone when some disastrous move occurs.

Forces:

- You need to create a snapshot of an object's state and also be able to restore the state of the object.
- You want a mechanism that saves and restores an object's state to be independent of the object's internal structure, so that the internal structure can change without having to modify the save/restore mechanism.

Question: Compare saving the state of an object by Serialization (as in Snapshot) and with the use of Memento Objects only: consider persistence, preservation of encapsulation, preservation of object identity, complexity of implementation, overhead, expertise required to implement.

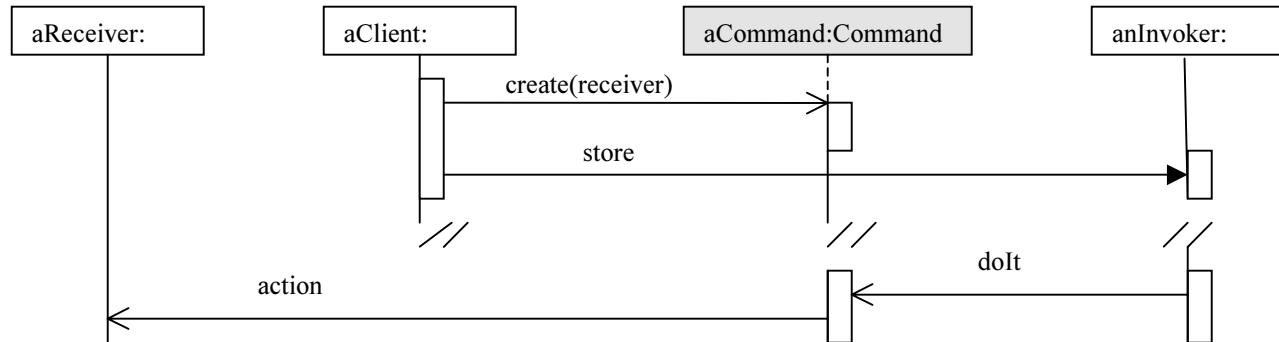
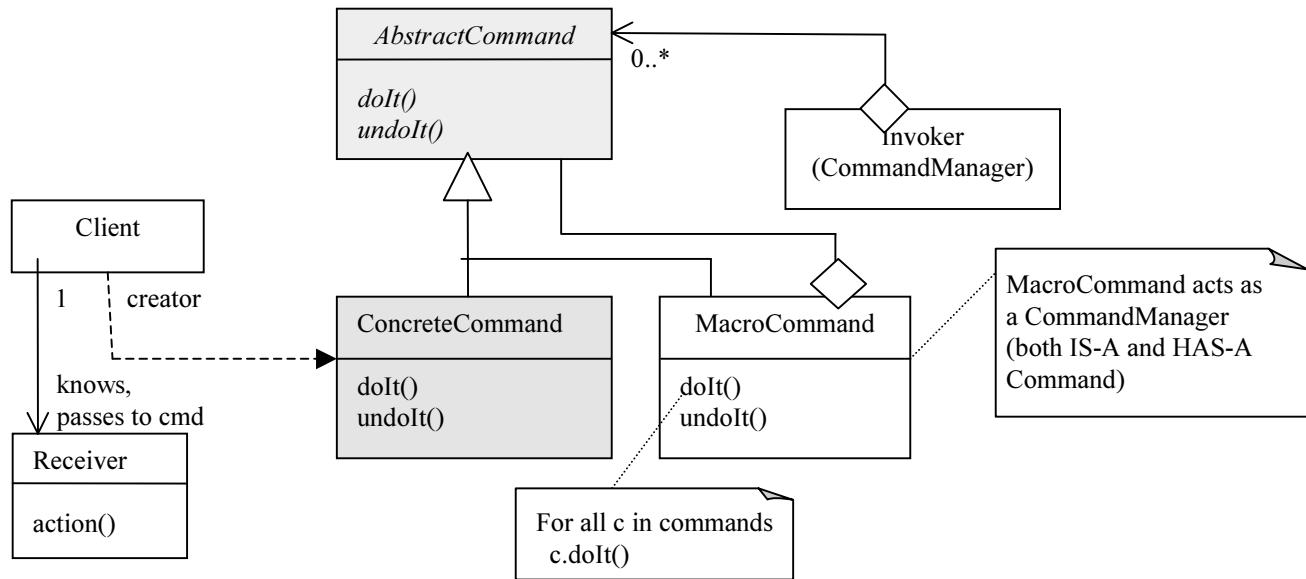
Consequences: Both forms of Snapshot keep a lot of the complexity of saving/restoring an object's state out of its class.

Notes: If you don't override the default behavior of `ObjectInputStream`, it puts the original Target object's state information in a new instance of the Target object's class. Not suitable for undoing a fine-grained sequence of commands; making many snapshots of an object can consume lots of storage – consider Command for this type of activity.

Notes

Comments: A callback object. Can return a command which undoes this one. Could replace with a closure if only one exec() method is needed.

Diagram:



Participants: AbstractCommand, ConcreteCommand, Invoker, CommandManager, Client, Receiver.

Usage: Related usage: java's Button and MenuItem classes have methods getActionCommand() and setActionCommand*(which allow you to access the associated command.

Solution: See Code, Appendix page B-10.

20. Command (aka: Action, Transaction) p. 233

Pattern Thumbnail: Parameterize commands as objects, thereby making them first-class citizens in the OO environment. Then you can queue, log, undo, redo, etc.

Example: WordProcessor commands (like Paste, Open in GoF text). Context-sensitive help: as context shifts, store a new Command instance with the Help button.

Forces:

- You want to control the sequencing, selection, timing of command executions;
- you want to support undo and redo;
- you want to maintain a persistent log of commands executed.

Question: In the Motivation section of the Command pattern, an application's menu system is described. An application has a Menu, which in turn has MenuItem s, which in turn execute commands when they are clicked.

What happens if the command needs some information about the application in order to do its job? How would the command have access to such information so that new commands could easily be written that would also have access to the information they need?



Consequences:

- Support for macros.
- Separation of invoker and executer adds flexibility in timing and sequencing.
- Easy to add new commands without having to change previous commands.
- Could use Marker Interface for undoable commands.

Notes

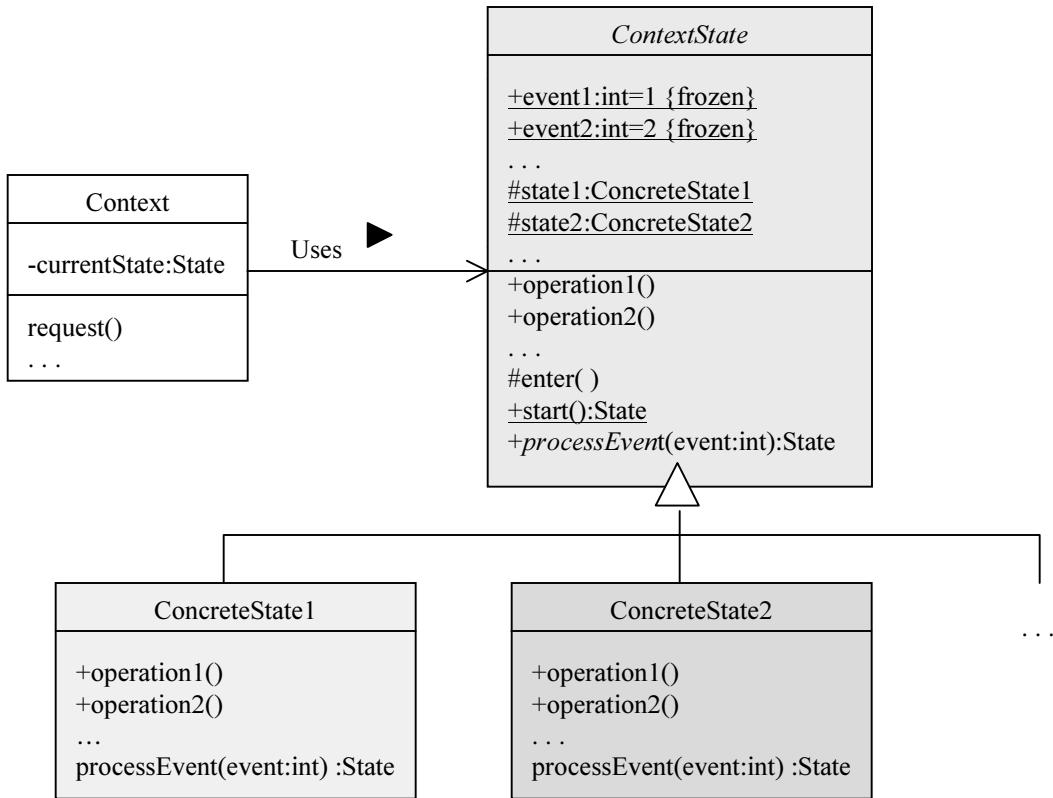
Exercises

1. How could **Memento** capture state in the midst of an iteration?
 2. How is **Snapshot** a variation of **Memento**?
 3. Examine the code on the **Command** pattern in the GoF book (p. 239+). Be prepared to present a code review on this example; be sure to also explain the **Marker** pattern to signify undoable commands.
 4. How could you use **Template Method** (instead of **Command**) to implement top-level undo logic?

Notes

Comments: Encapsulate state-dependent behavior into separate objects, each belonging to a separate subclass of an abstract state class. Switch-style forwarding.

Diagram:



Participants: Context (has-a currentState), State (abstract superclass), ConcreteState (subclasses).

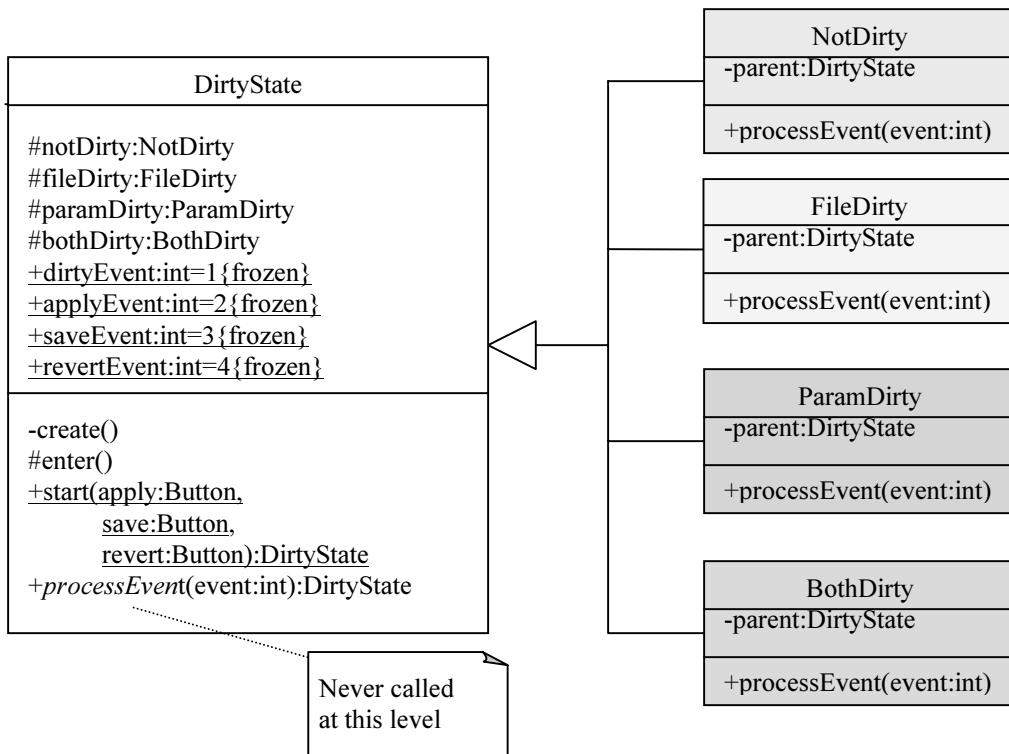
Usage: Think of interactive drawing programs where selecting a tool from the toolbar lets you perform different functions: Selection Tool, Drawing various shapes, setting Style, etc. Or the vi editor, where you are either in insert mode or command mode...

Solution: See Code, Appendix page B-29.

21. State (aka: Objects for States) GoF p. 305

Pattern Thumbnail: Objects alter their behavior when their internal state changes and the object appears to change its class.

Example: Writing a dialog for editing parameters of a program. The OK button (always enabled) saves the parameter values to a file and working memory; the Save button saves just to a file; the Apply button saves just to working memory; and the Revert button restores values from a file. Make the dialog stateful; enable buttons as appropriate.



Forces: An object's behavior is determined by an internal state that changes in response to events. The organization of logic that manages an object's state should scale up to many states without becoming complex and unmanageable.

Question: If something has only two to three states, is it overkill to use the State pattern?

Consequences:

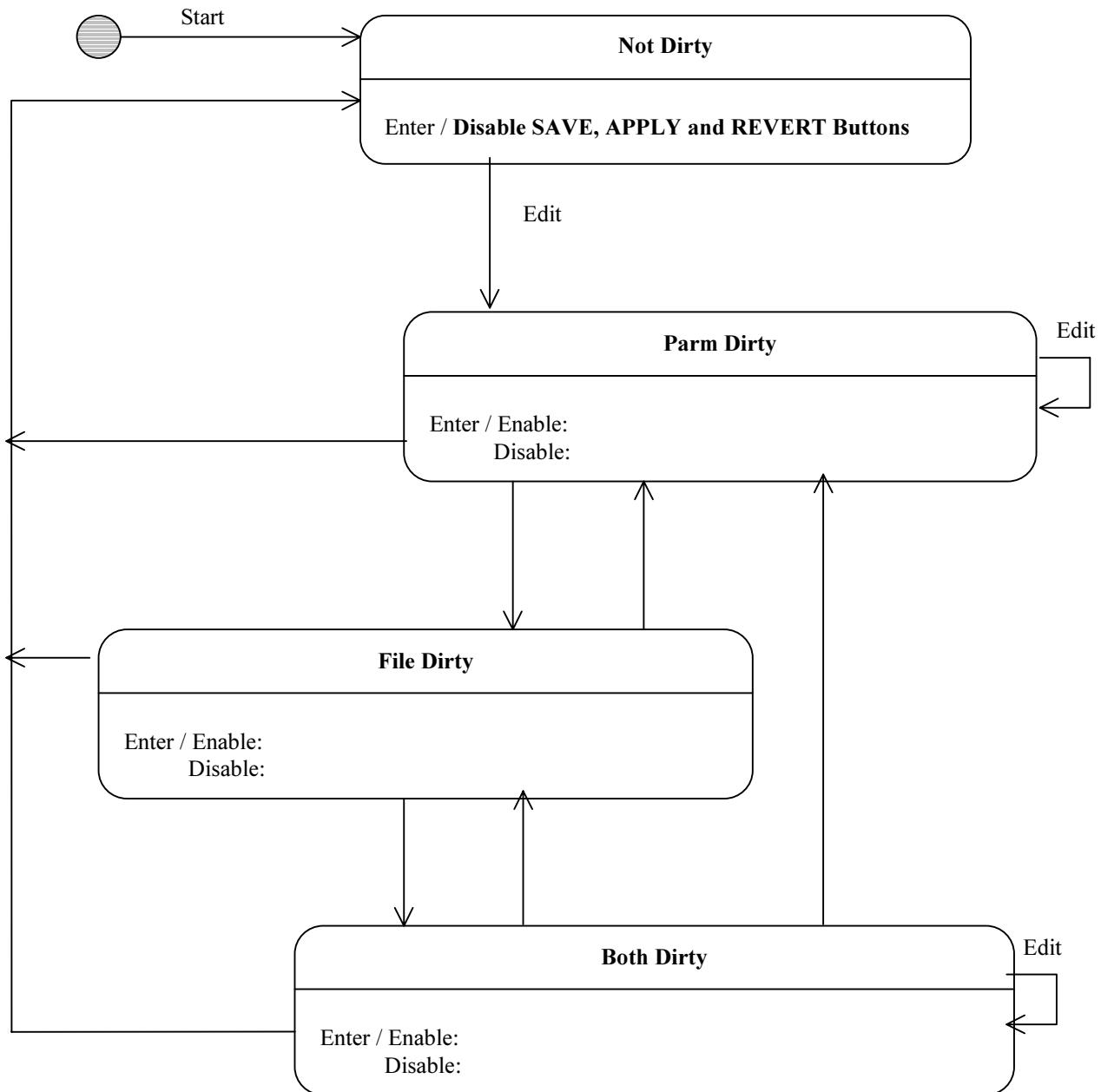
- The code for each state is in its own class, thereby localizing state-specific behavior.
- State transitions appear to be atomic to clients.
- State objects can be shared (as Singletons) and can use the Flyweight pattern.



Notes

Exercises

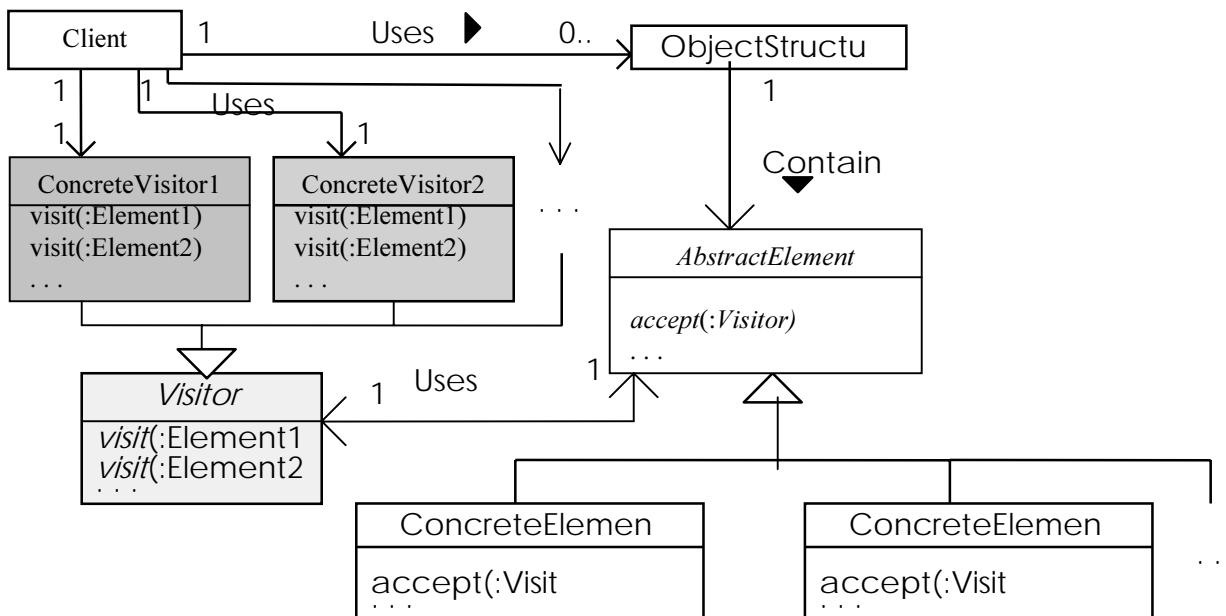
1. Complete the state diagram below for the 4 subclasses presented in the **State** pattern, showing enabled buttons in each (APPLY, SAVE, REVERT) and the button transitions which cause state changes. (Note: that means label all unlabeled lines and model the buttons enabled as has been done in Not Dirty; “edit” means parameters have been edited; **OK** button is always enabled and saves edited parm fields to working memory and to file and moves to next task.) Feel free to change lines/arrows...



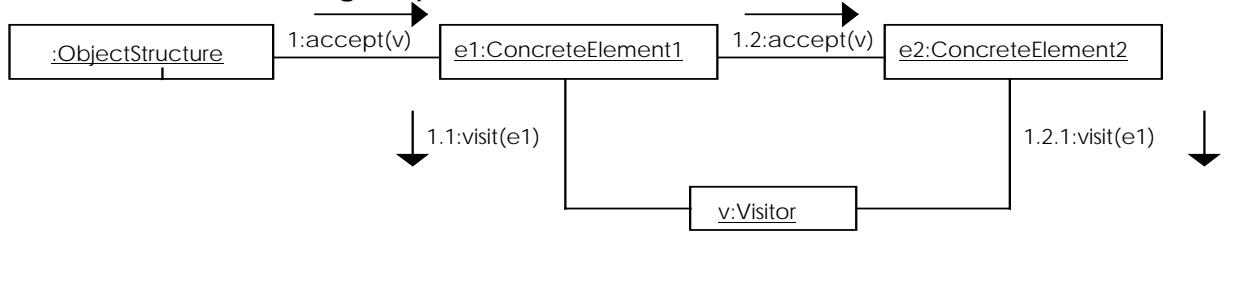
Notes

Comments: Add new operations to classes without changing the classes being operated on -- an artifact of multi-methods being missing from C++. If the object structure classes are very changeable, it is probably better to define the operations in those classes rather than use the Visitor pattern.

Diagram: (Ideal – less effort to implement and maintain if you can do it this way)



(Ideal Collaboration Diagram)

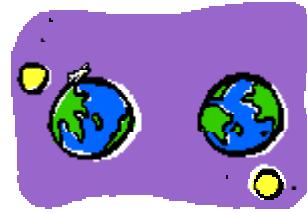


Participants: Visitor, ConcreteVisitor, Element, ConcreteElement, ObjectStructure

22. Visitor GoF p. 331

Pattern Thumbnail:

- One way to implement an operation that involves objects in a complex structure is to provide logic in *each* class of the structure to support the operation.
- Visitor provides an alternative by putting all the necessary logic in a separate Visitor class.
- Logic can also vary by using different Visitor classes.
- Therefore, you have two hierarchies – one of structure, one of operations, and do a dual dispatch.



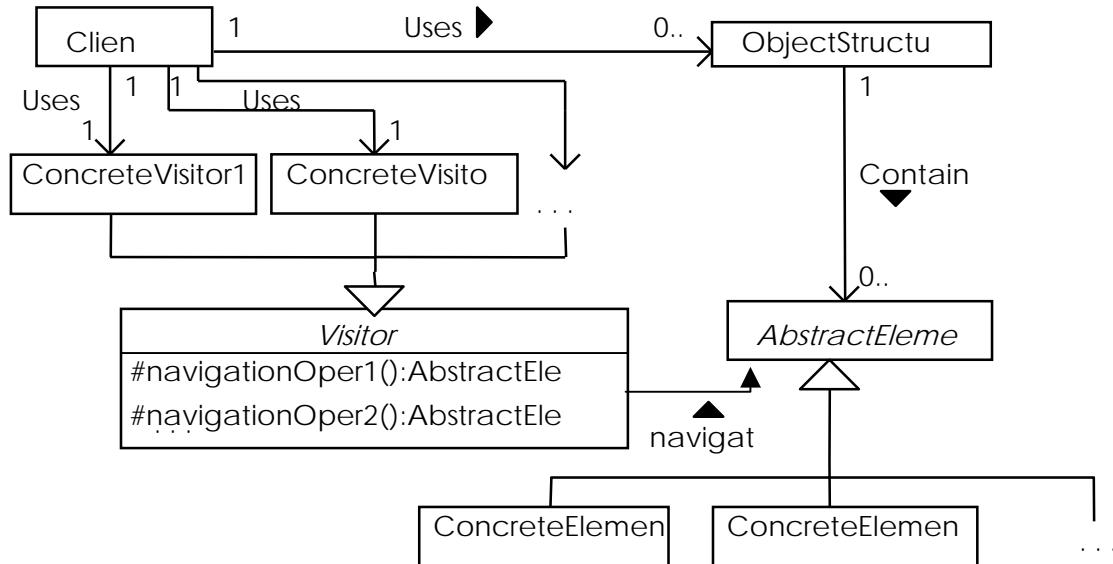
Example: (From www.ootips.org) “A transaction object is provided to the client and the client must select the right UI classes and pass the newly created transaction class to it. You know the type of transaction created (e.g., Deposit, Withdrawal) and you need to create the appropriate UI. You don’t want Transaction to have to know about UI classes, so use Visitor and pass the concrete Transaction to the UI constructor.”

Forces:

- There are a variety of operations that need to be performed on an object structure.
- The object structure is composed of objects that belong to different classes with differing interfaces, the operations you need to do depend on the concrete classes.
- The types of objects that occur in the object structure do not change often and the ways that they are connected are consistent and predictable.
- You want to avoid polluting the object structure classes with unrelated operations.

Notes

Diagram for Variation 2:



Participants: Visitor, ConcreteVisitor, Element, ConcreteElement, ObjectStructure

Usage: IRIS Inventor toolkit represents a 3-D scene as a hierarchy of nodes. Rendering a scene or mapping an input event require traversing this hierarchy in different ways. Inventor uses visitors called “actions” for rendering, event handling, searching, filing, and determining bounding boxes.

Solution: See Code, Appendix page B-33.

Visitor, continued



Variation 2 – There are many times you can't use the “ideal” version – either it won't work or it will be very inefficient. The second form works for a wider range of situations but has the expense of additional dependencies between classes.

Put as much logic as possible for navigation the object structure in the Visitor class rather than its subclasses to minimize the number of dependencies ConcreteVisitor objects have on the object structure. This will make maintenance easier.)

Question: One issue with the Visitor pattern involves cyclicity. When you add a new Visitor, you must make changes to existing code. How would you work around this possible problem? How would you compare Visitor with the Law of Demeter?

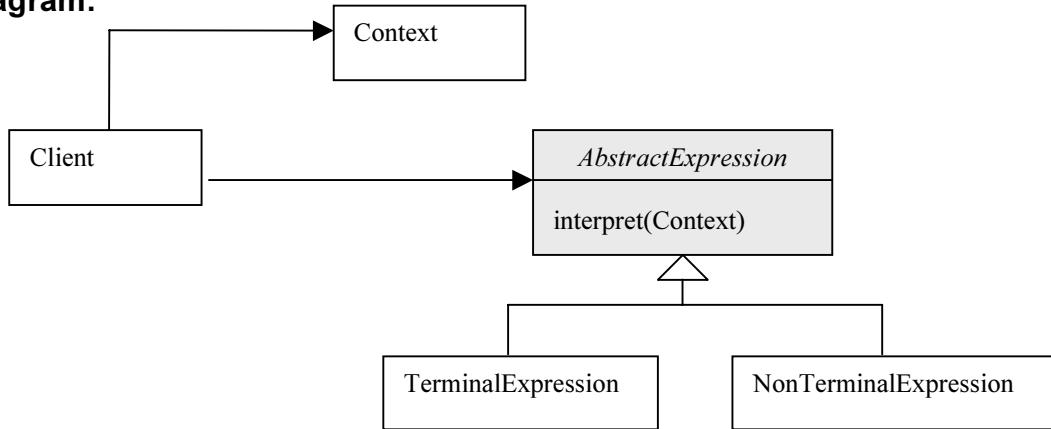
Consequences:

1. Adding new operations is easy; simply add a new Visitor.
2. A Visitor gathers related operations and separates unrelated ones, putting logic in one cohesive ConcreteVisitor class.
3. Adding new ConcreteElement classes is hard; ideal version requires you to add a new visit method to each ConcreteVisitor class for each ConcreteElement class you add.
4. Unlike Iterator, Visitor can visit objects that don't have a common parent class.
5. Visitors can accumulate state info as they visit each element in the object structure.
6. Number 5 violates good encapsulation since ConcreteElement classes must provide access to enough of their state to allow Visitor objects to perform their computations.

Notes

Comments: Stylized kind of Visitor. Distributes an operation over a hierarchy of composite patterns. Related patterns include Composite, Flyweight, Iterator, and Visitor.

Diagram:



Participants: AbstractExpression, TerminalExpression (literal), NonTerminalExpression (alternate expression), Context, Client

Usage: Compilers. Subclasses of `java.text.Format` use this pattern; constructors are passed a String which describes the format.

Solution: See Code, Appendix page B-19.

23. Interpreter (some variations aka: Little Language) GoF p. 243



Pattern Thumbnail: Given a language, define a representation for its grammar as well as an interpreter to interpret sentences in the language.

Example: A simple grammar or syntax where you can represent statements in the language as abstract syntax trees.

Forces:

- Need to identify, create, or format similar kinds of data using many different combinations of a moderate number of operations.
- A straightforward representation of combinations of operations can provide adequate performance.

Question: As the note says in Known Uses, Interpreter is most often used “in compilers implemented in object-oriented languages...” What are other uses of Interpreter and how do they differ from simply reading in a stream of data and creating some structure to represent that data?

Consequences:

- After defining an Interpreter (or Little Language) and designing the classes to implement, it is easy to change/extend the grammar, implement the grammar, and add new ways to interpret.
- Complex grammars are hard to maintain.

► **Explain** how a compiler would use this pattern. You can use a **for** loop as an example code fragment, or choose some other construct.



Notes

Exercises

Summary of GoF via Case Study (GoF Chapter 2) Designing a Document Editor

1. List the 7 problems in Lexi's (name of document editor) design (GoF p. 33):
 - (1.)
 - (2.)
 - (3.)
 - (4.)
 - (5.)
 - (6.)
 - (7.)

2. From GoF p. 38, define a **Glyph**:
What GoF pattern is obvious in the diagram?

3. From GoF p. 41, describe what the **Compositor** class does:
What GoF pattern is used here?

4. From GoF p. 43, we want to **embellish** the user interface two ways:
 - (1) add a border around the text editing area
 - (2) add scroll bars for viewingWhat GoF pattern can we use (and NOT use inheritance) for flexibility?
What other GoF pattern is shown in the figure on p. 45?

5. We want to support **multiple look-and-feel standards** (GoF p. 47-51).
What GoF pattern can accomplish this?

6. We want to also support **multiple window systems** (GoF p. 51-58).

Why can't we use an abstract factory for this? (GoF p. 52)

Separating windowing functionality into window and windowImpl hierarchies lets us have two separate hierarchies which can work together to create the needed window objects. What GoF pattern is this?

7. We want to allow **multiple user interfaces for the same operation** (buttons, menu selections, hot keys, etc.) and we also want to support undo and redo (GoF p. 58-64). What GoF pattern allows us to encapsulate a request and parameterize menu items?

8. In order to do **spell checking and hyphenation**, we will need to **access and traverse** sections of the document (GoF p. 64-70).

What GoF pattern helps here?

9. Beyond simply the access and traversal above, we need to do an **analysis of the glyphs** (GoF p. 70-76). This should be separate from the traversal as the same traversals could be used for various analyses. A given analysis must be able to distinguish different kinds of glyphs but we don't want to pollute the glyph interface with all kinds of new stuff. What GoF pattern makes sense to do this job?

10. (from GoF p. 358)

“It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound.” - Christopher Alexander, *A Pattern Language*, [p. xli]

Why do the best object-oriented designs use many design patterns that dovetail and intertwine?

What about this is difficult for us as programmers?

What about this is powerful for us as programmers?

Chapter 7 – Other Micro-Architecture Patterns

Notes

Chapter Objectives

- Learn Other Micro-Architecture Patterns

1. Object Pool
2. Dynamic Linkage
3. Cache Management
4. Type Object
5. Extension Object
6. Smart Pointers (C++)

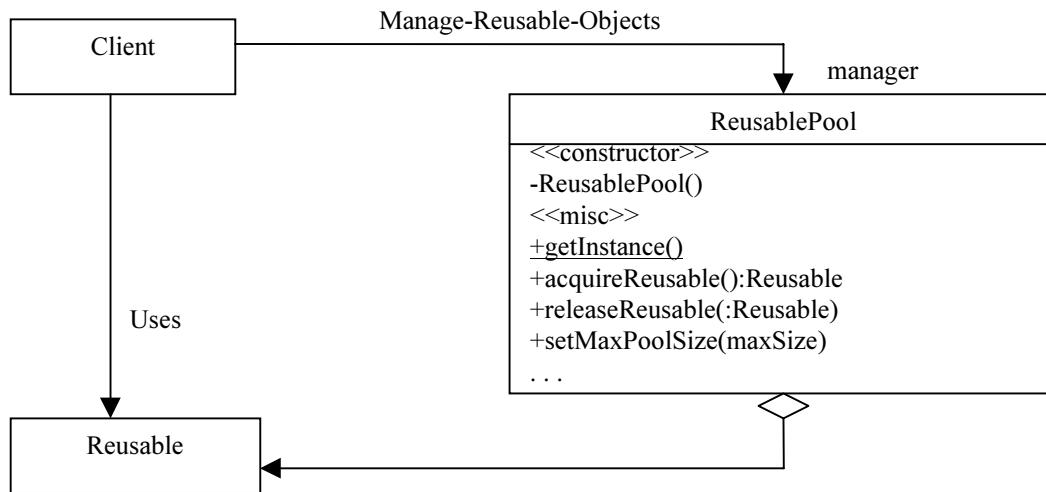


- Develop **incentive** to look for additional patterns

Notes

Comments: If instances of a class are reused, avoid creating new instances of the class. See also Cache Management pattern.

Diagram:



Participants: Reusable, Client, ReusablePool (a Singleton).

Usage: Database connections.

Solution: If instances of a class are reused, avoid creating new instances of the class. When there is a limit on the number of objects that may be created, you can use a simply array. When there is no limit on the pool size, consider a Vector.

1. Object Pool (aka: Instance Manager)

Pattern Thumbnail: Manages the reuse of objects when a type of object is expensive to create or only a limited number of objects can be created.

Example: Managing connections to a database. The connections are used by clients to send queries and retrieve results. You want to avoid having each program create its own connection: each creation takes several seconds, the more connections there are to a database the longer it takes to create new connections, and each database connection uses a network connection and some platforms limit the number of network connections allowed.

Solution: have a library manage the database connections on behalf of the applications.

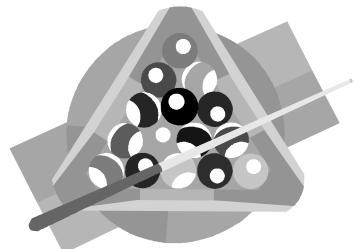
Forces:

- Program may not create more than limited number of instances for a particular class.
- If creating instances of a particular class is sufficiently more expensive, then creating new instances for that class should be avoided.
- A program can avoid creating some objects by reusing objects when it has finished with them rather than discarding them as garbage.

Question: How can you ensure that a class is instantiated only by the class managing the object pool? Suppose you don't have control over the structure of the managed class?

Consequences:

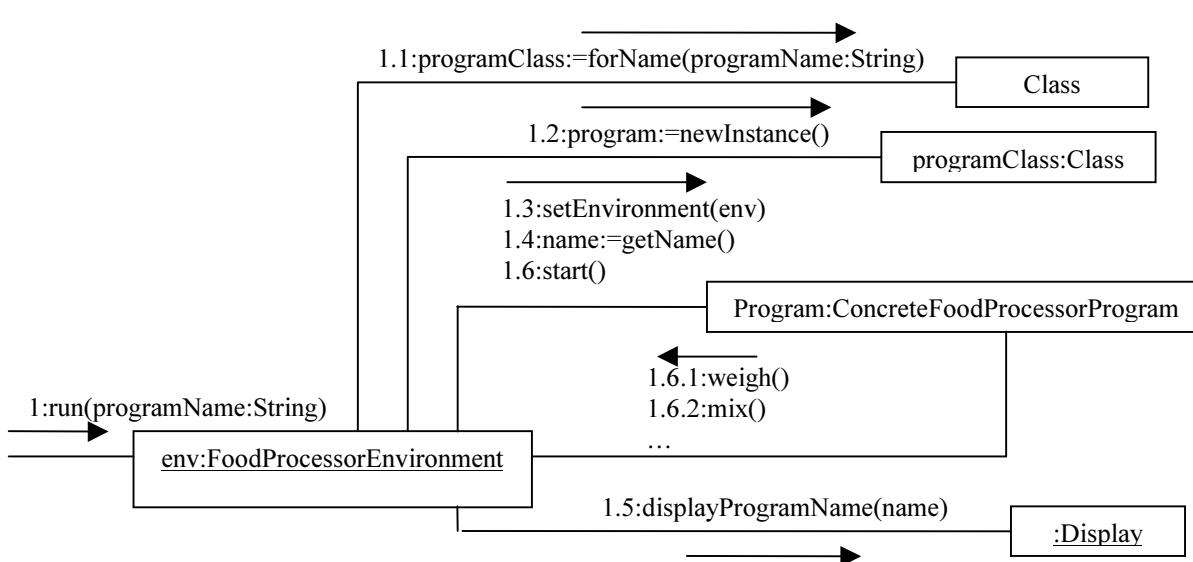
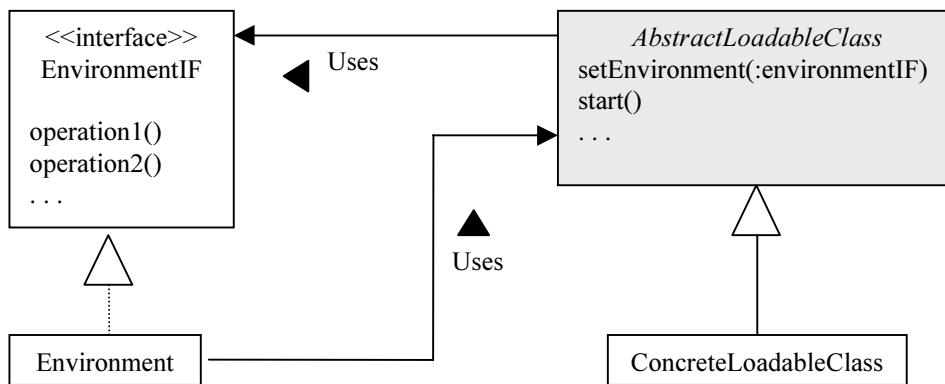
- Helps avoid object creation.
- Works best when demand for objects is fairly stable.
- Keeping the logic to manage creation/reuse of a class's instances in a separate class from the class instances being managed results in greater cohesion by eliminating interactions between implementation of a creation and reuse policy and the implementation of the managed class's functionality.



Notes

Comments: Flexibility vs. complexity trade-offs, implemented with interfaces and abstract base classes.

Diagram:



Participants: EnvironmentIF, Environment, AbstractLoadableClass, ConcreteLoadableClass

Usage: Web browsers use the Dynamic Linkage pattern to run Java applets. The browser environment accesses a subclass of Applet that it loads through the Applet class; loaded applet subclasses access the browser environment through the AppletStub interface.

2. Dynamic Linkage

Pattern Thumbnail: Allow a program, upon request, to load and use arbitrary classes that implement a known interface.

Example: A smart food processor program. This piece of equipment can load various programs to produce a great variety of foods – from baking bread to stir-fry shrimp. Due to size and variety not all the programs can be kept in memory, but are loaded from a CD-ROM. Methods need a way to call each other. (See facing diagram.)

Forces:

- A program must be able to load and use arbitrary classes that it has no prior knowledge of.
- An instance of a loaded class must be able to call back to the program that loaded it.

Question: How could Dynamic Linkage be used by Virtual Proxy to create its underlying object?



Implementation: This pattern requires that the environment know about the *AbstractLoadableClass* class and that the loaded class know about the *EnvironmentIF* interface. When less structure is needed other mechanisms for interoperation are possible – JavaBeans uses a combination of reflection classes and naming conventions to allow other classes to infer how to interact with a bean.

Another requirement of this pattern is that the Environment class knows the name of a class that it wants to load. In the example above the CD-ROM could contain a directory of programs; these could be displayed as a menu, allowing user selection. Other applications might hardwire names.

To avoid problems with incompatible versions of supporting classes, ensure that all supporting classes implicitly dynamically loaded along with an explicitly dynamically loaded class are not used by any other explicitly dynamically loaded class. Implement this strategy by using a different ClassLoader object for each dynamically loaded class.

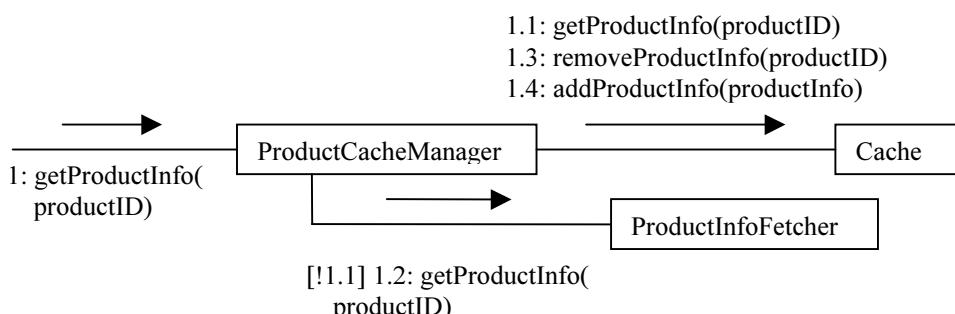
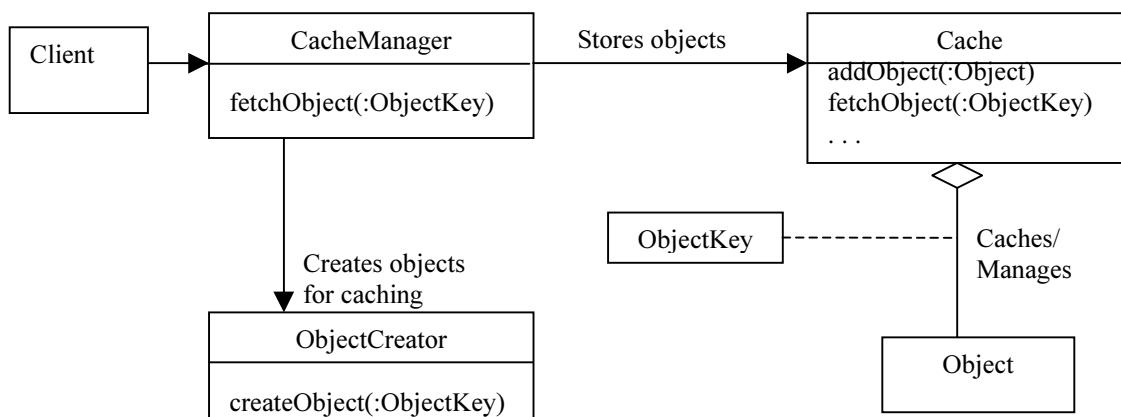
Consequences:

- Subclasses of the *AbstractLoadableClass* can be dynamically loaded.
- The operating environment and the loaded classes do not need any specific foreknowledge of each other.
- Dynamic linkage increases the total amount of time it takes for a program to load all of the classes that it uses. However, it does have the effect of spreading out, over time, the overhead of loading, which can make an interactive program seem more responsive.
- Note that the Environment class must know the name of a class that it wants to load.

Notes

Comments: Effectiveness of caching is measured by *hit rate*, the percentage of object fetch requests the cache manager can satisfy. Making optimal choices on which objects to store can involve statistical analysis, queuing theory and other mathematical analysis. Searching should be optimized for speed over addition/removal, but adding or removing objects should not be a lot more expensive than searching. (Consider a hash table.)

Diagram:



Participants: ObjectKey, CacheManager, ObjectCreator, Cache

Usage: Caching of server objects in CORBA architecture. Each server pool can be managed by an instance manager, which understands the requirements of an object and provides the appropriate run-time environment. The instance manager should be able to manage an object's state on transaction boundaries and provide load-balancing as well as smart activation services with object caching.

3. Cache Management

Pattern Thumbnail: Storing objects in working memory for fast access.

Example: Suppose you are writing a program for fetching information about products in a catalog. Fetching all the information for a product can take a few seconds as it is gathered from multiple sources. Once the info for a product is secured, you want to have it available for quick access if it is needed again soon.



Forces:

- There is a need for access to an object that takes a long time to construct.
- When the number of such objects is small enough to fit in local memory, keeping them there will provide the best results.
- It may be necessary to set an upper bound on the number of cached objects with an enforcement policy which determines who stays and who goes.

Question: How could you use Publish-Subscribe to ensure the read consistency of a cache? How could you use the Template Method pattern to keep the Cache class reusable across application domains?

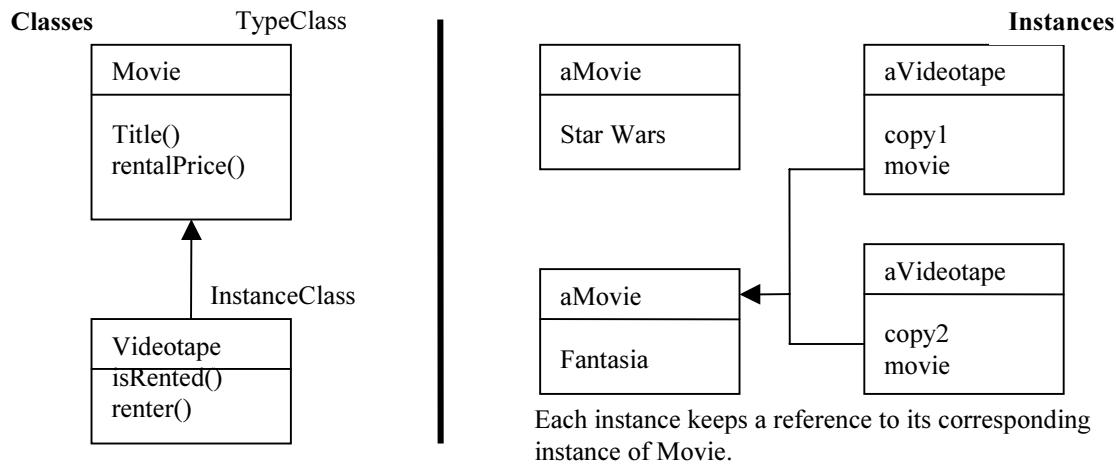
Consequences:

- The impact of Cache Management on the rest of a program is minimal.
- Positive: A program spends less time creating expensive objects.
- Negative: The cache may become inconsistent with the original data source:
read consistency means the objects fetched from cache always reflect updates to information in the original;
write consistency means the original object source always reflects updates to the cache.
- Some synchronization mechanism is required to achieve **absolute consistency**.
Relative consistency may suffice; its guarantee is that if an update occurs in the cache or the original data source, the other will reflect the update within some specified amount of time.

Notes

Comments: A class requires an unknown number of subclasses in addition to an unknown number of instances; you want to be able to create new subclasses without recompilation.

Diagram:



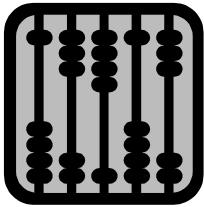
Participants: TypeClass, TypeObject, InstanceClass, InstanceObject

Usage: Java and other reflective systems – a type object is often called a metaobject. Used to model medical samples where each sample has 4 independent properties.

Implementation Issues:

1. InstanceObject references TypeObject – must be specified when instance is created.
2. An object's behavior can be implemented in its class or delegated to its TypeObject.
3. The messages an object understands are defined by its class, not its TypeObject.
4. A new InstanceObject is created by sending a request to the appropriate TypeObject instance (different).
5. An object could have multiple TypeObjects, but this is not common.
6. An object could dynamically change class (easier to change the reference than to mutate to new class).
7. It is possible to subclass either the Instanceclass or TypeClass – e.g., a Videodisk instance could reference same movie as a Videotape instance; 3 videotapes and 2 videodisks could all share the same movie.

4. Type Object (aka: Power Type)



Pattern Thumbnail: Replacing an entire class hierarchy with just two classes – a class for the type and a class for the instance.

Example: A video store where there are multiple copies of tapes. You don't want to store redundant information for each copy (instance). The total number of videotapes is not known ahead and changes often anyway.

Forces:

- Instances of a class need to be grouped together according to their common attributes and/or behavior.
- The class needs a subclass for each group to implement that group's common attributes/behavior.
- The class requires a large number of subclasses and/or the total variety of subclasses that may be required is unknown.
- You want to be able to create new groupings at runtime that were not predicted during design.
- You want to be able to change an object's subclass after it has been instantiated without having to mutate it to a new class.
- You want to be able to nest groupings recursively so that a group is itself an item in another group.

Question: How would you compare Type Object pattern to Strategy and State patterns? Decorator?

Consequences:

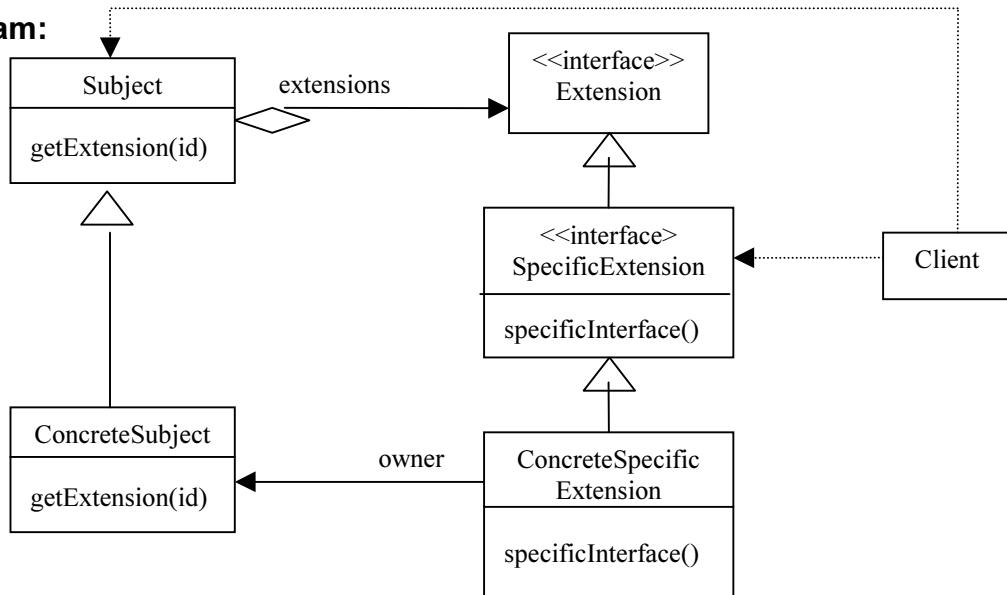
Positive: Runtime class creation, avoiding subclass explosion, hiding separation of instance and type, dynamic type changes (NewRelease can change to GeneralRelease), independent subclassing, multiple type objects.

Negative: Design complexity, implementation complexity, reference management is required by the application (usually each instance knows what type it is inherently).

Notes

Comments: Anticipate extensions to an object's interface.

Diagram:



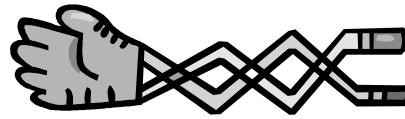
Participants: Subject, Extension, ConcreteSubject, SpecificExtension, ConcreteSpecificExtension.

Usage: The QueryInterface in COM (Component Object Model); interfaces behave as extensions.

Implementation Issues:

1. Static vs. dynamic extension objects.
2. Specifying extensions – clients must specify uniquely (in Java, use class literal expressions)
3. Demand loading of extensions.
4. Defining SpecificExtensions interfaces.
5. Freeing extensions in non-garbage collecting environments.

5. Extension Object (aka: Facet)



Pattern Thumbnail: Extension Object lets you add interfaces to a class, and lets clients choose and access the interfaces they need.

Example: A framework for compound documents made up of components such as text, graphics, spreadsheets, movies. To assemble components in various interesting ways there is a need for a common interface that provides operations to manage and arrange components. You want to add a spell checker --textless components would "do-nothing." Adding an interface to Component creates bloat. Therefore, define the spell check interface in a separate abstract class

Forces:

- You want to add new or unforeseen interfaces to existing classes and you don't want to impact clients that don't need this new interface.
- Clients perceive different roles for the same abstraction and the number of such roles is open-ended.
- A class should be extensible without being subclassed directly.

Question: Visitor and Decorator also address the problem of extending class functionality; how does Extension Object differ from these?

Consequences:

Positive: Facilitates adding interfaces, can prevent bloated interfaces, clients can perceive an abstraction differently.

Negative: Clients become more complex, the Subject interface doesn't express all of its behavior.

Notes

Comments: Smart pointers are objects that look and feel like pointers, but are smarter. They can address many memory management issues.

Code Example:

```
template <class T> class auto_ptr {
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() {delete ptr;}
    T& operator* () {return *ptr;}
    T* operator->() {return ptr;}
    // ...
};
```

Dangling pointer illustration:

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething();      // p dangling
p = NULL;               // p no longer dangling
q->DoSomething(); // Oops! q still dangling
```

For the user of auto_ptr, this means instead of:

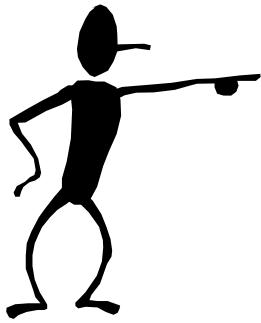
```
void foo() {
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

You can write:

```
void foo() {
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
} // and trust p to cleanup after itself.
```

```
template <class T> auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs) {
    if (this != &rhs) {
        delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = NULL;
        // set pointer to NULL on copy
    }
    return *this;
};
```

Usage: The Smart Pointer Library includes at least five smart pointer class templates:
see www.boost.org/libs/smart_ptr/



6. Smart Pointer (C++)

Pattern Thumbnail: Using **Proxy**, smart pointers implement the same interface as regular pointers. The key is the overload of the dereferencing operator. By wrapping an instance of a class in a template class, and returning the address of the wrapped instance from the dereferencing operator, you can still access any of the wrapped class's members.

Question: What happens in the first foo() on the facing page if DoSomething() throws an exception?



Implementation Issues: Must support all pointer operations, like dereferencing (operator *) and indirection (operator ->).

Consequences:

Positive: Pointers which handle common C++ bugs can save a lot of aspirin. Smart pointers can mean fewer bugs in programs and can do such tasks as automatic initialization, removal of dangling pointers (pointers to objects already deleted), creating new copies, transfer of ownership, reference counting, reference linking, copy-on-write, and garbage collection.

Negative: A bit more work, but probably worth it.

Notes

Exercises

1. Consider a factory with many different machines manufacturing many different products. Every order has to specify the kinds of products it requires; each kind of product has a list of parts and a list of the kinds of machines needed to make it. Instead of class hierarchies for kinds of machines and kinds of products (which would require programming every time you added a new kind of machine or product), how could you use **Type Objects**?
 2. Suppose a video store client wanted to browse all of the movies the store offers. How might you accomplish this without iterating through all of the videotapes (which include copy1, copy2, etc.)?

3. Draw the UML diagram for the **Extension Object** example of adding a SpellChecker interface. Consider classes such as Component, StandardTextComponent (or HTMLTextComponent, or both), ComponentExtension, TextAccessor, StandardTextAccessor (or HTMLTextAccessor, or both).

Chapter 8 – Concurrency Patterns

Notes

Chapter Objectives

- Create awareness of **special problems** in concurrent environments
- Examine several **patterns for concurrent programming**

Notes

Concurrency Patterns make sense to familiarize yourself with if you work in a multi-threaded environment.

(See *Concurrent Programming in Java, Second Edition*, by Doug Lea.)

Concurrency Patterns Overview

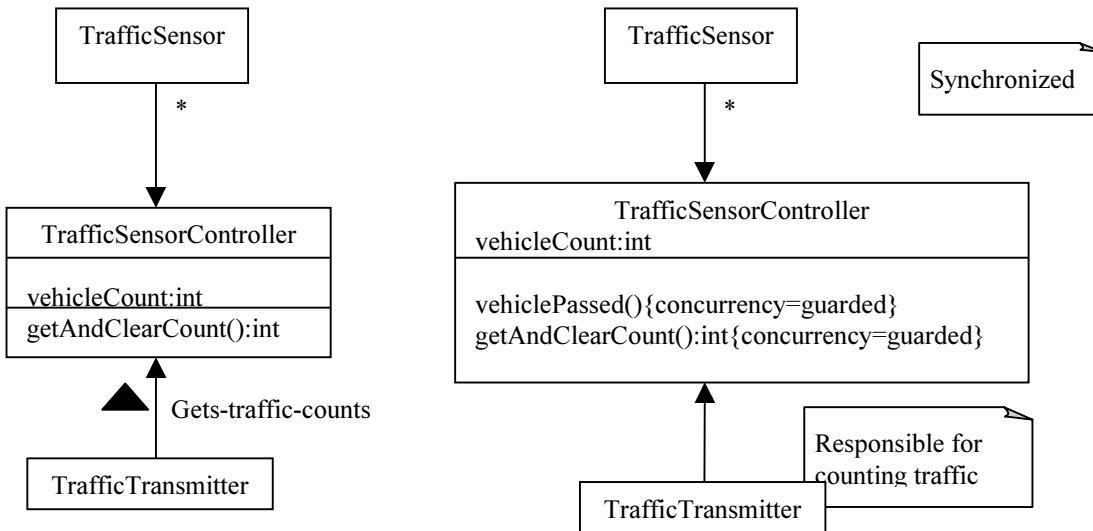
1. Single Threaded Execution
2. Guarded Suspension
3. Balking
4. Scheduler
5. Read/Write Lock
6. Producer/Consumer
7. Two-Phase Termination
8. Double-Checked Locking (note problems)



Notes

Comments: Some methods access data or other shared resources in a way that produces incorrect results if there are concurrent calls to a method and both calls access at the same time.

Diagram:



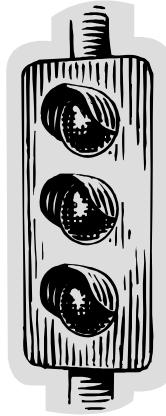
Participants: Resource.

Usage: Many of the methods of `java.util.Vector` are synchronized to ensure single threaded access to the internal data structures of `Vector` objects.

Implementation: Guarded methods are implemented in Java by declaring methods to be synchronized. If a synchronized method calls other methods, it is possible to optimize by having the called methods NOT be synchronized ("synchronization factoring"). However, if modifications allow concurrent calls to those unsynchronized methods, the program stops working correctly.

1. Single Threaded Execution (aka Critical Section)

Pattern Thumbnail: Prevents concurrent calls to a method from resulting in concurrent executions of the method.



Example: Monitoring flow of traffic on a major highway. Sensors monitor flow and send information to a central computer that controls electronic signs near interchanges. If it is possible for more than one TrafficSensor object to call vehiclePassed() at the same time, incorrect results occur.

Forces:

- class implements methods that update or set instance or class variables.
- class implements methods that manipulate external resources in a way that will work correctly only if the methods are executed by one thread at a time.
- The class's methods called concurrently by different threads.

Question: Describe various sequences of calls to show how the example facing (left) gives incorrect results.

Consequences:

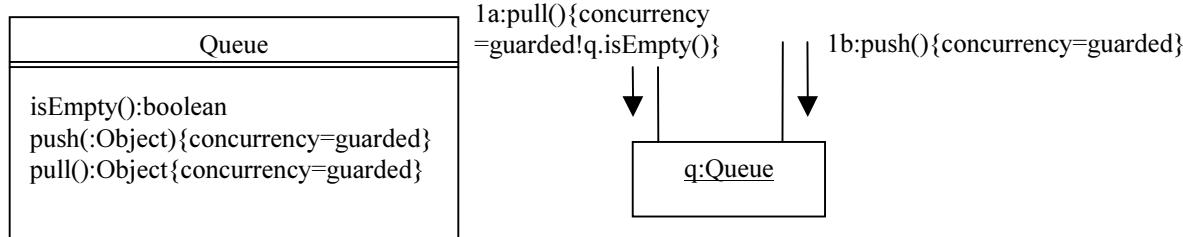
Positive: Guarded methods become thread-safe.

Negative: Overhead. Deadlock can occur when one synchronized thread calls another and each has exclusive use of a resource. Deadlock can involve more than two threads.

Notes

Comments: Use wait / notify methods and preconditions for safe concurrent calls.

Diagram:



```

public class Queue {
    private Vector data = new Vector();
    synchronized public void put (Object obj) {
        data.addElement(obj);
        notify();
    }
    synchronized public Object get() {
        while (data.size()==0){
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        Object obj = data.elementAt(0);
        data.removeElementAt(0);
        return obj;
    }
}
  
```

Participants: Two synchronized methods.

Usage: The QueryInterface in COM (Component Object Model); interfaces behave as extensions.

Implementation Issues:

If more than one thread is waiting, the notify method chooses one arbitrarily. notifyAll() notifies all waiting threads, but may wake up waiting threads whose preconditions have not been satisfied. See also Scheduler Pattern.

2. Guarded Suspension

Pattern Thumbnail: Suspends execution of a synchronized method call until a precondition is satisfied.

Example: A queue. It would be a mistake to perform pull() on an empty queue (or try to push an object onto a full queue unless there is some automatic resizing mechanism).



Forces:

- A class's methods must be synchronized to allow safe concurrent calls to them.
- An object may be in a state that makes it impossible for one of its synchronized methods to execute to completion.
- In order for the object to leave that state, a call to one of the object's other synchronized methods must execute.
- If a call to the first method is allowed to proceed while the object is in that state, deadlock will occur.

Question: Visitor and Decorator also address the problem of extending class functionality; how does Extension Object differ from these?

Consequences:

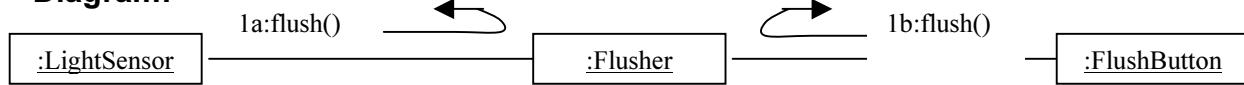
Positive: Allows a thread to cope with an object that is in the wrong state to perform an operation.

Negative: Note: Nested synchronization makes it difficult or impossible to use this pattern.

Notes

Comments: Collaboration and when to ignore a method call.

Diagram:



```

public class Flusher {
    private boolean flushInProgress = false;

    public void flush() {
        synchronized (this) { // synchronized – why?
            if (flushInProgress) return;
            flushInProgress = true;
        }
        // code to start a flush

        private void flushCompleted() { //NOT a synchronized method. Why?
            flushInProgress = false;
        }
    }
}

```



Implementation Issues:

1. If a method can balk, check state of object to determine if it should balk. (It may be possible for the object's state to change to an inappropriate state for a balking method to run while that method is running. In this case, consider Single Threaded Execution.)
2. Instead of telling its callers that it balked by passing a return value, you could throw an exception. If callers don't care, don't return anything.

3. Balking



Pattern Thumbnail: If an object's method is called when the object is not in an appropriate state to execute that method, have the method return without doing anything.

Example: An electronic toilet flusher program (see facing page). The flusher has a light sensor mounted on its front – when light level increases, it triggers a flush. The flusher also has a button which manually triggers a flush.

Forces:

- An object may be in a state in which it is inappropriate to execute a method call.
- Postponing execution of the call until the object is in an appropriate state is not a good solution, and calls made when the object is not in an appropriate state to execute the method may safely be ignored.

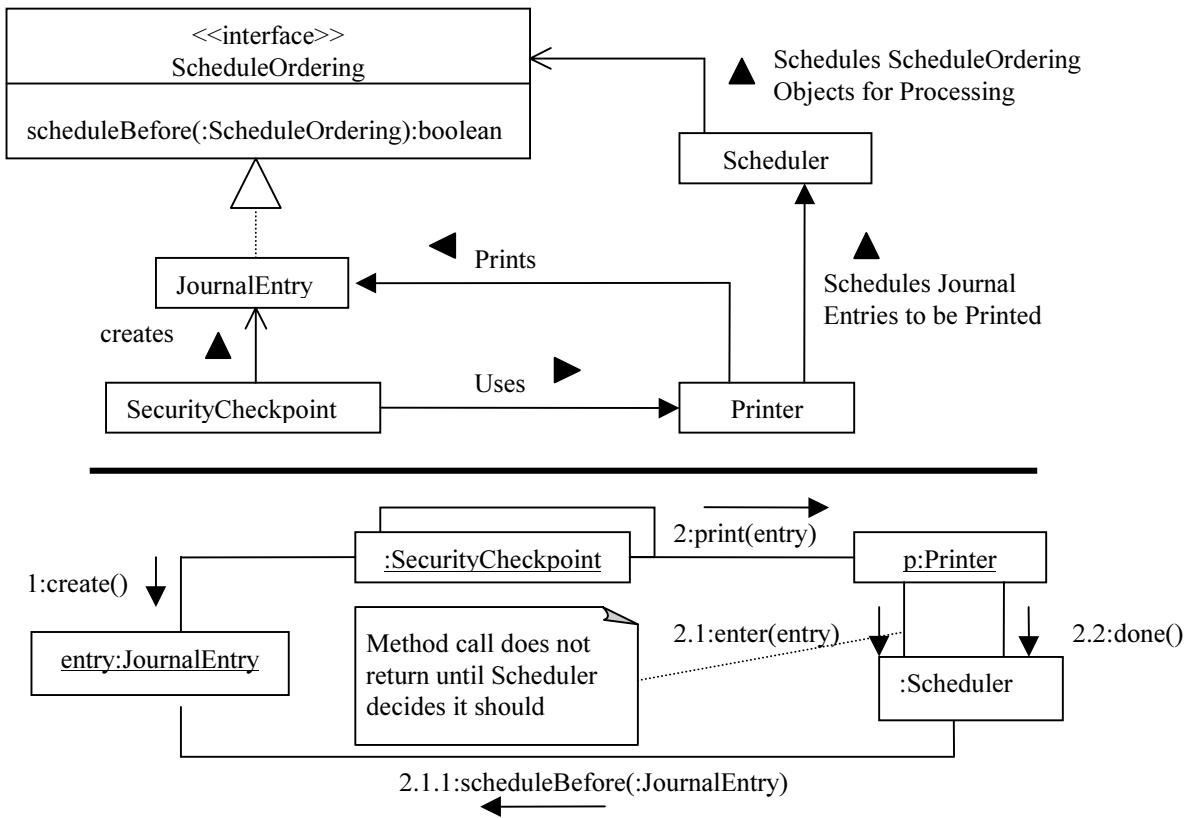
Consequences:

Positive: Method calls are not executed if they are made when their object is in an inappropriate state. Calling a method that can balk means the method may do nothing instead of its usual action.

Notes

Comments: A mechanism for implementing a scheduling policy; it is independent of any specific scheduling policy.

Diagram:



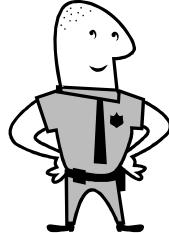
Participants: Request (JournalEntry), Processor (Printer), Scheduler, ScheduleOrdering.

Usage: The Read/Write Lock pattern often uses the Scheduler pattern to ensure fairness in scheduling.

Implementation Issues: If the policy allows calls to `enter()` to return in order, there is no need to pass Request objects into `enter()` or to have a `ScheduleOrdering` interface. Another policy might not care about order but could require a time period between the end of one task and the beginning of another.

4. Scheduler

Pattern Thumbnail: Controls the order in which threads are scheduled to execute single threaded code using an object that explicitly sequences waiting threads..



Example: A security system that supports checkpoints for badges through a scanner and prints corresponding entries on a log. When people go through several checkpoints at about the same time, the system needs to guarantee entries are logged in the same order as they were sent to the printer. The sequential queuing logic should be reusable.

Forces:

- Multiple threads may need to access a resource at the same time and only one thread at a time may access it.
- Program requirements imply constraints on the order in which threads should access the resource.

Consequences:

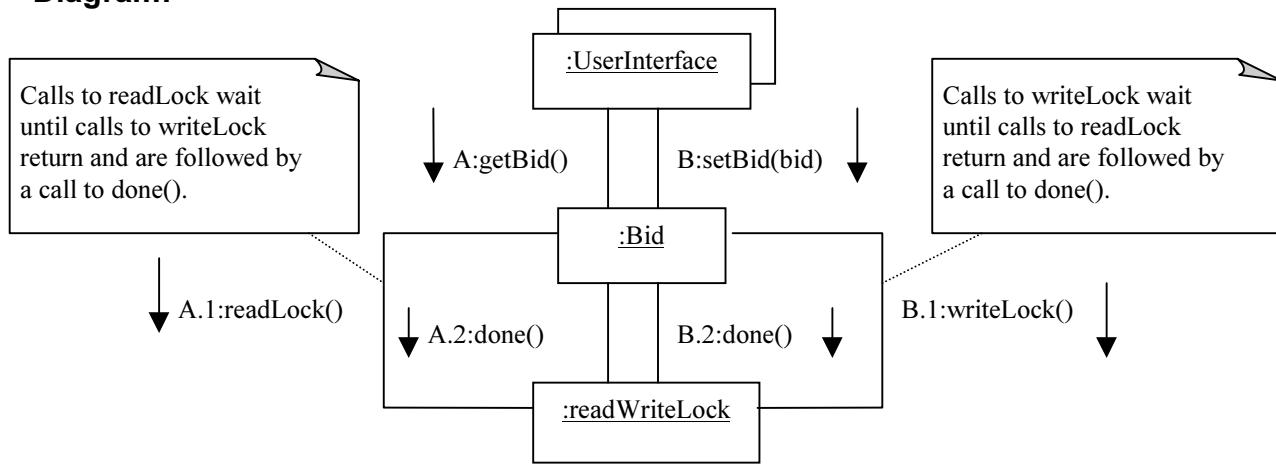
Positive: A way to explicitly control when threads may execute a piece of code. The scheduling policy is encapsulated in its own class and is reusable.

Negative: Using this adds significant overhead beyond synchronized method calls.

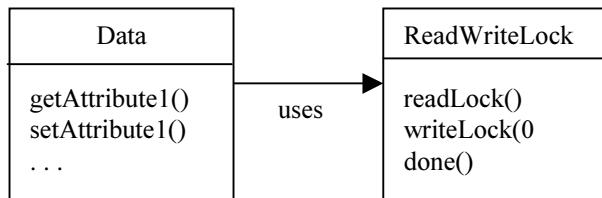
Notes

Comments: A specialized form of the Scheduler Pattern.

Diagram:



Participants: `ReadWriteLockObject` (encapsulates the coordinating logic), `DataObject` (`Bid`).



Implementation Issues: The `done()` method considers 3 cases:

- (1) there are outstanding read locks, which implies no outstanding write lock;
- (2) there is an outstanding write lock,
- (3) there are no outstanding locks -- throw exception. Use `notifyAll()` in `done()` to guarantee a write thread will wake up.

Note that because read locks and write locks do not contain any information, there is no need to represent them as explicit objects; it is sufficient to just count them.

5. Read/Write Lock



Pattern Thumbnail: Allow concurrent read access to an object but require exclusive access for write operations.

Example: A software system for conducting online auctions where there are more requests to read the current bid for an item than to update it with a new bid. The auction closes at some predetermined time.

Forces:

- You need to read and write to an object's state information.
- Any number of read operations may be performed concurrently as long as there are no write operations executing at the same time.
- Write operations should be performed one at a time.
- Allowing concurrent read operations will improve responsiveness and throughput.
- The logic for coordinating read/write operations should be reusable.

Consequences:

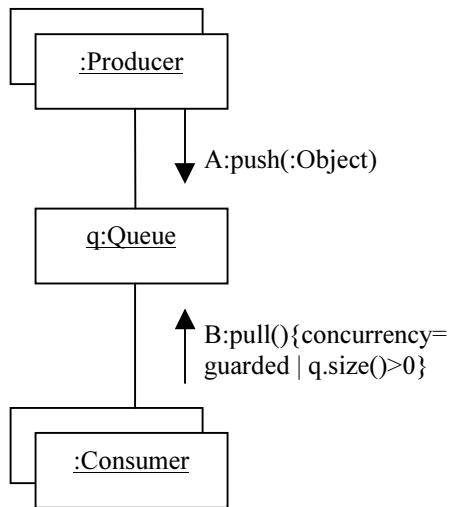
Positive: Coordinates concurrent calls to an object's get and set methods so that set calls do not interfere with each other or with calls to get methods. If there are many concurrent calls to an object's get methods, using this pattern can result in better responsiveness and throughput than Single Threaded Execution because the get methods can execute concurrently.

Negative: If there are relatively few concurrent calls to an object's get methods, this pattern results in lower throughput than Single Threaded Execution because more time is spent managing individual calls.

Notes

Comments: Think of trying to pop an empty stack or push a full stack.

Diagram:



```

public class Queue {
    private ArrayList data = new ArrayList();
    synchronized public void push(TroubleTicket tkt) {
        data.add(tkt);
        notify();
    }
    synchronized public TroubleTicket pull() {
        while (data.size() == 0) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        TroubleTicket tkt = (TroubleTicket)data.get(0);
        data.remove(0);
        return tkt;
    }
    public int size() {
        return data.size();
    }
}
  
```

Participants: Producer, Queue, Consumer.



Usage: `java.io.PipedInputStream` and `java.io.PipedOutputStream`. Together these two streams implement a variant of this pattern called the Pipe pattern. The Pipe pattern involves only one Producer object (data source) and only one Consumer object (data sink). The two classes jointly fill the role of the Queue class, allowing one thread to write a stream of bytes to one other thread asynchronously (unless the internal buffer they use is empty or full).

Implementation Issues: Whether or not the queue is limited.

6. Producer/Consumer



Pattern Thumbnail: Coordinates the asynchronous production and consumption of information.

Example: Customers entering trouble tickets through web pages. Dispatchers review the tickets and forward them. (Assume the queue is not limited here...)

Forces:

- Objects are produced or received asynchronously of their use or consumption.
- When an object is received or produced, there may not be any object available to use or consume it.

Question: How is this pattern related to yet different from Scheduler?

Consequences:

- Producer objects are able to deliver the objects they produce to a Queue object without having to wait for a Consumer object.
- When there are objects in the Queue object, Consumers are able to pull an object out of the queue without waiting. (However, when the queue is empty, the pull method must wait().)

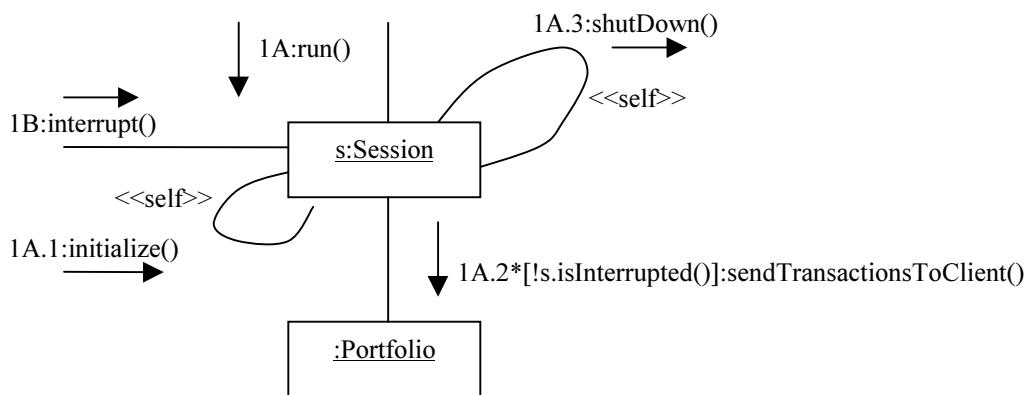


Related Patterns: Guarded Suspension, Pipe (special case that involves only one Producer object – data source - and one Consumer object – data sink), Scheduler

Notes

Comments: In Java, other Thread methods, such as sleep() and wait(), put a thread in a wait state. To support the shutdown of a process, when a thread dies, if there are no threads still alive that are not daemon threads, then the process shuts down.

Diagram:



```
public void run() {  
    initialize(); // private method  
    while (!myThread.interrupted()) { // this is the latch  
        portfolio.sendTransactionsToClient(mySocket);  
    }  
    shutdown(); //private method  
} // run()
```

Usage: Java's Thread class provides `interrupt()` and `isInterrupted()`. Note that `stop()` has been deprecated—for good reason—and this pattern is the suggested method for terminating a Thread.

Implementation Issues: If there is any uncertainty that the thread or process will actually terminate after the request to do so, it should be forcibly terminated after some predetermined amount of time. Methods that set a termination latch do not need to be synchronized.

7. Two-Phase Termination

Pattern Thumbnail: Provides for the orderly shutdown of a thread or process through the setting of a latch. The thread or process checks the value of the latch at strategic points in its execution.



Example: A server that provides middle-tier logic for a stock-trading workstation. A client connects to the server to obtain current stock prices, etc. The server has a thread for each client served. When the server is asked to disconnect from the client (or if the server is shutting down), the client thread must be shut down and resources used released.

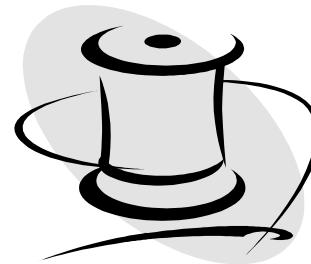
Forces:

- A thread or process is designed to run indefinitely.
- There could be unfortunate results if the thread is forcibly shut down without first cleaning up.
- When the thread or process is asked to shut down, it is acceptable to take a reasonable amount of clean-up time first.

Question: Why did Thread.stop() get deprecated in Java 2 and receive serious warnings in the docs?

Consequences:

- Positive: Allows threads to clean up before they terminate.
Negative: Can delay the termination for an unpredictable amount of time.



Notes

Comments: Many Java programmers are familiar with this idiom. However, it is ***NOT guaranteed to work***. This is because of little-known details of the Java memory model. The most effective way to fix the DCL idiom is to avoid it. In multithreaded programs, ensure any reference to a variable written by another thread is properly synchronized.

Example Code:

```
class SomeClass {
    private Resource resource = null;

    public Resource getResource() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

with DCL

```
class SomeClass {
    private Resource resource = null;

    public Resource getResource() {
        if (resource == null) {
            synchronized {
                if (resource==null) {
                    resource = new Resource();
                }
            }
        }
    }
}

// After 1st call to getResource(), resource is initialized, avoiding
// synch hit, & 1 thread. DCL averts race condition with 2nd check.
```

Usage: Lazy initialization does work as expected without synchronization for static singletons. When the initialized object is a static field of a class with no other methods or fields, the JVM effectively performs lazy initialization automatically. In the following example, the Resource will not be constructed until the field resource is first referenced by another class, and any memory writes resulting are automatically visible to all threads.

```
class MySingleton {
    Public static Resource resource = new Resource();
}
```

Note: Check the JMM to see if the problem has been fixed. In the meantime, carefully examine your multithreaded programs to ensure that any reference to a variable written by another thread is properly synchronized.

8. Double-Checked Locking, DCL

Pattern Thumbnail: Allows lazy initialization (deferred until needed) with reduced synchronization overhead.



Forces:

- Some expensive resource may never actually be needed, so avoid the overhead of creating it. Or, defer creation to improve start-up time.
- In multi-threaded environments two threads could simultaneously execute the test and initialize the resource twice.
- Synchronized methods run slower – maybe 100 times slower – and you don't want to pay that price.

Question: What does synchronized really mean?

Consequences: In the Java memory model, compilers, processors, and caches are free to take all sorts of liberties with our programs and data, as long as they don't affect the result of the computation. Sequences of program operations are rearranged to achieve higher performance. DCL relies on an unsynchronized use of the resource field.

Imagine that thread A is inside the synchronized block (facing page, right), executing the statement “resource = new Resource();” while thread B is just entering getResource(). Memory for the new Resource object will be allocated; the constructor will be called, initializing the member fields and the field resource of SomeClass will be assigned a reference to the newly created object.

Since thread B is not executing inside a synchronized block, it may see these memory operations in a different order than the one thread A executes. It could be that B sees these events in the following order: allocate memory, assign reference to resource, call constructor. Suppose thread B comes along after the memory has been allocated and the resource field is set, but before the constructor is called. It sees that resource is not null, skips the synchronized block, and returns a reference to a partially constructed Resource.

There are other scenarios which also give bogus results.

Notes

Chapter 9 – Patterns-Oriented Software Architecture (POSA)

Notes

Chapter Objectives

- Create an **overview** of the **basic architectural patterns**
- Examine **patterns** at a **higher level**
- Build **foundation** for **J2EE patterns** in next section

Notes

Factors driving the need for comprehensive software architecture:

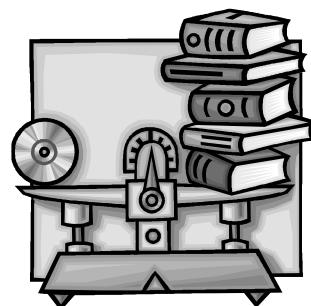
- Increasing use of software components
- Enterprise-wide technology implementations
- Rapid rate of technological change
- Importance of technology-to-business strategy
- Difficulty of connecting stable business strategies to unstable technology

Patterns-Oriented Software Architecture

Overview: Pattern-oriented software architecture (POSA) accomplishes at a system level what design patterns do at the subsystem level. Patterns can document the problem, its context, and a tried-and-true solution.

Software architecture provides benefits:

- basis for conceptual view of technological solutions
- solutions can be planned independently from specific technologies
- provides structure within which components can be assembled meaningfully
- clarifies requirements to design path
- means of communicating overall design



Three fundamental issues to address in creating software architecture:

1. What are the major structural elements?
2. How do they relate to each other?
3. What are the guiding principles for system organization?
(i.e., What do we hope to accomplish and what is the underlying rationale?)

"Patterns must support the development, maintenance and evolution of complex, large-scale systems. They must also support effective industrial software production, otherwise they remain just an interesting intellectual concept, but useless for constructing software."¹

¹ Buschmann, et al, *Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996, p. 21.

Notes

Architectural Patterns

(8 in 4 categories, from *Pattern-Oriented Software Architecture*)

I. Mud to Structure Patterns: (system partitioning, cooperating subtasks)

(1) Layers

(2) Pipes & Filters

(3) Blackboard

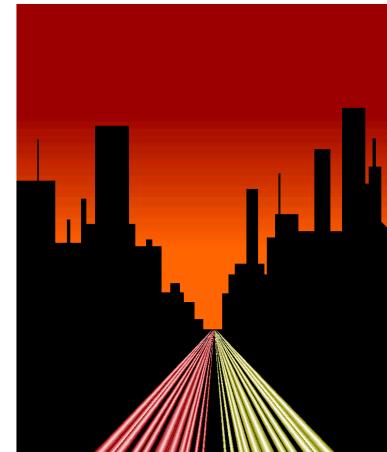
II. Distributed:

(4) Broker (standardized by OMG as “CORBA”)

[also Distributed:

(2) Pipes & Filters

(8) Microkernel]



III. Interactive:

(5) MVC (Model, View, Controller)

(6) PAC Agents (Presentation, Abstraction, Component)

IV. Adaptable: (evolving technology and changing functional requirements)

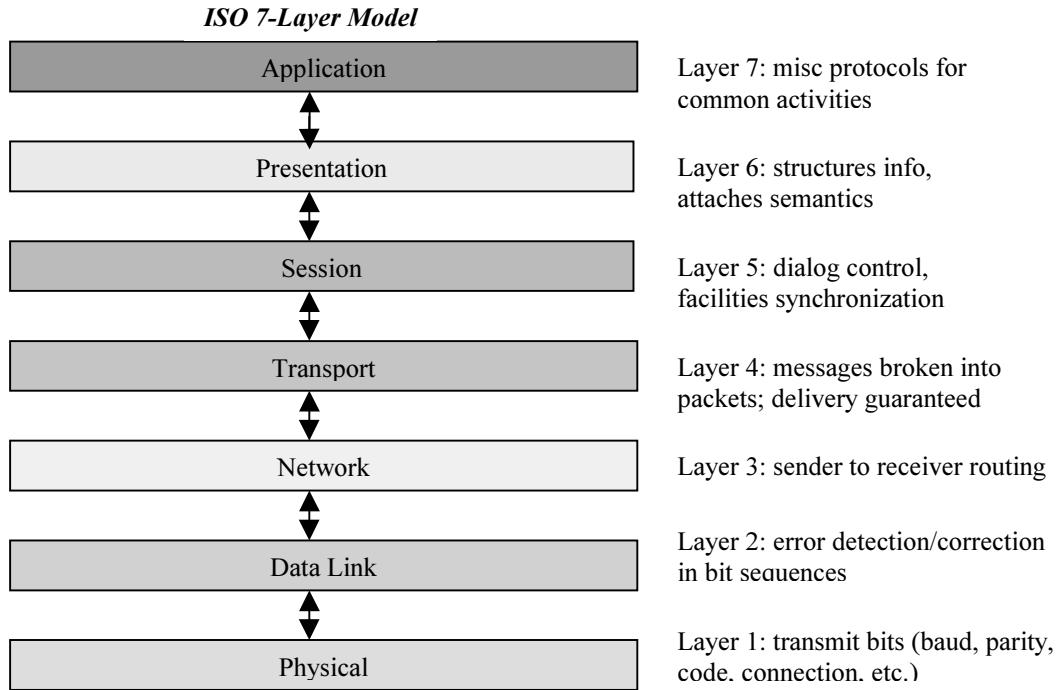
(7) Reflection

(8) Microkernel

Notes

Comments: Similar to a stack, or an onion, with each layer distinct and separating aspects of the system. Though tiered view systems are widespread, they are not the appropriate solution in every case.

Diagram:



Participants (presented as CRC Cards):

Class: Layer J	
Responsibilities <ul style="list-style-type: none"> ● Provide services used by Layer J+1 ● Delegate subtasks to Layer J-1 	Collaborators Layer J-1

Usage: Java's low level byte-code, platform independent, is delivered to a platform-dependent JVM. APIs provide another example of this approach – such as Java's JDBC layer. TCP/IP, the prevalent networking protocol, can reuse individual layers in different contexts – ftp, telenet, etc.



1. Layers Architecture

Pattern Thumbnail: Decomposition of complex systems by breaking into subtasks. Each subtask can be seen as a stack of layers where layers draw on services from below and/or provide services to layers above.

Example: J2EE has Presentation Tier, Business Tier (with processing logic), and Data/Integration Tier.

Forces:

- Mix of high and low level issues.
- Don't want late code changes to ripple through system.
- Keep interfaces stable.
- Have plug-compatible, exchangeable parts; facilitate reuse.
- Want to group components for cohesion and maintenance; crossing component boundaries may impede performance.
 - There is no standard component granularity; complex components need further decomposition.
 - Application needs to be built by a team and work must be subdivided along clear boundaries.

Question: What GoF patterns might have relevance in using a layered architecture?

Notes

Variants:

- Relaxed Layered System

use services of all layers below (more efficient but harder to maintain)

Examples: X Windows, Windows NT

- Layering Through Inheritance

higher layers inherit from base class in lower layers (easy to modify lower-level services but greater coupling between layers and changes in base classes require recompile of a lot)

Layers Architecture, continued

Implementation: (Top down vs. bottom up vs. Yo-yo)

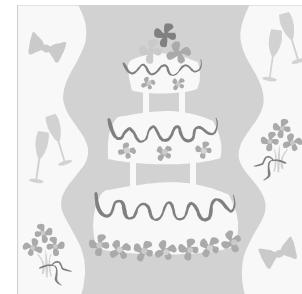
1. Decide abstraction criteria
2. Number of levels
(too many=overhead; too few = poor structuring)
3. Name levels and assign tasks
4. Specify services
5. Refine Layering (repeat steps 1-4)
6. Specify interfaces for each layer
7. Structure individual layers
i.e., use **Bridge** to support multiple implementations of services
use **Strategy** to support dynamic exchange of algorithms used by layer
8. Specify communication between adjacent layers
9. Decouple adjacent layers: push vs. pull; callbacks
10. Design error-handling strategy



Consequences:

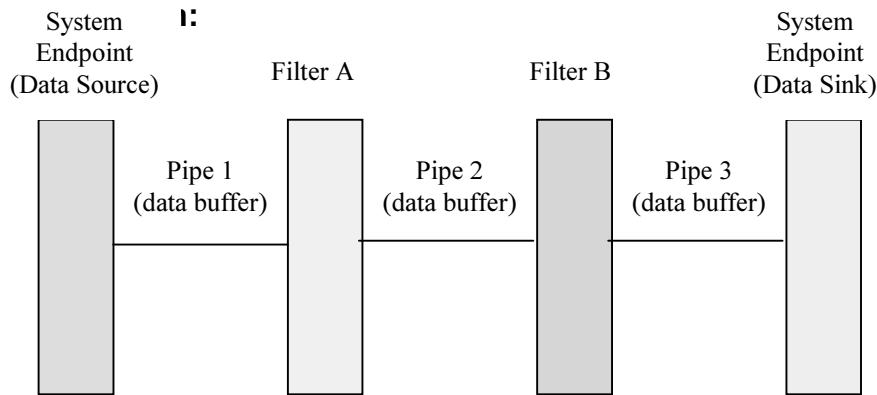
Positive: potential reuse of layers, support for standardization, dependencies kept local, exchangeability options.

Negative: Cascades of changing behavior, lower efficiency (diminished performance at runtime), unnecessary work (more programming effort), getting the right granularity for layers.



Notes

Comments: A filter consumes and delivers data incrementally; a sequence of filters combined is a processing pipeline.



Participants:

Class: Filter	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Get input data • Perform function on input data • Supply output data 	Pipe

Class: Pipe	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Transfer data • Buffer data • Synchronize active neighbors 	Data Source Data Sink Filter

Usage:

UNIX.

Implementation:

Straightforward. Use system services such as message queues or UNIX pipes.

- Steps:
1. Divide system's task into sequence of stages
 2. Define data format to be passed along each pipe
 3. Decide how to implement each pipe connection
 4. Design and implement the filters (a filter should do ONE THING WELL)
 5. Design error handling (consider resynchronization)
 6. Set up processing pipeline

2. Pipes & Filters Architecture

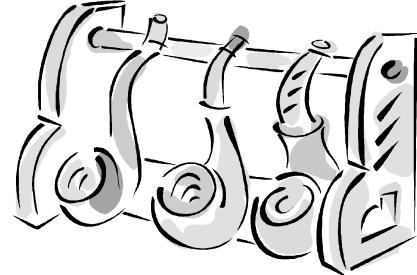
Pattern Thumbnail: A structure for systems that process a stream of data where each processing step is encapsulated in a filter component.

Example: Wireless applications.

Forces:

- You need to accommodate a variety of input and output.
- Non-adjacent steps do not share information.
- Future system enhancements should be possible by exchanging processing steps or by recombining steps, even by users.
 - Different sources of input data exist, such as network connections or hardware sensors. You want to present or store final results in various ways.
 - Storing intermediate results in files is error-prone if done by users and creates clutter.
 - You may want to consider parallel or quasi-parallel processing

Question: How do you maintain the integrity of the process, especially if it is interrupted?



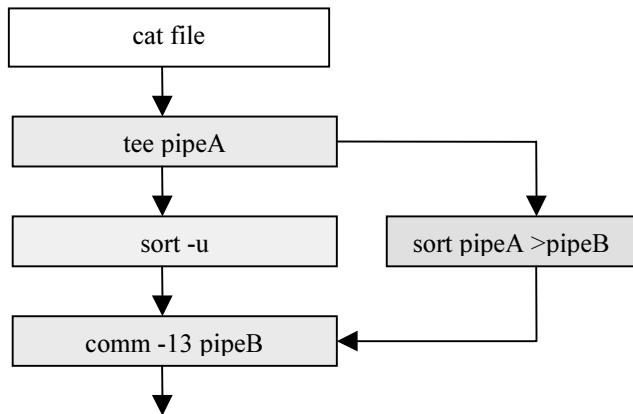
Consequences:

Positive: No intermediate files necessary, though possible. Flexibility by exchanging or recombining filters. Reuse of filters. Ability to do rapid prototyping of pipelines. Parallel processing potential.

Negative: Sharing state information is expensive/inefficient; overhead of data transformation; efficiency gain by parallel processing is often an illusion; error handling is a bear.

Notes

Tee and join pipelines are another variant.



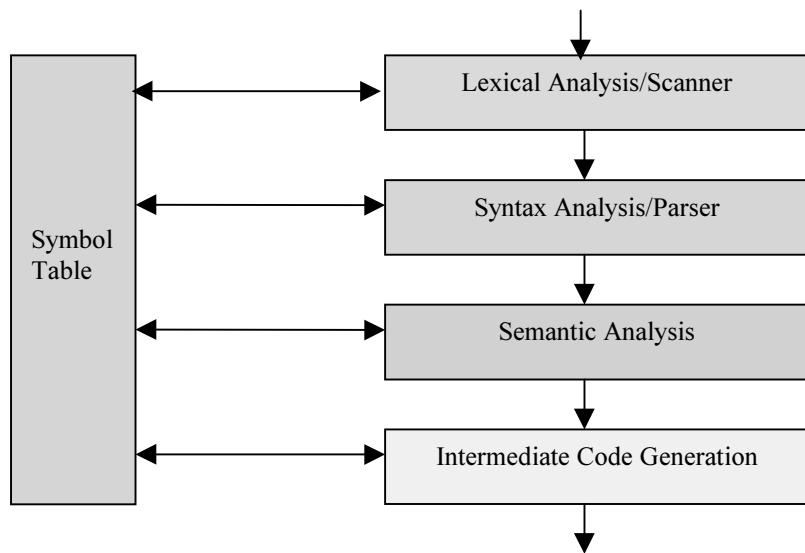
```

# first create two auxiliary named pipes
mknod pipeA p
mknod pipeB p
# now do the processing using avail UNIX
# filters
# start side fork of processing in background
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | com -13 - pipeB
  
```

Pipes & Filters Architecture, continued

Variants:

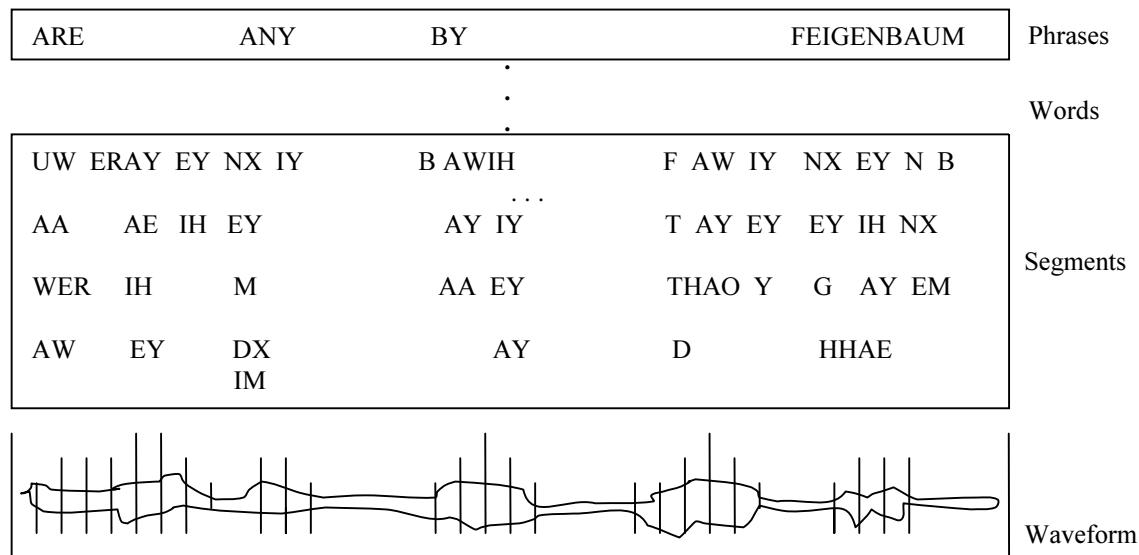
- If processing phases share some global data, such as a symbol table, there can be a combination of several filters.
- Many compilers combine initial steps into a single program because they all access and modify the symbol table.
- Passing lots of complex information along the pipeline imposes performance penalty.



Notes

Comments: Comes from the **AI community** and applies to applications where the domain is new and immature (and thus poorly structured and information is incomplete). However, you can reason to a solution even if it is not the optimal solution. The limitations of any Blackboard System should be carefully documented. As a domain matures, blackboard architectures are often abandoned in favor of predefined, closed solution approaches.

Diagram: (an image for example opposite)



Participants:

Class: Blackboard	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Manage central data 	

Class: Knowledge Source	
Responsibilities	Collaborators

Class: Control (Inference Engine)	
Responsibilities	Collaborators

3. Blackboard Architecture

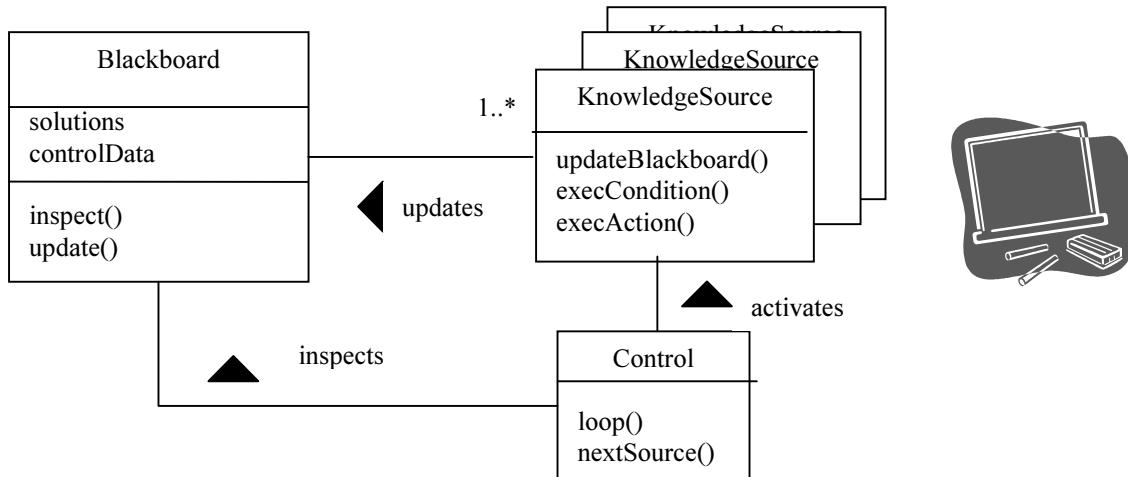
Pattern Thumbnail: No deterministic solution strategies are known and specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Example: Speech recognition where input is speech recorded as a waveform.

Forces:

- Complete search of solution space is not feasible;
- Uncertain data and approximate solutions are involved;
- Domain is immature;
- Different algorithms solve partial problems;
- Algorithms usually work on results of other algorithms;
- Using disjoint algorithms induces potential parallelism;
- All partial problems are solved using the same knowledge representation;
- Inputs (as well as intermediate and final results) have different representations.

Dynamics: Start main loop of Control component. Control calls nextSource() to select next KnowledgeSource. nextSource() checks Blackboard to determine which sources are potential contributors then invokes the condition-part of each candidate knowledge source. Depending on results of condition-part, Control chooses a hypothesis (or set of hypotheses) to be worked on. The action-part of the selected knowledge source is applied to the hypothesis and the blackboard is updated.



Question: How might the Strategy Pattern relate to the control structure? Do you know some of the basic strategies commonly used by expert systems?

Notes

Usage: HEARSAY-II, developed for speech recognition in the early 1970s; numerous expert systems.

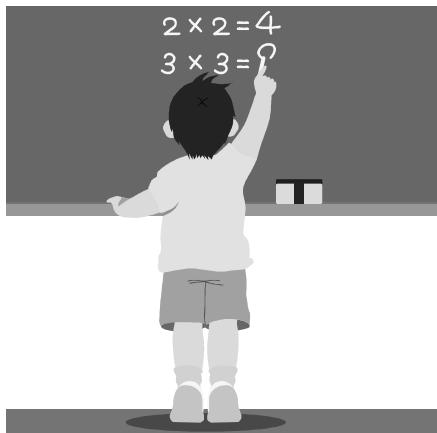
Solution: A collection of specialized independent programs that work cooperatively on a common data structure.

Implementation Steps:

1. Define problem – specify domain, scrutinize input to system, define output (including standards for correctness and fail-safe behavior), detail how user interacts with system.
2. Define solution space for problem – specify different abstraction levels of solutions and organize into hierarchies, find subdivisions that can be worked on independently (i.e., words of a phrase)
3. Divide solution process into steps – specify how to prune the tree
4. Divide knowledge into specialized knowledge sources with certain subtasks
5. Define vocabulary of the blackboard
6. Specify the control of the system
7. Implement the knowledge sources – different technologies for rule-based, neural network, set of conventional functions, etc.

Note: Knowledge sources may be organized according to design patterns: Layers, Reflection, wrap non-object-oriented sources with Façade.

Blackboard Architecture, continued



Variants:

Production System - Variant subroutines are represented as condition-action rules and data is globally available in working memory. Complicated scheduling algorithms are used for conflict-resolution.

Repository – This variant is a generalization of the Blackboard pattern. Repository architecture may be controlled by user input or by an external program; a traditional database can be considered as a repository and application programs working on the database correspond to the knowledge sources in the Blackboard architecture.

Consequences:

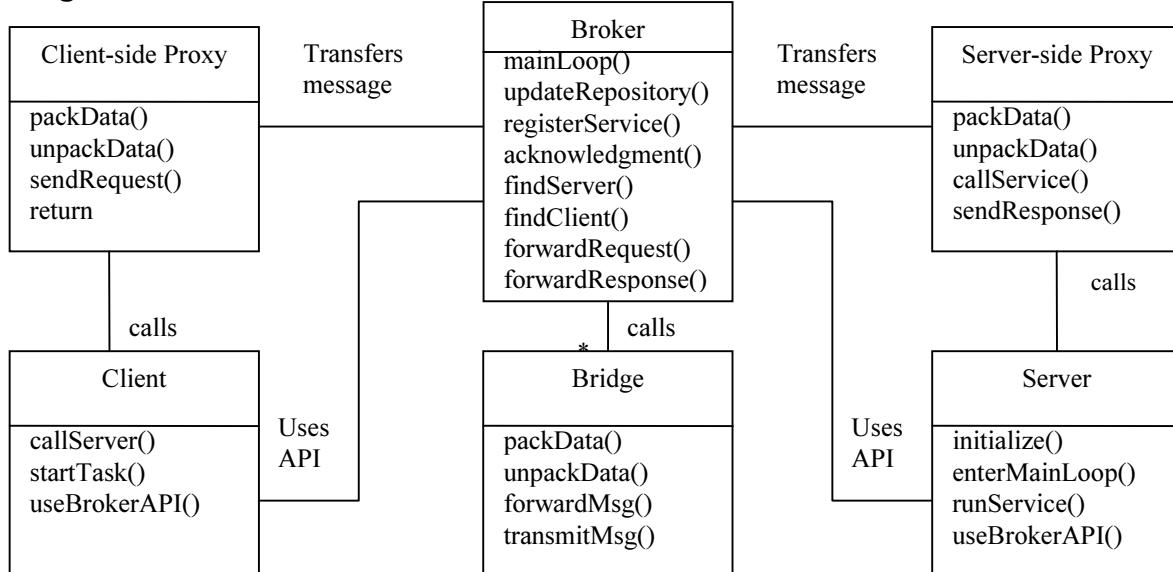
Positive: Experimentation with different algorithms and control heuristics possible, support for changeability and maintainability, reusable knowledge sources, support for fault tolerance and robustness (all results are just hypotheses).

Negative: Difficulty of testing, lack of guarantee of any good solution, difficulty of honing control strategy, low efficiency, high development effort, no support for parallelism.

Notes

Comments: The advantages of distributed systems are many: economics, performance and scaleability, inherent distribution (e.g. Client-Server apps), reliability. However, distributed systems need radically different software than do centralized systems. Consequently, many consortia (e.g. OMG) and companies (e.g. Microsoft) have developed their own technologies for distributed computing.

Diagram:



Participants: Besides Client-side Proxy and Server-Side Proxy:

Class: Client		
Responsibilities	Collaborators	
<ul style="list-style-type: none"> Implement user functionality Send requests to server thru client-side proxy 	Client-side Proxy	
Class: Server		
Responsibilities	Collaborators	
<ul style="list-style-type: none"> Implement services Register with Broker Send responses and exceptions to client 	Server-side Proxy Broker	
Class: Broker		
Responsibilities	Collaborators	
<ul style="list-style-type: none"> (Un-)register servers Offer APIs Transfer messages Error recovery Interoperate w/other brokers thru bridges Locate servers 	Client Server Client-side Proxy Server-side Proxy Bridge	
Class: Bridge		
Responsibilities	Collaborators	
<ul style="list-style-type: none"> Encapsulate network-specific functionality Mediate between local broker and bridge of remote broker 	Broker Bridge	

4. Broker

Pattern Thumbnail: Structure distributed systems with decoupled components that interact by remote services invocations. A Broker component is responsible for coordinating communications and transmitting results and exceptions.



Example: A city information system designed to run on a wide-area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments, public transportation, etc. Computer terminals in kiosks are connected and tourists throughout the city can retrieve information of interest via a browser. Data is distributed across the network and not maintained in the terminals.

Forces:

- Distributed components need a way to communicate.
- Components should be able to access services provided by others through remote invocations,
- You may need to exchange/add/remove components at run-time,
- The architecture should hide system and implementation-dependent details from users.

Question: How would you compare Broker with **Mediator**?



Notes

Usage: CORBA, IBM's SOM/DSOM, OLE 2.x, World Wide Web with hypertext browsers as brokers and www servers as service providers.

Solution: A broker component achieves better decoupling of clients and servers. Servers register with the broker and make their services available through method interfaces. Clients send requests to servers via the broker. Distribution becomes transparent to the developer through a flexible architecture that allows dynamic change.

Implementation Steps:

1. Define an object model or use an existing model.
2. Decide which kind of component-interoperability system should offer (binary standard, IDL, etc.)
3. Specify APIs provided by Broker for collaborating with clients and servers.
4. Use proxy objects to hide implementation details.
5. Design broker component along with steps 3 and 4.
Iterate systematically through the following steps:
 - 5.1. Specify detailed protocol for interacting with proxy components.
 - 5.2. A local broker must be available for every participating machine in the network.
 - 5.3. May need means to return results and exceptions back to original client.
 - 5.4. If proxies do not provide for marshaling/unmarshaling parameters/results, broker must provide that.
 - 5.5. If asynchronous communication, provide message buffers within broker or within the proxies
 - 5.6. Include a directory service for associating server IDs with physical location of corresponding servers in broker. (e.g., Internet port number in TCP/IP)
 - 5.7. If arch requires system-unique IDs to be dynamically generated during server registration, broker must offer a name service.
 - 5.8. If system supports dynamic method invocation, broker needs way to maintain type info about servers.
6. Develop IDL compilers



Broker, continued

Variants:

- Direct Communication Broker System
- Message Passing Broker System
- Trader System
- Adapter Broker System
- Callback Broker System

Consequences:

Positive: Location transparency, changeability and extensibility of components, portability, interoperability between different Broker systems, reusability.

Negative: Restricted efficiency, lower fault tolerance, debugging is **a bear**.

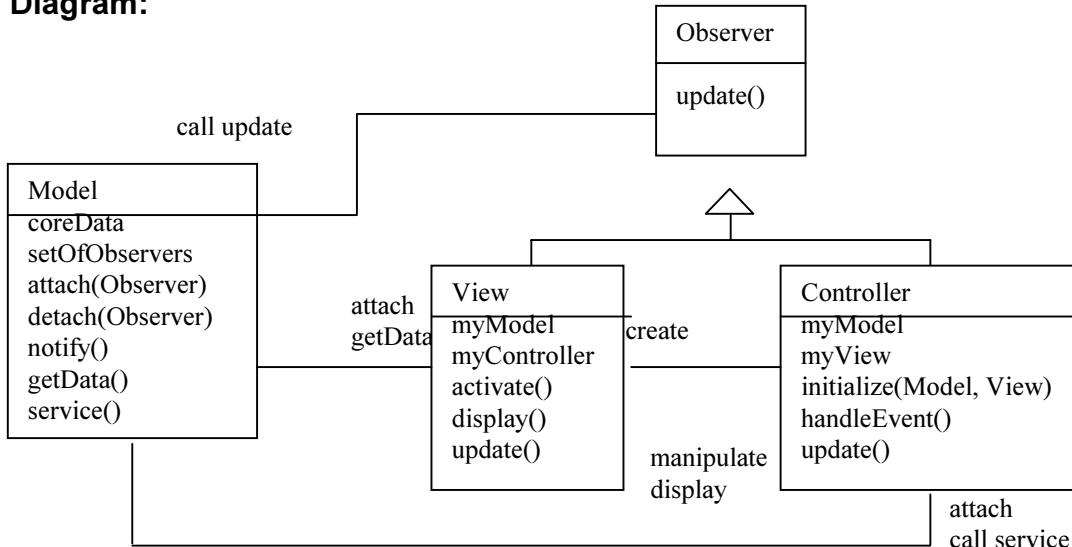


(However, note that a client application developed from tested services is more robust and easier itself to test.)

Notes

Comments: Though popular for many years, many toolkits define their own flow of control and it is difficult to retrofit toolkit components or the output of user interface layout tools to MVC.

Diagram:



Participants:

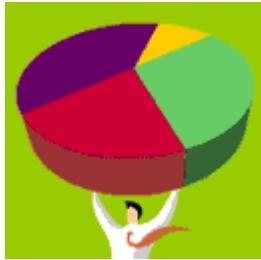
Class: Model	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Provide core of appl Register dependent views & controllers Notify dep. when data changes 	View Controller

Class: View	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Create/initialize its assoc controller Display info to user Implement update Retrieve model data 	Controller Model

Class: Controller	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Accept user input Translate Events to service for model or display for view Implement update 	View Model

5. Model-View-Controller (MVC)

Pattern Thumbnail: MVC divides an interactive application into three components: model contains core functionality and data, views display information to user, controllers handle user input. V+C=user interface.



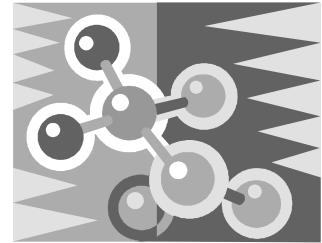
Example: Looking at the same data (model) as a pie chart, bar chart, spreadsheet, or other view. When any view is changed, all change, and it appears each knows about the other when in reality the views are decoupled.

A chat server is another example – clients appear to know of one another, but are decoupled and sent update notices by a controller component when model data changes.

Forces:

- The same information is presented differently in different windows
- Data manipulations must be reflected in display and behavior immediately;
- Changes to user interface should be easy and even possible at runtime;
- Supporting different 'look and feel' standards should not affect code in the model.

Question: Why does the user interface platform of long-lived systems represent a moving target? Discuss how customers, managers, various users, porting to new platforms, upgrading to new releases, etc. might enter into the picture.



Notes

Usage: Smalltalk, MFC.

Solution: MVC divides an interactive application into 3 areas: processing, output, and input.

Implementation:

Steps:

1. Separate human-computer interaction from core functionality.
2. Implement the change-propagation mechanism.
(Use Publisher-Subscriber pattern.)
3. Design and implement the views.
4. Design and implement the controllers.
5. Design and implement the view-controller relationship.
6. Implement the initialization set-up of MVC.
7. Dynamic view creation – if views can be opened/closed dynamically, design component to manage.
8. Pluggable controllers – i.e., novice vs. expert, disabled accessibility, etc.
9. Create infrastructure for hierarchical views and controllers.
10. Further decouple system dependencies (consider Bridge pattern and abstract display class, etc.)

Model-View-Controller (MVC), continued



Variants:

Document-View, where Document corresponds to Model and View combines MVC's View + Controller. Visual C++ does this.

Consequences:

Positive: Multiple views of the same model, synchronized views, pluggable views and controllers, exchangeability, framework potential.

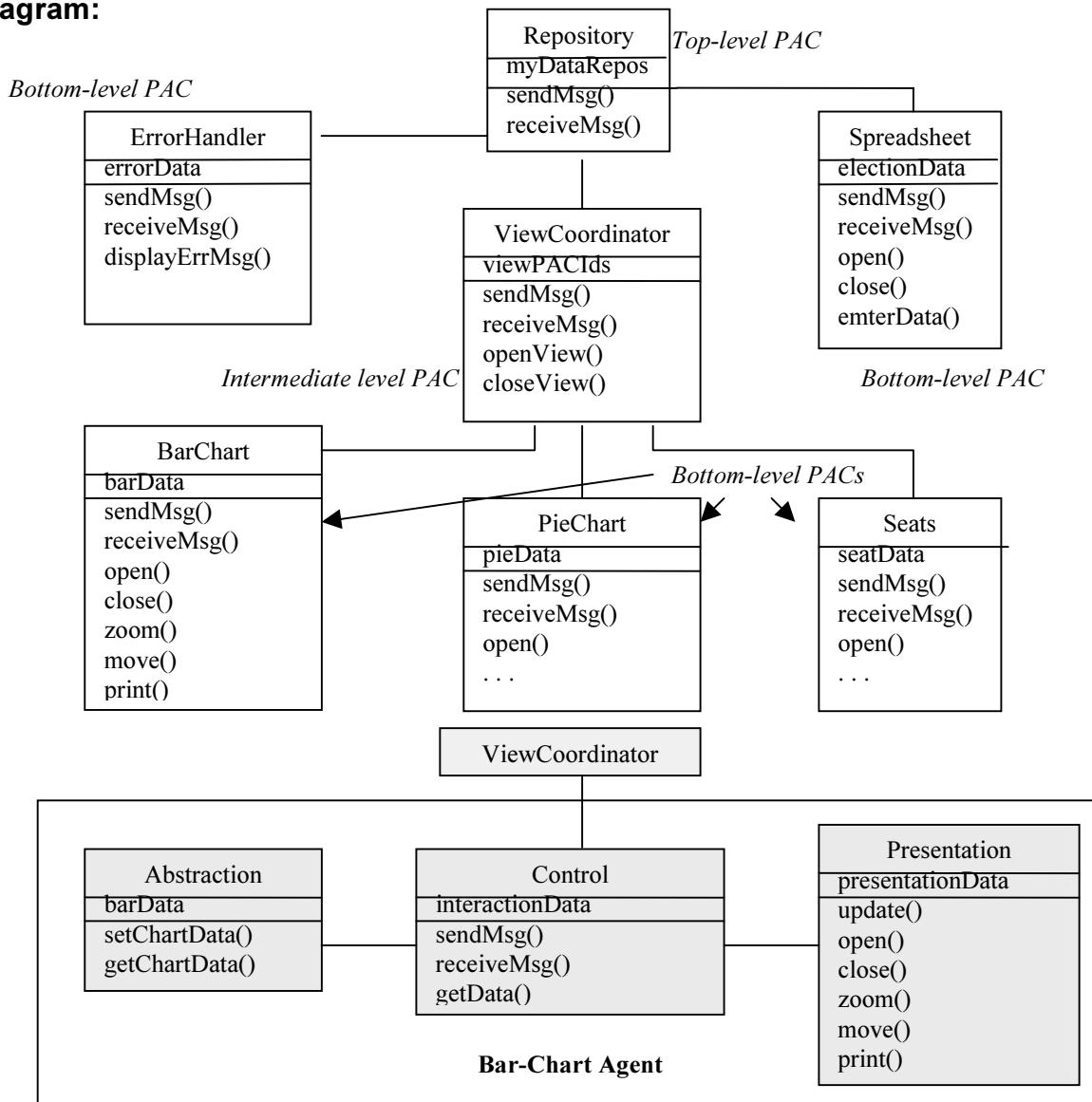
Negative: Increased complexity, potential for excessive number of updates, intimate connection between view and controller, close coupling of views and controllers to a model,

inefficiency of data access in view, inevitability of change to view and controller when porting, difficulty of using MVC with modern user interface tools.

Notes

Comments: This architecture captures both a horizontal and vertical decomposition of a system.

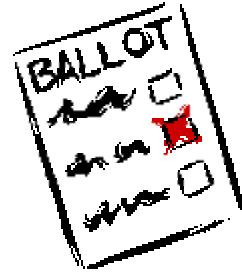
Diagram:



6. Presentation-Abstraction-Control (PAC)

Pattern Thumbnail: PAC defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Example: A simple information system for political elections with proportional representation. Data is entered via a spreadsheet; presentation is via tables and charts; users interact via a GUI.



Forces:

- Agents often maintain their own state and data;
- Interactive agents provide their own user interface;
- Systems evolve over time (especially the presentation aspects).

Question: How would you compare PAC and MVC? What aspects are similar? What is different?

Notes

Participants:

Class: Top-level Agent		Class: Intermediate-level Agent	
Responsibilities	Collaborators	Responsibilities	Collaborators
<ul style="list-style-type: none"> Provide functional core of the system Control PAC hierarchy 	<p>Intermediate-level Agent Bottom-level Agent</p>	<ul style="list-style-type: none"> Coordinate lower-level PAC Agents Compose lower-level PAC Agents to single unit of abstr. 	<p>Top-level Agent Intermediate-level Agent Bottom-level Agent</p>
Class: Bottom-level Agent			
Responsibilities	Collaborators		
<ul style="list-style-type: none"> Provide specific view of software or a system service, including its assoc human-computer interaction 	<p>Top-level Agent Intermediate-level Agent</p>		

Usage: Network Traffic Management. The system includes functions for: gathering traffic data from switching units, threshold checking and generating overflow exceptions, logging and routing of exceptions, visualization of traffic flow, displaying various user-configurable views of the network, statistical evaluations, access to historic traffic data, system administration and configuration, etc.

Solution: Structure the application as a tree-like hierarchy of PAC agents. There should be one top-level agent, several intermediate-level agents, and many bottom-level agents. Each agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control.

Implementation: Steps:

- Define a model of the application. Which services to provide? Which components can fulfill? What are relationships between components and how do they collaborate? What data do components operate on? UI?
- Define a general strategy for organizing the PAC hierarchy.
- Specify top-level PAC agent.
- Specify bottom-level PAC agents.
- Specify bottom-level PAC agents for system services.
- Specify intermediate-level PAC agents to compose lower-level PAC agents (if appropriate).
- Specify intermediate-level PAC agents to coordinate lower-level PAC agents (i.e., layout vs. edit views).
- Separate core functionality from human-computer interaction.
- Provide the external interface.
- Link the hierarchy together.

Presentation-Abstraction-Control (PAC), continued

Variants: Many large applications, especially interactive ones, are multi-user systems.

Two variants address multi-tasking issues.

- (1) PAC agents as active objects - every PAC agent runs in its own thread.
- (2) PAC agents as processes – PAC agents located in different processes or on remote machines, locally represented by proxies.

Consequences:

Positive:

- Separation of concerns,
- Support for change and extension,
- Support for multi-tasking.



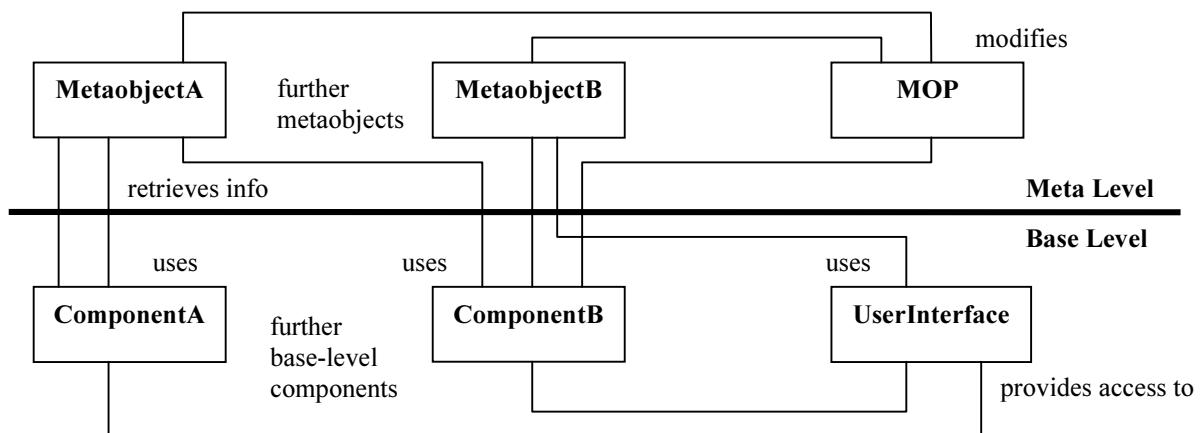
Negative:

- Increased system complexity,
- complex control component,
- loss of efficiency,
- lots of communication overhead;
- if agents are distributed, lots of marshaling/ unmarshaling/ fragmentation of data),
- applicability (small atomic concepts may have lots of similarity in UI and result in a complex fine-grain structure hard to maintain).

Notes

Comments: Building systems that support their own modification a priori – self-aware software – in a two-tiered architecture..

Diagram: (Note similarities to a layered system, but with mutual dependencies between both layers.)



Participants:

Class: Metaobject Protocol	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Offer interface for specifying changes to meta-level Perform specified changes 	Meta Level Base Level

Class: Base Level	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Implement application logic Use info provided by meta level 	Meta Level

Class: Meta Level	
Responsibilities	Collaborators
<ul style="list-style-type: none"> Encapsulate system internals that change Provide interface to facilitate mods to meta-level. 	Base Level

7. Reflection (aka Open Implementation, Meta-Level Architecture)

Pattern Thumbnail: Reflection provides a mechanism for changing structure and behavior of software systems dynamically. It supports modification of fundamental aspects such as type structures and function call mechanisms. Applications are split into two parts: (1) a meta level that provides information about system properties and (2) a base level that includes the application logic. Its implementation builds on the meta level; changes to information there affect subsequent base-level behavior.

Example: Developing a persistence component for a C++ application that is independent of specific type structures.
To store/read arbitrary objects you would need dynamic access to the internal structure.



Forces:

- Changing software is tedious, error prone, and often expensive.
- Adaptable software systems generally have a complex inner structure and aspects subject to change are encapsulated within separate components.
- The more techniques used to keep a system changeable (parameterization, subclassing, mix-ins, copy-and-paste), the more awkward/complex modification is.
- Changes can be of any scale – providing shortcuts to adapting frameworks.
- Even fundamental aspects of software systems can change, such as communication mechanisms between components.

Question: In order to implement a C++ persistence component (as in the example above), what would it have to know and do in order to read and store arbitrary data structures? What if you change runtime type information of base-level components?

Notes

Usage: CLOS is the classic example. OLE 2.0 provides functionality for exposing and accessing type information. JavaBeans and departures from that model, which uses introspection.

Solution: A self-aware software system with selected aspects the structure and behavior accessible for adaptation and change. The meta level provides a self-representation of the software to provide knowledge of its own structure and behavior. Metaobjects encapsulate/represent information about the software, such as type structures, algorithms, function call mechanisms. The base level defines the application logic. Its implementation uses the metaobjects to remain independent of those aspects likely to change. The interface for manipulating metaobjects is called the metaobject protocol (MOP); it allows clients to specify changes and is responsible for checking change correctness and performing the change. Changing the metaobjects changes behavior in base-level components without modifying base-level code.

Implementation Steps:

1. Define a model of the application – Which services to provide? Which components fulfill? What data?
2. Identify varying behavior, stable behavior – IPCs, shared memory, exceptions, real time constraints, etc.
3. Identify structural aspects of system, which when changed, should not affect base level implementation.
4. Identify system services – resource allocation, garbage collection, page swapping, object creation, etc.
5. Define the metaobjects – for every aspect in the 3 previous steps
6. Define metaobject protocol
7. Define the base level according to analysis model from step 1

Reflection, continued

Variants:

Reflection with several meta levels where metaobjects might depend on each other.

Consequences:

Positive: No explicit modification of source code, changing a software system is easy, support for many kinds of change.

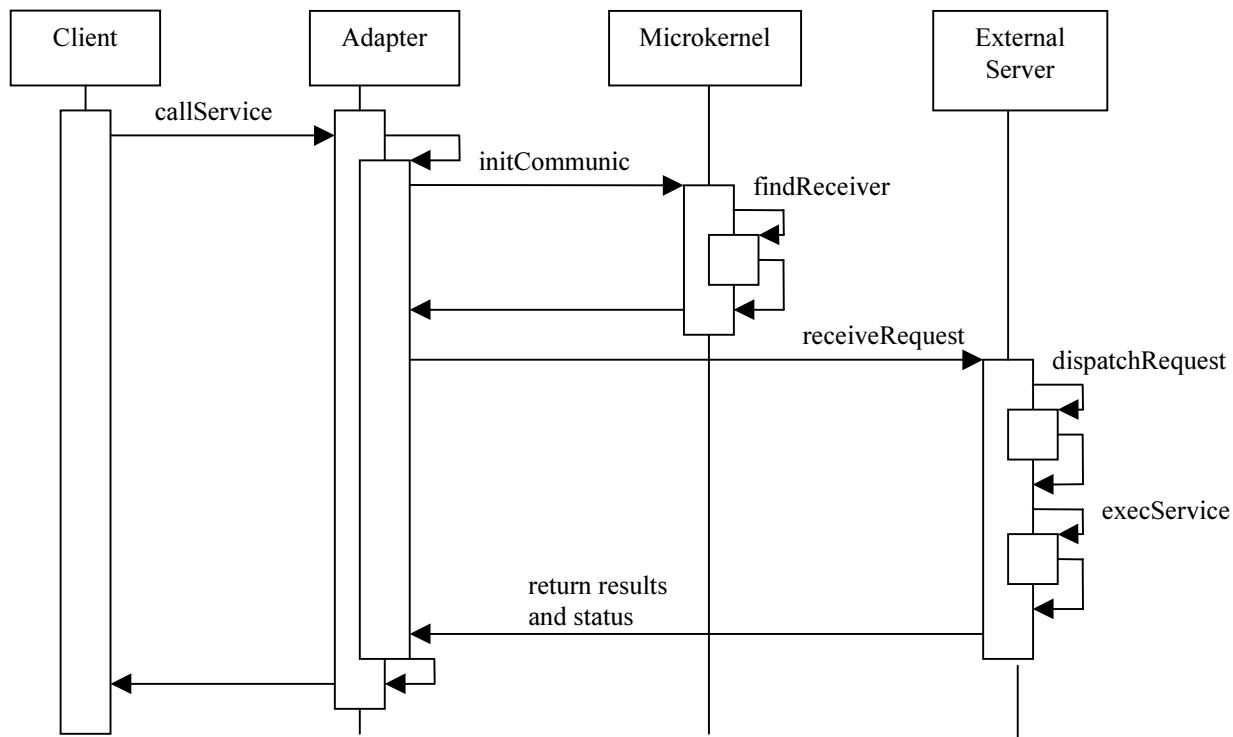
Negative: Modifications at the meta level may cause damage, increased number of components, lower efficiency, not all languages support reflection.



Notes

Comments: Support for ‘pluggable’ parts which can be exchanged.

Diagram: (for Client calling a service of an external server)





8. Microkernel

Pattern Thumbnail: The Microkernel architectural pattern supports change by providing a mechanism for extending the software with additional or customer-specific functionality. The microkernel serves as a socket for plugging in such extensions and for coordinating their collaboration.

Example: Creating a new O/S for desktop computers that allows the running of applications written for other popular operating systems such as NeXTSTEP, Windows, UNIX, etc. A user should be able to choose which O/S is desired from a pop-up menu prior to starting an application.

Forces:

- The application platform must cope with continuous hardware and software evolution.
- The application should also be portable, extensible and adaptable to allow easy integration of emerging technologies.
- The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.
- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

Solution:

- Encapsulate fundamental services of your application platform in a microkernel component that includes functionality that enables other components running in separate processes to communicate with each other.
- The microkernel also maintains system-wide resources such as files or processes and provides interfaces for other components to access its functionality.

Notes

Participants:

Class: Microkernel	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Provide core mechan. • Offer communication • Encapsulate system dependencies • Manage/control resources 	Internal Server

Class: External Server	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Provide programming interfaces for clients 	Microkernel

Class: Client	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Represent an application 	Adapter

Class: Internal Server	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Implement additional services • Encapsulate some system specifics 	Microkernel

Class: Adapter	
Responsibilities	Collaborators
<ul style="list-style-type: none"> • Hide system depend. like communication facilities from client • Invoke methods of ext servers for clients 	External Server Microkernel

Usage: Windows NT, an O/S for 3 high performance external servers – OS/2.1x , POSIX, Win32.

Implementation Steps:

1. Analyze the application domain. If necessary, perform a domain analysis and identify core functionality.
2. Analyze external servers – what policies they will provide
3. Categorize the services – build categories of operations not directly related to application domain.
4. Partition the categories – microkernel vs. internal servers' services.
5. Find a consistent and complete set of operations and abstractions for every category.
6. Determine strategies for request transmission and retrieval.
7. Structure the microkernel component – consider the Layers Pattern for system specific vs. system independent parts and hide system dependencies from higher layers.
8. Specify the programming interfaces of the microkernel – separate process vs. shared module.
9. Implement strategies for managing all system resources (memory, devices, device contexts, etc)
10. Design and implement the internal servers - as separate processes or shared libraries.
11. Implement the external servers – each as a separate process with its own service interface.
12. Implement the adapters – adapters package all relevant info and forward to an external server
13. Develop client applications (or use existing ones)

Microkernel, continued

Question: Part 1. What types of applications might this pattern be applicable to other than operating systems?

Part 2: Discuss differences between Broker and Microkernel.
Between Layers and Microkernel.



Variants: Microkernel System with indirect Client-Server connections. Microkernel establishes a communication path and then serves as a message backbone. Thus all requests pass through the microkernel and could control security, etc.

Distributed Microkernel System. Every machine is a distributed system uses its own microkernel implementation and a microkernel acts as a message backbone for sending/receiving messages. Distribution is transparent to users. In this variant, further benefits include scalability, reliability, and transparency.

Consequences:

Positive: Portability, flexibility and extensibility, separation of policy and mechanism.

Negative: Performance (price of flexibility), complexity of design and implementation.

Notes



Summary: How to Select an Architecture

Application needs

Understand the context and requirements:

- general properties
- non-functional requirements (i.e., changeability, reliability)

Explore Alternatives

Which patterns best address the problems?

Look at Trade-Offs

Compare candidate patterns

(i.e., MVC is more efficient than PAC; PAC better at multi-tasking)

Combine as Appropriate

(i.e., Broker for distributed clients and MVC as Server)

Follow/adapt implementation guidelines for selected pattern(s)

THEN,

further specify/refine

integrate functionality

identify design patterns

Exercises

1. Select an Architecture for the following example:

You need to develop Sales Force Automation software so that sales agents in the field can submit orders to an office-based server via a handheld device. Multiple business partners are involved in fulfilling an order, meaning varying formats and requirements. Information on the state of the order will be communicated back to the sales agent after the order is placed.

- a. List the requirements for this application.

- b. Note any associated problems you can think of.

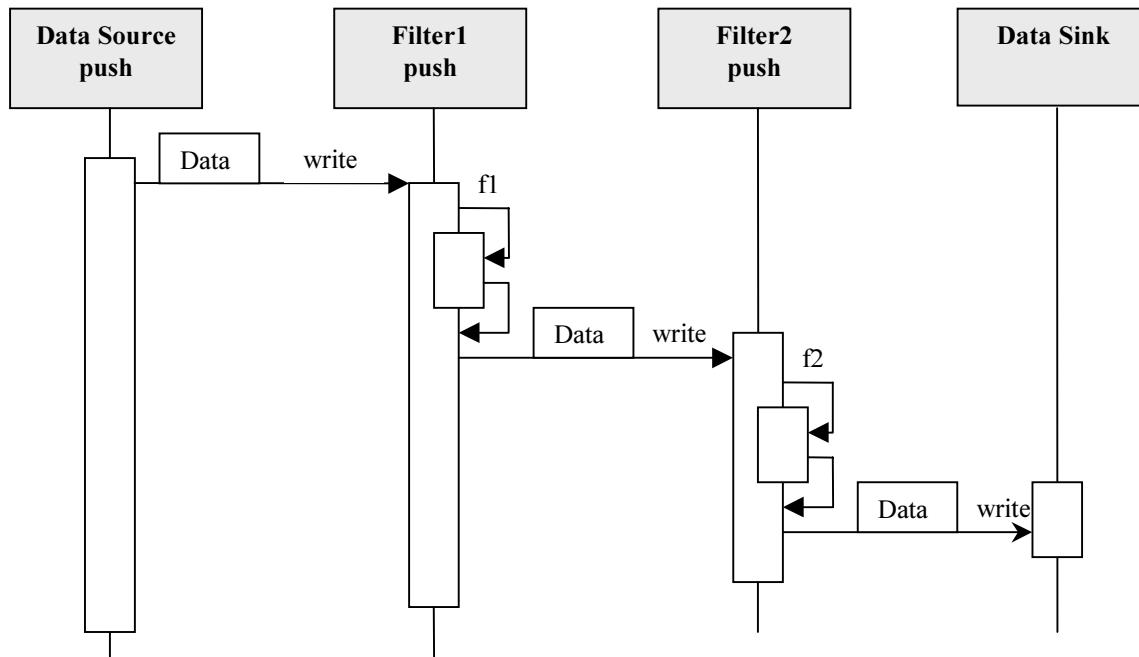
- c. Which architectural patterns best address the need here?

- d. Compare your candidate pattern(s) and select one.

- e. What are the implementation guidelines for the pattern(s) chosen?

- f. Define the processing steps you will need to include.

2. If the diagram below represents a “push” pipes & filters model, redesign it for a “pull” model.



Notes

Chapter 10 – Catalog of J2EE Patterns

Notes

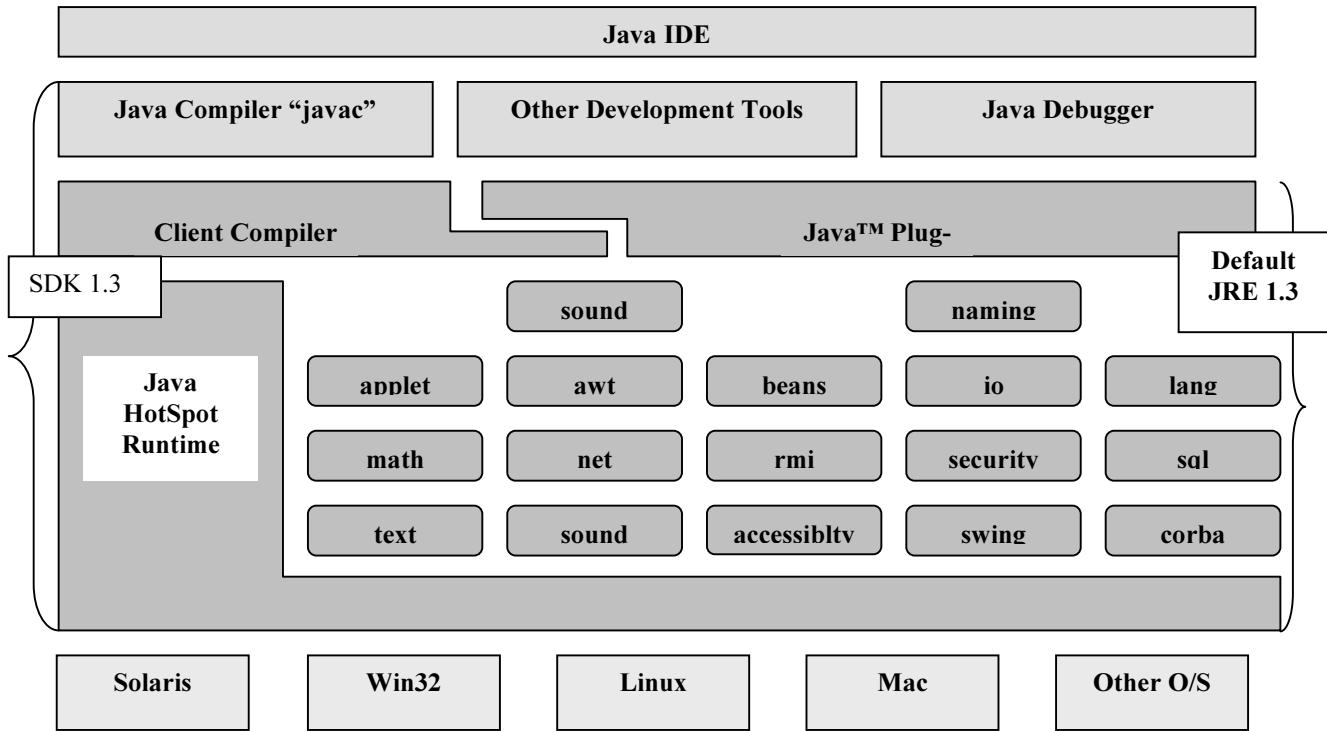
Chapter Objectives

- Learn **common solutions** to problems within the J2EE platform
- Build **vocabulary**
- Prevent **re-inventing the wheel** in J2EE design
- Be able to identify design patterns for **each tier**:
 - **presentation tier** (6)
 - **business tier** (7)
 - **integration tier** (2)
- Increase list of **resources for J2EE**

Notes

J2SE is the underlying base platform for J2EE.

J2SE platform components:



The J2EE component container supports application components in the J2EE platform. A container is a service that provides the necessary infrastructure and support for a component to exist and for the component to provide its own services to clients.

Notice that the layers on the facing page contain the following numbers of patterns:

Presentation – 6

Business – 7

Integration – 2

Catalog of J2EE Patterns

Purpose	Design Pattern	Aspect(s) That Can Vary
<i>Presentation Tier</i>	Intercepting Filter	Facilitates preprocessing and post-processing of a request
	Front Controller	Provides a centralized controller for managing the handling of a request
	View Helper	Encapsulates logic not related to presentation formatting into Helper components
	Composite View	Creates an aggregate View from atomic subcomponents
	Service to Worker	Combines Dispatcher component with Front Controller and View Helper patterns
	Dispatcher View	Combines Dispatcher component with Front Controller and View Helper patterns, Deferring activities to View processing
<i>Business Tier</i>	Business Delegate	Decouples presentation and service tiers, provides façade and proxy interfaces
	Value Object	Facilitates data exchange between tiers by reducing network chattiness
	Session Façade	Hides business object complexity; centralizes workflow handling
	Composite Entity	Represents best practice for coarse-grained entity beans by grouping parent-dependent objects into single entity bean
	Value Object Assembler	Assembles a composite value object from multiple data sources
	Value List Handler	Manages query execution, results caching, and results processing
	Service Locator	Encapsulates complexity of business service lookup/creation; locates factories
<i>Integration Tier</i>	Data Access Object	Abstracts data sources; provides transparent access to data
	Service Activator	Facilitates asynchronous processing for EJB components

Notes

Some of the J2EE Patterns sound very similar in structure; however the intent varies.

The core focus of the J2EE patterns presented here is the design and architecture of applications using servlets, JSPs, and enterprise bean components.

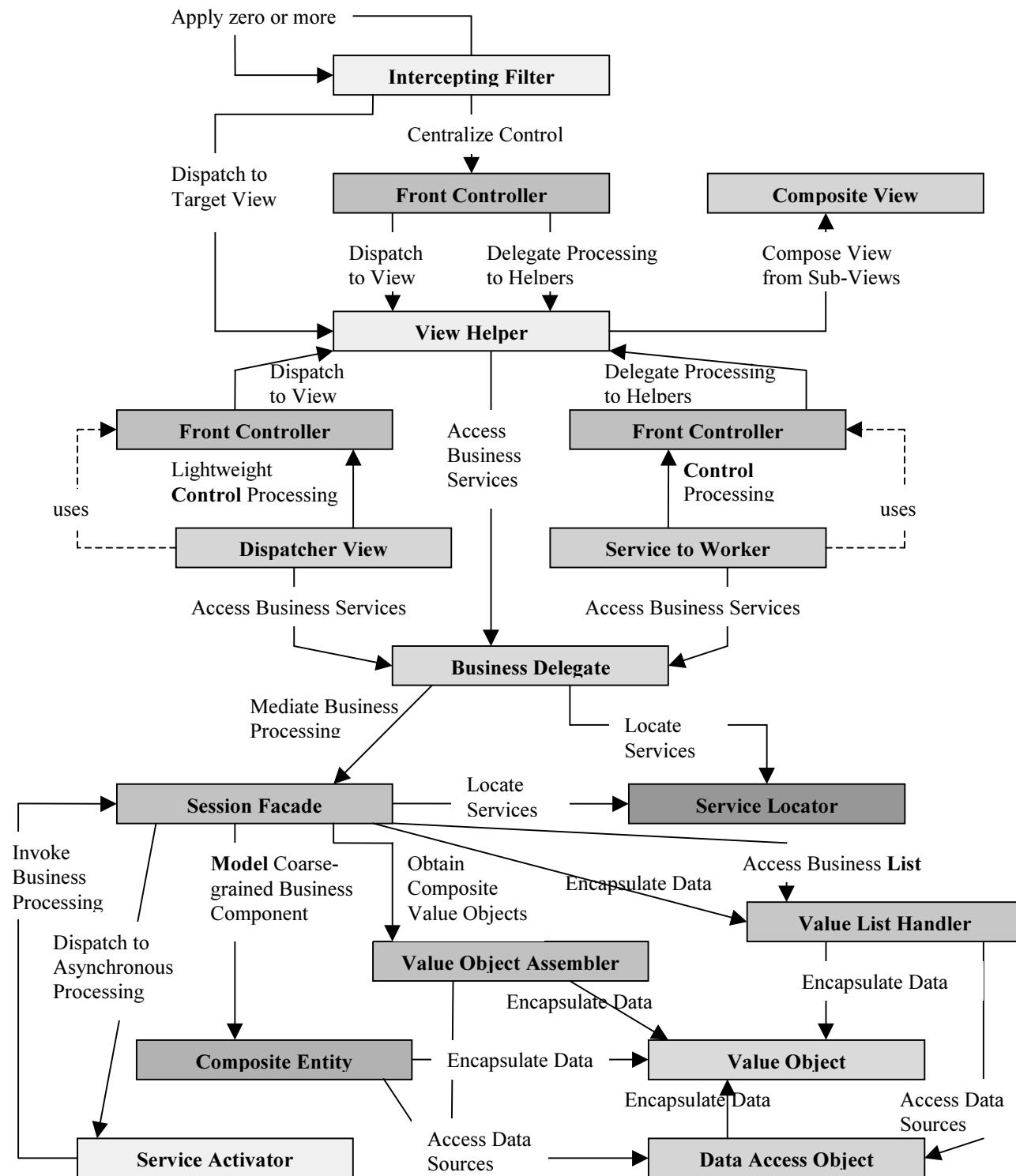
The J2EE platform uses a set of defined roles to conceptualize the tasks related to the various workflows in the development and deployment life cycle of an enterprise application. It standardizes a number of different technologies to provide a robust platform for building distributed multitier enterprise class applications.

J2EE roles (provide logical separation of team responsibilities):

- J2EE product provider
- Application component provider
- Application assembler
- Application deployer
- System administrator
- Tool provider

The final chapter in *Core J2EE Patterns* explores example use cases to demonstrate how many patterns come together to form a patterns framework to realize a use case.

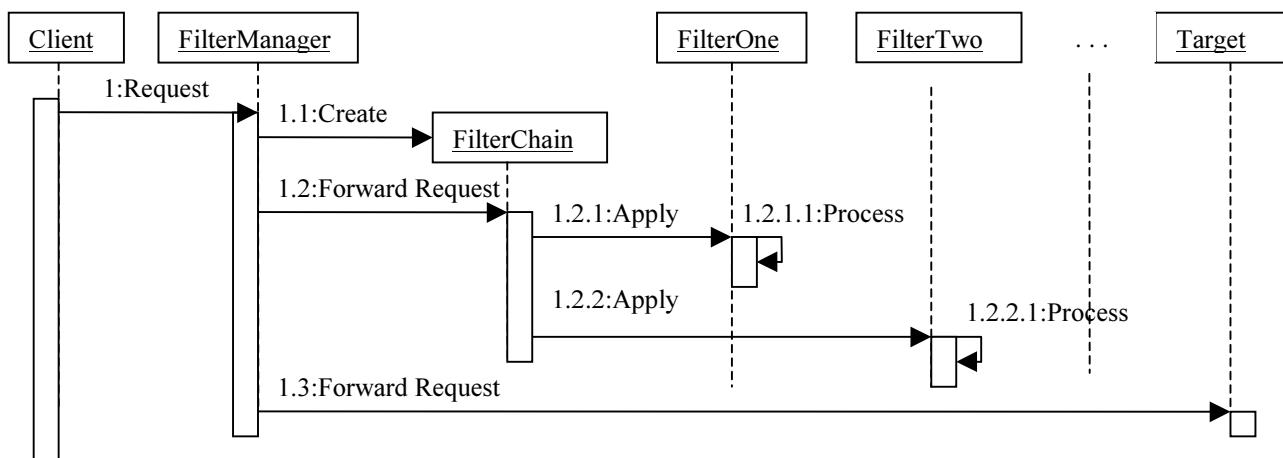
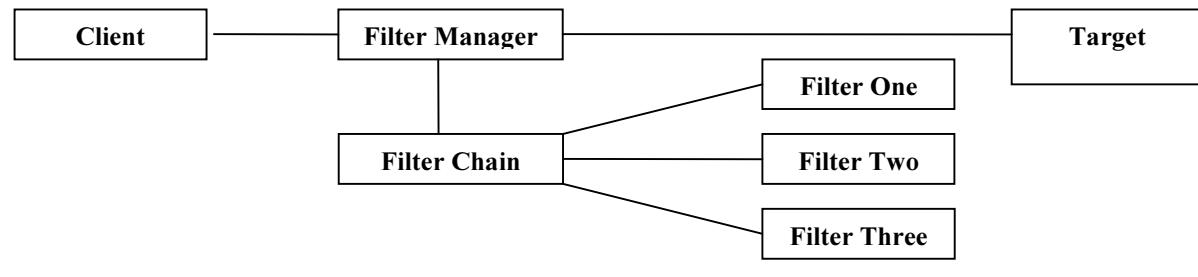
J2EE Pattern Relationships



- J2EE Patterns do not exist in isolation – they form larger framework of interdependence.
- It is important to understand the big picture.

Notes

Structure:



Participants/Responsibilities: FilterManager (creates FilterChain with appropriate filters/order); FilterChain (collection of independent filters that coordinates); FilterOne (individual filters mapped to a target); Target (resources requested).

Code: For code examples for Chapter 10, see *Core J2EE Patterns*, by Alur, Crupi, and Malks. Find additional information on this book and downloads for all code at:

<http://authors.phptr.com/corej2eepatterns/index.html>

For Intercepting Filter code, see *Core J2EE Patterns*, examples 7.1 - 7.13 (pages 152-171 in the book.)

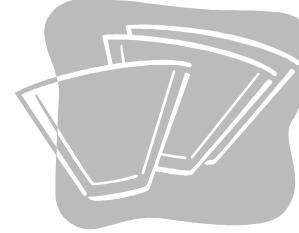
1. Intercepting Filter (J2EE p. 152)

Context: The presentation tier receives many different requests, each requiring different processing from simple forwarding to modification, auditing, uncompressing of data, etc.

Problem:

Incoming requests must be tested for many things:

- Client authenticated?
- Client has valid session?
- Client's IP address is from trusted network?
- Request path doesn't violate any constraints?
- Encoding used by client to send data?
- Client's browser supported?

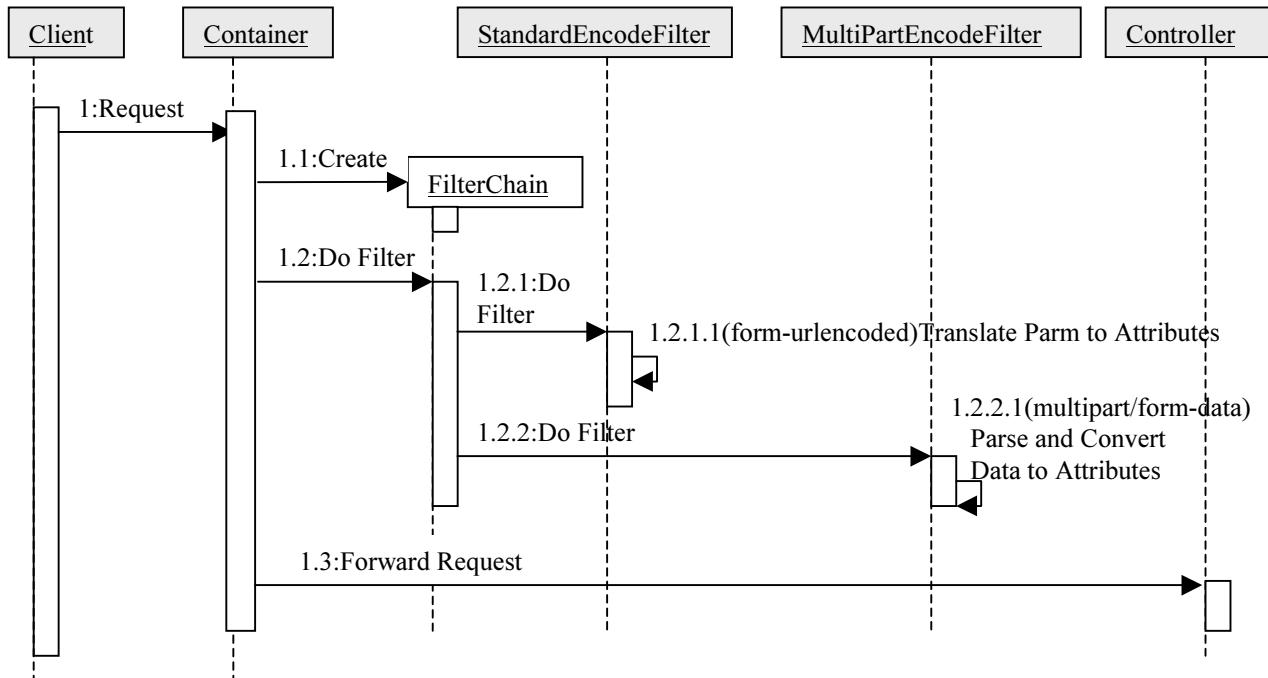


Forces: Want to centralize common logic but common processing varies with each request. Services should be configurable for a variety of combinations—logging, authentication, debugging, transformation of output, converting input, etc.

Solution: Decorate main processing with a variety of filters for common services; components may be added/removed.

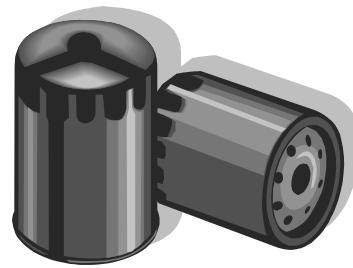
Notes

Sample Sequence Diagram for Standard Filter Strategy, encoding conversion example:



Related Patterns: Front Controller, Decorator, Template Method, Interceptor, Pipes & Filters

Intercepting Filter, continued



Consequences:

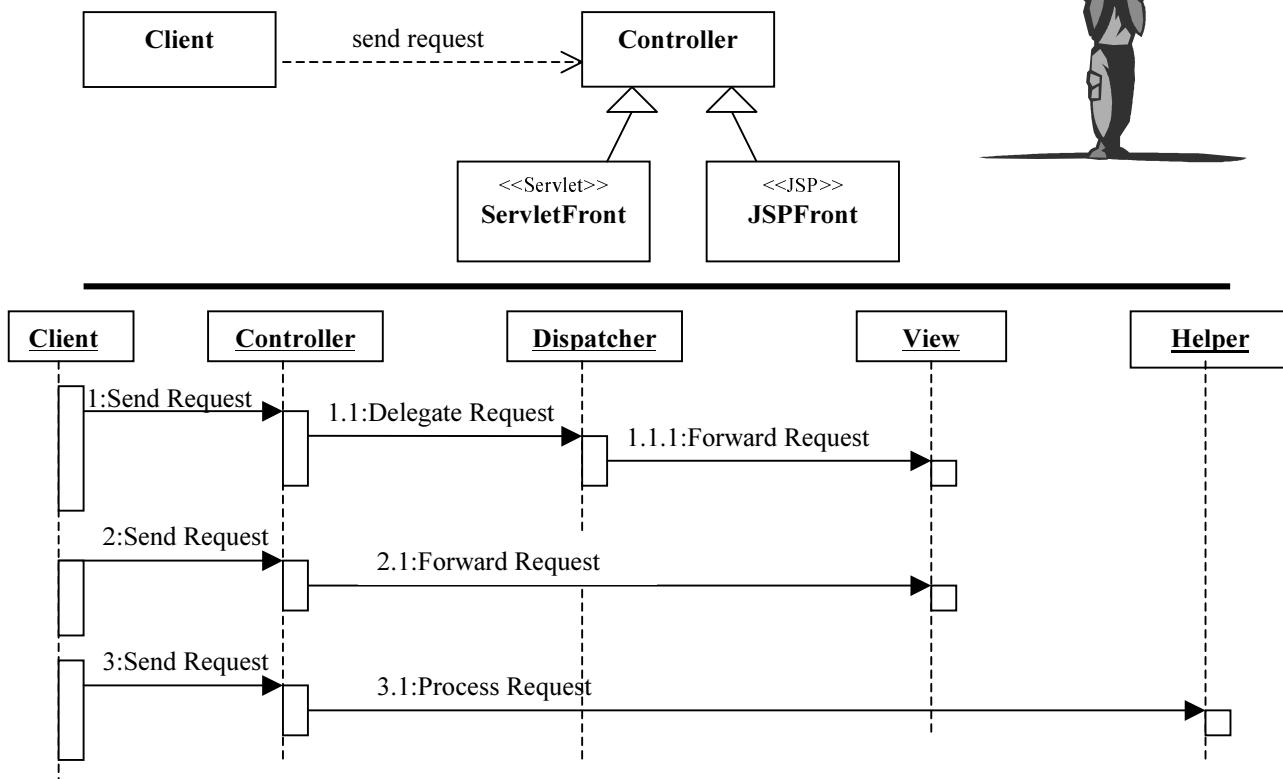
- Centralizes control with loosely coupled handlers
- Improves reusability with pluggable interceptors due to standard interface
- Declarative and flexible configuration – can combine various services without recompile of core base
- Inefficient information sharing – since filters are loosely coupled – if filters need to share, reconsider strategy.

Strategies:

- Custom Filter (less flexible/powerful—uses Decorator to wrap filters);
- Standard Filter (servlet 2.3 spec includes mechanism for building filter chains);
- Base Filter (common superclass for all filters); and
- Template Filter (base filter provides a template method).

Notes

Structure:



Participants/Responsibilities: Controller (initial point of contact for requests); Dispatcher (responsible for view management/navigation); Helper (typically implemented as JavaBeans and custom tags – can gather/store values, adapt data for views or other forms, etc.); View (display info to client).

2. Front Controller (J2EE p. 172)

ONE WAY

Context: Presentation tier request handling has to control/coordinate processing of each user across multiple requests.

Problem: Need centralized point of access to handle presentation tier requests and support integration of underlying services. If user accesses view directly several problems may result:

- each view must provide own system services (duplicated code, a No-No)
- view navigation becomes responsibility of the views with potential mixing of view content and view navigation
- distributed control is harder to maintain

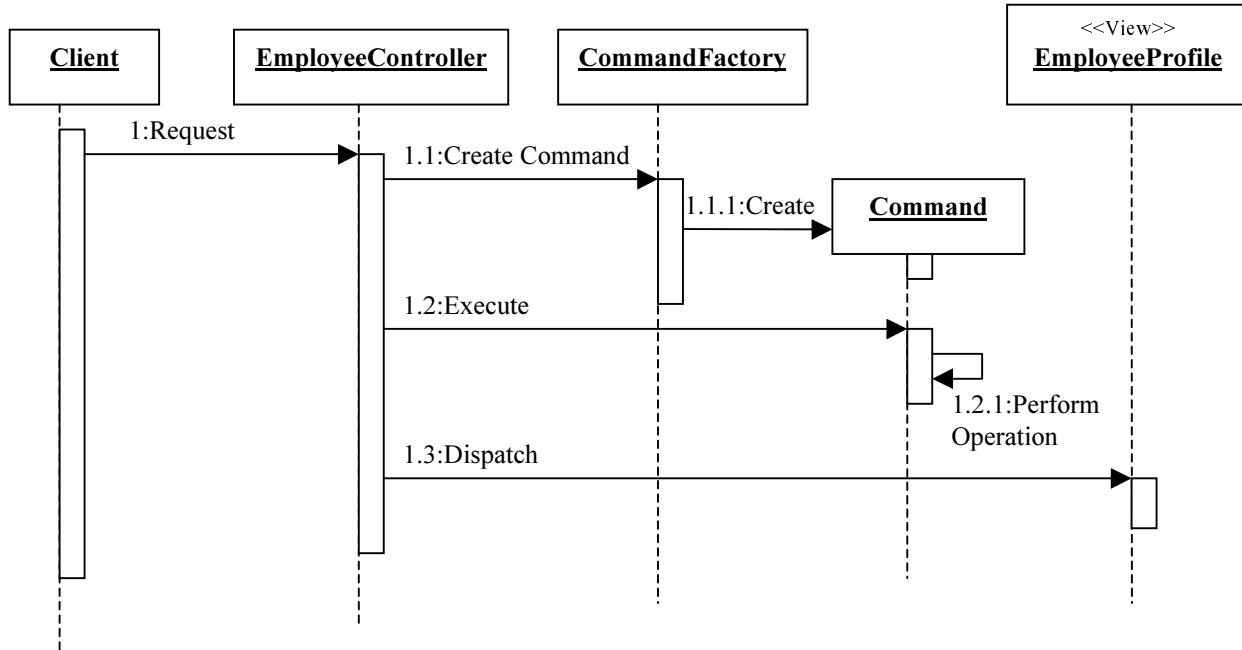
Forces:

- Common system services processing completes per request; logic best handled centrally becomes replicated within many views.
- Decision points exist with respect to retrieval/manipulation of data.
- Multiple views used to respond to similar business requests.
- Centralized contact for requests may be useful for things like controlling/logging user progress through site.
- System services and view management logic are relatively sophisticated.

Solution: Use a controller as initial point of contact for handling requests. Controller then responsible for managing security services, delegation of business processing, choosing appropriate view, error handling, managing strategies for creation of content, etc.

Notes

Example of Command and Controller strategy sequence diagram:



Code: See *Core J2EE Patterns*, examples 7.14 - 7.16 (pages 172-185)

Front Controller, continued

Strategies:

- ServletFront (implementing controller as servlet),
- JSPPFront,
- Command & Controller (based on GoF Command),
- Physical or Logical Resource Mapping,
- Multiplexed Resource Mapping,
- Dispatcher folded into Controller,
- Base Front (controller hierarchy), implementing as a filter.

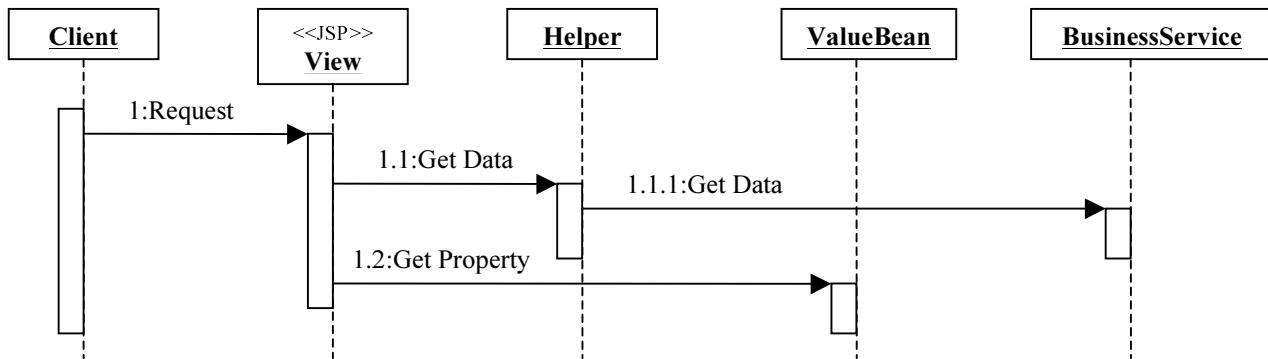
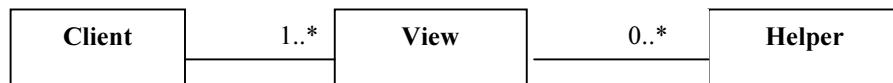


Consequences: Centralizes control; improves manageability of security; improves reusability.

Related Patterns: View Helper, Intercepting Filter, Dispatcher View and Service to Worker.

Notes

Structure:



Participants/Responsibilities: View (represent/display information to client); Helper (assist in processing: gather/store data, adapting, etc.); ValueBean (helper responsible for holding intermediate model state for use by a view); BusinessService (role fulfilled by service client seeks)

Code: See *Core J2EE Patterns*, examples 7.17 - 7.21 (pages 186-202)

3. View Helper (J2EE p. 186)

Context: The system creates presentation content, which requires processing of dynamic business data.



Problem: Presentation tier changes occur often. If presentation formatting logic and business data access logic are interwoven then maintenance becomes difficult, modularity is reduced and there is poor separation of roles among Web production and software development teams.

Forces:

- Business data assimilation requirements are nontrivial
- Maintenance issues arise when business logic is embedded in view – typically using copy-and-paste reuse
 - Clean separation of labor is desirable among Web team and software team
 - One view is commonly used to respond to a particular business request

Solution: A view contains formatting code and delegates processing responsibilities to its helper classes (usually JavaBeans or custom tags). Helpers store view's intermediate data model and serve as business data adapters.

Strategies: JSP as view component (preferred), Servlet as view (if tags embedded in Java code is more difficult to maintain), JavaBean Helper (business logic factored out of view), Custom Tag Helper (business logic factored out and encapsulated in this component—requires more upfront work), Business Delegate as helper (hides underlying implementation details for distributed invocations to business tier – can be implemented as JavaBean), transformer helper (eXtensible Stylesheet Language Transformer, useful with XML models).

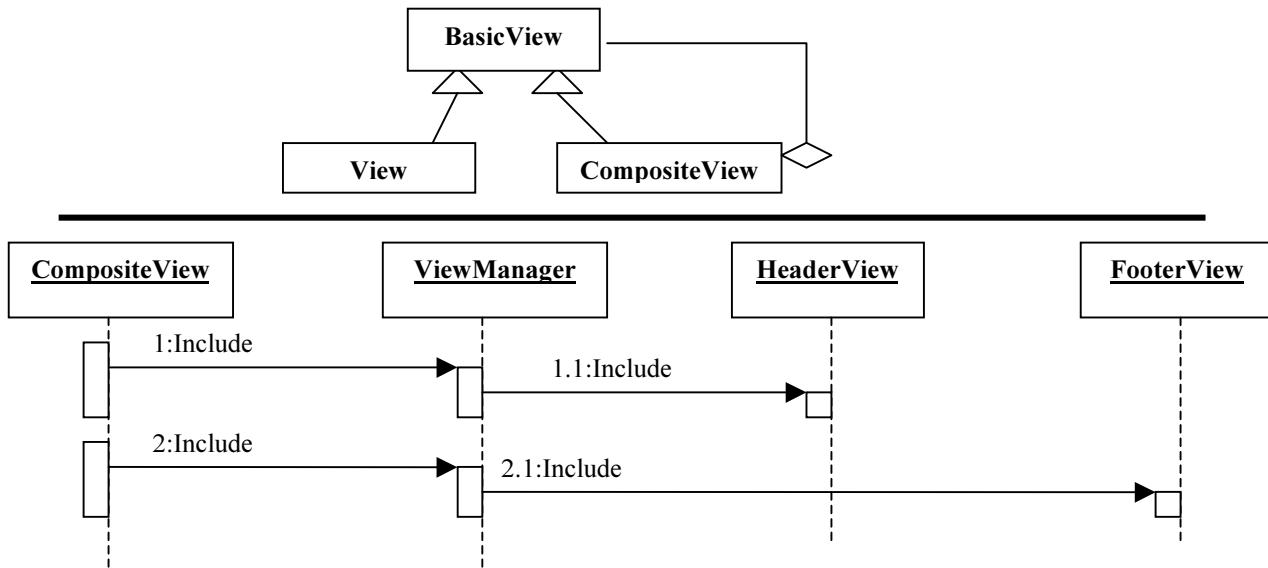
Consequences:

- Improves application partitioning, reuse, and maintainability
- Improves role separation

Related Patterns: Business Delegate, Dispatcher View and Service to Worker, Front Controller

Notes

Structure:



Participants/Responsibilities: Composite View (an aggregate of subviews); View Manager (manages inclusion of subparts); Included View (a subview, one atomic piece or a composite).

Code: See *Core J2EE Patterns*, examples 7.22 - 7.28 (pages 203 - 215)

4. Composite View (J2EE p. 203)



Context: Present multiple subviews as a single display page.

Problem: Web pages are built by embedding formatting code directly within each view—with resulting difficulty of modification and code duplication leading to increased errors.

Forces:

- Atomic portions of view content change often;
- many composite views use similar subviews but with different text;
- layout changes are harder to manage and code harder to maintain;
- embedding frequently changing text portions directly into views may affect availability/administration of system (i.e., may need to restart server to see mods).

Solution: Use composite views composed of multiple atomic subviews, including each component dynamically into layout, which can then be managed independently of content.

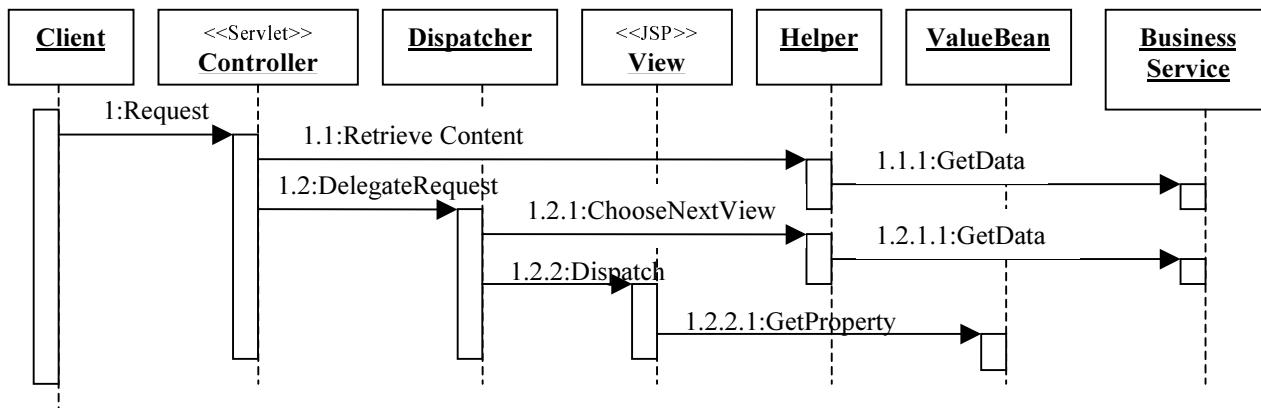
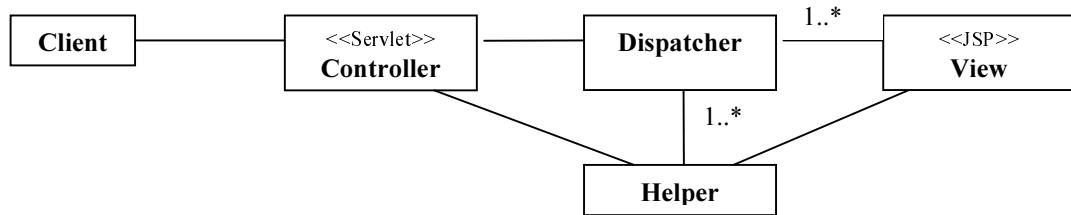
View Management Strategies: JSP, Servlet, JavaBean, Standard Tag, Custom Tag, Transformer, Early-Binding Resource, Late-Binding Resource

Consequences: Improves modularity and reuse; enhances flexibility, maintainability and manageability; performance impact; aggregating introduces potential for display errors (manageability issue).

Related Patterns: View Helper, Composite.

Notes

Structure:



Participants/Responsibilities: Controller (initial point of contact for requests); Dispatcher (resp. for view mgmt/navig); View (represent/display info to client); Helper (processing for view or controller); ValueBean (helper responsible for state used by view); BusinessService (role fulfilled by service client wants).

Code: See *Core J2EE Patterns*, examples 7.29 - 7.33 (pages 216 - 230)

5. Service to Worker (J2EE p. 216)



Context: The system controls flow of execution and access to business data, from which it creates presentation content.

Problem: No centralized component for managing access control, retrieval of content, and management of views; duplicate control code exists in various views; business and formatting logic are mixed within views. System becomes inflexible, unreusable, unmodular, less resilient to change, and there is poor separation of Web roles.

Forces:

- Authentication/authorization checks completed per request;
- scriptlet code within views should be minimized;
- business logic should be encapsulated in non-view components;
- control flow is relatively complex and based on values from dynamic content
- view management logic is relatively sophisticated—multiple views may map to the same request.

Solution: Combine a controller and dispatcher with views and helpers to handle client requests and prepare dynamic presentation. (Similar structural solution to Front Controller and View Helper—different usage.)

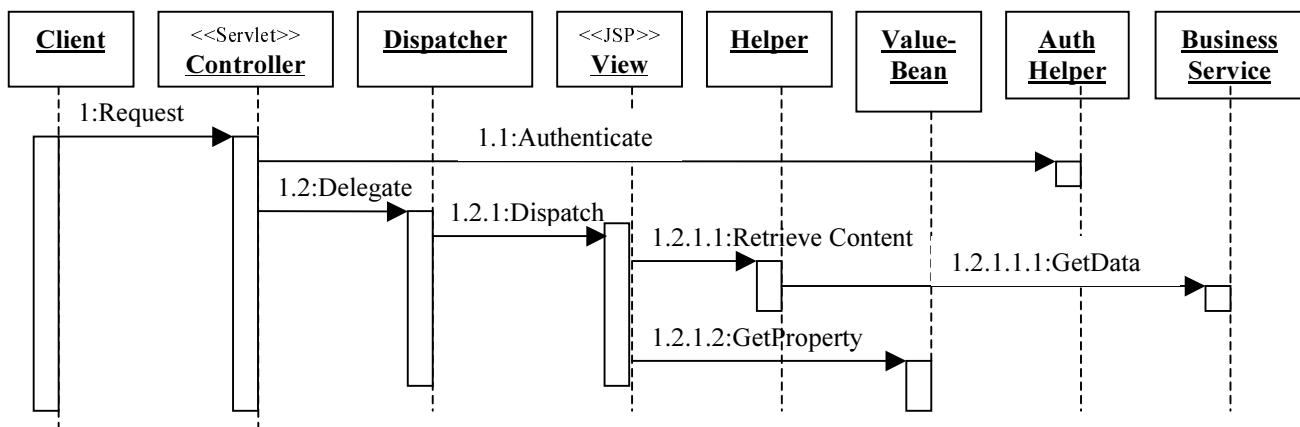
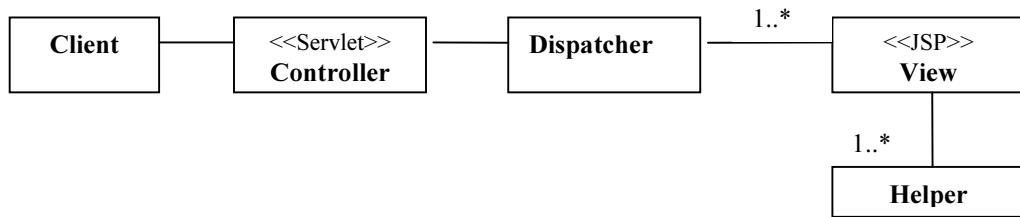
Strategies: ServletFront, JSP Front; JSP View; Servlet View; JavaBean Helper; Custom Tag Helper; Dispatcher in Controller; Transformer Helper.

Consequences: Centralizes control, improves modularity/reuse; improves partitioning of application; improves role separation for team.

Related Patterns: FrontController and View Helper, Dispatcher View.

Notes

Structure:

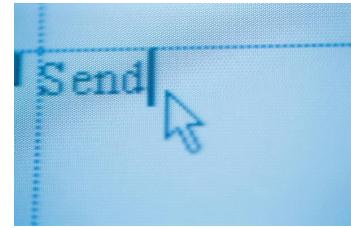


Participants/Responsibilities: Controller (initial point of contact for requests); Dispatcher (resp. for view mgmt/navig); View (represent/display info to client); Helper (processing for view or controller); ValueBean (helper responsible for state used by view); BusinessService (role fulfilled by service client wants).

Code: See *Core J2EE Patterns*, examples 7.34 - 7.35 (pages 231 – 245)

6. Dispatcher View (J2EE p. 231)

Context: System controls flow of execution and access to presentation processing, which is responsible for generating dynamic content.



Problem: Combination of problems solved by FrontController and ViewHelper patterns in presentation tier: no centralized component for managing access control, content retrieval, view management—plus duplicate scattered code and mixing of business/presentation logic in views.

Forces:

- Authentication/authorization checks completed per request;
- scriptlet code within views should be minimized;
- business logic should be encapsulated in non-view components;
- control flow is relatively complex and based on values from dynamic content
- view management logic is limited in complexity.

Solution: Combine a controller and dispatcher with views and helpers to handle client requests and prepare dynamic presentation. Combination of Front Controller and View Helper with Dispatcher component, dispatcher responsible for view management/navigation. Content retrieval deferred to time of view processing.

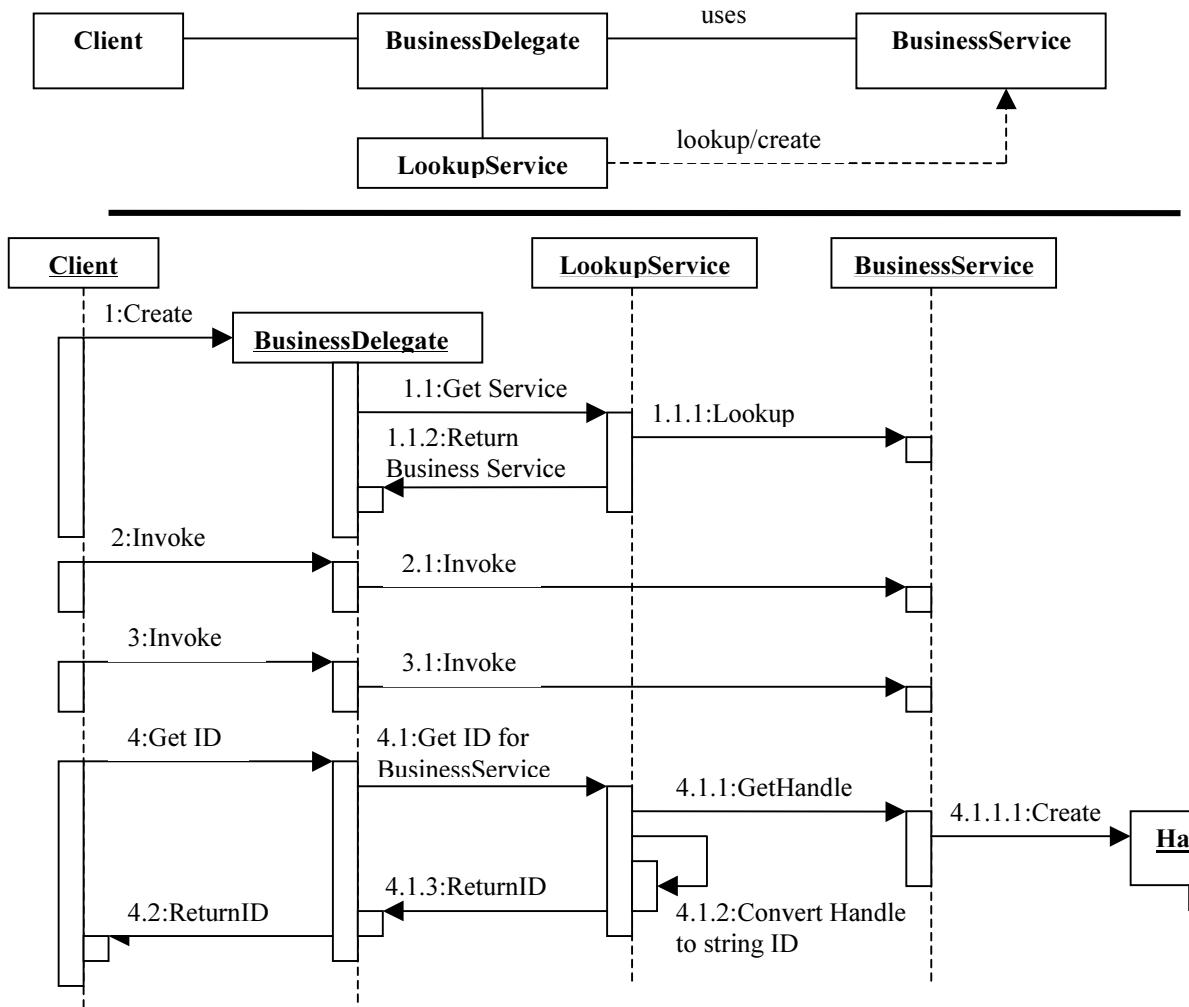
Strategies: ServletFront, JSP Front; JSP View; Servlet View; JavaBean Helper; Custom Tag Helper; Dispatcher in Controller, Dispatcher in View, Transformer Helper.

Consequences: Centralizes control and improves reuse and maintainability, improves application partitioning, improves role separation.

Related Patterns: Front Controller, View Helper, Service to Worker.

Notes

Structure:



Participants/Responsibilities: Business Delegate uses LookupService for locating the BusinessService (used to invoke methods for client). BusinessDelegate role is control and protection for BusinessService;.

Code: See *Core J2EE Patterns*, examples 8.1 - 8.2 (pages 248 - 260)

7. Business Delegate (J2EE p. 248)

Context: A multilayered, distributed system requires remote method invocations (RMI) to send/receive data across tiers; clients are exposed to the complexity of dealing with distributed components.



Problem: Presentation tier interacts directly with business services, exposing its implementation details. Therefore presentation-tier components vulnerable to changes in implementation of business service layer. Lots of invocations have performance impact.

Forces:

- Presentation-tier clients need access to business services
- different clients (devices, Web clients, thick clients) need access to business services
- want to minimize coupling between presentation and business services, hide details (e.g. lookup and access)
- clients may need to cache business service information
- want to reduce network traffic between client and business services

Solution: Use a Business Delegate to reduce coupling, hide implementation details. A client-side business abstraction.

Strategies: Delegate Proxy, Delegate Adapter (good fit in B2B when communicating with J2EE services, possibly using XML as integration language for disparate systems).

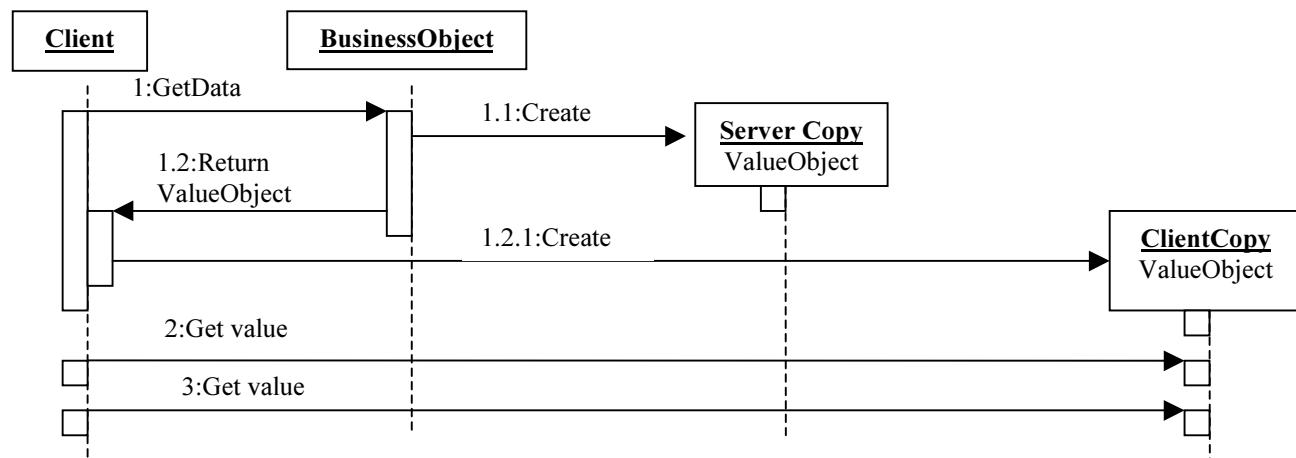
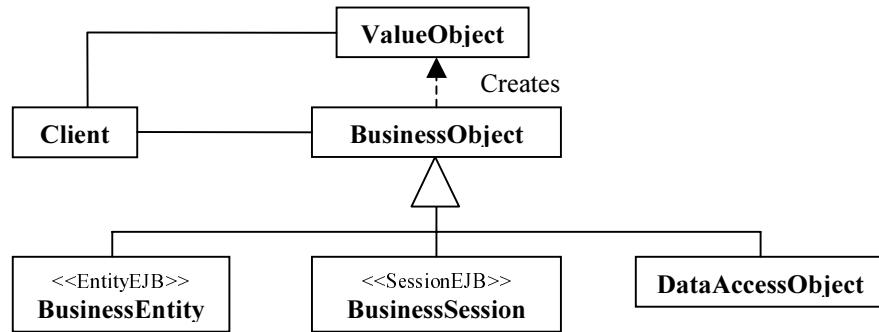
Consequences:

- reduces coupling, improves manageability
- translates Business Service exceptions
- implements failure recovery and thread synchronization
- exposes simpler, uniform interface to business tier
- impacts performance
- introduces additional layer
- hides remoteness

Related Patterns: Service Locator, Proxy, Adapter, Broker

Notes

Structure:



Participants/Responsibilities: Client (of enterprise bean)—can be end-user appl or Business Delegate, BusinessObject—can be session or entity bean or DAO; ValueObject—arbitrary serializable Java object

Code: See *Core J2EE Patterns*, examples 8.3 - 8.14 (pages 261 - 290)

8. Value Object (J2EE p. 261)

Context: Application clients need to exchange data with enterprise beans.

Problem: Multiple calls to “get” methods that return single attribute values is inefficient for obtaining data values from an enterprise bean.



Forces:

- all access to enterprise bean performed via remote interfaces (with overhead)
- applications have more read transactions than updates
- client usually requires values for several attributes or objects (multiple calls)
- increased traffic between client and server tiers degrades network performance

Solution: Use a Value Object to encapsulate the business data; a single method call sends and retrieves the value object.

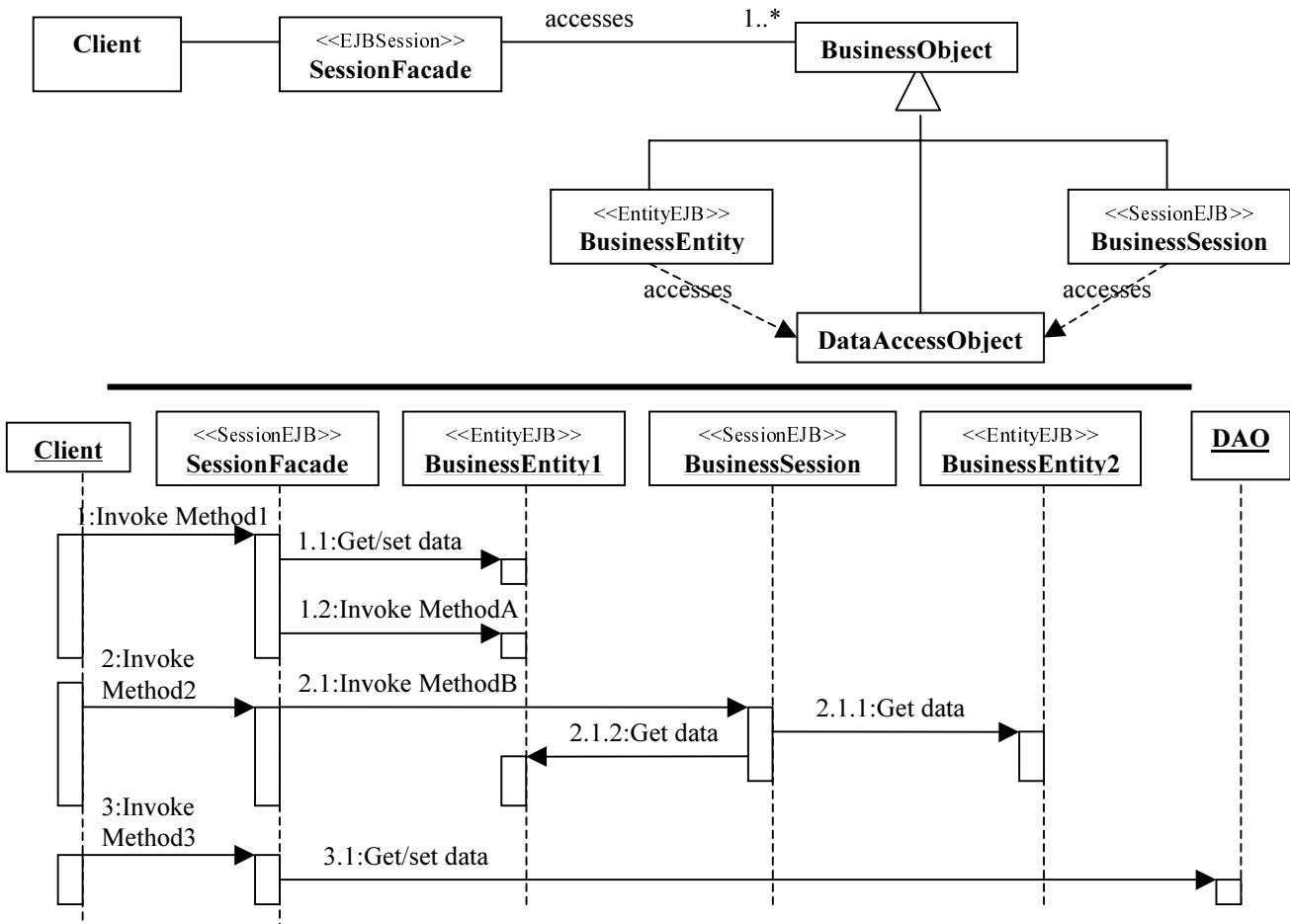
Strategies: Updatable Value Objects, Multiple Value Objects, Entity Inherits Value Object, Value Object Factory.

Consequences: simplifies entity bean and remote interface, transfers more data in fewer remote calls, reduces network traffic, reduces code duplication, may introduce stale value objects (cached invalid instances), may increase complexity due to synchronization and version control, concurrent access and transactions.

Related Patterns: Session Façade, Value Object Assembler, Value List Handler, Composite Entity.

Notes

Structure:



Participants/Responsibilities: SessionFacade is implemented as a session bean and manages relationships between various BusinessObjects—provides higher level abstraction to client (another session bean or Business Delegate).

Code: See *Core J2EE Patterns*, examples 8.15 - 8.17 (pages 291 - 309)

9. Session Facade (J2EE p. 291)

Context: Enterprise beans encapsulate business logic/data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.

Problem: In multitiered J2EE, tight coupling leads to direct dependencies; too many method invocations lead to performance problems; lack of uniform client access strategy exposes business objects to potential misuse.



Forces:

- provide simpler interface to clients, hiding complex interactions between business components
- reduce number of exposed business objects
- hide underlying interactions and interdependencies from client — better manageability and ease of change
- provide uniform coarse-grained service layer to separate implementation from abstraction of business
- avoid exposing underlying business objects directly to client; minimize coupling

Solution: Use a session bean as façade to encapsulate complexity of interactions between business objects in workflow.

Strategies: Business Objects can be either session or entity bean, DAO, or regular Java object; Session Façade can be stateless or stateful.

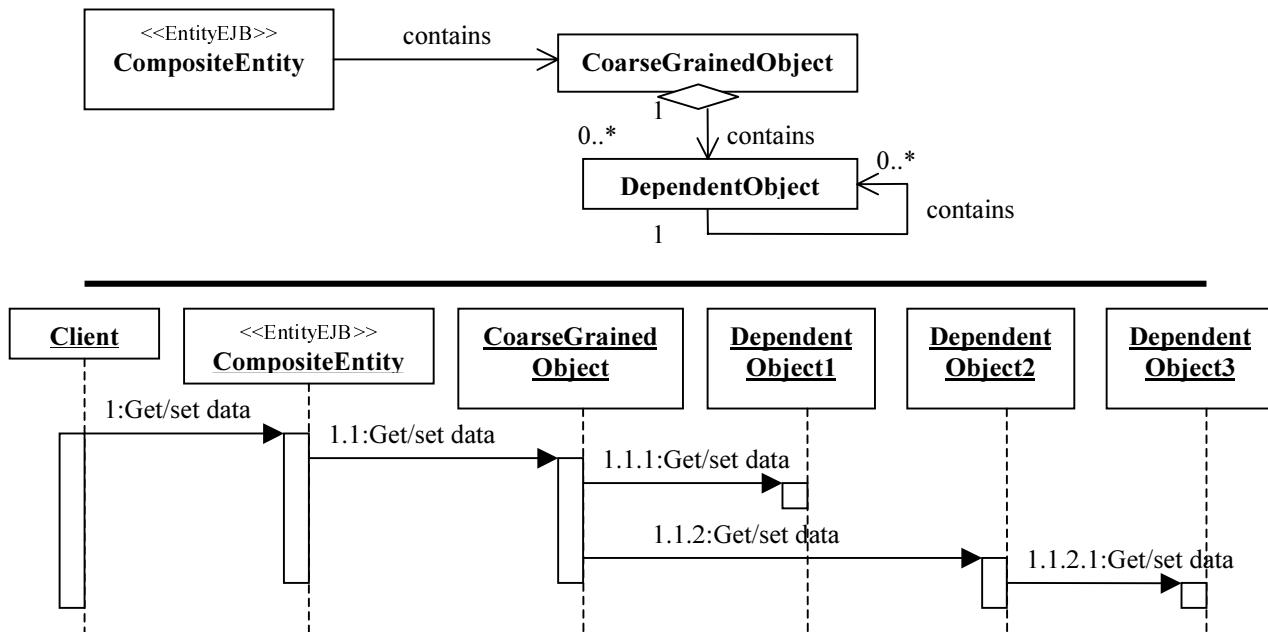
Consequences: introduces business-tier controller layer, exposes uniform interface, reduces coupling, increases manageability, improves performance by reducing fine-grained methods, provides coarse-grained access, centralizes security management, centralizes transaction control, exposes fewer remote interfaces to clients.

Related Patterns: Façade, Data Access Object, Service Locator, Business Delegate, Broker

- How would you identify Session Facades through studying use cases? (A one-to-one mapping will defeat the intention of having fewer coarse-grained session beans.) Consider, for example, a banking application...and aggregate a group of related interactions into a single Session Façade.

Notes

Structure:



Participants/Responsibilities: **CompositeEntity** (the coarse-grained object or a reference to it); **CoarseGrainedObject** (manages relationships to other objects); **DependentObjectxx** (has its life cycle managed by coarse-grained object) and may contain other dependent objects (thus creating a tree).

Code: See *Core J2EE Patterns*, examples 8.18 - 8.23 (pages 310 - 338)

10. Composite Entity (J2EE p. 310)

Context: Entity beans are not intended to represent every persistent object in the object model; entity beans are better as coarse-grained persistent business objects.



Problem: Every client invocation of entity beans potentially routes through network stubs and skeletons, even in same JVM, OS, or machine. Object granularity impacts data transfer; clients mostly need larger chunks.

Forces:

- entity beans best as coarse-grained objects
- apps that directly map RDBMS schema to entity beans have many fine-grained entity beans
 - direct mapping of object-to-EJB model yields fine-grained entity beans
 - clients don't need schema implementation to use/support entity beans; don't want schema/beans tightly coupled
 - fine-grained entity beans increases chattiness of applications—performance bottlenecks

Solution: Use Composite Entity to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained beans.

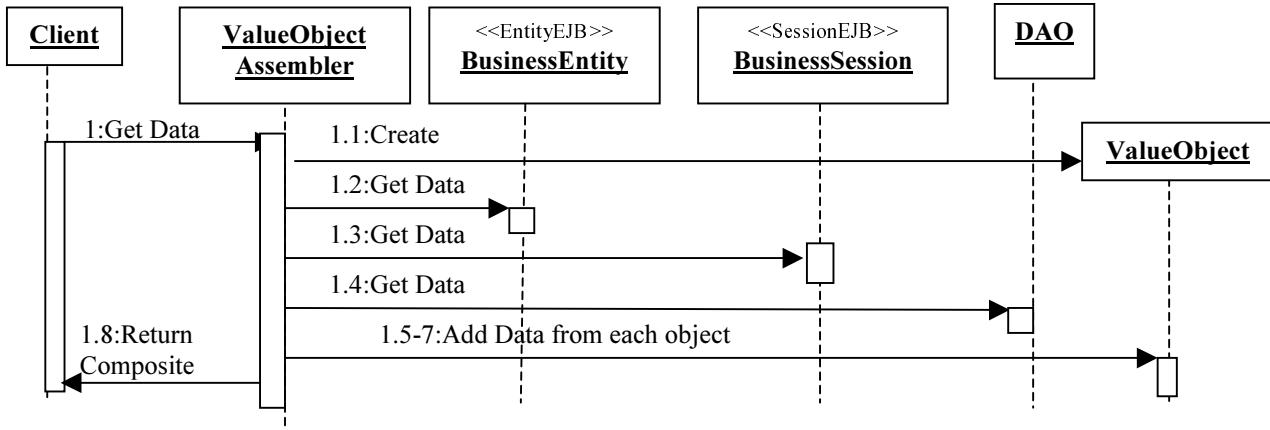
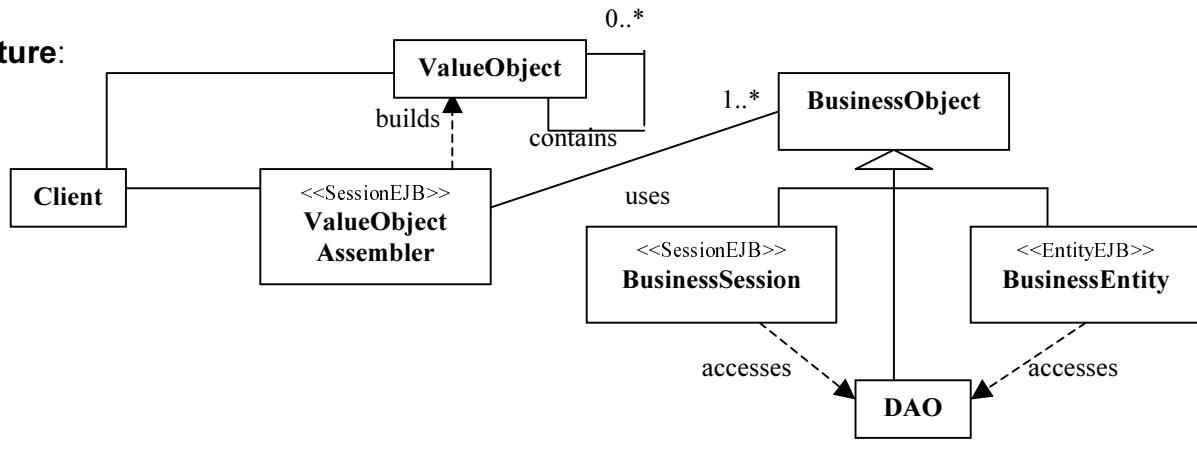
Strategies: Composite entity contains coarse-grained Object; Implements coarse-grained object; lazy loading strategy; store optimization (dirty marker); composite value object.

Consequences: eliminates inter-entity relationships, improves manageability by reducing entity beans, improves performance, reduces DB schema dependency, increases object granularity, facilitates composite value object creation, overhead of multi-level dependency object graphs.

Related Patterns: Value Object, Session Façade, Value Object Assembler.

Notes

Structure:



Participants/Responsibilities: ValueObjectAssembler is main class—constructs new value object as composite ValueObject; Client typically a Session Façade or Business Delegate; BusinessObject provides data to VOA.

Code: See *Core J2EE Patterns*, examples 8.24 - 8.28 (pages 339 - 352)

11. Value Object Assembler (J2EE p. 339)



Context: In a J2EE app, server-side business components are implemented using session beans, entity beans, DAOs, etc. App clients often need to access data that is composed from multiple objects.

Problem: Clients must access each distributed component individually, a tight coupling affecting maintenance. Network performance can degrade. Client needs necessary business logic to reconstruct the model from parts.

Forces:

- Need separation of business logic between client and server-side components,
- with distributed components, want to reduce number of RMI calls over network,
- client may only need to present completed model to user – extra chattiness undesirable,
- even with updates, usually only selected portions are updated, not entire model,
- clients don't need to be aware of low level dependencies,
- want clients to have minimal business logic.

Solution: Use a Value Object Assembler to build required model or submodel, using Value Objects to retrieve data from various parts of the model.

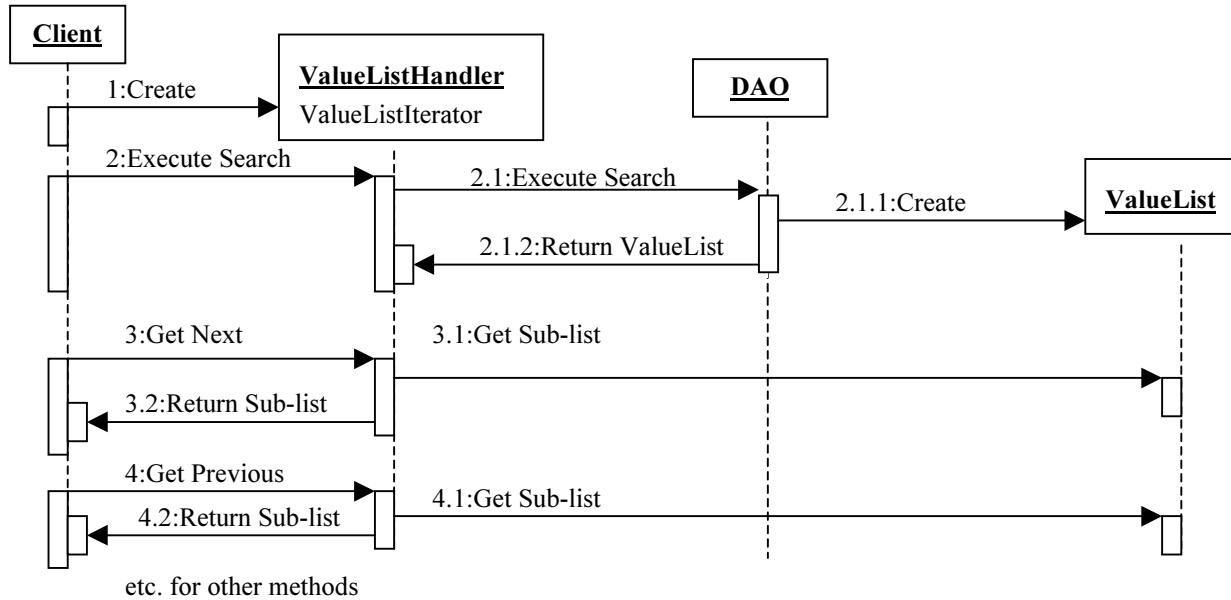
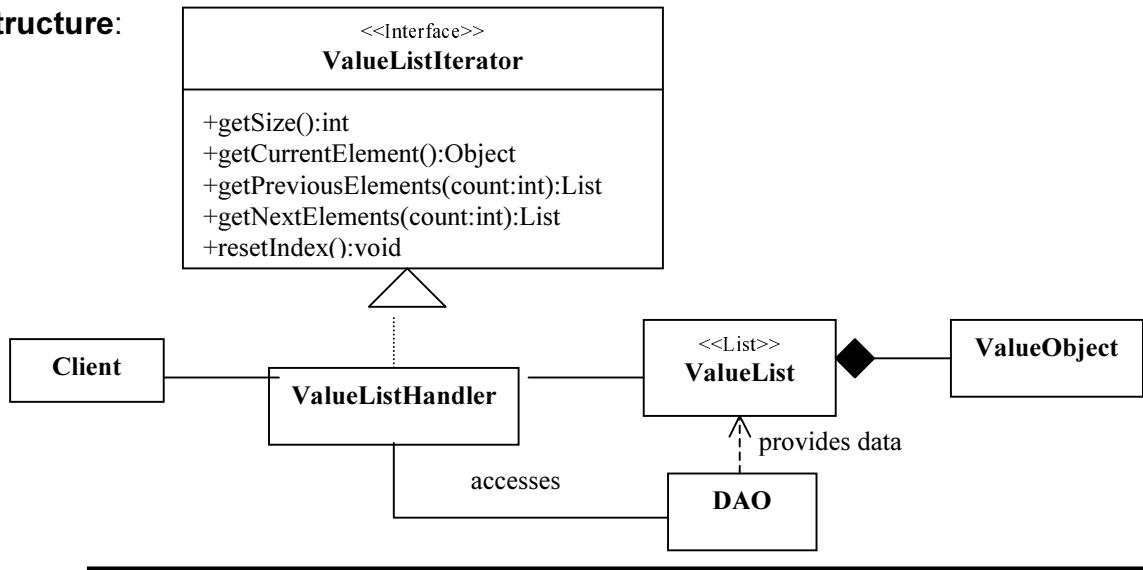
Strategies for implementing a Value Object Assembler: Java object, Session Bean, Business Object.

Consequences: Separates business logic, reduces coupling between clients and app model, improves network/client/ transaction performance, may introduce stale value objects.

Related Patterns: Value Object, Composite Entity, Session Façade, Data Access Object, Service Locator.

Notes

Structure:



Participants/Responsibilities: **ValueListHandler** interface (provides iteration facility), **ValueListHandler** (implements interface, executes queries/manipulates results), **DAO** (database access), **ValueList** (holds queries as **ValueObjects**).

Code: See *Core J2EE Patterns*, examples 8.29 - 8.32 (pages 353 - 366)

12. Value List Handler (J2EE p. 353)



Context: Client requires a list of items from the service for presentation. The number of items is unknown and could be large.

Problem: Most J2EE apps have a search and query requirement to search and list certain data—could yield large results. Impractical to return full set if client will traverse or only wants first few items, etc.

Forces:

- App needs efficient query facility to avoid querying each remote object;
- server-tier caching mechanism needed; repeated queries could be optimized;
- EJB finder methods not suitable for large result sets, have lots of overhead, can't cache results, and have preset query constructs;
- client wants to scroll forward/back within result set.

Solution: Use a Value List Handler to control the search, cache, provide results to client to meet client needs.

Strategies:

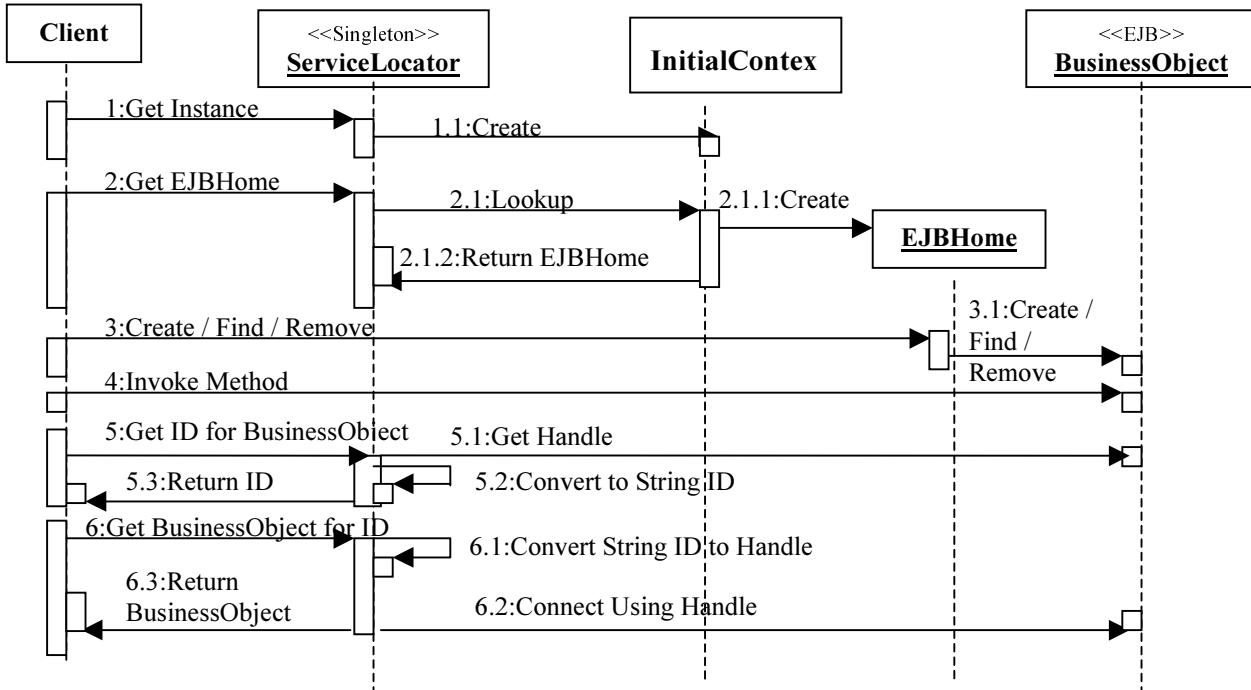
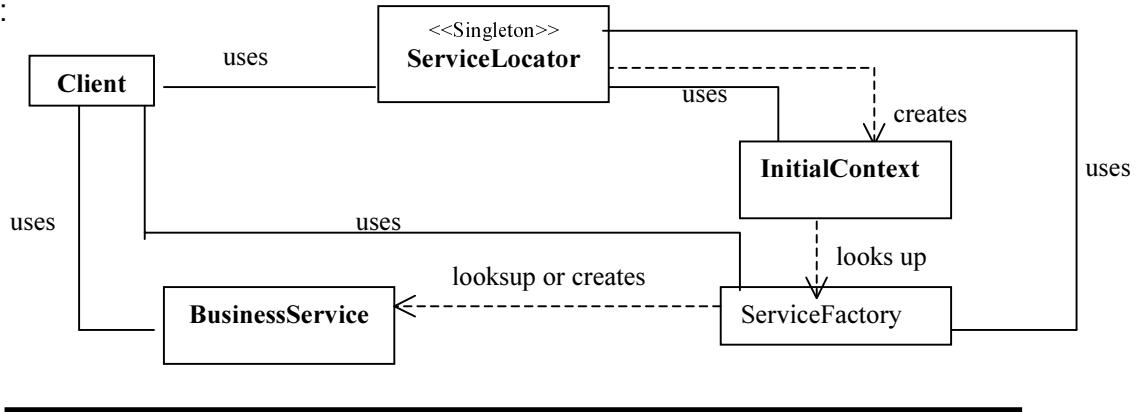
- Java object for apps not using EJBs but servlets, JSPs, etc.;
- stateful Session Beans.

Consequences: Alternative to EJB finders for large queries; caches query results on server side; better query flexibility; improves network performance; allows deferring entity bean transactions.

Related Patterns: Iterator, Session Façade.

Notes

Structure:



Participants/Responsibilities: Client (needs access to business objects), Service Locator (abstracts the API, vendor dependencies, lookup complexities, business object creation, and provides simple interface to clients), InitialContext (start point for lookup/creation), ServiceFactory (provides life cycle mgmt for BusinessService objects).

Code: See *Core J2EE Patterns*, examples 8.33 - 8.35 (pages 367 – 386)

13. Service Locator (J2EE p. 367)

Context: Service lookup/creation involves complex interfaces and network operations.

Problem: J2EE clients interact with service components to provide services and persistence; clients must locate the service component or create new one, JNDI facility services are common to all clients needing access to service objects.



Forces:

- EJB clients use JNDI API to look up EJBHome objects,
- JMS clients use JNDI to look up JMS components like connection factories, queues, topics,
- context factory for initial JNDI context is vendor-dependent and varies by object being looked up,
- lookup/creation of service components could be complex and need to be done multiple times,
- context creation and service lookups resource-intensive, impact performance,
- EJB clients may need to reestablish connection to previously accessed EJB instance.

Solution: Use a Service Locator to abstract all JNDI usage and hide complexities of context creation, EJB lookup and re-creation. Multiple clients can reuse Service Locator to reduce code complexity, give one point of control, cache.

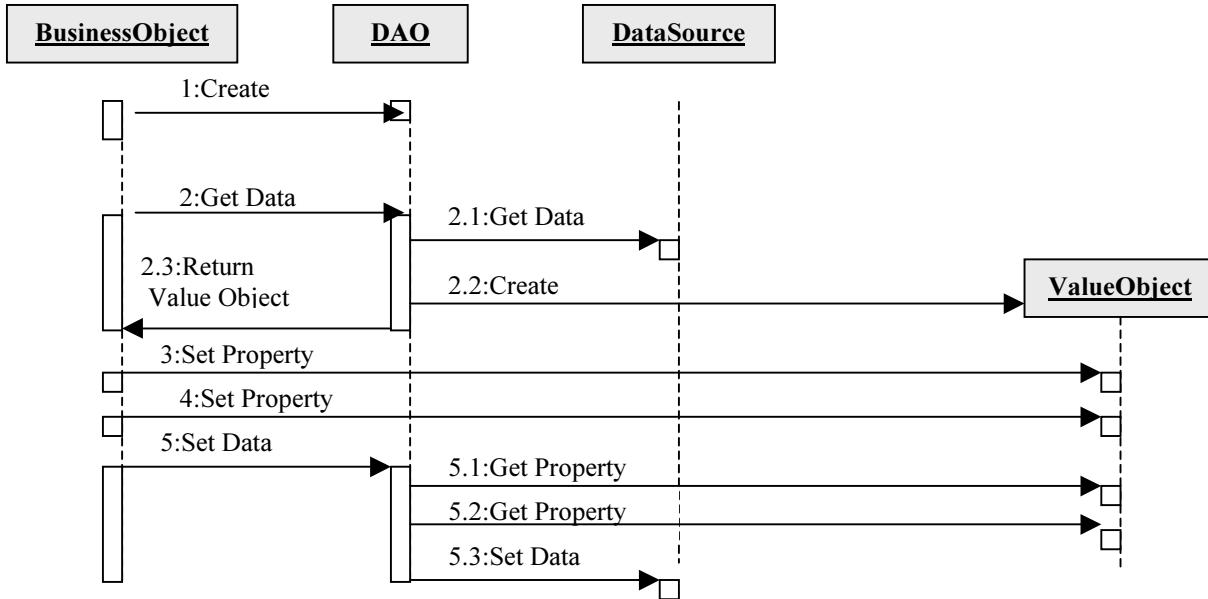
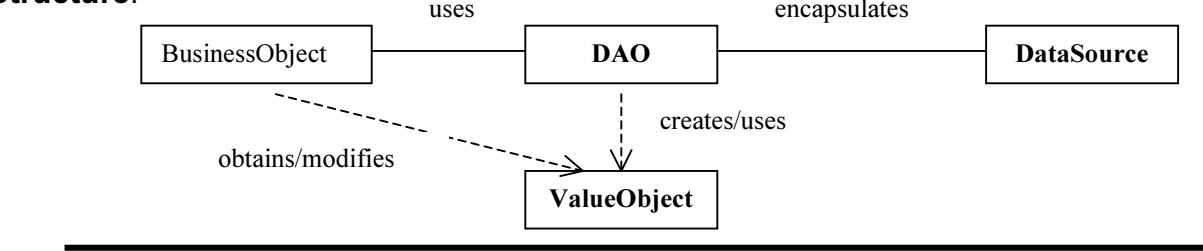
Strategies: EJBService Locator, JMS Queue Service Locator, JMS Topic Service Locator, Combined EJB/JMS Locator, Type Checked Service Locator, Service Locator Properties.

Consequences: Abstracts complexity, provides uniform service access to clients, facilitates addition of new business components, improves network/client performance.

Related Patterns: Business Delegate, Session Façade, Value Object Assembler.

Notes

Structure:



Participants/Responsibilities: BusinessObject (represents the data client), DAO (abstracts underlying data access implementation for BusinessObject), DataSource (DBMS, XML, flat file, etc), ValueObject (data carrier).

Code: See *Core J2EE Patterns*, examples 9.1 - 9.6 (pages 390 - 407)



14. Data Access Object (J2EE p. 390)

Context: Access to data varies depending on the source of the data. Access to persistent storage (database) varied greatly depending on type of DB and vendor.

Problem: J2EE apps need persistent data, implemented with varying mechanisms. Including connectivity in components makes them unreusable and hard to maintain.

Forces:

- Components (entity/ session beans, servlets, JSP helpers) to retrieve/store information from legacy systems,
- DB APIs vary by vendor – great lack of uniformity,
- components use proprietary APIs to access legacy systems,
- portability of components affected when access included in the components,
- components need to be transparent to data store for migration, varying storage/source types.

Solution: Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain/store data.

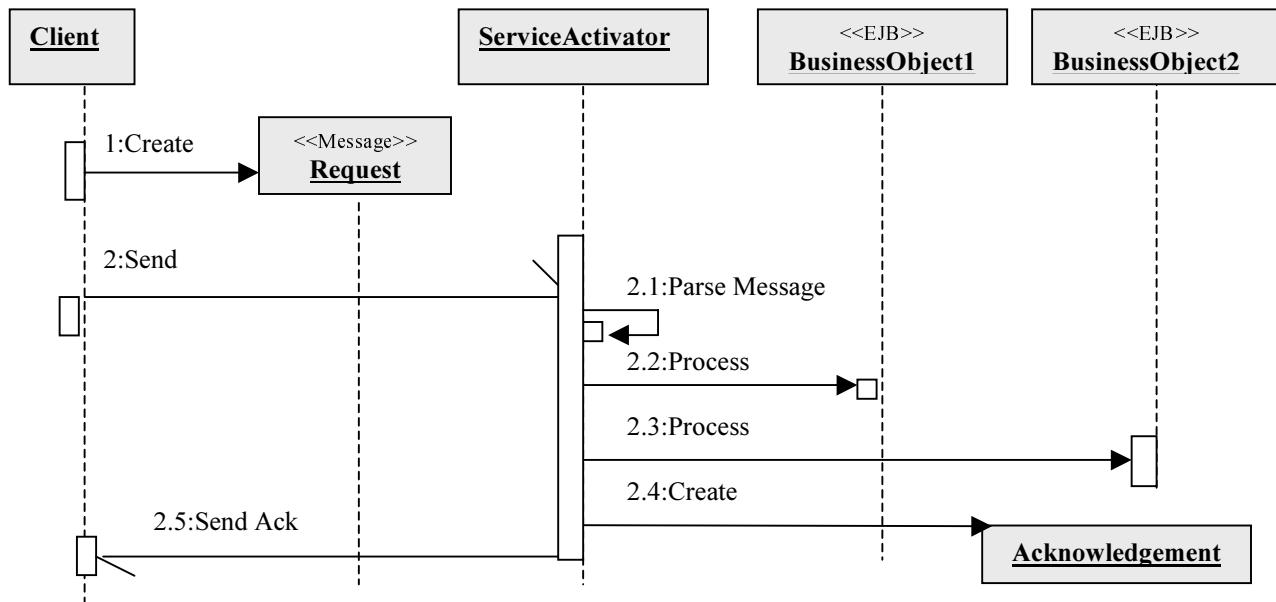
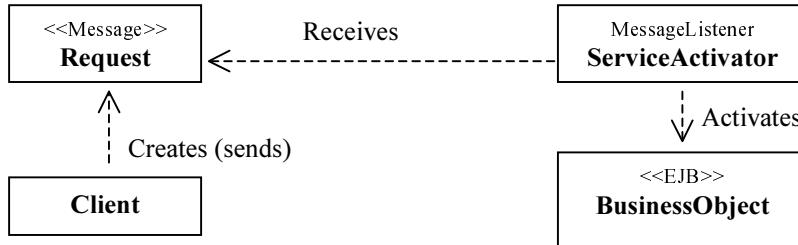
Strategies: Automatic DAO Code Gen, Factory for DAOs (Factory Method or Abstract Factory).

Consequences: Transparency, easier migration, reduced code complexity, centralized data access into separate layer, not useful for container-managed persistence, adds extra layer, needs class hierarchy design (if using DAO factory).

Related Patterns: Value Object, Factory Method, Abstract Factory, Broker.

Notes

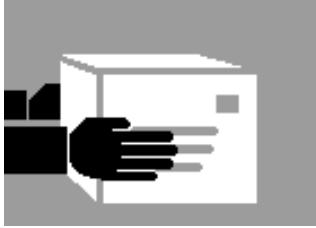
Structure:



Participants/Responsibilities: Client (requires asynch processing), Request (message created and sent) to Service Activator (implements JMS listener interface with onMessage() method), BusinessObject (target of client).

Code: See *Core J2EE Patterns*, examples 9.7 - 9.9 (pages 408 - 420)

15. Service Activator (J2EE p. 408)



Context: Enterprise beans and other business services need way to be activated asynchronously.

Problem: Client looks up bean's home object for access and gets remote reference, then invokes business method calls. All calls are synch and client must wait.

Forces:

- Enterprise beans exposed via remote interfaces with only synch access,
- container manages enterprise beans and does not allow direct access,
- appl needs to provide messaging framework so clients can make requests for asynchronous processing,
- clients need asynch processing to send requests without waiting for results,
- clients want to use message-oriented middleware interfaces of JMS but pre-EJB 2.0 products not integrated,
- appl needs to provide daemon-like service so bean can "rest" until event/message triggers activity,
- enterprise beans subject to container life cycle mgmt with passivation for time-outs, inactivity, resource mgmt,
- EJB 2.0 has message-driven bean as stateless session bean --not possible to invoke other types of beans asynch.

Solution: Use a Service Activator to receive asynchronous client requests/messages. When message received, Service Activator locates/invokes necessary business methods on business service components to do request asynchronously.

Strategies: Entity Bean, Session Bean, ServiceActivator server, Enterprise Bean as Client.

Consequences: Integrates JMS into pre-EJB 2.0, provides asynch processing for enterprise beans, standalone process.

Related Patterns: Session Façade, Business Delegate, Service Locator, Half-Synch/Half-Asynch [POSA2].

Notes

Chapter 11 – Selected Process Patterns (from PLoP)

Notes

PLoP is different than other conferences. Its focus is *improving* the expression of patterns. Authors have the opportunity to refine and extend their patterns with help from knowledgeable and sympathetic fellow pattern enthusiasts.

[See website: <http://jerry.cs.uiuc.edu/~plop/>]

"The PLoP conference is an excellent venue to get exposed to the practical use of patterns in OO programming. Not only is it recognition of the use of patterns but also an identification of patterns in pattern use. Being an open and free environment it is a great place to network and share ideas with some of the most talented people in the OO programming society.

There is genius in simplicity as I always say and George demonstrated that with his team building games. His games were simple in nature but provided a means for us to work together as a team in a very fun way. I left this conference with a better understanding of OO design, patterns of patterns, and a better gamer. I would highly recommend this conference to anyone wanting to further their skills in object-oriented technology."

Paul Saletzki

Chapter Objectives

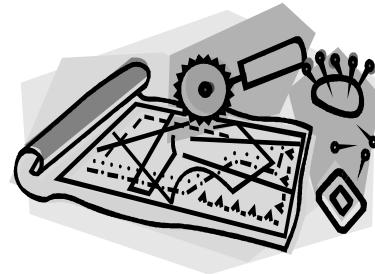
- Create incentive to study **other patterns** and **pattern sources**
- Learn about **PLoP**, the Pattern Languages of Programs
- Examine a small set of **PLoP patterns**

Notes

PLoP had their first conference in rural Illinois in August 1994. The Patterns Community provided software architects with something they never had before: an audience.

This group has produced many patterns. Some are already included in this course. Others are variations, address specific areas or domains, and there are even patterns on writing patterns. Problems in software design and programming are not the only problems faced by software engineers.

Note: I have included the Process Patterns section from PLoP, Volume III, as an example of other types of patterns available. **Remember that all patterns come from experience.**



PLoP Patterns

Process Patterns Overview

PLoP is an acronym for Pattern Languages of Program Design.

- **The Selfish Class:** code itself as a DNA-like propagation vehicle
- **Patterns for Evolving Frameworks:**
- **Patterns for Designing in Teams:**
- **Patterns for System Testing:**



Notes

Comments: Programs, and the artifacts from which they are built, have lifecycles that evolve within and beyond the applications that spawn them. Software is seldom built from the ground up anymore. What qualities encourage the reproduction of software?

Re: aisles and buttresses, it is the artifacts themselves which are the durable, evolving repositories of architectural insight—not the patterns existing in the minds of the architects.

Participants/ Related Patterns:

- Works Out of the Box (aka Batteries Included) – Design objects with reasonable behavior with default arguments. Provide everything a programmer needs; make it as easy as possible for designers to see a working example. Installation should be effortless and minimize user configurations.
- Low Surface-to-Volume Ratio (aka Simple Interface) – Design objects with small external interfaces that encapsulate a large volume of internal complexity.
- Gentle Learning Curve (aka Fold-Out Interfaces) – Design artifacts that allow users to start with a simple subset of capabilities, and permit them to gradually master more complex capabilities as they go along. Simple defaults, templates, examples – ability to drill-down later.
- Programming-By-Difference (aka Wrap It-Don't Bag It) – Use translators, subclasses and/or wrappers to supply new states or behavior while leaving the original artifact intact. The Open/Closed Principle – extend for new concepts without changing the old code.
- First One's Free (aka Freeware/Shareware; Netscape Now; Doom) – Give the artifact away.
- Winning Team (aka Piggybacking; Hitch Your Wagon to a Star) – Strive to become bundled with a popular platform. (Interestingly, every commercially successful object-oriented framework has been distributed with full source code.)



1. The Selfish Class (aka Software Darwinism, Plumage)



Pattern Thumbnail: A code's-eye view of software reuse and evolution (the authors' description). A code-level artifact must be able to attract programmers in order to survive and flourish.

Problem: Software artifacts that cannot attract programmers are not reused, and fade into oblivion.

Example: Aisle and buttress in many individual cathedrals.

Forces:

- Availability of existing, potentially reusable code, including cost.
- Utility – does the code do what you want?
- Suitability for task at hand.
- Comprehensibility – how easy to understand?
- Reliability. Fragility. Visibility.

Question: What might an object do to encourage programmers to (re-)use it, as opposed to using some other object, or building new ones?

Solution: Design artifacts that programmers will want to reuse. Make them widely available. Make sure they reliably solve a useful problem in a direct and comprehensible fashion.

Participants / Related Patterns:

- Works Out of the Box (aka Batteries Included)
- Low Surface-to-Volume Ratio (aka Simple Interface)
- Gentle Learning Curve (aka Fold-Out Interfaces)
- Programming-By-Difference (aka Wrap It-Don't Bag It)
- First One's Free (aka Freeware/Shareware; Netscape Now; Doom)
i.e., Give the artifact away.
- Winning Team (aka Piggybacking; Hitch Your Wagon to a Star)



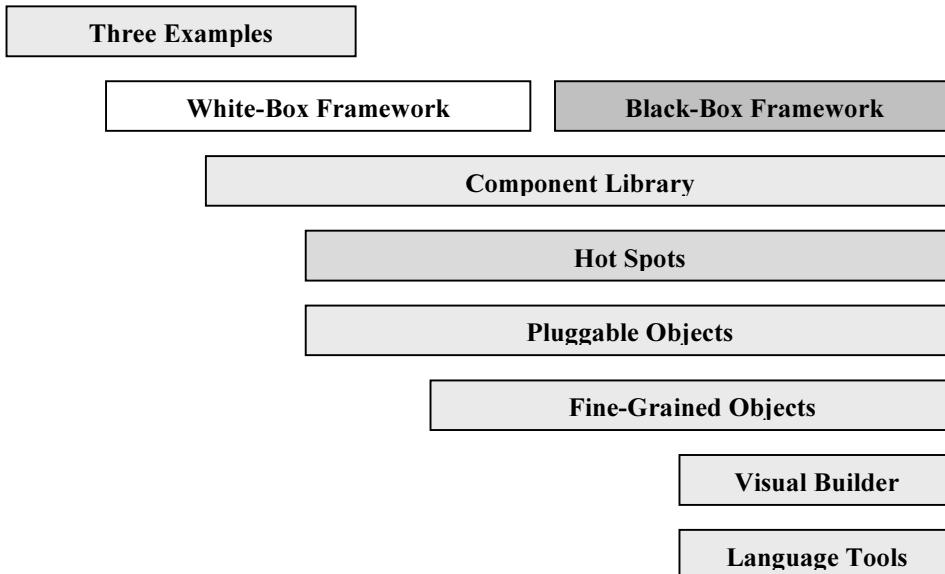
Consequences: Replicators behave as if their only interest is their own survival, to the exclusion of other considerations -- such as altruism or the good of a clan. Software that does not evolve will die.

Notes

Comments: Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate.

Example: A company experienced in developing control systems for ships sought to dominate a certain segment of its market. Using OO methods, the company developed a framework that it could adapt to the particular requirements of its expected customers. The resulting architecture was applied to ships from five different countries and represented a substantial return on investment for the company.

Diagram of Participant Patterns:



Expanded Solution: (1) Develop three applications you believe the framework should help you build. These applications should pay for their own development.

- (2) Use inheritance; build a white-box framework by generalizing classes.
- (3) Start a library of obvious objects and add as you need more.
- (4) Separate stable from changing code and encapsulate the variations using composition.
- (5) Design adaptable subclasses that can be parameterized with messages, indexes, blocks to evaluate, etc.
- (6) Continue breaking objects into finer granularities.
- (7) Use inheritance to organize the component library and composition to combine components into applications.
- (8) Make a graphical program that lets you specify the objects and interconnections for an application and which will generate code.
- (9) Create inspecting/debugging tools.

2. Patterns for Evolving Frameworks

Pattern Thumbnail: A set of patterns used together to address the problems of creating frameworks.

Problem: How do you start designing a framework?



Forces:

- People develop abstractions by generalizing from concrete examples.
- The more examples you look at, the more general your framework will be.
- Designing applications is hard—too many examples and you'll never finish
- Having a framework makes it easier to develop applications, even when the framework is only marginally useful.
- Even the first version makes it easier to develop more examples.
- Projects that take a long time to deliver anything tend to get canceled. There is usually a window of market opportunity that must be met.
- If the project doesn't start producing/saving money, the organization will run out of \$.

Question: What is a pattern language? How do these patterns (above) form a language?

Solution:

Develop three applications you believe the framework should help you build. These applications should pay for their own development.

Consequences:

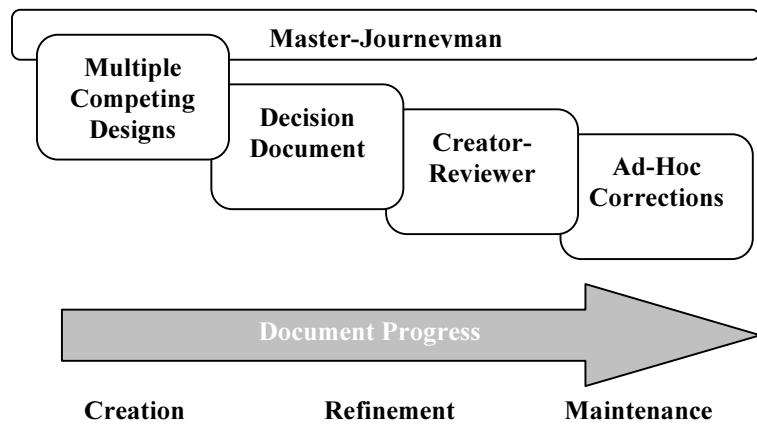
Positive: Reuse on a much bigger scale—both design and code. Development costs can be reduced by an order of magnitude.

Negative: This is hard work and the failure rate has been high.

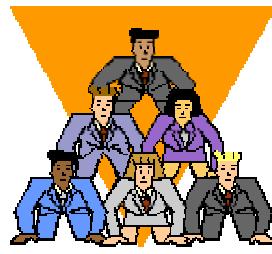
Notes

Comments: The purpose of analysis is to agree on the right thing to build. The purpose of design is to agree on how to build the thing right.

Diagram of five effective patterns:



3. Patterns for Designing in Teams



Pattern Thumbnail:

Techniques to use teams effectively in software design.

Forces:

- A team can think itself into a cul-de-sac—*inertia sets in*.
- There may be several possible options – how to choose?
- People make mistakes—it is difficult to see problems in your own work.
- Many projects are so large they must be partitioned to be approached, yet a unified design is necessary and a small team (or one) is the major system architect.
- Design projects are seldom up to date as documents fail to be updated.

Question: What teams you have been on worked well? Which ones didn't work? What were the differences?

Solution Patterns:

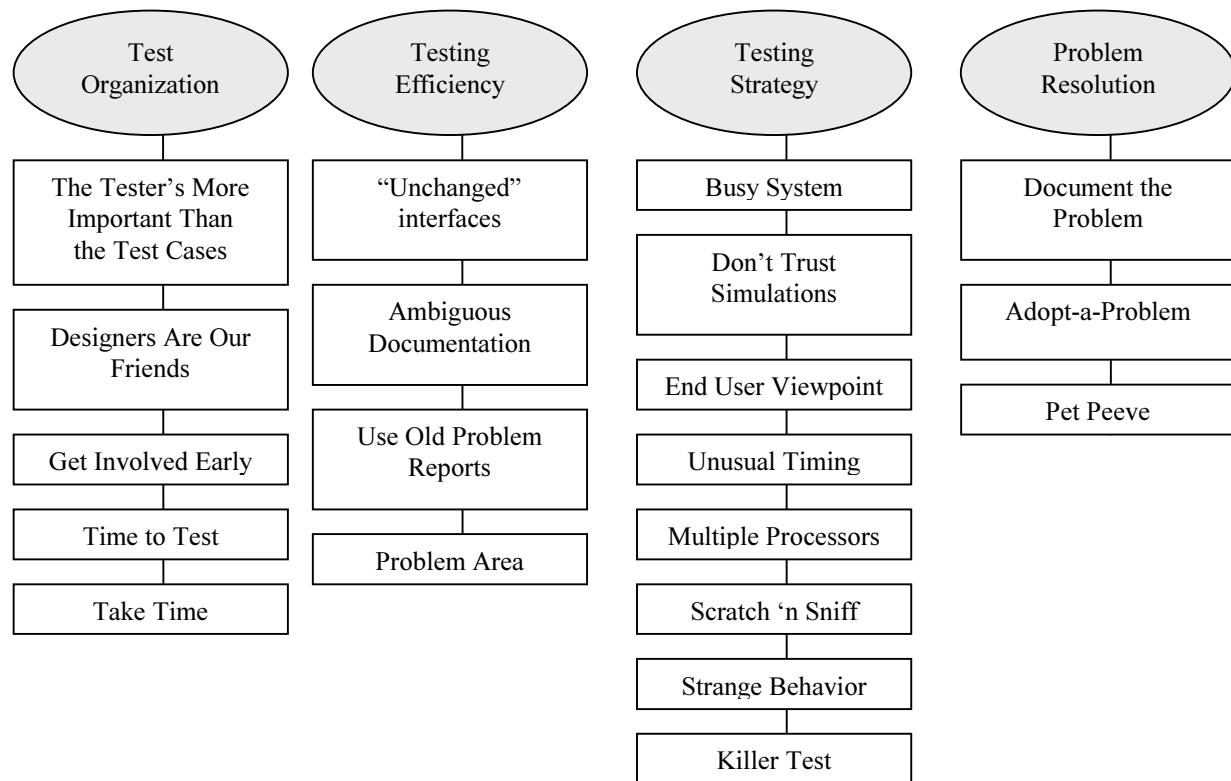
- Multiple Competing Designs:** Designers working independently or in pairs to generate a variety of design ideas.
 - Decision Document:** Produce a document via a facilitator to compare/contrast two different designs.
 - Creator-Reviewer:** Use different skills available to provide effective feedback to designers.
 - Master-Journeyman:** Divide a large development effort into manageable components to allow many design teams to work in parallel. Consider shared areas: core architecture, architectural vision, interfaces, specification control, error handling policy, coding guidelines, testing strategy, 3rd party libraries, multithreading strategy.
 - Ad-Hoc Corrections:** Use a corrections copy of a design document to coordinate design changes by different members of the implementation team.

Consequences: A variety of roles and skills are needed. Documentation is essential to capture/communicate decisions. Egos have to be put aside (or managed). Arbitrariness is taken out as the team searches for the best design. Information is available to the entire team.

Notes

Comments: Testing of systems is more of an art than a science.

Diagram of System Test Pattern Categories:



4. Patterns for System Testing



Pattern Thumbnail: A series of testing patterns.

Forces:

- The time for testing is short.
- Compressing the development schedule risks introducing problems.
- Compressing the test interval risks not finding all critical problems.
- Not all problems can be found.
- A release with some non-critical problems is acceptable.
- Specifications are often unclear or ambiguous.
- Problems should be found and corrected as early as possible.
- Reporting problems is often negatively viewed.
- Testing resources, such as prototype systems or person-hours, are at a premium.
- Software load stability is critical to progress.
- The content of a given software load is uncertain.
- Designers traditionally have a higher value within organization than system testers.
- Designers always want more time.
- Designers are primarily concerned with their own code; system testers must have a global view of the system and represent the customer and end users.
- End users don't always use features as specified by the customer.
- Etc., etc., etc.

Question: What's your best testing strategy?

Solution:

- Assign tasks to system testers based on experience and talent.
- Document problems,
- Don't attack designers; designers and testers approach them with a common goal.
- Start testing as soon as possible.
- Follow the patterns on the facing page.

Consequences: More upfront effort usually produces a better product in the end.

Notes

Chapter 12 – Selected Anti-Patterns

Notes

The AntiPatterns home page is located at the following website:
www.antipatterns.com.

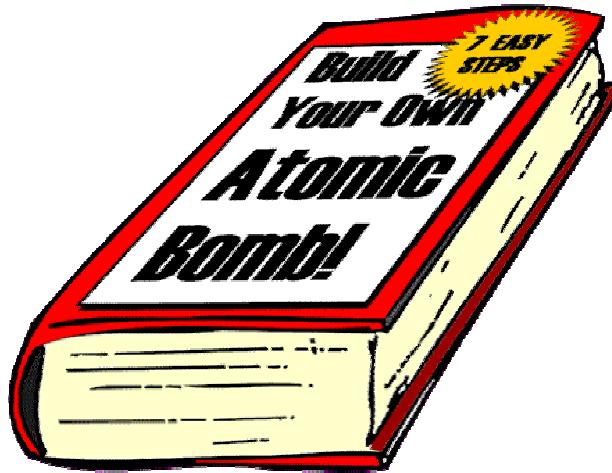
Chapter Objectives

- Point out that you can learn by **negative examples**
- Address many **existing practices**
- Understand that **refactoring** applies to more than just code
- Provide **stress release** in the form of shared misery for common problems

Notes

"AntiPatterns represent the latest concept in a series of revolutionary changes in computer science and software engineering thinking. As we approach the 50-year mark in developing programmable digital systems, the software industry has yet to resolve some fundamental problems in how humans translate business concepts into software applications. The emergence of design patterns has provided the most effective form of software guidance yet available, and the whole patterns movement has gone a long way in codifying a concise terminology for conveying sophisticated computer science thinking.

"While it is reasonable to assume that the principle reason we write software is to provide solutions to specific problems, it is also arguable that these solutions frequently leave us worse off before we started. In fact, academic researchers and practitioners have developed thousands of innovative approaches to building software: from exciting new technologies to progressive processes, but with all these great ideas, the likelihood of success for practicing managers and developers is grim.



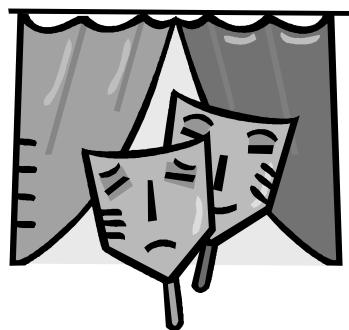
"A survey of hundreds of corporate software development projects indicated that five out of six software projects are considered unsuccessful. About a third of software projects are canceled. The remaining projects delivered software that was typically twice the expected budget and took twice as long to develop as originally planned [Johnson 95]. These repeated failures, or "negative solutions", are highly valuable, however in that they provide us with useful knowledge of what does not work, and through study: why. Such study, in the vernacular of Design Patterns can be classified as the study of Anti-Patterns.¹

¹ From the website, www.antipatterns.com



Anti-Patterns Overview

- **Negative patterns** of behavior exist in all walks of life
(especially software development – design, architecture, management, etc.)
- **Anti-Patterns** tell you what to avoid and how to fix it
- **Comedy** is the most serious **tragedy**



*... the state of software engineering today
is mostly a tragedy ...*

With patterns, benefits exceed consequences.
With anti-patterns, consequences exceed benefits.

Notes

- There is a tutorial on anti-patterns at the website.

1. Stovepipe System (aka: Legacy System, Uncle Sam Special, Ad Hoc Integration)

Most Frequent Scale: System

Refactored Solution Name: Architecture Framework

Refactored Solution Type: Software

Root Causes: Haste, Avarice, Ignorance, Sloth

Unbalanced Forces: Management of Complexity, Change

Anecdotal Evidence:

- The software project is way over budget;
- it has slipped its schedule repeatedly;
- my users still don't get the expected features;
- and I can't modify the system.

Every component is a stovepipe.

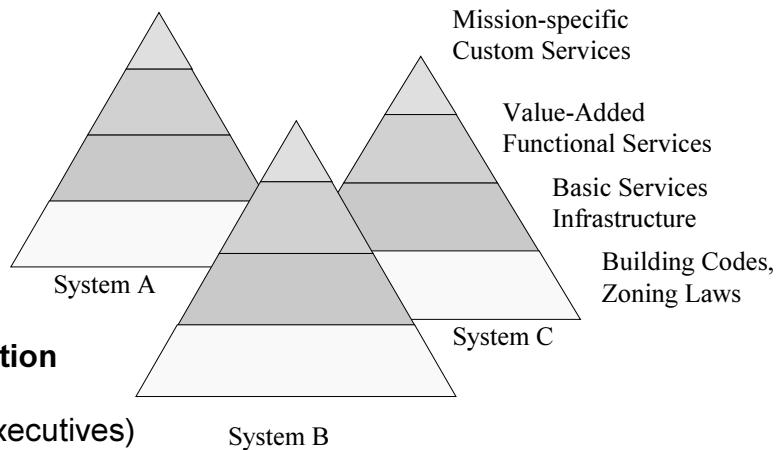


Notes

1b. Stovepipe Enterprise (aka: Islands of Automation)

Typical Causes:

- Lack of **Enterprise Technology Strategy**
- Lack of incentive for cooperation across system developments (competing business areas, executives)
- Lack of communication between system development projects
- Lack of knowledge of the technology standard being used
- Absence of horizontal interfaces in system integration solutions



Symptoms & Consequences:

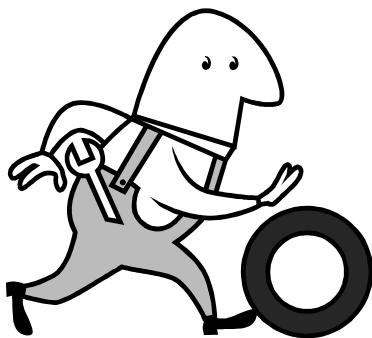
- Incompatible terminology, approaches, technology;
- brittle, monolithic system architectures and undocumented architectures;
- inability to extend systems to support business needs;
- incorrect use of a technology standard;
- lack of reuse between systems;
- lack of interoperability (even with same standards);
- excessive maintenance costs due to changes;
- employee turnover and project discontinuity with maintenance problems.

► *Describe a refactored solution:*

Notes

“Immature artists imitate. Mature artists steal.” -- Lionel Trilling

- **Certification** is required for many professionals – why not software architects?



2. Reinvent the Wheel

(aka Design in a Vacuum, Greenfield System—assuming a build from scratch)

Most Frequent Scale: System

Refactored Solution Name: Architecture Mining

Refactored Solution Type: Process

Root Causes: Pride, Ignorance

Unbalanced Forces: Management of Change, Technology Transfer

Anecdotal Evidence:

- “Our problem is unique.”
- Software developers generally have minimal knowledge of each other’s work.
- Even widely used software packages available in source code rarely have more than one experienced developer for each program.

Known Exceptions: This pattern may be suitable for a research environment where developers with different skills work at logically remote sites.

- *What would typical causes include?*
- *Describe symptoms and consequences?*
- *What would a refactored solution look like?*

Notes



3. Golden Hammer (aka: Old Yeller, Head-in-the-sand)

Most Applicable Scale: Application

Refactored Solution Name: Expand your horizons

Refactored Solution Type: Process

Root Causes: Ignorance, Pride, Narrow-Mindedness

Unbalanced Forces: Management of Technology Transfer

Anecdotal Evidence: "I have a hammer and everything else is a nail." "Our database is our architecture." "Maybe we shouldn't have used Excel macros for this job after all."

General Form: A software development team has gained competence in a particular solution or vendor product (the Golden Hammer). It may be a mismatch for the problem, but minimal effort is devoted to exploring alternative solutions.

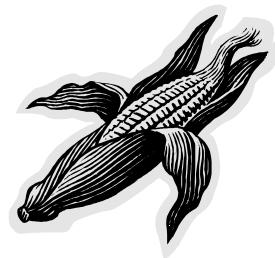
Typical Causes:

- Several successes have used this particular approach
- Large investment in training with product or technology
- Group is isolated from industry, other companies
- Reliance on proprietary product features not available in other industry products
- "Cornucob" (prevalent but difficult person) proposing solution

► *What would be an exception to this anti-Pattern?*

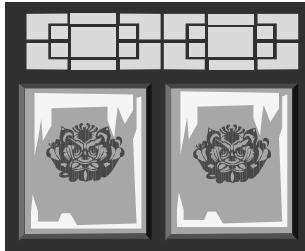
► *Describe possible symptoms and consequences:*

► *What would be a refactored solution?*



Notes

It is essential to baseline early and rarely. Otherwise the ability to track changes is lost.



4. Death by Planning (aka: glass Case Plan, Detailitis Plan)

Most Frequent Scale: Enterprise

Refactored Solution Name: Rational Planning

Refactored Solution Type: Process

Root Causes: Avarice, Ignorance, Haste

Unbalanced Forces: Management of Complexity

Anecdotal Evidence:

- "We can't get started until we have a complete program plan."
- "The plan is the only thing that will ensure our success."
- "As long as we follow the plan and don't diverge from it, we will be successful."
- "We have a plan ... just need to follow it."

Typical Causes/Symptoms:

- Lack of a pragmatic, **common-sense approach** to planning, to schedules and capture of progress
- Ignorance of **basic project-management principles**
- Sales aid** for contract acquisition
- Forced customer compliance** (or executive management)
- No up-to-date plan** that shows software component deliverables and their dates



- *Describe a refactored solution:*
- *How is the analysis paralysis anti-pattern related?*

Notes

Some cynics contend that all software projects are death-march projects...



5. Death March

Yourdon has described the death march project as one with unreasonable commitments -- any project with goals or resources that are scoped 50% outside of reasonable norms...

- **Schedule:** 50% too short
- **Staff:** half what's needed
- **Budget:** 50% too small
- **Number of features:** 50% greater than comparable successful projects.
- _____ (fill in the blanks)

► ***What's your best war story of this anti-pattern?***

► ***Describe a refactored solution:***

Notes

Note more books in addition to the original on anti-patterns (see references):

Anti-Patterns in Project Management and *Anti-Patterns and Patterns in Software Configuration Management*", both published by J.S. Wiley and Sons.

Refactoring IT Business Failure Before the Money is All Gone is coming out soon (as of this writing).

Additional Management AntiPatterns

- **Blowhard Jamboree:** influence of so-called industry experts
- **Viewgraph Engineering:** developers doing materials instead of real development
- **Smoke and Mirrors:** demonstration systems which create misperceptions
- **Throw It Over the Wall:** the code is finished! (no testing, no documentation)



- **Fire Drill:** months of boredom followed by demands for immediate delivery
- **E-mail is Dangerous:** (aka blame-storming)
email is an inefficient mode for complex/sensitive topics
- **The Feud:** personality conflicts between managers, aka dueling corncobs...
- **Intellectual Violence:** some expert uses knowledge to intimidate others in meetings
- **Project Mismanagement:** “All you need in this life is ignorance and confidence; then success is sure.” - Mark Twain



Notes

Chapter 13 – Patterns Summary

Notes

Chapter Objectives

- Summarize Patterns
- Final Exam!

Notes

Patterns Summary

None of us is as smart as all of us!

“Each pattern describes a problem that 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.”¹

- **Expertise captured and shared**
- **Communication improved** through well-defined vocabulary
- Design **decisions** and **code** take on new character:
 - **decisions** can be made more quickly
 - **trade-offs** naturally included
 - **documentation** made easier
- **Reuse** has a more flexible foundation

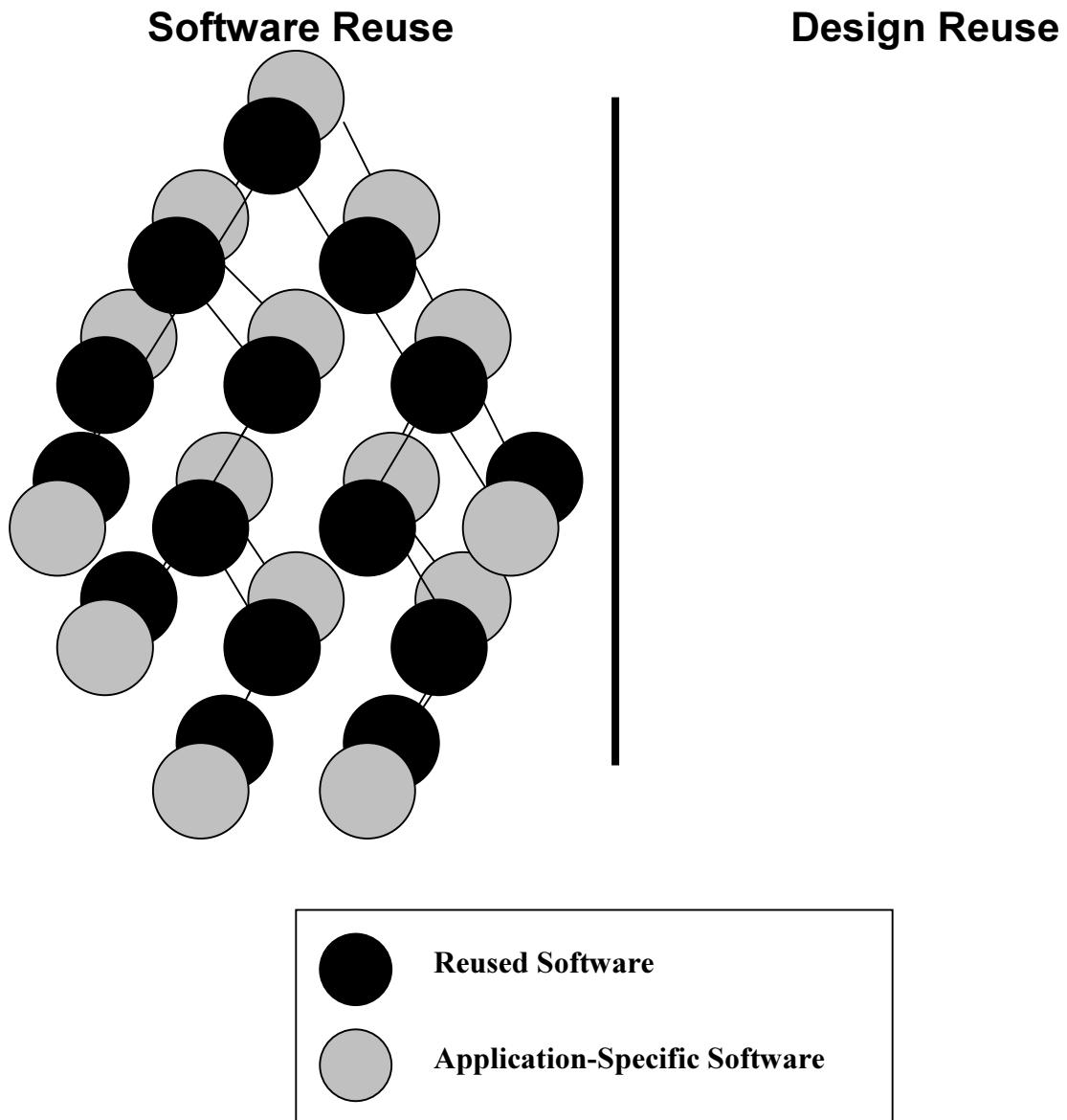


- Each application of a pattern **solves a problem**,
and creates a new context with new problems requiring other patterns
- **Patterns** are **artifacts** discovered in multiple systems

¹ Alexander, Christopher, et al., *A Pattern Language*, New York: Oxford University Press, 1977.

Notes

FINAL EXAM: Code Reuse vs. Design Reuse



- ▶ Explain the above diagram ...
- ▶ What is the significance on overall development cost of a system?

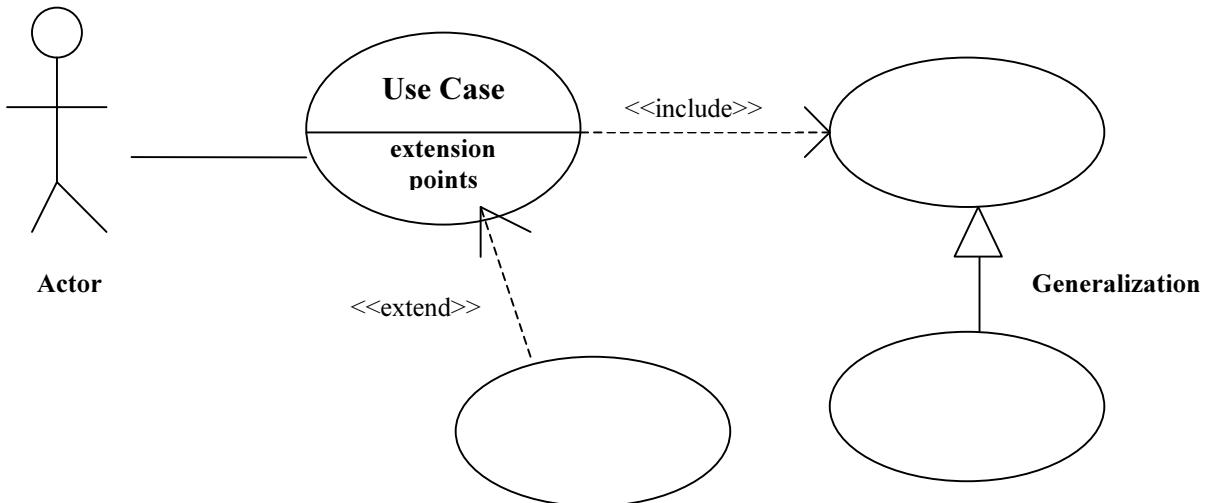
Notes

Appendix A – UML Review

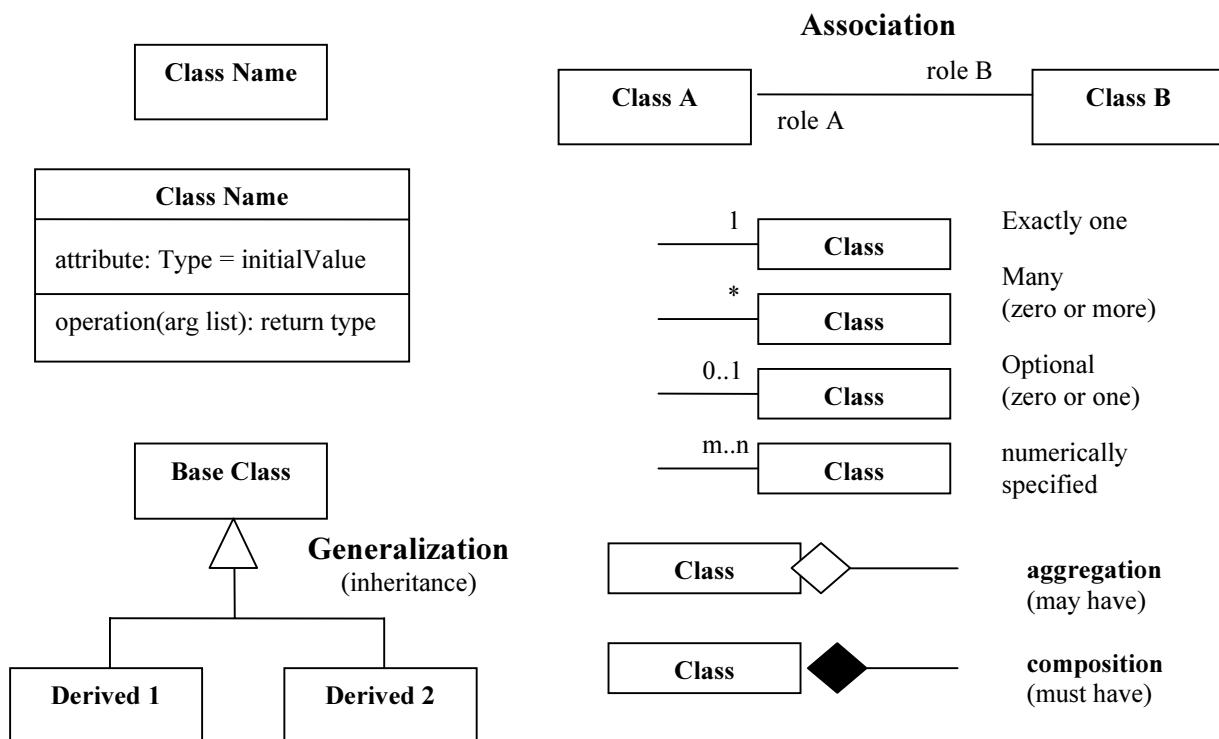
Notes

This appendix is not a complete set of UML notation, but includes enough for the purposes of this course.

Use Case Diagram notation

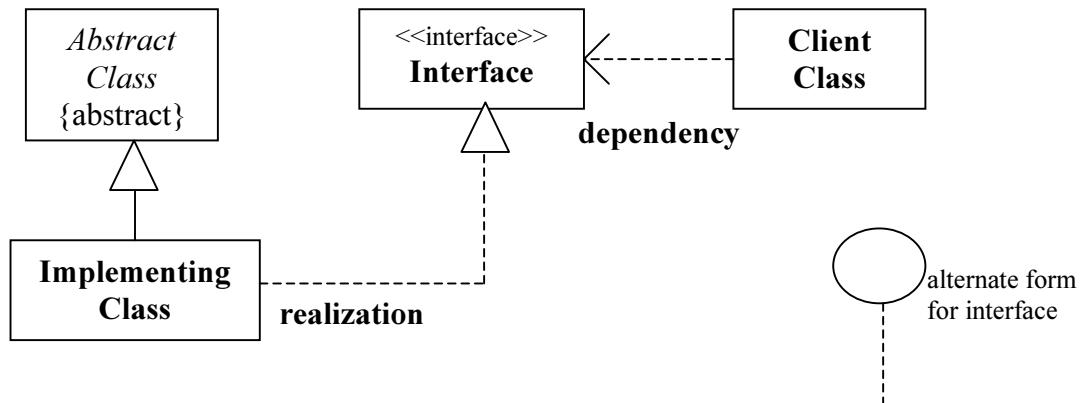


Class Diagram notation

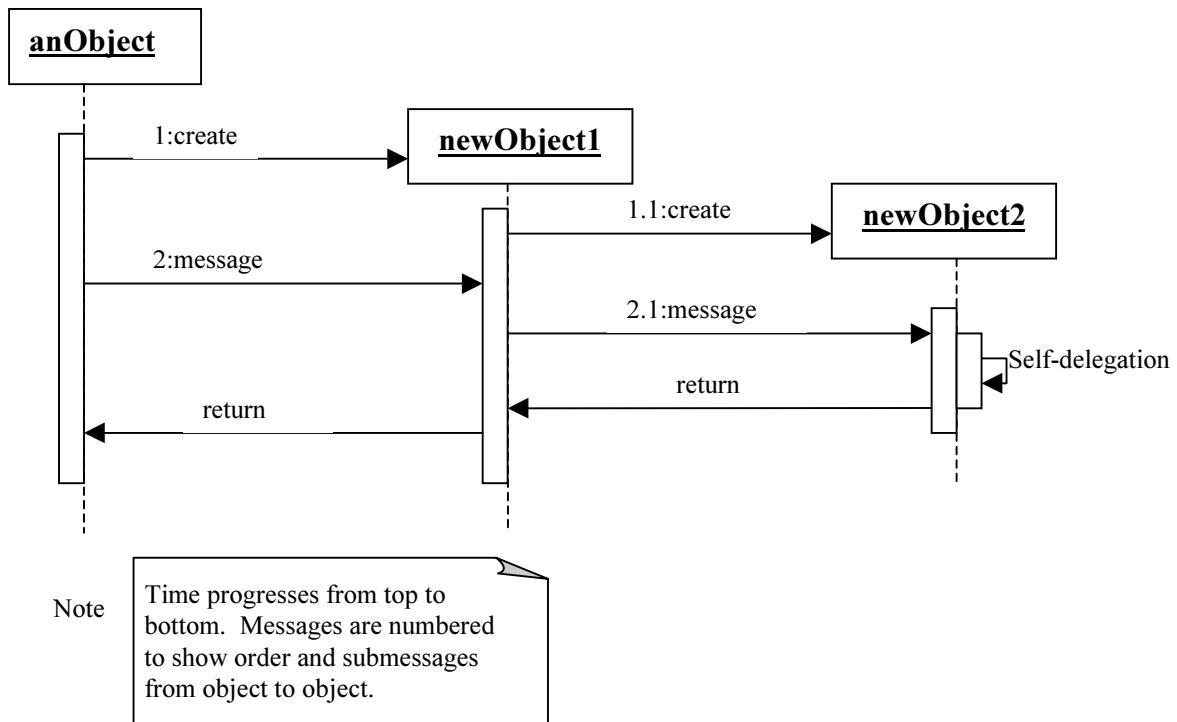


Notes

Class Diagram: Interface notation

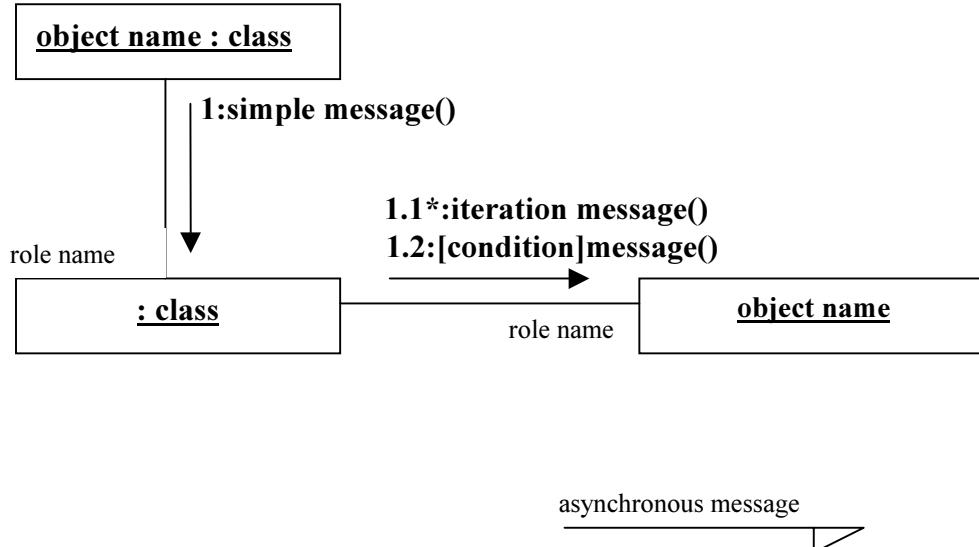


Sequence Diagram notation



Notes

Collaboration Diagram notation



Notes

Appendix B – GoF Code Examples in C#

Notes

// Abstract Factory

```

// Intent: "Provide an interface for creating families of related or
// dependent objects without specifying their concrete classes."

/* Notes:
 * When the construction needed involves many objects, possibly organized
 * in multi-faceted arrangements, the entire construction can be delegated
 * to an abstract factory. This exposes standardized creation functionality
 * which can be customized in concrete implementation to suit your specific
 * needs, and avoid embedding this information in higher level code - it
 * just needs to know how to call the abstract factory.
 *
 * In this sample, we have a framework with 3 abstract operating classes,
 * called DPDocument, DPWorkspace and DPView and one abstract construction
 * class, called DPFactory. An application-level class, called DPApplication
 * is responsible for construction.
 *
 * We have a series of application-level operating classes derived from this
 * framework - MyDocument, MyWorkspace and MyView. For design reasons we
 * assume we wish to instantiate these from inside DPApplication. As there
 * are multiple objects needed and they could be arranged in different
 * lattices, we use a factory, MyFactory (in our example, there are all
 * simple siblings), which is called inside DPApplication.
*/
namespace AbstractFactory_DesignPattern {
    using System; //similar to Java's package statement

    // These classes could be part of a framework,
    // which we will call DP
    // =====

    abstract class DPDocument {
        abstract public void Dump();
    }

    abstract class DPWorkspace {
        abstract public void Dump();
    }

    abstract class DPView {
        abstract public void Dump();
    }

    abstract class DPFactory {
        abstract public DPDocument CreateDocument();
        abstract public DPView CreateView();
        abstract public DPWorkspace CreateWorkspace();
    }

    abstract class DPApplication {
        protected DPDocument doc;
        protected DPWorkspace workspace;
        protected DPView view;

        public void ConstructObjects(DPFactory factory) {
            // Create objects as needed
            doc = factory.CreateDocument();
            workspace = factory.CreateWorkspace();
            view = factory.CreateView();
        }
    }
}

```

4 Design Patterns

```
abstract public void Dump();

public void DumpState() {
    if (doc != null) doc.Dump();
    if (workspace != null) workspace.Dump();
    if (view != null) view.Dump();
}

}

// These classes could be part of an application
class MyApplication : DPApplication {
    MyFactory myFactory = new MyFactory();

    override public void Dump() {
        Console.WriteLine("MyApplication exists");
    }

    public void CreateFamily() {
        MyFactory myFactory = new MyFactory();
        ConstructObjects(myFactory);
    }
}

class MyDocument : DPDocument {
    public MyDocument() {
        Console.WriteLine("in MyDocument constructor");
    }

    override public void Dump() {
        Console.WriteLine("MyDocument exists");
    }
}

class MyWorkspace : DPWorkspace {
    override public void Dump() {
        Console.WriteLine("MyWorkspace exists");
    }
}

class MyView : DPView {
    override public void Dump() {
        Console.WriteLine("MyView exists");
    }
}

class MyFactory : DPFactory {
    override public DPDocument CreateDocument() {
        return new MyDocument();
    }
    override public DPWorkspace CreateWorkspace() {
        return new MyWorkspace();
    }
    override public DPView CreateView() {
        return new MyView();
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        MyApplication myApplication = new MyApplication();
```

```

        myApplication.CreateFamily();
        myApplication.DumpState();
        return 0;
    }
}

// Adapter

// 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."

/* Notes:
 * Adapters are often used when client code is written to expect classes
 * from one framework, and it meant to work with classes from a totally
 * different framework. Assume you cannot change code of either framework.
 * The solution is for you to write an adapter, which appears like a
 * native class to each framework.
 *
 * There are two different types of adapters - class adapter and object
 * adapter. Class adapters are based on multiple inheritance - specifically
 * the interface of the target class and the implementation of the adaptee.
 * Unfortunately C# supports multiple inheritance for interfaces but not
 * for classes. Object adapters derive from the target (single inheritance)
 * and maintain a private instance of the adoptee.
 *
 * The sample code here shows an object adapter. We have a class called
 * FrameworkYAdaptee which we wish to use, yet the (bulk of) the client code
 * (GenericClientCode) is written to expect a class called FrameworkXTarget.
 * To solve the problem we create an Adapter class, FrameworkXTarget
 * to the client, and it calls FrameworkYAdaptee.
*/
namespace Adapter_DesignPattern {
    using System;

    class FrameworkXTarget {
        virtual public void SomeRequest(int x) {
            // normal implementation of SomeRequest goes here
        }
    }

    class FrameworkYAdaptee {
        public void QuiteADifferentRequest(string str) {
            Console.WriteLine("QuiteADifferentRequest = {0}", str);
        }
    }

    class OurAdapter : FrameworkXTarget {
        private FrameworkYAdaptee adaptee = new FrameworkYAdaptee();
        override public void SomeRequest(int a) {
            string b;
            b = a.ToString();
            adaptee.QuoteADifferentRequest(b);
        }
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {

```

6 Design Patterns

```
void GenericClientCode(FrameworkXTarget x) {
    // We assume this function contains client-side code that
    // only knows about FrameworkXTarget.
    x.SomeRequest(4);
    // other calls to FrameworkX go here
    // ...
}

public static int Main(string[] args) {
    Client c = new Client();
    FrameworkXTarget x = new OurAdapter();
    c.GenericClientCode(x);
    return 0;
}
}
```

// Bridge

```
// Intent: "Decouple an abstraction from its implementation so that the
// two can vary independently."

/* Notes:
 * Coupling between classes and class libraries is a major maintenance
 * headache. To ease this problem, often the client talks to an
 * abstraction description, which in turn calls an implementation.
 * Sometimes these must evolve - when one changes there can be a need
 * to change the other. The bridge design pattern lets the abstraction
 * and its implementation evolve separately.
 *
 * So, what is the difference between a bridge and an interface? Interfaces
 * can be used when creating bridges - but it should be noted that bridges
 * have additional possibilities. Both the abstraction and the
 * implementation may evolve over time and be the parent of derived classes.
 * The operations needed in the implementation could be defined in an
 * interface if there are no standard methods which are available at the
 * top-level of the implementation.
 */

namespace Bridge_DesignPattern {
    using System;

    class Abstraction {
        protected Implementation impToUse;

        public void SetImplementation(Implementation i) {
            impToUse = i;
        }

        virtual public void DumpString(string str) {
            impToUse.DoStringOp(str);
        }
    }

    class DerivedAbstraction_One : Abstraction {
        override public void DumpString(string str) {
            str += ".com";
            impToUse.DoStringOp(str);
        }
    }
}
```

```

class Implementation {
    public virtual void DoStringOp(string str) {
        Console.WriteLine("Standard implementation: print string as is");
        Console.WriteLine("string = {0}", str);
    }
}

class DerivedImplementation_One : Implementation {
    override public void DoStringOp(string str) {
        Console.WriteLine("DerivedImplementation_One - don't print
string");
    }
}

class DerivedImplementation_Two : Implementation {
    override public void DoStringOp(string str) {
        Console.WriteLine("DerivedImplementation_Two - print string twice");
        Console.WriteLine("string = {0}", str);
        Console.WriteLine("string = {0}", str);
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    Abstraction SetupMyParticularAbstraction() {
        // we localize to this method the decision which abstraction
        // and which implementation to use. These need to be decided
        // somewhere and we do it here. All the rest of the client
        // code can work against the abstraction object.
        Abstraction a = new DerivedAbstraction_One();
        a.SetImplementation(new DerivedImplementation_Two());
        return a;
    }

    public static int Main(string[] args) {
        Client c = new Client();
        Abstraction a = c.SetupMyParticularAbstraction();

        // From here on client code thinks it is talking to the
        // abstraction, and will not need to be changed as
        // derived abstractions are changed.

        // more client code using the abstraction goes here
        // ...
        a.DumpString("Clipcode");
        return 0;
    }
}
}

// Builder

// Intent: "Separate the construction of a complex object from its
// representation so that the same construction process can create
// different representations."

/* Notes:
 * Builder is an object creational design pattern that codifies the
 * construction process outside of the actual steps that carries out the
 * construction - thus allowing the construction process itself to be

```

8 Design Patterns

```
* reused.  
*/  
  
namespace Builder_DesignPattern {  
    using System;  
  
    // These two classes could be part of a framework,  
    // which we will call DP  
    // ======  
  
    class Director {  
        public void Construct(AbstractBuilder abstractBuilder) {  
            abstractBuilder.BuildPartA();  
            if (1==1) { //represents local decision inside director  
                abstractBuilder.BuildPartB();  
            }  
            abstractBuilder.BuildPartC();  
        }  
    }  
  
    abstract class AbstractBuilder {  
        abstract public void BuildPartA();  
        abstract public void BuildPartB();  
        abstract public void BuildPartC();  
    }  
  
    // These two classes could be part of an application  
    // ======  
  
    class ConcreteBuilder : AbstractBuilder {  
        override public void BuildPartA() {  
            // Create some object here known to ConcreteBuilder  
            Console.WriteLine("ConcreteBuilder.BuildPartA called");  
        }  
  
        override public void BuildPartB() {  
            // Create some object here known to ConcreteBuilder  
            Console.WriteLine("ConcreteBuilder.BuildPartB called");  
        }  
  
        override public void BuildPartC() {  
            // Create some object here known to ConcreteBuilder  
            Console.WriteLine("ConcreteBuilder.BuildPartC called");  
        }  
        /// <summary>  
        /// Summary description for Client.  
        /// </summary>  
        public class Client {  
            public static int Main(string[] args) {  
                ConcreteBuilder concreteBuilder = new ConcreteBuilder();  
                Director director = new Director();  
                director.Construct(concreteBuilder);  
  
                return 0;  
            }  
        }  
    }  
}
```

// Chain Of Responsibility

```

// 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."

/* Notes:
 * The client sends a request and it will be operated on by one of a number
 * of potential receivers, in a chain. The client does not know (and does
 * not need to know) which receiver handles the request. The receivers are
 * in a chain, and the request is passed from one to the next, until one
 * receiver actually performs the request.
 */

namespace ChainOfResponsibility_DesignPattern {
    using System;

    abstract class Handler {
        protected Handler successorHandler;
        abstract public void HandleRequest(Request request);
        public void SetSuccessor(Handler successor) {
            successorHandler = successor;
        }
    }

    class ConcreteHandler1 : Handler {
        override public void HandleRequest(Request request) {
            // determine if we can handle the request
            if (request.RequestType == 1){// complex decision making!
                // request handling code goes here
                Console.WriteLine("request handled in ConcreteHandler1");
            }
            else {
                // not handled here - pass on to next in the chain
                if (successorHandler != null)
                    successorHandler.HandleRequest(request);
            }
        }
    }

    class ConcreteHandler2 : Handler {
        override public void HandleRequest(Request request) {
            // determine if we can handle the request
            if (request.RequestType == 2){// complex decision making!
                // request handling code goes here
                Console.WriteLine("request handled in ConcreteHandler2");
            }
            else {
                // not handled here - pass on to next in the chain
                if (successorHandler != null)
                    successorHandler.HandleRequest(request);
            }
        }
    }

    class ConcreteHandler3 : Handler {
        override public void HandleRequest(Request request) {
            // determine if we can handle the request
            if (request.RequestType == 3){// complex decision making!
                // request handling code goes here
                Console.WriteLine("request handled in ConcreteHandler3");
            }
        }
    }
}

```

10 Design Patterns

```
        else {
            // not handled here - pass on to next in the chain
            if (successorHandler != null)
                successorHandler.HandleRequest(request);
        }
    }
}

class Request {
    private int iRequestType;
    private string strRequestParameters;

    public Request(int requestType, string requestParameters){
        iRequestType = requestType;
        strRequestParameters = requestParameters;
    }

    public int RequestType {
        get {
            return iRequestType;
        }
        set {
            iRequestType = value;
        }
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        // Set up chain (usually one need to be done once)
        Handler firstHandler = new ConcreteHandler1();
        Handler secondHandler = new ConcreteHandler2();
        Handler thirdHandler = new ConcreteHandler3();
        firstHandler.SetSuccessor(secondHandler);
        secondHandler.SetSuccessor(thirdHandler);

        // After setting up the chain of responsibility, we can
        // now generate requests and pass them off to the
        // chain to be handled

        // generate and fire request
        Request newRequest = new Request(2,"This are the request parameters");
        firstHandler.HandleRequest(newRequest);

        return 0;
    }
}
}
```

// Command

```
// Intent: "Encapsulate a request as an object, thereby letting you
// parameterize clients with different requests, queue or log
// requests, and support undoable operations."
```

```
/* Notes:
 * Commands are at the heart of a modern GUI app.
 * They must be nameable, undoable, recordable,
 * configurable, executable and repeatable.
```

```
*  
* In addition to the command here, a command type could also be useful.  
* It could store the name of a command, information about its icon, etc.  
*/  
  
namespace Command_DesignPattern {  
    using System;  
  
    abstract class Command {  
        abstract public void Execute();  
        protected Receiver r;  
        public Receiver R {  
            set {  
                r = value;  
            }  
        }  
    }  
  
    class ConcreteCommand : Command {  
        override public void Execute() {  
            Console.WriteLine("Command executed");  
            r.InformAboutCommand();  
        }  
    }  
  
    class Receiver {  
        public void InformAboutCommand() {  
            Console.WriteLine("Receiver informed about command");  
        }  
    }  
  
    class Invoker {  
        private Command command;  
        public void StoreCommand(Command c) {  
            command = c;  
        }  
        public void ExecuteCommand() {  
            command.Execute();  
        }  
    }  
  
    /// <summary>  
    /// Summary description for Client.  
    /// </summary>  
    public class Client {  
        public static int Main(string[] args) {  
            // Set up everything  
            Command c = new ConcreteCommand();  
            Receiver r = new Receiver();  
            c.R = r;  
            Invoker i = new Invoker();  
            i.StoreCommand(c);  
  
            // now let application run  
  
            // the invoker is how the command is exposed for end-user  
            // (or a client) initiates the command,  
            // (e.g. toolbar button, menu item)  
            i.ExecuteCommand();  
            return 0;  
        }  
    }  
}
```

```

// Composite

// Intent: "Compose objects into tree structures to represent part-whole
// hierarchies. Composite lets clients treat individual objects and
// compositions of objects uniformly."

/* Notes:
 * We often have tree structures where some nodes are containers of
 * other nodes, and some nodes are leafs. Rather than have separate client
 * code to manage each type of node, Composite lets a client work with
 * either using the same code.
 *
 * There is an excellent discussion to be found in "Pattern Hatching",
 * Vlissides, ISBN: 0-201-43293-5, describing how the Composite design
 * pattern can be used as the basis file system management
 * (directory = composite, Leaf = file, Component = iNode)
 */

namespace FactoryMethod_DesignPattern {
    using System;
    using System.Collections;

    abstract class Component {
        protected string strName;

        public Component(string name) {
            strName = name;
        }

        abstract public void Add(Component c);
        public abstract void DumpContents();
        // other operations for delete, get, etc.
    }

    class Composite : Component {
        private ArrayList ComponentList = new ArrayList();

        public Composite(string s) : base(s) {}

        override public void Add(Component c) {
            ComponentList.Add(c);
        }

        public override void DumpContents() {
            // First dump the name of this composite node
            Console.WriteLine("Node: {0}", strName);

            // Then loop through children; get them to dump contents
            foreach (Component c in ComponentList) {
                c.DumpContents();
            }
        }
    }

    class Leaf : Component {
        public Leaf(string s) : base(s) {}

        override public void Add(Component c) {
            Console.WriteLine("Cannot add to a leaf");
        }
    }
}

```

```

        public override void DumpContents() {
            Console.WriteLine("Node: {0}", strName);
        }
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        Component SetupTree() {
            // here we have to create a tree structure,
            // consisting of composites and leafs.
            Composite root = new Composite("root-composite");
            Composite parentcomposite;
            Composite composite;
            Leaf leaf;

            parentcomposite = root;
            composite = new Composite("first level - first sibling - composite");
            parentcomposite.Add(composite);
            leaf = new Leaf("first level - second sibling - leaf");
            parentcomposite.Add(leaf);
            parentcomposite = composite;
            composite = new Composite("second level - first sibling - composite");
            parentcomposite.Add(composite);
            composite = new Composite("second level - second sibling - composite");
            parentcomposite.Add(composite);

            //we will leave the second level - first sibling empty, and
            // start populating the second level - second sibling
            parentcomposite = composite;
            leaf = new Leaf("third level - first sibling - leaf");
            parentcomposite.Add(leaf);

            leaf = new Leaf("third level - second sibling - leaf");
            parentcomposite.Add(leaf);
            composite = new Composite("third level - third sibling - composite");
            parentcomposite.Add(composite);

            return root;
        }
    }

    public static int Main(string[] args) {
        Component component;
        Client c = new Client();
        component = c.SetupTree();

        component.DumpContents();
        return 0;
    }
}
}

```

// Decorator

```

// Intent: "Attach additional responsibilities to an object dynamically.
// Decorators provide a flexible alternative to subclassing for
// extending functionality."

/* Notes:
 * Dynamically wrap functionality with additional functionality, so that
 * the client does not know the difference.

```

14 Design Patterns

```
*/  
  
namespace Decorator_DesignPattern {  
    using System;  
  
    abstract class Component {  
        public abstract void Draw();  
    }  
  
    class ConcreteComponent : Component {  
        private string strName;  
        public ConcreteComponent(string s) {  
            strName = s;  
        }  
  
        public override void Draw() {  
            Console.WriteLine("ConcreteComponent - {0}", strName);  
        }  
    }  
  
    abstract class Decorator : Component {  
        protected Component ActualComponent;  
  
        public void SetComponent(Component c) {  
            ActualComponent = c;  
        }  
        public override void Draw() {  
            if (ActualComponent != null)  
                ActualComponent.Draw();  
        }  
    }  
  
    class ConcreteDecorator : Decorator {  
        private string strDecoratorName;  
        public ConcreteDecorator (string str) {  
            // how decoration occurs is localized inside this decorator  
            // For this demo, we simply print a decorator name  
            strDecoratorName = str;  
        }  
        public override void Draw() {  
            CustomDecoration();  
            base.Draw();  
        }  
        void CustomDecoration() {  
            Console.WriteLine("In ConcreteDecorator: decoration goes here");  
            Console.WriteLine("{0}", strDecoratorName);  
        }  
    }  
  
    /// <summary>  
    /// Summary description for Client.  
    /// </summary>  
    public class Client  
    {  
        Component Setup() {  
            ConcreteComponent c = new ConcreteComponent("This is the real  
component");  
            ConcreteDecorator d = new ConcreteDecorator("This is a decorator for  
the component");  
            d.SetComponent(c);  
            return d;  
        }  
    }  
}
```

```

public static int Main(string[] args) {
    Client client = new Client();
    Component c = client.Setup();
    // The code below will work equally well with the real
    // component, or a decorator for the component

    c.Draw();

    return 0;
}
}

// Facade

// Intent: "Provide a unified interface to a set of interfaces in a
// subsystem. Facade defines a higher-level interface that makes the
// subsystem easier to use."

/* Notes:
 * Many subsystems are complex to manage - and this is a disincentive for
 * client code to use them. A Facade design pattern provides a simplified
 * high-level API to the client, thus shielding it from direct contact
 * with the (more complex) subsystem classes.
 *
 * Often the code that is inside a facade would have to be inside client
 * code without the facade. The subsystem code beneath the facade can
 * change, without affecting the client code.
 */

namespace Facade_DesignPattern {
    using System;

    class SubSystem_class1 {
        public void OperationX() {
            Console.WriteLine("SubSystem_class1.OperationX called");
        }
    }

    class SubSystem_class2 {
        public void OperationY() {
            Console.WriteLine("SubSystem_class2.OperationY called");
        }
    }

    class SubSystem_class3 {
        public void OperationZ() {
            Console.WriteLine("SubSystem_class3.OperationZ called");
        }
    }

    class Facade {
        private SubSystem_class1 c1 = new SubSystem_class1();
        private SubSystem_class2 c2 = new SubSystem_class2();
        private SubSystem_class3 c3 = new SubSystem_class3();

        public void OperationWrapper() {
            Console.WriteLine("The Facade OperationWrapper carries out complex
decision-making");
            Console.WriteLine("which in turn results in calls to the subsystem
classes");
        }
    }
}

```

```

        c1.OperationX();
        if (l==1 /*some really complex decision*/) {
            c2.OperationY();
        }
        // lots of complex code here . . .
        c3.OperationZ();
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        Facade facade = new Facade();
        Console.WriteLine("Client calls Facade OperationWrapper");
        facade.OperationWrapper();
        return 0;
    }
}
}

```

// Factory Method

// Intent: "Define an interface for creating an object, but let subclasses
// decide which class to instantiate. Factory Method lets a class defer
// instantiation to subclasses."

```

/* Notes:
 * This design pattern is often used in a framework architecture, which
 * contains a number of related classes inside the framework, and which
 * expects application developers to derive custom classes from them.
 * If the instantiation of some custom application classes needs to be
 * carried out inside framework code, then the Factory Method design
 * pattern is an ideal solution.
 *
 * The sample below is based on a document-view architecture.
 * It shows the framework providing classes DPApplication and DPDокумент
 * and the application providing MyApplication and MyDocument, derived
 * from the corresponding framework classes. Note how the construction
 * of MyDocument is initiated from inside the framework's DPApplication,
 * yet the framework was not written specifically for it.
 *
 * It could be extended for views and workspaces along the same lines.
 */

```

```

namespace FactoryMethod_DesignPattern {
    using System;

    // These two classes could be part of a framework,
    // which we will call DP
    // =====

    class DPDocument {

    abstract class DPApplication {
        protected DPDocument doc;

        abstract public void CreateDocument();

        public void ConstructObjects() {

```

```

// Create objects as needed
// . . .

// including document
CreateDocument();
}
abstract public void Dump();
}

// These two classes could be part of an application
// =====

class MyApplication : DPAplication {
    override public void CreateDocument() {
        doc = new MyDocument();
    }

    override public void Dump() {
        Console.WriteLine("MyApplication exists");
    }
}

class MyDocument : DPDocument {
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        MyApplication myApplication = new MyApplication();

        myApplication.ConstructObjects();
        myApplication.Dump();

        return 0;
    }
}
}

// Flyweight

// Intent: "Use sharing to support large numbers of fine-grained objects
effectively."

/* Notes:
 * Useful pattern when you have a small(ish) number of objects that might be
 * needed a very large number of times - with slightly different
 * information that can be externalized outside those objects.
 */

namespace Flyweight_DesignPattern {
    using System;
    using System.Collections;

    class FlyweightFactory {
        private ArrayList pool = new ArrayList();

        //the flyweightfactory can create all entries in pool at startup
        // (if pool is small, and it is likely all will be used), or as
        // needed, if pool is large and likely some will never be used
        public FlyweightFactory() {

```

18 Design Patterns

```
        pool.Add(new ConcreteEvenFlyweight());
        pool.Add(new ConcreteUnevenFlyweight());
    }

    public Flyweight GetFlyweight(int key) {
        // here we would determine if flyweight identified by key
        // exists, and if so return it. If not, we create it.
        // As in this demo we have implemented all the possible
        // flyweights we wish to use, we return the suitable one.
        int i = key % 2;
        return((Flyweight)pool[i]);
    }
}

abstract class Flyweight {
    abstract public void DoOperation(int extrinsicState);
}

class UnsharedConcreteFlyweight : Flyweight {
    override public void DoOperation(int extrinsicState) {

    }
}

class ConcreteEvenFlyweight : Flyweight {
    override public void DoOperation(int extrinsicState) {
        Console.WriteLine("In ConcreteEvenFlyweight.DoOperation: {0}",
extrinsicState);
    }
}

class ConcreteUnevenFlyweight : Flyweight {
    override public void DoOperation(int extrinsicState) {
        Console.WriteLine("In ConcreteUnevenFlyweight.DoOperation: {0}",
extrinsicState);
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        int[] data = {1,2,3,4,5,6,7,8};

        FlyweightFactory f = new FlyweightFactory();

        int extrinsicState = 3;
        foreach (int i in data) {
            Flyweight flyweight = f.GetFlyweight(i);

            flyweight.DoOperation(extrinsicState);
        }

        return 0;
    }
}
}
```

// Interpreter

```

// Intent: "Given a language, define a representation for its grammar along
// with an interpreter that uses the representation to interpret
// sentences in the language."

/* Notes:
 * This is used to implement a language using a class hierarchy
 */

namespace Interpreter_DesignPattern {
    using System;
    using System.Collections;

    class Context {

    }

    abstract class AbstractExpression {
        abstract public void Interpret(Context c);
    }

    // class for terminal symbol
    class TerminalExpression : AbstractExpression {
        override public void Interpret(Context c) {
        }
    }

    // class for grammar rule (one per rule needed)
    class NonterminalExpression : AbstractExpression {
        override public void Interpret(Context c) {
        }
    }
    // to extend grammar, just add other NonterminalExpression classes

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        public static int Main(string[] args) {
            Context c = new Context();
            ArrayList l = new ArrayList(); //really need a tree here!

            // build up context information
            // . .

            // Populate abstract syntax tree with data
            l.Add(new TerminalExpression());
            l.Add(new NonterminalExpression());

            // interpret
            foreach (AbstractExpression exp in l) {
                exp.Interpret(c);
            }
            return 0;
        }
    }
}

```

```

// Iterator

// Intent: "Provide a way to access the elements of an aggregate object
// sequentially without exposing its underlying representation."

/* Notes:
 * Here we wish to separate node traversal from the nodes themselves.
 * STL in ISO C++ is a highly successful application of this pattern.
 * Generic programming is a great way to implement iterators. As this is
 * not yet in C#, we use inheritance.
 */

namespace Iterator_DesignPattern {
    using System;
    using System.Collections;

    class Node {
        private string name;
        public string Name {
            get {
                return name;
            }
        }
        public Node(string s) {
            name = s;
        }
    }

    class NodeCollection {
        private ArrayList list = new ArrayList();
        private int nodeMax = 0;

        // left as a student exercise - implement collection
        // functions to remove and edit entries also
        public void AddNode(Node n) {
            list.Add(n);
            nodeMax++;
        }
        public Node GetNode(int i) {
            return ((Node) list[i]);
        }

        public int NodeMax {
            get {
                return nodeMax;
            }
        }
    }

    /*
     * The iterator needs to understand how to traverse the collection
     * It can do that as way it pleases - forward, reverse, depth-first,
     */
    abstract class Iterator {
        abstract public Node Next();
    }

    class ReverseIterator : Iterator {
        private NodeCollection nodeCollection;
        private int currentIndex;

        public ReverseIterator (NodeCollection c) {

```

```

        nodeCollection = c;
        currentIndex = c.NodeMax - 1; // array index starts at 0!
    }

    // note: as the code stands, if the collection changes,
    // the iterator needs to be restarted
    override public Node Next() {
        if (currentIndex == -1)
            return null;
        else
            return(nodeCollection.GetNode(currentIndex--));
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        NodeCollection c = new NodeCollection();
        c.AddNode(new Node("first"));
        c.AddNode(new Node("second"));
        c.AddNode(new Node("third"));

        // now use iterator to traverse this
        ReverseIterator i = new ReverseIterator(c);

        // the code below will work with any iterator type
        Node n;
        do {
            n = i.Next();
            if (n != null)
                Console.WriteLine("{0}", n.Name);
        } while (n != null);

        return 0;
    }
}
}

```

// Mediator

```

// Intent: "Define an object that encapsulates how a set of objects interact.
// Mediator promotes loose coupling by keeping objects from referring to each
// other explicitly, and it lets you vary their interaction independently."

/* Notes:
 * Consider a mediator as a hub, which objects that need to talk -
 * but do not wish to be interdependent - can use.
 */

namespace Mediator_DesignPattern {
    using System;

    class Mediator {
        private DataProviderColleague dataProvider;
        private DataConsumerColleague dataConsumer;
        public void IntroduceColleagues(DataProviderColleague c1,
                                         DataConsumerColleague c2) {
            dataProvider = c1;
            dataConsumer = c2;
        }
    }
}

```

```

public void DataChanged() {
    int i = dataProvider.MyData;
    dataConsumer.NewValue(i);
}
}

class DataConsumerColleague {
    public void NewValue(int i) {
        Console.WriteLine("New value {0}", i);
    }
}

class DataProviderColleague {
    private Mediator mediator;
    private int iMyData=0;
    public int MyData {
        get {
            return iMyData;
        }
        set {
            iMyData = value;
        }
    }
    public DataProviderColleague(Mediator m) {
        mediator = m;
    }
    public void ChangeData() {
        iMyData = 403;

        // Inform mediator that I have changed the data
        if (mediator != null)
            mediator.DataChanged();
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        Mediator m = new Mediator();
        DataProviderColleague c1 = new DataProviderColleague(m);
        DataConsumerColleague c2 = new DataConsumerColleague();
        m.IntroduceColleagues(c1,c2);

        c1.ChangeData();

        return 0;
    }
}
}

// Memento

// Intent: "Without violating encapsulation, capture and externalize an
// object's internal state so that an object can be restored to this
// state later."

/* Notes:
 * We often have client code that wishes to record the current state of an

```

```
* object, without being interested in the actual data values (needed for
* undo and checkpointing). To support this behavior, we can have the object
* record its internal data in a helper class called a memento, and the
* client code can treat this as an opaque store of the object's state.
* At some later point, the client can pass the memento back into the
* object to restore it to the previous state.
*/
```

```
namespace Memento_DesignPattern {
    using System;

    class Originator {
        private double manufacturer=0;
        private double distributor = 0;
        private double retailer = 0;

        public void MakeSale(double purchasePrice) {
            // We assume sales are divided equally among the three
            manufacturer += purchasePrice * .40;
            distributor += purchasePrice *.3;
            retailer += purchasePrice *.3;
            // Note: to avoid rounding errors for real money handling
            // apps, we should be using decimal integers
            // (but hey, this is just a demo!)
        }

        public Memento CreateMemento() {
            return (new Memento(manufacturer, distributor, retailer));
        }

        public void SetMemento(Memento m) {
            manufacturer = m.A;
            distributor = m.B;
            retailer = m.C;
        }
    }

    class Memento {
        private double iA;
        private double iB;
        private double iC;

        public Memento(double a, double b, double c) {
            iA = a;
            iB = b;
            iC = c;
        }

        public double A {
            get {
                return iA;
            }
        }

        public double B {
            get {
                return iB;
            }
        }

        public double C {
            get {
                return iC;
            }
        }
    }
}
```

```

        }
    }

class caretaker {
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        Originator o = new Originator();

        // Assume that during the course of running an application
        // we set various data in the originator
        o.MakeSale(45.0);
        o.MakeSale(60.0);

        // Now we wish to record the state of the object
        Memento m = o.CreateMemento();

        // We make further changes to the object
        o.MakeSale(60.0);
        o.MakeSale(10.0);
        o.MakeSale(320.0);

        // Then we decide to change our minds, and revert to the saved
        // state (and lose the changes since then)
        o.SetMemento(m);

        return 0;
    }
}
}

```

// Observer

```

// Intent: "Define a one-to-many dependency between objects so that when one
// object changes state, all its dependents are notified and
// updated automatically."

/* Notes:
 * Often used for Document-View architectures, where a document stores
 * data and one or more views renders the data.
 */

namespace Observer_DesignPattern {
    using System;
    using System.Collections;

    class Subject {
        private ArrayList list = new ArrayList();

        private string strImportantSubjectData = "Initial";

        public string ImportantSubjectData {
            get {
                return strImportantSubjectData;
            }
            set {
                strImportantSubjectData = value;
            }
        }
    }
}

```

```
        }

    }

    public void Attach(Observer o) {
        list.Add(o);
        o.ObservedSubject = this;
    }

    public void Detach(Observer o) {
    }

    public void Notify() {
        foreach (Observer o in list) {
            o.Update();
        }
    }
}

class ConcreteSubject : Subject {
    public void GetState() {
    }

    public void SetState() {
    }
}

abstract class Observer {
    protected Subject s;
    public Subject ObservedSubject {
        get {
            return s;
        }
        set {
            s = value;
        }
    }
    abstract public void Update();
}

class ConcreteObserver : Observer {
    private string observerName;

    public ConcreteObserver(string name) {
        observerName = name;
    }

    override public void Update() {
        Console.WriteLine("In Observer {0}: data from subject = {1}",
            observerName, s.ImportantSubjectData);
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        // Set up everything
        ConcreteSubject s = new ConcreteSubject();
        ConcreteObserver o1 = new ConcreteObserver("first observer");
        ConcreteObserver o2 = new ConcreteObserver("second observer");
    }
}
```

```

        s.Attach(o1);
        s.Attach(o2);

        // make changes to subject
        s.ImportantSubjectData = "This is important subject data";

        // Notify all observers
        s.Notify();
        return 0;
    }
}

// Prototype

// Intent: "Specify the kinds of objects to create using a prototypical
// instance and create new objects by copying this prototype."

/* Notes:
 * When we are not in a position to call a constructor for an object
 * directly, we could alternatively clone a pre-existing object
 * (a prototype) of the same class.
 *
 * This results in specific class knowledge being only required in
 * one area (to create the prototype itself), and then later cloned
 * from code that knows nothing about the cloned prototype, except
 * that it exposed a well-known cloning method.
*/
namespace Prototype_DesignPattern {
    using System;

    // Objects which are to work as prototypes must be based on classes
    // which are derived from the abstract prototype class
    abstract class AbstractPrototype {
        abstract public AbstractPrototype CloneYourself();
    }

    // This is a sample object
    class MyPrototype : AbstractPrototype {
        override public AbstractPrototype CloneYourself() {
            return ((AbstractPrototype)MemberwiseClone());
        }
        // lots of other functions go here!
    }

    // This is the client piece of code which instantiate objects
    // based on a prototype.
    class Demo {
        private AbstractPrototype internalPrototype;
        public void SetPrototype(AbstractPrototype thePrototype) {
            internalPrototype = thePrototype;
        }

        public void SomeImportantOperation() {
            // During some important operation, imagine we need to
            // instantiate an object - but we do not know which. We
            // use the predefined prototype object; ask it to clone itself.

            AbstractPrototype x;
            x = internalPrototype.CloneYourself();
            // now have 2 instances of class which acts as prototype
        }
    }
}

```

```
        }
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        public static int Main(string[] args) {
            Demo demo = new Demo();
            MyPrototype clientPrototype = new MyPrototype();
            demo.SetPrototype(clientPrototype);
            demo.SomeImportantOperation();
            return 0;
        }
    }
}
```

// Proxy

```
// Intent: "Provide a surrogate or placeholder for another object to
// control access to it."
/*
 * Notes:
 * When there is a large CPU/memory expense attached to handling an object
 * directly, it can be useful to use a lightweight proxy in front of it,
 * which can take its place until the real object is needed.
 */
namespace Proxy_DesignPattern {
    using System;
    using System.Threading;

    /// <summary>
    /// Summary description for Client.
    /// </summary>
    abstract class CommonSubject {
        abstract public void Request();
    }

    class ActualSubject : CommonSubject {
        public ActualSubject() {
            // Assume constructor does some operation that takes a
            // while - hence the need for a proxy - to avoid incurring
            // this delay until (and if) the actual subject is needed
            Console.WriteLine("Starting to construct ActualSubject");
            Thread.Sleep(1000); // represents lots of processing!
            Console.WriteLine("Finished constructing ActualSubject");
        }
        override public void Request() {
            Console.WriteLine("Executing request in ActualSubject");
        }
    }

    class Proxy : CommonSubject {
        ActualSubject actualSubject;

        override public void Request() {
            if (actualSubject == null)
                actualSubject = new ActualSubject();
            actualSubject.Request();
        }
    }
}
```

```

public class Client {
    public static int Main(string[] args) {
        Proxy p = new Proxy();

        // Perform actions here
        // . . .

        if (1==1)      // at later point based on a condition,
            p.Request(); // determine if we need to use subject

        return 0;
    }
}
}

// Singleton

// Intent: "Ensure a class only has one instance, and provide a global
// point of access to it."

/* Notes:
 * If it makes sense to have only a single instance of a class (a so-called
 * singleton), then it makes sense to enforce this (to eliminate potential
 * errors, etc).
 *
 * A class based on the singleton design pattern protects its constructor,
 * so that only the class itself (e.g. in a static method) may instantiate
 * itself. It exposes an Instance method which allows client code to
 * retrieve current instance, and if it does not exist to instantiate it.
 */
namespace Singleton_DesignPattern {
    using System;

    class Singleton {
        private static Singleton _instance;

        public static Singleton Instance() {
            if (_instance == null)
                _instance = new Singleton();
            return _instance;
        }
        protected Singleton(){}
        // Just to prove only a single instance exists
        private int x = 0;
        public void SetX(int newVal) {x = newVal;}
        public int GetX(){return x;}
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        public static int Main(string[] args) {
            int val;
            // can't call new, because constructor is protected
            Singleton FirstSingleton = Singleton.Instance();
            Singleton SecondSingleton = Singleton.Instance();

            // Now we have 2 variables; both should refer to the same
        }
    }
}

```

```

        // object. Let's prove this, by setting a value using one
        // variable and (hopefully!) retrieving same value using           //
the 2nd variable.
        FirstSingleton.SetX(4);
        Console.WriteLine("Using 1st singleton var, set x to 4");
        val = SecondSingleton.GetX();
        Console.WriteLine("Using 2nd variable for singleton, value
retrieved = {0}", val);
        return 0;
    }
}
}

```

// State

// Intent: "Allow an object to alter its behavior when its internal state
// changes. The object will appear to change its class."

```

/* Notes:
 *
 * A finite state machine is an appropriate construct to use for a class
 * whose reaction to stimuli depends on previous behavior. In the past, each
 * method in the class was coded as a switch statement, switching on the
 * state. If a new state was added, then each method had to be edited.
 *
 * With the state design pattern, functionality specific to a state is
 * placed in a helper class, and the main class delegates those methods that
 * are state-specific to such helper classes.
*/

```

```

namespace State_DesignPattern {
    using System;

    abstract class State {
        protected string strStatename;

        abstract public void Pour();
        // do something state-specific here
    }

    class OpenedState : State {
        public OpenedState () {
            strStatename = "Opened";
        }
        override public void Pour() {
            Console.WriteLine("...pouring...");
            Console.WriteLine("...pouring...");
            Console.WriteLine("...pouring...");
        }
    }

    class ClosedState : State {
        public ClosedState() {
            strStatename = "Closed";
        }
        override public void Pour() {
            Console.WriteLine("ERROR - bottle is closed - cannot pour");
        }
    }

    class ContextColaBottle {
        public enum BottleStateSetting {

```

```

        Closed,
        Opened
    };

    // If the state classes had large amounts of instance data,
    // we could dynamically create them as needed - if this demo
    // they are tiny, so we just create them as data members
    OpenedState openedState = new OpenedState();
    ClosedState closedState = new ClosedState();

    public ContextColaBottle () {
        // Initialize to closed
        CurrentState = closedState;
    }

    private State CurrentState;

    public void SetState(BottleStateSetting newState) {
        if (newState == BottleStateSetting.Closed) {
            CurrentState = closedState;
        }
        else {
            CurrentState = openedState;
        }
    }

    public void Pour() {
        CurrentState.Pour();
    }
}

/// <summary>
///     Summary description for Client.
/// </summary>
public class Client {
    public static int Main(string[] args) {
        ContextColaBottle contextColaBottle = new ContextColaBottle();
        Console.WriteLine("initial state is closed");
        Console.WriteLine("Now trying to pour");

        contextColaBottle.Pour();
        Console.WriteLine("Open bottle");

        contextColaBottle.SetState(
            ContextColaBottle.BottleStateSetting.Opened);

        Console.WriteLine("Try to pour again");
        contextColaBottle.Pour();

        return 0;
    }
}
}

```

// Strategy

```

// Intent: "Define a family of algorithms, encapsulate each one, and make
// them interchangeable. Strategy lets the algorithm vary independently
// from clients that use it."

/* Notes:
 * Ideal for creating exchangeable algorithms.

```

```

/*
namespace Strategy_DesignPattern {
    using System;

    abstract class Strategy {
        abstract public void DoAlgorithm(); }

    class FirstStrategy : Strategy {
        override public void DoAlgorithm() {
            Console.WriteLine("In first strategy");
        }
    }

    class SecondStrategy : Strategy {
        override public void DoAlgorithm() {
            Console.WriteLine("In second strategy");
        }
    }

    class Context {
        Strategy s;
        public Context(Strategy strat) {
            s = strat;
        }

        public void DoWork() {
            // some of the context's own code goes here
        }

        public void DoStrategyWork() {
            // now we can hand off to strategy to do some more work
            s.DoAlgorithm();
        }
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        public static int Main(string[] args) {
            FirstStrategy firstStrategy = new FirstStrategy();
            Context c = new Context(firstStrategy);
            c.DoWork();
            c.DoStrategyWork();
            return 0;
        }
    }
}

// Template Method

// Intent: "Define the skeleton of an algorithm in an operation, deferring
// some steps to subclasses. Template Method lets subclasses redefine
// certain steps of an algorithm without changing the algorithm's
// structure."

/* Notes:
 * If you have an algorithm with multiple steps, and it could be helpful to
 * make some of those steps replaceable, but not the entire algorithm, then
 * use the Template method.
*/

```

32 Design Patterns

```
* If the programming language in use supports generics / templates (C# does
* not), then they could be used here. It would be educational to take a
* good look at the way algorithms in ISO C++'s STL work.
*/
namespace TemplateMethod_DesignPattern {
    using System;

    class Algorithm {
        public void DoAlgorithm() {
            Console.WriteLine("In DoAlgorithm");

            // do some part of the algorithm here

            // step1 goes here
            Console.WriteLine("In Algorithm - DoAlgoStep1");
            // . . .

            // step 2 goes here
            Console.WriteLine("In Algorithm - DoAlgoStep2");
            // . . .

            // Now call configurable/replaceable part
            DoAlgoStep3();

            // step 4 goes here
            Console.WriteLine("In Algorithm - DoAlgoStep4");
            // . . .

            // Now call next configurable part
            DoAlgoStep5();
        }

        virtual public void DoAlgoStep3() {
            Console.WriteLine("In Algorithm - DoAlgoStep3");
        }

        virtual public void DoAlgoStep5() {
            Console.WriteLine("In Algorithm - DoAlgoStep5");
        }
    }

    class CustomAlgorithm : Algorithm {
        public override void DoAlgoStep3() {
            Console.WriteLine("In CustomAlgorithm - DoAlgoStep3");
        }

        public override void DoAlgoStep5() {
            Console.WriteLine("In CustomAlgorithm - DoAlgoStep5");
        }
    }

    /// <summary>
    ///     Summary description for Client.
    /// </summary>
    public class Client {
        public static int Main(string[] args) {
            CustomAlgorithm c = new CustomAlgorithm();
            c.DoAlgorithm();

            return 0;
        }
    }
}
```

```
}
```

// Visitor

```
// Intent: "Represent an operation to be performed on the elements of an
// object structure. Visitor lets you define a new operation without
// changing the classes of the elements on which it operates."

/* Notes:
 * If you have a number of elements, and wish to carry out a number of
 * operations on them, the Visitor design pattern can be helpful. It lets
 * you extract the operations to be carried out on elements from the
 * elements themselves. It means operations can change without affecting
 * the elements.
 */

namespace Visitor_DesignPattern {
    using System;

    abstract class Visitor {
        abstract public void VisitElementA(ConcreteElementA a);
        abstract public void VisitElementB(ConcreteElementB b);
    }

    class ConcreteVisitor1 : Visitor {
        override public void VisitElementA(ConcreteElementA a) {
        }

        override public void VisitElementB(ConcreteElementB b) {
        }
    }

    abstract class Element {
        abstract public void Accept(Visitor v);
    }

    class ConcreteElementA : Element {
        public Visitor myVisitor;
        override public void Accept(Visitor v) {
            myVisitor = v;
        }

        public void OperationA() {
        }

        public void DoSomeWork() {
            // do some work here
            // . . .

            // Get visitor to visit
            myVisitor.VisitElementA(this);

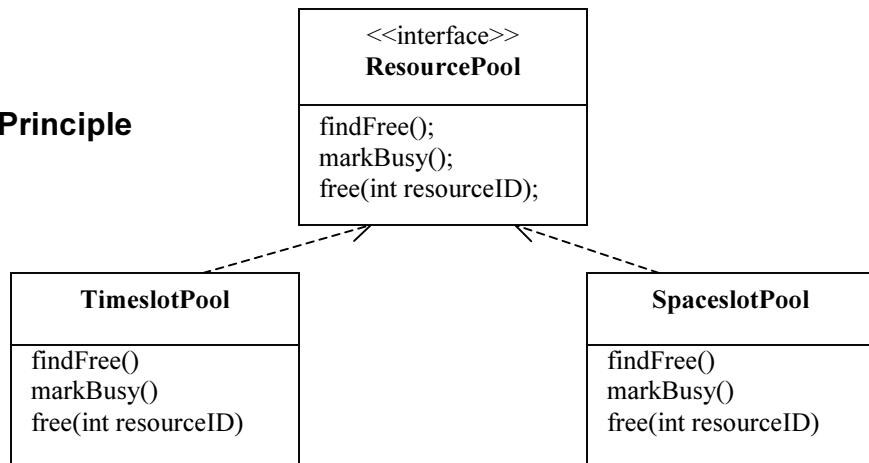
            // do some more work here
            // . . .
        }
    }

    class ConcreteElementB : Element {
        override public void Accept(Visitor v) {
        }
    }
}
```

```
public void OperationB() {  
}  
}  
  
/// <summary>  
/// Summary description for Client.  
/// </summary>  
public class Client {  
    public static int Main(string[] args) {  
        ConcreteElementA eA = new ConcreteElementA();  
        ConcreteElementB eB = new ConcreteElementB();  
        ConcreteVisitor1 v1 = new ConcreteVisitor1();  
        eA.Accept(v1);  
        eA.DoSomeWork();  
        return 0;  
    }  
}
```

Appendix C – Possible Solutions for Exercises

Notes

Pages 64-68:**1. Open-Closed Principle**

```

class ResourceAllocator {
    ResourcePool *m_pResourcePool[MAX_RESOURCE_POOLS];
public:
    int Allocate(int resourceType) {
        int resourceId;
        resourceId = m_pResourcePool[resourceType].FindFree();
        m_pResourcePool[resourceType].MarkBusy(resourceId);
    }

    int Free(int resourceType, int resourceId) {
        m_pResourcePool[resourceType].MarkBusy(resourceId);
    }

    void AddResourcePool(int resourceType, ResourcePool *pPool) {
        m_pResourcePool[resourceType] = pPool;
    }
};

class ResourcePool {
public:
    virtual int FindFree() = 0;
    virtual int MarkBusy() = 0;
    virtual Free(int resourceId) = 0;
};

class TimeslotPool : public ResourcePool {
public:
    int FindFree();
    int MarkBusy();
    Free(int resourceId);
};

class SpaceslotPool : public ResourcePool {
public:
    int FindFree();
    int MarkBusy();
    Free(int resourceId);
};
  
```

2. Circle/Ellipse Dilemma

Problems for Circle: Circle has only a single center, not two foci, and only a radius, not a major-axis and a minor-axis.

Fixes: You would have to guarantee that `focusA == focusB` in a circle and collapse major-axis and minor-axis to be simply radius.

If method `f()` is called, passed an instance of a Circle and the above fixes have been coded for a Circle, the `assert()` statements fail. The programmer who wrote method `f()` assumed that changing `focusA` has no bearing on the length of `focusB`.

LSP has been violated; the derived Circle is NOT substitutable for the base Ellipse. Also, Bertrand Meyer's statement on pre/post conditions is NOT true – a Circle expects MORE than does an Ellipse (i.e., `focusA = focusB`) and thus has a stronger precondition. And likewise, the postcondition of Circle is weaker than the postcondition of Ellipse: for `setFoci(a, b)` does not enforce the constraint that, after setting `focusB`, `getFocusB()` would return Point B.

By deriving a Circle from an Ellipse, Circle violates an invariant of Ellipse. The IS-A relationship should describe behavior that clients can depend on.

Consequently, you would have to do a check on the type of Shape—perhaps a case statement or something like:

```
if (x instanceof Ellipse) {
    . . .
}
else if (x instanceof Circle) {
    . . .
}
```

and the derived class (Circle) is NOT substitutable for the base class (Ellipse).

3. ISP: ATM Transaction Hierarchy

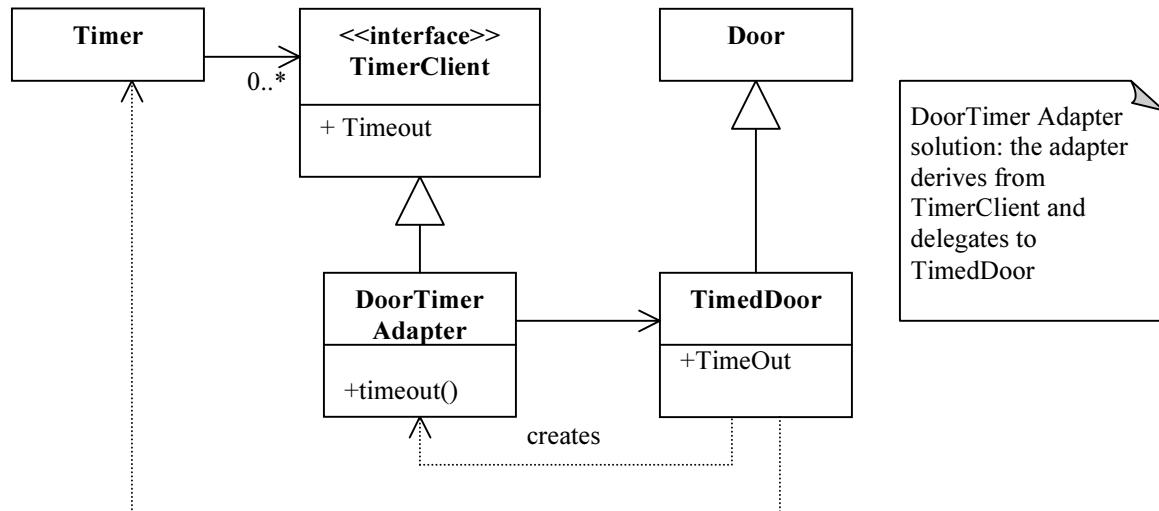
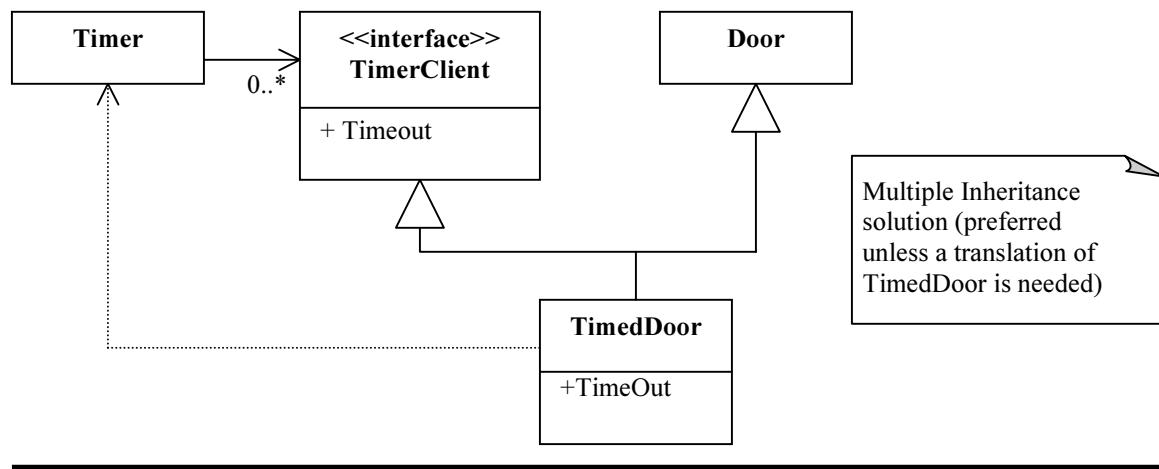
Potential UI methods: deposit(Money, Account); withdraw(Money, Account); transfer(Money, FromAccount, ToAccount); informInsufficientFunds(); etc.

The UI (in this diagram) has to define ALL methods, so each class that implements the UI interface has to have code for ALL methods even though many of the methods defined would have no-ops for that particular class. This “stinks.”

Adding subclasses only makes the problem worse in this model, as it increases the number of methods defined in the interface – and ALL methods would have to be defined by classes implementing that interface.

It is better to break up the UI into sub-interfaces, such as DepositUI, WithdrawalUI, TransferUI.

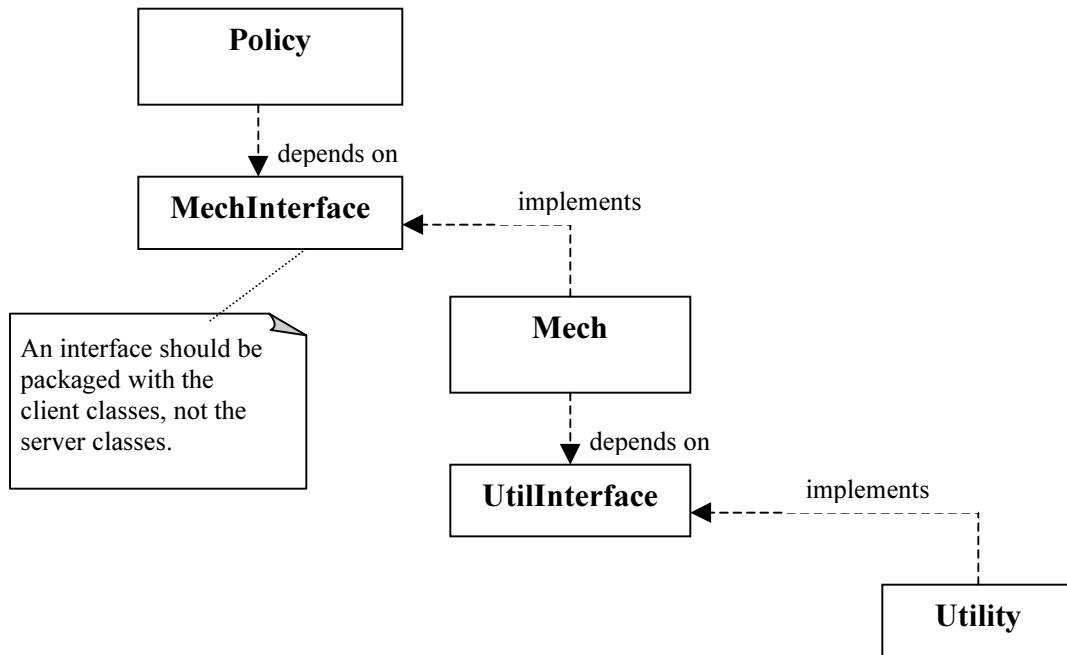
UML for TimedDoor (page 58):



4. DIP

If you make a change in the Utility Layer, all of the above dependent layers are potentially affected and would need to be tested for side-effects.

A more appropriate model (notice how dependencies are inverted):

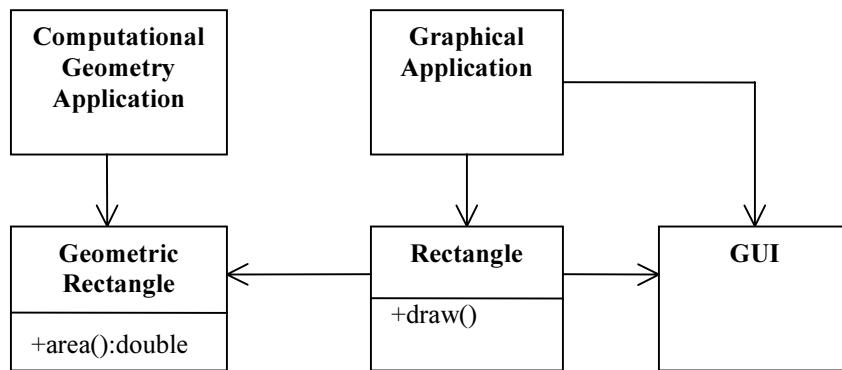


5. SRP

Potential Problems: You have to include the GUI classes in the Computational Geometry Application -- in C++, that would mean linking in the GUI; in Java the .class files for the GUI would have to be deployed to the target platform.

If a change to the Graphical Application necessitates a change to Rectangle, the change could cause us to rebuild, retest, and redeploy the Computational Geometry Application. Forgetting to do this could make that application break in unpredictable ways.

UML redrawn to separate the responsibilities of Rectangle:



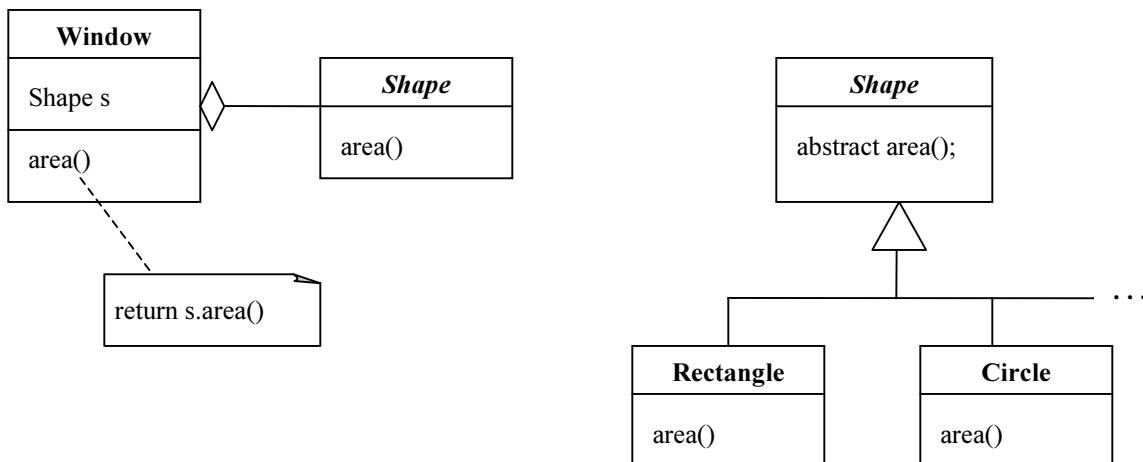
The computational aspects of Rectangle are moved into GeometricRectangle; now any changes made to the way rectangles are drawn cannot affect the Computational Geometry Application.

Page 86:**1. Martin Metrics:**

$Ca = 4; Ce = 3; I = 3 / 7.$

Pages 102-104:**1. Composition and Delegation**

A Window IS-A Rectangle vs. A Window HAS-A Rectangle (or any other shape). If you want to find the area() of aWindow, its area() method simply calls the area() method of the referenced shape. Any shape could be passed to a constructor and subsequently could be changed dynamically.



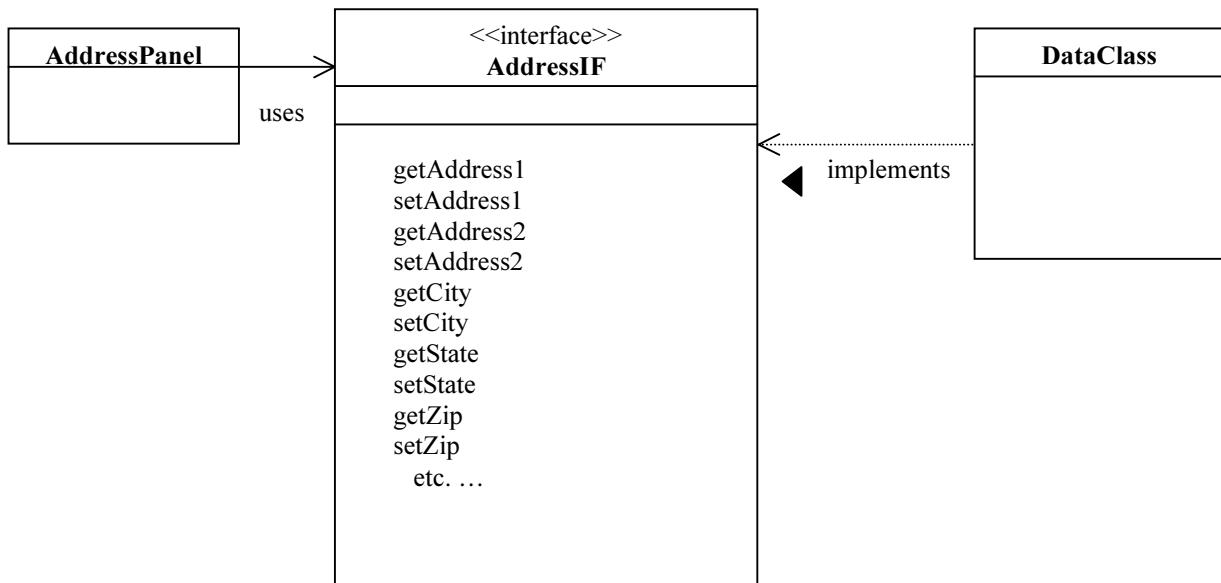
```

Window w = new Window(Shape s);
float area = w.s.area();
  
```

2. Interface

Interfaces used vary with the individual. Java programmers use lots of them... check for IO, RMI, use of the Collection classes, etc.

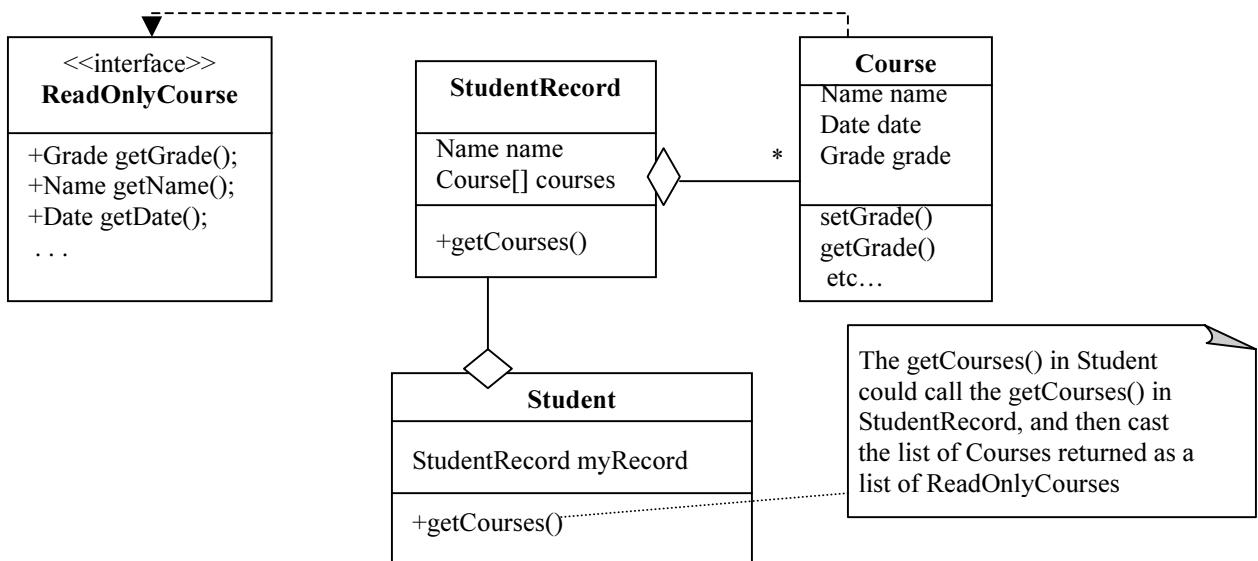
Address Interface:



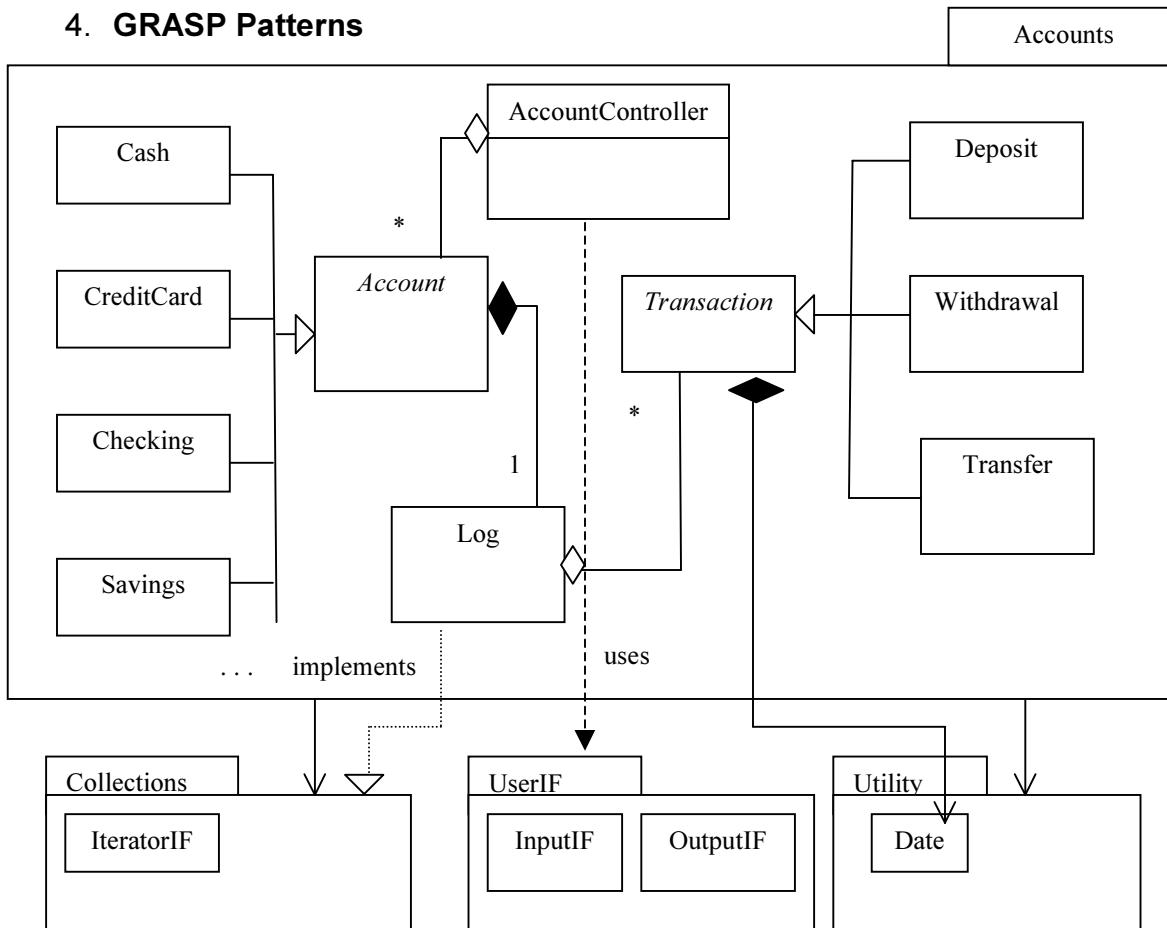
3. Immutable

Create an interface called `ImmutableCourse` or `ReadOnlyCourse`, with just the getter methods (no setter methods). When a student, unauthorized for changes, calls `getCourses()` on their `StudentRecord` instance, the method returns a list of `ReadOnlyCourse` instances instead of returning a list of `Course` instances.

`Course` implements the new interface (in Java) or inherits from it (C++), so `Course` can be returned in either fashion. A student with a list of `ReadOnlyCourse` instances will be limited to the methods defined in that interface – only the getters.



4. GRASP Patterns



(This is just a start at a possible solution... not necessarily the best, but one that does illustrate the GRASP patterns.)

Create **AccountController** to handle system events such as `createAccount()`. This method could be called by the **UserIF** with a button press (Command Pattern) and pass the necessary info as parameters. **AccountController** has a list of all **Account** objects. Make **Account** an abstract class with abstract methods [such as `enterTransaction(Transaction t)`] which then have to be implemented appropriately for each concrete subclass. Each **Account** does, however, have its own **Log**. Therefore, let **Account** also have a method `createLog()`. **AccountController** should not have to know or care which kind of **Account** it is talking to – each **Account** knows what to do -- polymorphic behavior here.

Similarly, make **Transaction** an abstract class with methods such as `display()` which could be implemented differently for each concrete subclass. `display()` could be both polymorphic and overloaded, and should send output to some **Output** class (looser coupling with the user interface – Proxy Pattern) rather than talking directly to any output device.

Other than each class talking to Output, Controller only talks to Account; Account only talks to Log [and temporarily Transaction with a method such as enter(Transaction)]; Log only talks to Transaction; Transaction only talks to Date.

We could now begin to talk about the following GRASP patterns:

Information Expert (Controller acts as the glue with the UserIF and receives the information)

Creator (Controller creates Account, Account creates Log)

Controller (We fabricated a class for this task. If the class got “bloated”, we should break the task up and have perhaps a controller class for each use case, then an overall controller to direct traffic.)

Low Coupling (We have minimized the lines of connection in the diagrams.)

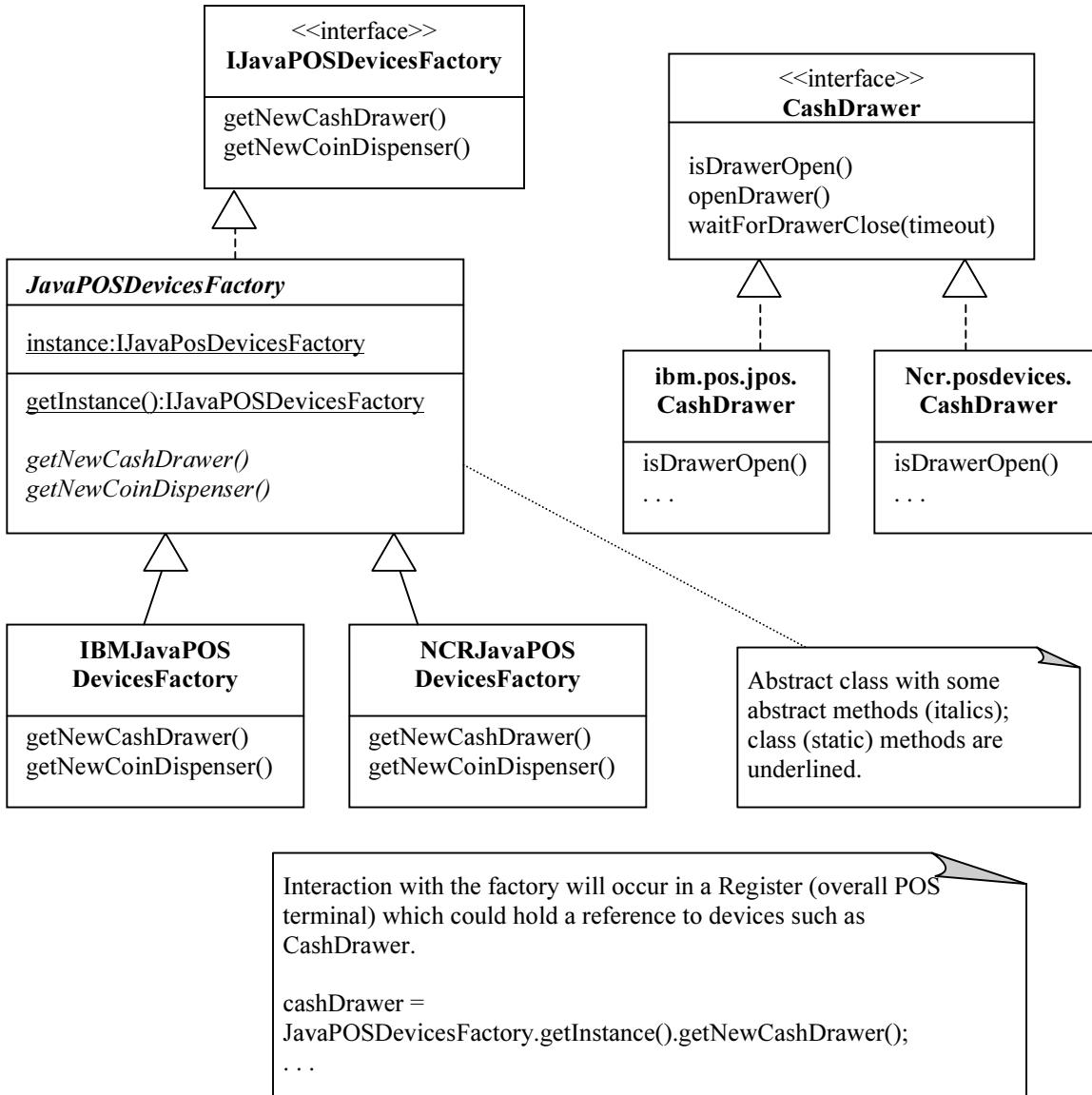
High Cohesion (Each class is responsible for the display of its own info.)

Polymorphism (Account and Transaction methods, could be polymorphic.)

Pure Fabrication (AccountController is one example; an Output class that implements the OutputIF could be another...)

Indirection (When Log uses the IteratorIF to sort the Transactions, it needs the Date information, but must go through some method in Transaction since Log does not direction talk to the Date class. Therefore we need a method in Transaction whereby a Transaction could query its Date and the Date of another Transaction and return appropriate sorting information to the caller, Log.)

Protected Variations (Input could easily change; Output could change. We create stable interfaces for both and protect ourselves from potential changes. The UserIF could also change. The type of data structure for the Log could change – the other classes should not have to know or care – Log knows its type and can create an appropriate Iterator for itself.)

Pages 122-125:**1a, b, c, d. UML for POS (point of sale) interfaces/classes:**

1e. The appropriate factory to instantiate could be determined by reading a system property such as "*jposfactory.classname*" in a properties file.

2. a. Benefits of the **Singleton** pattern:

- Controlled access to the sole instance
- Reduced name space (closest option to a global, yet within class)
- Singleton permits refinement of operations and representation – can be subclassed (though awkward in Java and the constructor cannot be private and static methods, such as `getInstance()`, cannot be overridden)
- Permits a variable number of instances – you can change your mind
- More flexible than class operations – static member functions in C++ are never virtual so subclasses can't override them polymorphically

b. If you decide later you want multiple copies simply modify the method `getInstance()`. Within this method you can control the number of copies allowed.

c. You consider at least one private constructor in a **Singleton** so the compiler doesn't automatically create the default constructor, which could then be called either accidentally or maliciously and create multiple copies.

d. A registry of singletons could add flexibility to a design.

In C++ the singleton class choice would need to be made at link-time instead of being deferred to a runtime decision. Conditional statements would hard-wire the set of choices – a bit more flexible yet still imperfect.

A registry would provide a map between string names and singletons. The `getInstance()` method could access the registry and look for a map with a string entered at runtime, returning the appropriate singleton if it exists. Singletons could register themselves in their constructor – you could define a static instance of each singleton.

e. Garbage collection could be a problem in Java if a singleton is used only occasionally and needs to do something like collect performance statistics. When no current references exist the instance would be a candidate for garbage collection. If it were deleted, the next reference to the singleton would be to a newly created instance instead of the old one and instance variables would be re-initialized. (See part f for example code that would keep a singleton around.)

f. Adding comments:

```

public class ObjectPreserver implements Runnable {
    // This keeps this class and everything it references from
    // being garbage collected

    private static ObjectPreserver lifeline
        = new ObjectPreserver();

    // Since this class won't be garbage collected, neither will
    // this HashSet or the object that it references

    private static HashSet protectedSet = new HashSet();

    private ObjectPreserver() {
        new Thread(this).start();
    } // constructor

    public void run() {
        try {
            wait();
        } catch (InterruptedException e) { }
    }

    /**
     * Garbage collection of objects passed to this method will be
     * prevented until they are passed to the unpreserveObject method
     */
    public static void preserveObject( Object o ) {
        protectedSet.add( o );
    } // preserveObject(Object)

    /**
     * Objects passed to this method lose the protection that the
     * preserveObject method gave them from garbage collection
     */
    public static void unpreserveObject( Object o ) {
        protectedSet.remove( o );
    } // unpreserveObject(Object)
} // class ObjectPreserver

```

3. Builder pattern code review session:

aClient creates a new ConcreteBuilder (which implements some standard AbstractBuilder interface) and passes it to the constructor of a new Director object. aClient then calls the construct() method on the Director. The Director is responsible for making the calls to the ConcreteBuilder for each subpart, i.e., buildPartA(), buildPartB(), etc. Parts are added to the product as appropriate. When the construct() method returns, aClient can then retrieve the finished product (which meets some standard AbstractProduct interface) with a method such as getResult().

4. Sample Java code fragments to illustrate Prototype

```

public abstract class Character implements Cloneable {
    protected Image image;
    protected String name;

    public Image getImage(){return image;}
    public void setImage(Image i){image = I;}
    public String getName(){return name;}
    public void setName(String n){name = n;}

    // override clone to make it public; inherited clone() is not public
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {}
    }
} //class Character

public class Hero extends Character {
    private int bravery = 10;

    public Hero(){
        super.image = HeroImage.jpg;
    }
    public void setBravery(int b){bravery = b;}
    public int getBravery(){return bravery; }
    .
}

} //class Hero

public class Monster extends Character {
    private int ferocity = 10;

    public Monster(){
        super.image = MonsterImage.jpg;
    }
    public void setFerocity(int f) { ferocity = f;}
    public int getFerocity() { return ferocity; }
    .
}

} //class Monster

public class CharacterManager{
    private Vector characters = new Vector();

    Character getRandomCharacter(){
        int i = (int)(characters.size() * Math.random());
        return (Character) ((Character)characters.elementAt(i)).clone();
    }
}

```

```
void addCharacter(Character char) {  
    characters.addElement(character);  
}  
} // class CharacterManager
```

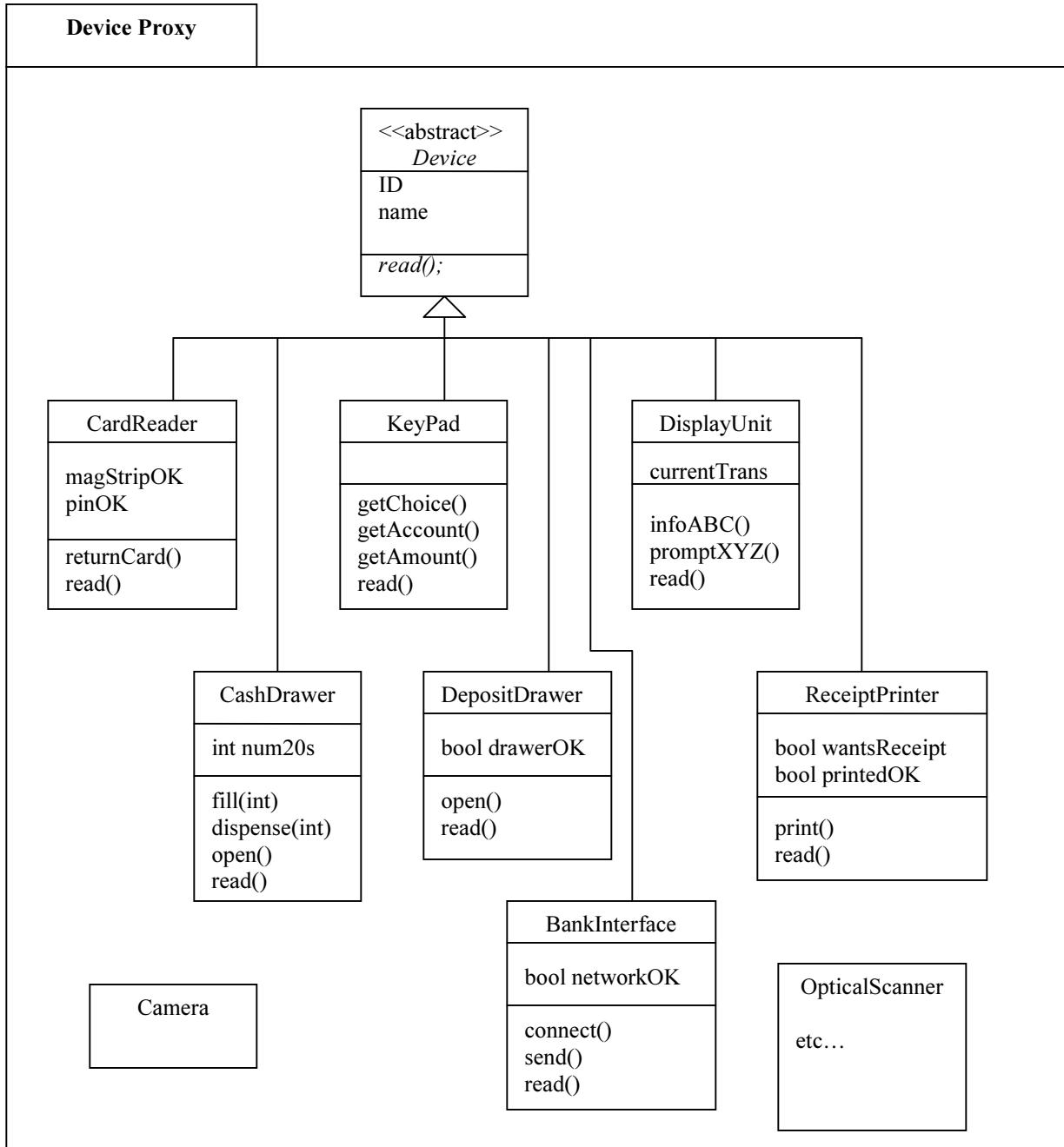
A CharacterLoader class could keep an instance of CharacterManager as a private variable and have a method called loadCharacters(String fname) that could load characters from a file using Java's ObjectInputStream. and return the number of characters loaded.

Each prototype character could then be specialized as desired.

Pages 132-137:

1. **Adapter** changes the interface of the class or object it adapts. **Decorator** changes the object's responsibility, not its interface. **Proxy** use the same interface as its target or real object, but manages the services of that real object, in some fashion adding to or enhancing the services, or controlling access, etc.
2. A **class adapter** only adapts at that level, not adapting derived classes; it overrides some behavior; it introduces only one object and no pointer indirection. An **object adapter** may reference the adaptee object and any of its subclass objects; it can add responsibilities to all levels; it is harder with an object adapter to override behavior as you have to then subclass the adaptee.
3. The issues when designing an **adapter** include:
 - (1) how much adapting to do -- i.e., how close is the adaptee to the target you need;
 - (2) making pluggable adapters – there are three ways to implement in adapting an interface;
 - (3) whether or not to make the adapter a 2-way adapter for use with 2 different clients by implementing 2 different interfaces.
4. **Proxy** implements the SAME interface as the real object; if it doesn't add anything more or do anything different, then what would be the point? You would have one more "door" to go through but it would buy you nothing extra for the trouble.
5. With the LargeHashTable, if another client reads the same instance, no problem – no one has changed anything and it is fine at that point to share the same instance instead of going through the trouble of cloning the structure. As soon as a client calls the put() method, however, then things start to happen. put() calls copyOnWrite() which then checks the **proxy** count. If more than one client exists, then (a) synchronize (lock for exclusive access) so no other changes than yours can be made to the HashTable during this copy, (b) decrease the proxy count – you are going to get your own copy and no longer share; (c) clone the HashTable and cast it as a ReferenceCountedHashTable; (d) if there is an error during the clone, put yourself back as a proxy and know your call to put() failed.

6. This is just a start...



The idea here is to model each of the actual devices connected with a device proxy. The device proxy is the software stand-in for the real object and has the responsibility for connecting with the real object. WHEN the real device changes, only the proxy class need be affected.

7. Remote Proxy example

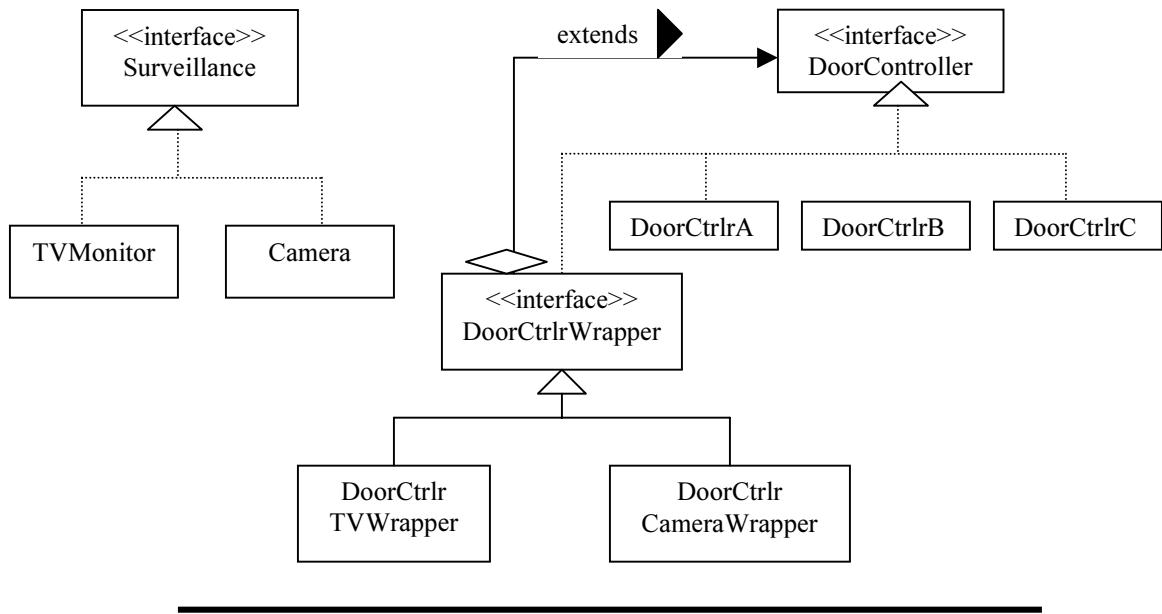
Identifiable patterns: **Proxy** (failover or redirection proxy – structure of proxy is the same – variations are related to what the proxy does once called), **Adapter**, **Singleton**.

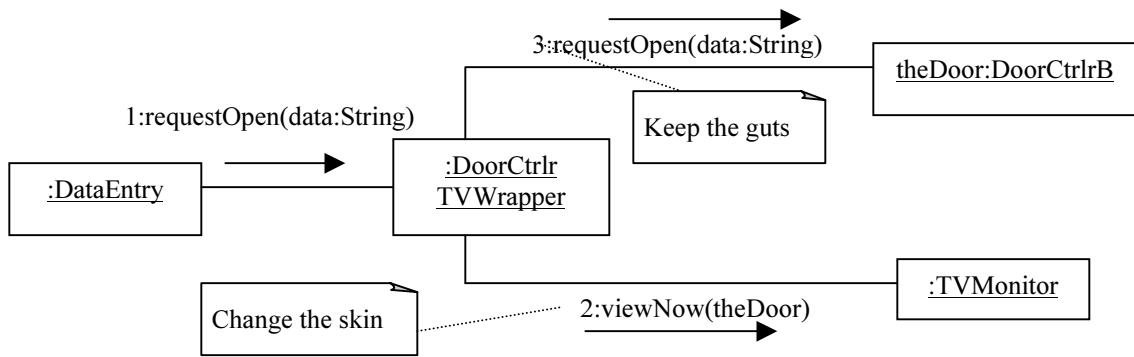
Whether or not to try the local service first depends on the application. In the example shown, the local service is always tried first. If there is some caching of products locally, that could avoid the overhead of going out to the database through the DBProductsAdapter. However, consider the situation of posting sales to an accounting service – business wants sales posted as soon as possible for real-time tracking of store and register activity. In that case, you might go to the external service first, and the local only as a back-up if the connection to the external failed.

Both the LocalProducts class and the DBProducts Adapter must implement the same interface (for example, <>interface>> ProductsAdapterIF).

Error Handling: Objects (represented by :Object in the diagram) could represent an ErrorLog, something wrapped in the UI such as a modal dialog, a text console, a sound beeper, a speech generator, etc., or combinations of any of the previous. Both the idea of having a centralized Error Logging facility and an Error Dialog as a Singleton-accessed, application-independent, non-UI object for notification could be considered patterns. A Factory reading from system parameters could create the appropriate UI objects.

8. Door Controller UML (Decorator)





Comments:

With three kinds of doors installed and two different kinds of surveillance monitors, you write two subclasses of each of the door controller classes. But instead of writing six subclasses, use the **Decorator** pattern, which uses delegation instead of inheritance.

The new class, AbstractDoorCtrlrWrapper, implements all methods of the DoorController interface with implementations that simply call the corresponding method of another object that implements that same interface. DoorCtrlrTVWrapper and DoorCtrlrCameraWrapper are concrete wrapper classes which extend the behavior of the requestOpen() method declared in the DoorController interface (as in call 2 above).

Doorways viewed by multiple cameras can have multiple wrappers in front of the DoorController interface object.

Page 141:

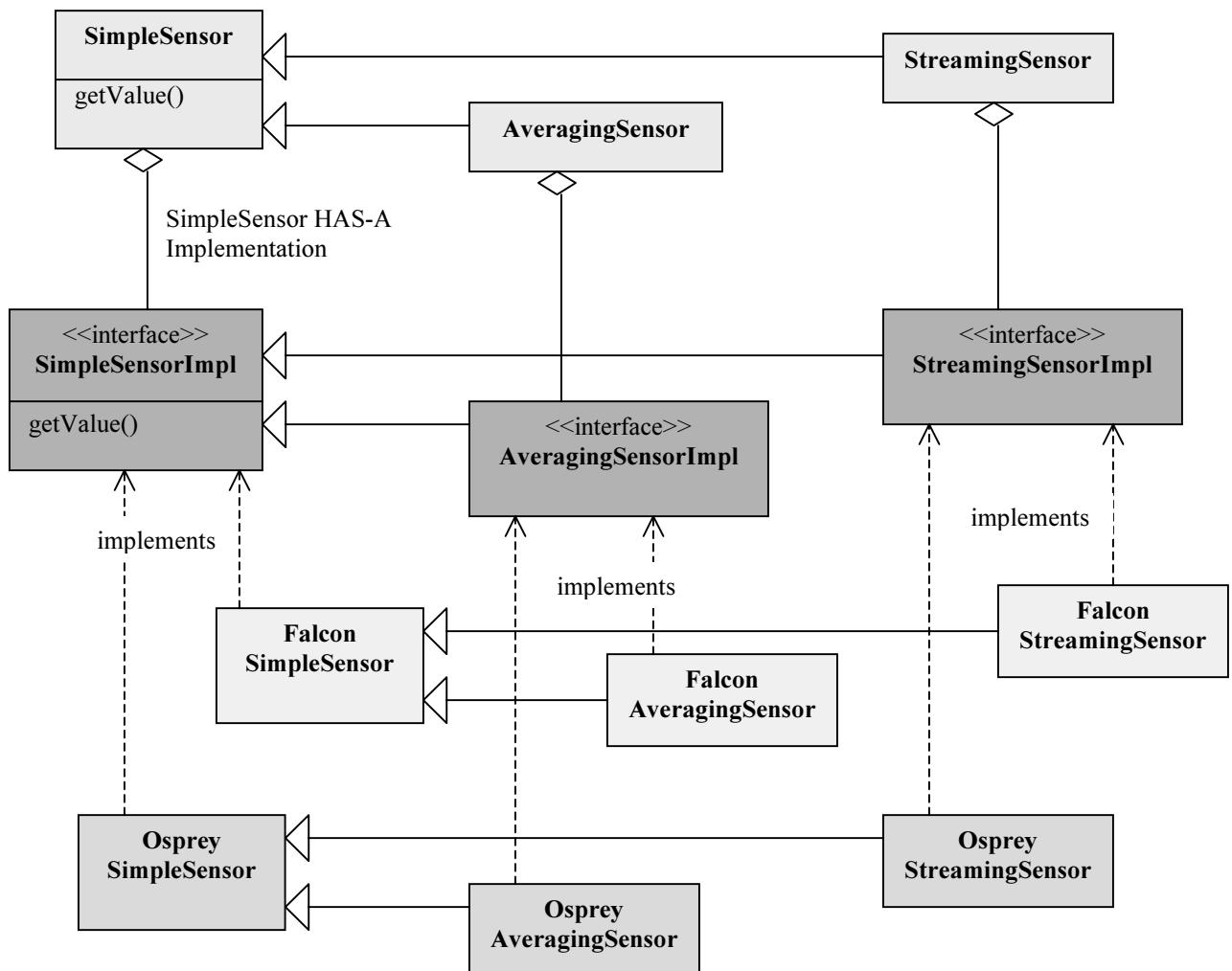
1. Advantages: Stabilize the interface to the subsystem through a single point of entry. When things change (and change is guaranteed), hopefully only that one façade class will be the focus of re-coding. You reduce/hide complexity and minimize coupling. The pattern does not preclude direct communication with subsystem objects if that is needed. You can create pluggable subsystems. You could even define an interface for the façade for more flexibility and substitute one façade class for another. Façade is often a Singleton.

Disadvantages: You add another class – the dependencies are moved and the communication may be harder to understand all the pieces even though it is easier to modify them.

2. Proxy is for access to another object (remote, virtual, device – etc); **Façade** is to control access to a subsystem. Proxy uses the same interface as the real object with transparent management of the services of that object; proxy requires a combination of the interfaces needed for accessing multiple objects through that single point of access.

Page 153:

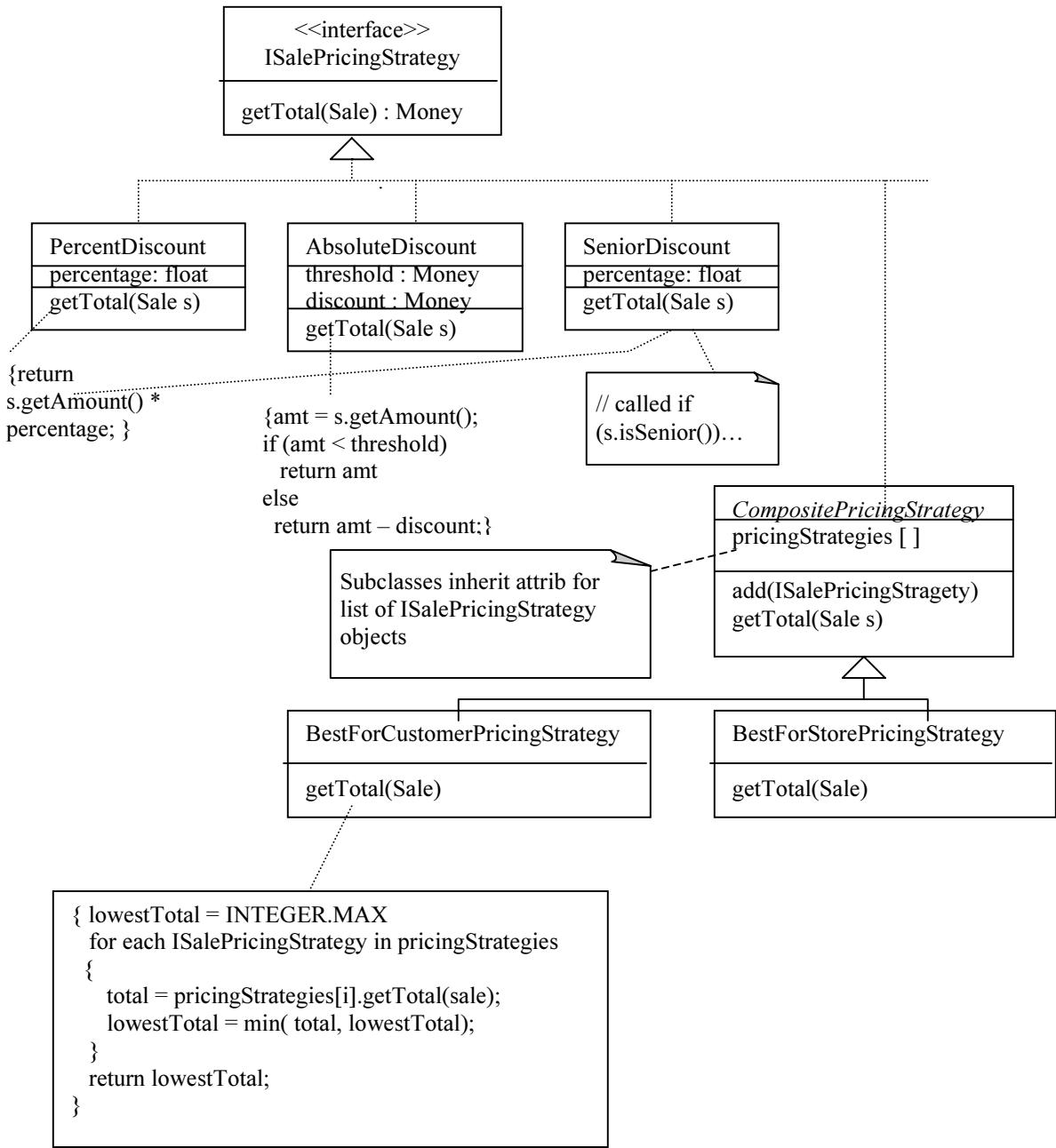
1. With **Chain of Responsibility**, an object can send a command to a chain of potential command handlers without knowing or caring what object(s) might field it. The order of the handlers can be shifted or prioritized dynamically. Reduces coupling, provides flexibility, but there is no guarantee there is a suitable handler.
2. Common logic is in the top generic hierarchy; specific logic is in the manufacturer-specific hierarchy; the interface hierarch in the middle provides the **Bridge**.



Page 157-159:

1. **Strategy** allows for separating algorithms and interchanging them as needed. This is sort of like adapting behavior (rather than class or object adaptation) in order to get the necessary behavior.
2. A Macro both IS-A command and HAS-A command – and is therefore a **Composite**. Likewise, CompositeHoliday both IS-A holiday and HAS-A list of holidays when a requester wants to display multiple holidays.
3. Implementation issues for **Composite**:
 - (1) Explicit parent references – simplifies traversal and management but invariant relationship between parent/child must be maintained
 - (2) Sharing components – difficult if (1) above is implemented – consider flyweight to avoid storing parent refs
 - (3) Maximizing the Component interface – may conflict with having meaningful operations for subclasses
 - (4) Declaring the child management operations – should add() and remove() be in Component (and thus need to be meaningful for Leaf classes) or only in Composite classes? Trade-offs in safety and transparency
 - (5) Should Component implement a list of Components? Would have a penalty for every leaf...
 - (6) Child ordering – do children need to be ordered? How should this be done?
 - (7) Caching for better performance – when are cached objects invalid/valid?
 - (8) Who should delete components? (In languages without garbage collection)
 - (9) What's best data structure for storing Components? Linked lists, trees, arrays, hash tables? Depends on what's most efficient for your application...
4. **Null Object** pattern used with Holiday:
 If none of the behaviors in a ConcreteStrategy classes are appropriate, the Client object could have a reference to a Null object, which would implement a do-nothing behavior for the method being called. Because of LSP, a NullHoliday object would be substitutable for the Holiday class and have a NO-op for getHolidays().
5. Consequences of using **Strategy**:
 - You create families of related algorithms which various contexts can reuse.
 - This is a more flexible alternative to subclassing – the algorithm can vary independently of the context and is therefore easier to change, easier to understand, and easier to extend; common behavior should be factored into a common superclass.
 - You eliminate conditional statements such as switch, making for easier maintenance and helping preserve the **Open/Closed Principle**.
 - You can choose various implementations for the same behavior
 - There is more overhead in coordinating strategy and context
 - There are more objects, therefore more overhead and more indirection, harder to understand.

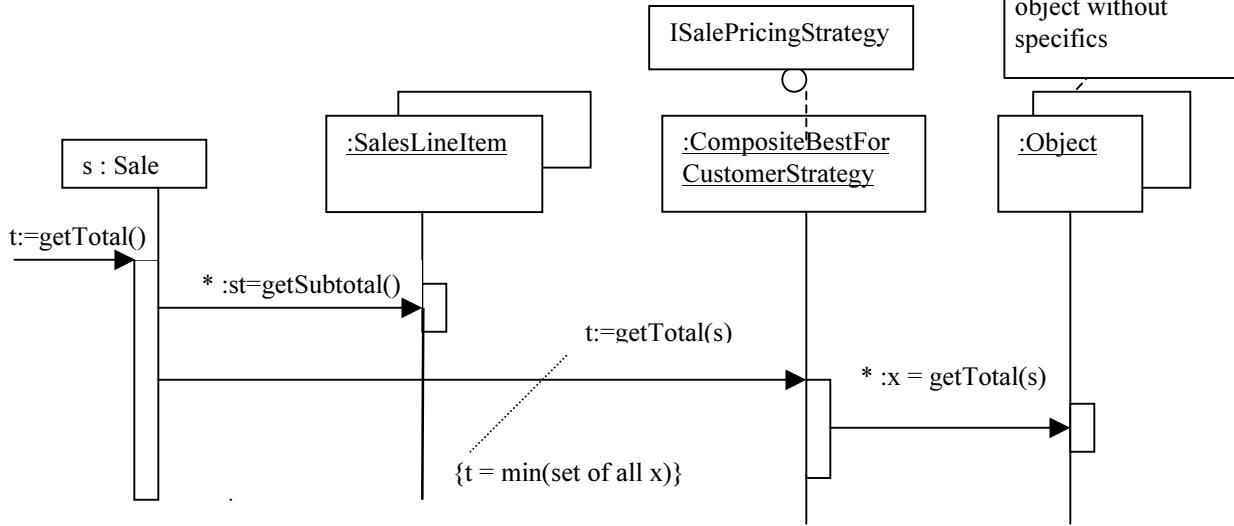
6a, 6b, 6c. Possible partial UML Class Diagram:



6d. Participants: Context = Sale; **Strategy** = `ISalePricingStrategy`; **ConcreteStrategy** = `PercentDiscount`.

7. More flexible – can update each day or more often. Keeps Sale class simpler, more cohesive to separate out the strategy.

8. Collaboration Diagram additions:



Sample Java Code:

```
// superclass so all subclasses inherit a List of strategies, but abstract to
// defer getTotal()
public abstract class CompositePricingStrategy implements ISalePricingStrategy {
    protected List pricingStrategies = new ArrayList();
    public add( ISalePricingStrategy s ) {
        pricingStrategies.add( s );
    }
    public abstract Money getTotal ( Sale sale );
}

// a Composite Strategy that returns the lowest total of its inner
// SalePricingStrategies
public class BestForCustomerPricingStrategy extends CompositePricingStrategy {
    public Money getTotal ( Sale sale ) {
        Money lowestTotal = new Money( Integer.MAX_VALUE );
        // iterate over all inner strategies
        ISalePricingStrategy strategy; // temp
        for ( Iterator I = pricingStrategies.iterator(); i.hasNext(); ) {
            strategy = (ISalePricingStrategy) i.next();
            Money total = strategy.getTotal ( sale );
            lowestTotal = total.min (lowestTotal );
        }
        return lowestTotal;
    }
}
```

Note what other pattern is built in to the previous page of code – **Iterator!**

Pages 164-165:

1.a. Pseudo-code for the four **Iterator** interface methods `first()`, `next()`, `current()`, `done()` for both an array and a singly linked list could look something like this.

<u>Array</u>	<u>Singly Linked List</u>
<pre>void first(){ index = 0; //index in Iterator obj array[0]; } void next(){ index = index + 1; } Object current(){ return array[index]; } // cast object to type of array boolean done(){ index == sizeof(Array) - 1; }</pre>	<pre>void first(){ traversalNode = head; //assume traversalNode in Iterator } void next(){ traversalNode = traversalNode.nextPtr; } Object current(){ //return data at that node } boolean done(){ traversalNode.nextPtr == null; }</pre>

Pseudo-code for a client to traverse each of the two concrete data structures:

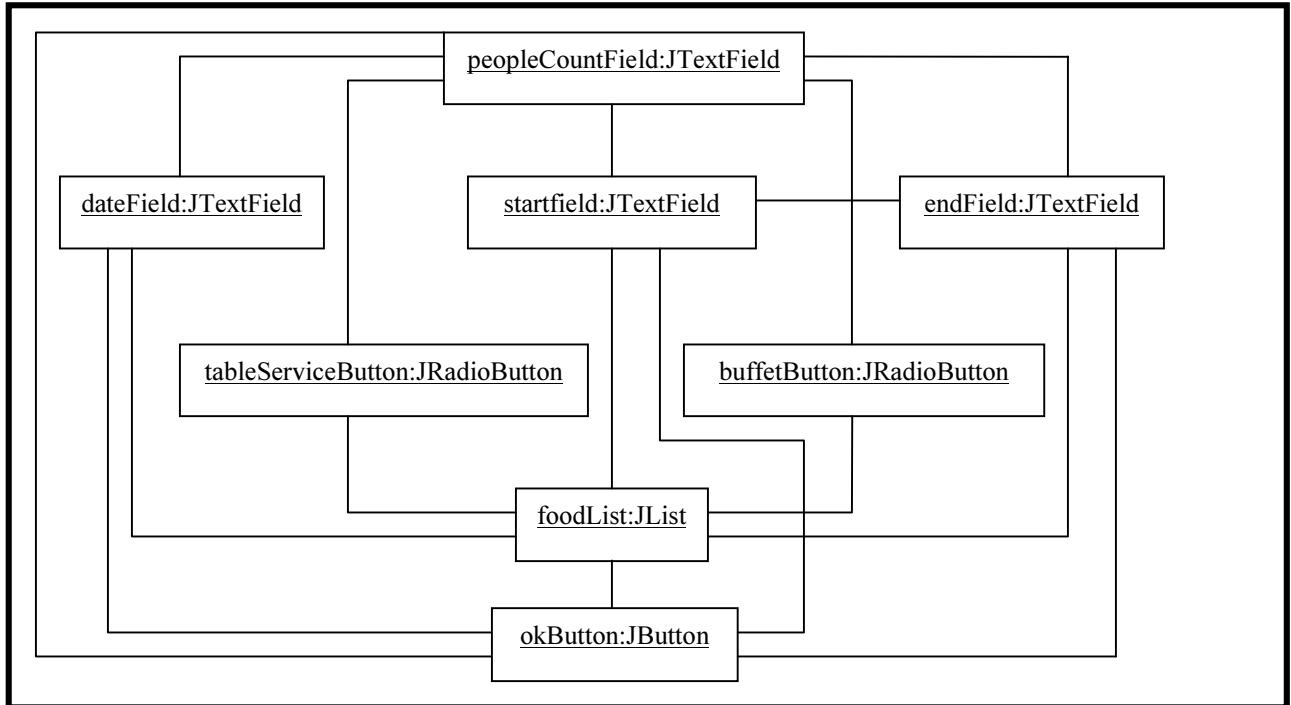
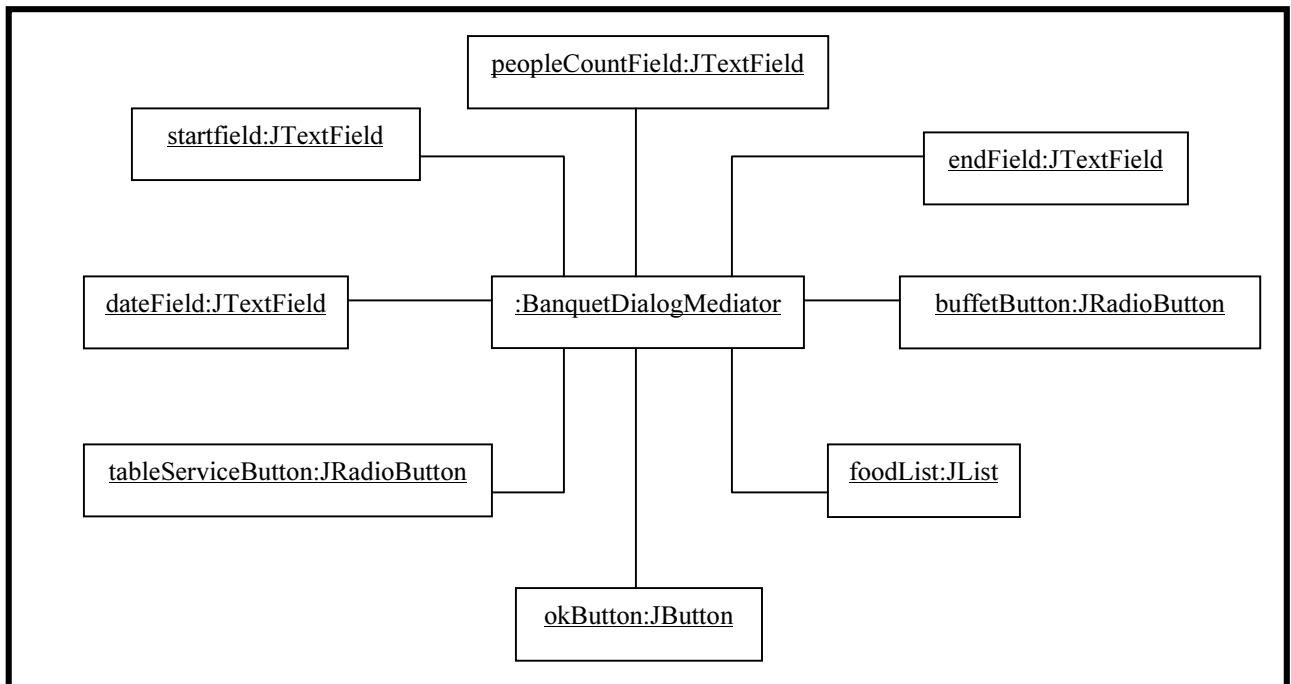
```
for (instanceName.first(); !instanceName.done(); instanceName.next()){
    //get instanceName.current() and examine for whatever...
}
```

b. No changes needed in client code to traverse a binary tree. You might have to know something about the tree – i.e., preorder, inorder, not ordered, etc. Different Iterators could exist for variations.

To have two traversals going at the same time on the array: simply create another instance of the Iterator for the instance of the data structure (each iterator has its own index, etc.)

c. Factory Method be used to determine what kind of **Iterator** to instantiate: Each data structure could have its own, appropriate factory method called `getIterator()`, which would return the appropriate Iterator instance for that particular data structure. All `getIterator()` methods would return an instance of an iterator which would meet the product interface for iterators.

2. **Iterator** is a specialized version of the **Adapter** pattern: it allows you to sequentially access the contents of collection objects, adapting the iterator interface to the structure of the collection.
3.
 - a. “Don’t call us, we’ll call you,” the Hollywood principle relates to the **Template Method** pattern as follows: The parent class (where the template method is defined) calls the submethods of its various subclasses for specialized behavior as needed within the framework of the template method.
 - b. The kinds of operations **Template** methods tend to call:
concrete methods, concrete AbstractClass methods, primitive methods (abstract methods which must be overridden in the subclasses), factory methods, and hook operations (with default behavior that subclasses can extend if necessary but which generally default to a No-op).
 - c. Implementation issues:
 - In C++ primitive operations called from a template method can be declared protected members to ensure they are ONLY called by the template method. Primitive operations that must be overridden are declared pure virtual, but the template method itself should be nonvirtual so it will NOT be overridden.
 - Minimize the primitive operations to be overridden so the task doesn’t get too tedious for clients.
 - Naming conventions can help identify the types of operations – i.e., MacApp framework prefixes template method names with “Do-“ such as DoRead.

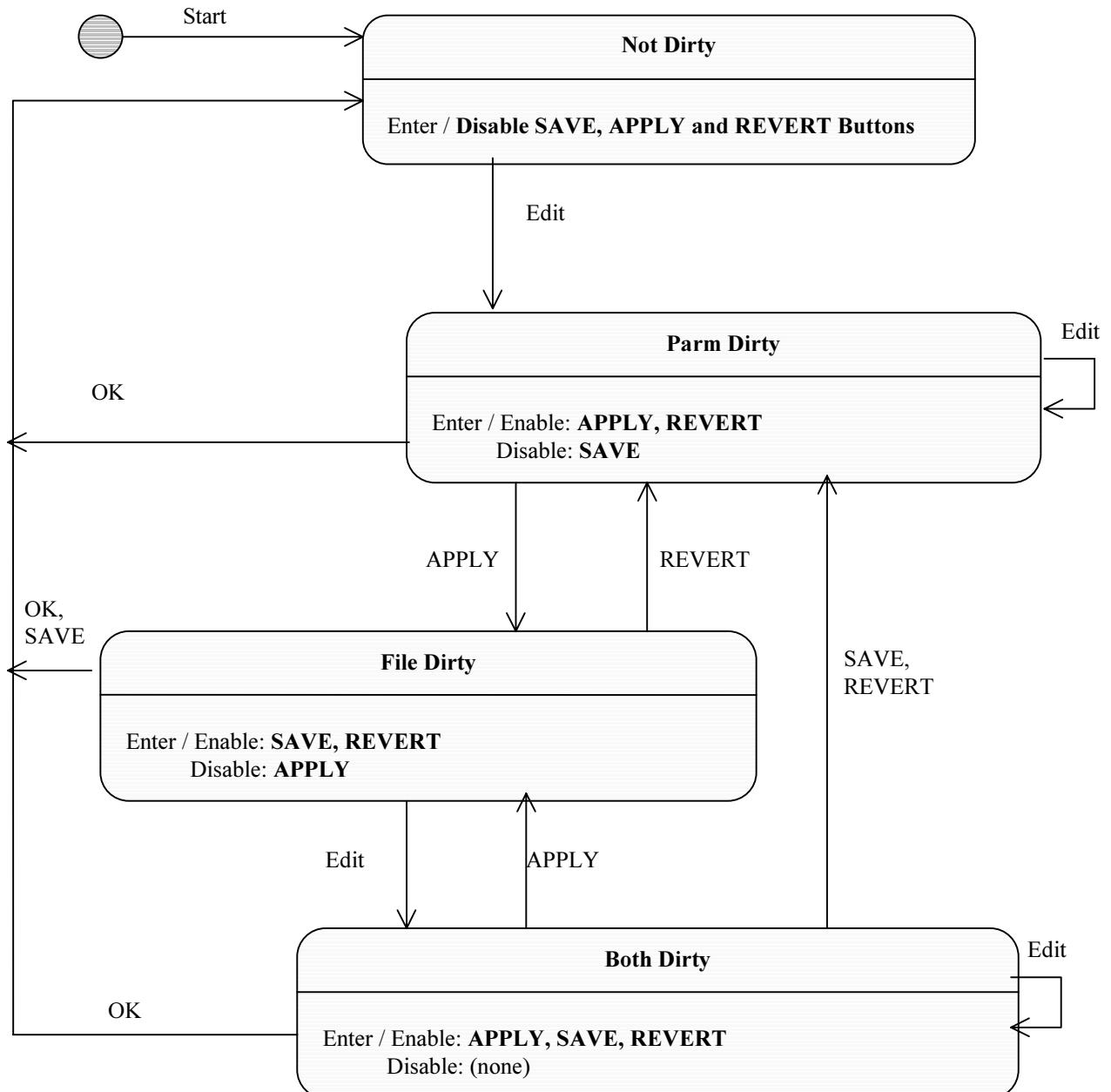
Page 171:**1. Mediator UML dependency diagrams:****Decentralized dependency management****Centralized dependency management with Mediator**

2. ChatServer uses the **Observer** pattern in a multithreaded environment. There is also an example of Singleton with the DocumentRegistry.

If multithreading was not used, the first client to connect to the server would then own the only connection thread, and no other client could connect until the first client disconnected.

Page 179:

1. **Memento** could capture state in the midst of an iteration by storing only the incremental changes. Since iteration is a predictable sequence there is a specific order in which mementos could be restored as needed.
2. **Snapshot** a larger variation of the simpler **Memento** pattern. Memento includes only the portion that uses Memento objects (no persistence) while Snapshot considers storage and retrieval of Mementos, which brings in classes such as (in Java) Serializer, FileOutputStream (bytes) or ObjectOutputStream (objects), Deserializer, and FileInputStream or ObjectInputStream. Snapshot requires more expertise and adds overhead if objects are to be serialized. Snapshot can also create nonpersistent copies stored in memory with the Caretaker instance, as does the simpler Memento.
3. **Command** pattern code review example (for starters):
 First, the abstract Command class with virtual destructor and execute(), the constructor is protected.
 OpenCommand inherits publicly from Command and is passed a reference to the current Application instance in its constructor. Its execute method calls its own helper method, AskUser() to get the name of the document to open. The Document is created and opened.
 PasteCommand inherits publicly from Command and receives a reference to the current Document in its constructor. Execute takes whatever is on the clipboard and pastes it into the current Document (wherever the cursor is).
 SimpleCommand is set up as a template class to create commands which aren't undoable and don't require arguments – receiver and action references are both passed to the constructor. Execute applies the action to the receiver.
 MacroCommand inherits publicly from Command and keeps a list of Commands as well. Add() appends a new command on the list; Remove() takes a command off the list. Execute walks the list and calls Execute() for each command on the list. It would be possible to use a Marker interface for commands that are undoable – walk the list in reverse order and call Undo() or Unexecute() for commands with the Marker.
4. **Template Method** (instead of **Command**) could implement top-level undo logic by managing the overall sequencing and major steps for a class, then calling undo() or unexecute() on subclasses for specific variations where the undo will work.

Page 183:**1. Complete the State Diagram**

Pages 191-192:

1. The 7 problems in Lexi's design (GoF Chapter 2):

- (1) Document structure – how we organize and represent the document internally.
- (2) Formatting – how we arrange text and graphics into lines and columns.
- (3) Embellishing the user interface – we want to be able to add and remove embellishments easily as needed.
- (4) Supporting multiple look-and-feel standards – the editor should be able to adapt to differing standards easily (such as Motif, Presentation Manager).
- (5) Supporting multiple window systems – windowing systems vary – our editor should be as independent of the window system as possible.
- (6) User operations -- we want a uniform way to access various user interfaces (buttons, pull-down menus, etc.) and undoing effects.
- (7) Spell-check and hyphenation – we want to minimize the changes when we add a new analysis such as these two.

2. **Glyph**: an abstract class for all objects that can appear in a document structure.

What GoF pattern is obvious? **Composite**, both IS-A and HAS-A.

3. The **Compositor** class is used to encapsulate a formatting algorithm

What GoF pattern is used? **Strategy**.

4. To embellish the user interface, but not use inheritance, use a **Decorator**.

What other GoF pattern is shown? **Composite**.

5. For multiple look-and-feel standards use an **Abstract Factory**.6. To support multiple window systems use the **Bridge** pattern.

We can't use an **abstract factory** for this because we already have vendor-specific class hierarchies, one for each look-and-feel standard, and therefore probably won't have common abstract product classes for each kind of widget. So, we'll create our own abstract product class hierarchy.

Separating windowing functionality into window and windowImpl hierarchies lets us have two separate hierarchies which can work together through our abstract product hierarchy to create the needed window objects; this is an example of the **Bridge** pattern.

7. For multiple user interfaces for the same operation (buttons, menu selections, hot keys, etc.) and support for undo and redo, use the **Command** pattern. This pattern also allows us to encapsulate a request and parameterize menu items.8. To do spell checking and hyphenation and to access and traverse sections of the document we can use the **Iterator** pattern.9. To do an analysis of the glyphs as we traverse them, we can use the **Visitor** pattern. This analysis is then separate from the traversal so the same traversals could be used for

various analyses. A given analysis can distinguish different kinds of glyphs but we don't pollute the glyph interface with all kinds of new stuff. **Visitor** allows us to "visit" an encapsulated analysis class over each glyph, and the number of potential visitors is open-ended.

10. The best object-oriented designs use many design patterns that dovetail and intertwine because patterns do not appear in isolation – one pattern calls for another, and that creates a context for yet another. It is the overlap that creates the richness and elegance in a software design.

This is **difficult** for us as programmers because it is no longer enough to simply know a given pattern – we have to also know relationships among patterns, trade-offs, issues, etc. Again, we can learn these combinations from the experts, but we have to also know the individual patterns to be able to put them together appropriately.

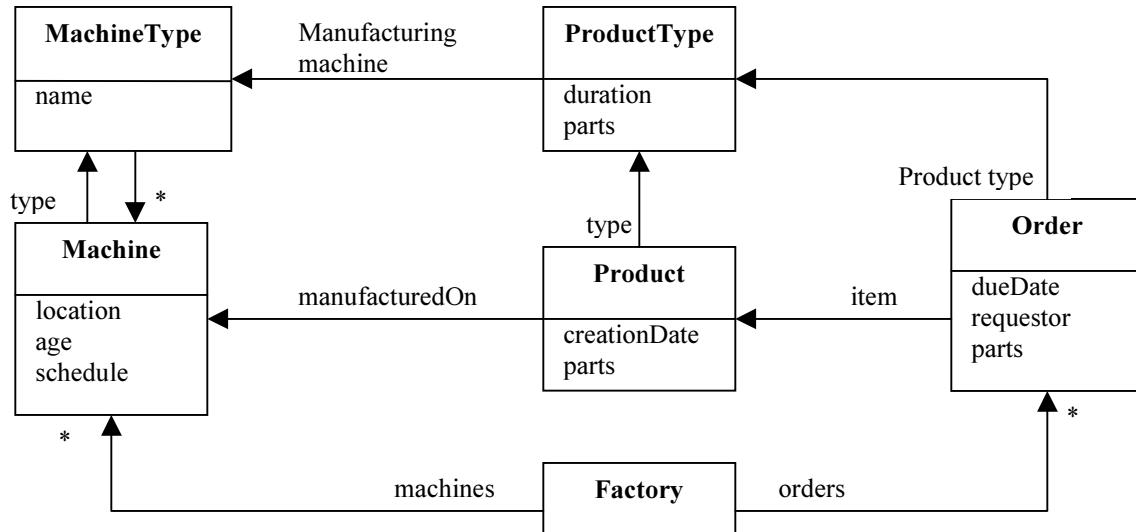
This is also **powerful** because it combines practical experience (ours and our mentors' – including those we know as authors) with our creativity and insight to develop a niche for ourselves as top-notch programmers – we make ourselves valuable. We avoid dead-ends. We shape our vocabulary, our thoughts, our approaches to solving software problems. We train ourselves to look for what works, document it, share. All on the job.

Pages 209-210:

1. Using Type Objects:

An alternative to creating a class hierarchy for kinds of machines and another for kinds of products...

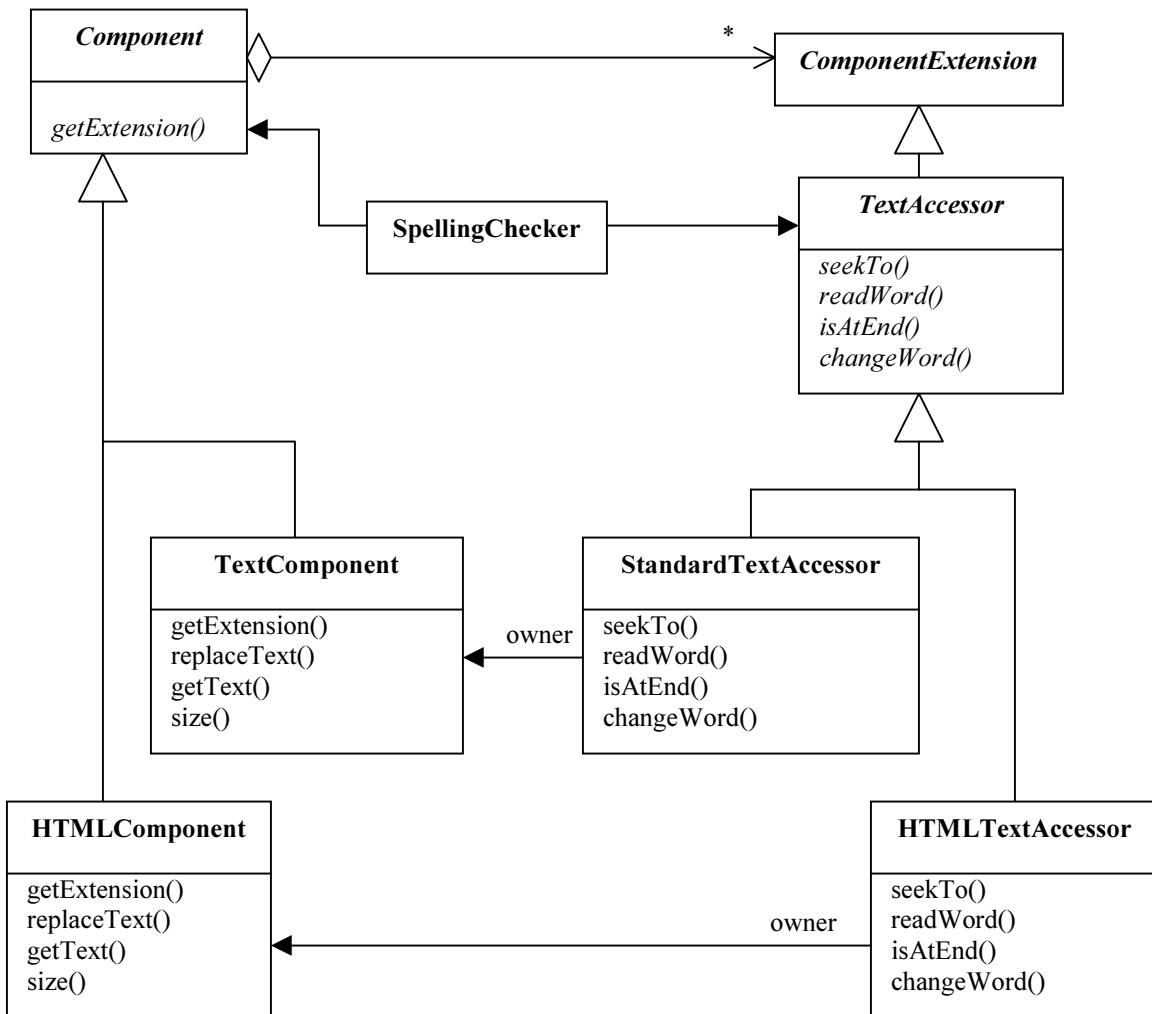
Create classes for *Machine*, *Product*, *MachineType* and *ProductType*. A *ProductType* has a “manufacturing plan” which knows the *MachineTypes* that make it. Since a particular instance of *Product* was made on a particular set of *Machines*, you can identify which machine is at fault when a product is defective. When orders come in, the system figures out the earliest it can fill the order. Each order knows what kind of product it needs to produce (and product is probably an example of Composite since a product may be made up of other products, which will probably require many machines). Additional classes could be *Order* and *Factory*. Order’s constructor gets passed *ProductType*, dueDate, requestor, parts, item. Factory’s constructor gets passed Machines and Orders and Factory acts as a Façade, creating the order and then scheduling it.¹



2. Browse all of the movies the store offers by simply going through a list of all of the TypeObjects (which might have references to its various copies).

¹ *Pattern Languages of Program Design 3*, Addison Wesley, p. 59, 60.

3. UML diagram for the **Extension Object** example of adding a SpellChecker interface.



ComponentExtension is the common base class for extensions in this example.² It exists to define the return type for GetExtension(). TextComponent overrides GetExtension to return a TextAccessor object. The SpellChecker would be implemented as follows: (1) traverse components in the document; (2) ask each component for its TextAccessor extension; (3) if the component returns a TextAccessor extension object, use it to spellcheck the component (downcasting it to a TextAccessor if necessary); otherwise skip the component.

² *Pattern Languages of Program Design 3*, Addison Wesley, p. 80.

Pages 274-275:**1: Sales Force Automation Software Architecture initial considerations³:****a. Requirements:**

- means for sales agents to submit orders via handheld device (wireless)
- submit orders to office-based order-coordinating server
- server interacts with systems of multiple business partners involved in order fulfillment
- responses communicated back to mobile client regarding state of the order

b. Associated Problems:

- wireless connections cannot be guaranteed
- multiple partners involved might have differing data formats for the systems with which server must integrate

c. Candidate Architecture(s):

Although elements of the needed system can be seen in nearly every category, the system falls primarily into the category of distributed systems. There are several possibilities: Broker, Pipes and Filters...

d. Comparisons

Pipes and Filters is an excellent match for many aspects of the system. For example, the order fulfillment process is essentially a series of data processing steps. The lack of a guaranteed connection requires a store-and-forward mechanism for transfer of orders.— a buffering pipe addresses this. Filters would also address the transformation of data from a transport format into a presentation format on the mobile devices and into multiple formats of the business partner system.

Though Pipes and Filters is the central pattern, a Layers pattern could be adopted in the mobile client to wrap the low-level message-queuing services and expose them to the client-based order entry application. This would allow the client application to be easily modified because such a wrapper provides a well-defined interface to messaging services by following an industry standard supported by multiple development tools.

e. Implementation Guidelines:

(see pattern – one of the guidelines is to define processing steps)

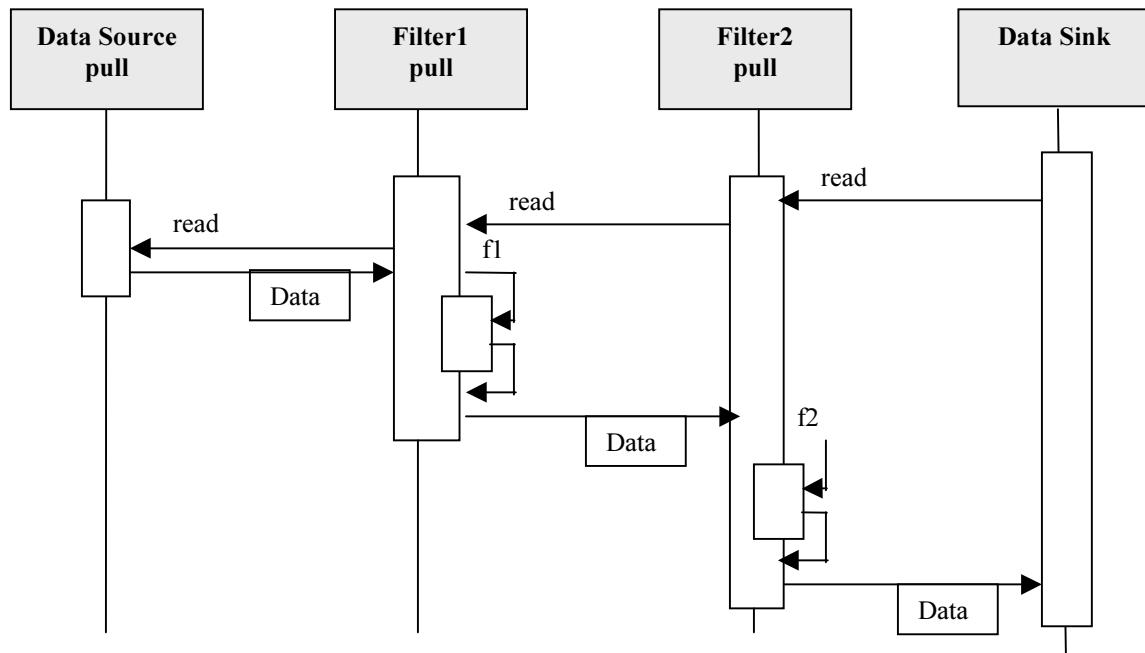
f. Processing Steps:

- order input on mobile device
- transformation of order to transport format (such as XML)
- queuing, transmission, and acknowledgement of messages (such as message queuing with buffering capability)
- transformation of transport formats into partner system formats (such as Microsoft's Biztalk Server)

³ See *Application Development Trends*, “Building on Patterns,” by Robert Hartman, May 2001.

2: Pull Scenario for Pipes & Filters:

Control flow is started by the data sink calling for data.



Notes

Index

abstract, abstraction 28, 32, 33, 34, 40, 41, 44, 45, 50, 56, 57, 58, 64, 81, 85, 102, 109, 110, 114, 119, 128, 139, 142-143, 146, 148-149, 150, 159, 162-163, 167, 180, 189, 198-199, 205, 243, 250, 258, 260-263, 270, 281, 301, 304-305, 312-313, 314-315, 326, 327

agent 32, 260-263

aggregate, aggregation 100, 109, 160-161, 281, 294, 305

algorithm 45, 109, 155, 243, 249, 251, 266

architecture, architectural 31, 63, 148, 200, 237-267, 282, 324, 326, 329, 337, 339, 341, 343, 345

association 35

Audio Clip example 120-121

balance 28, 59, 339, 343, 345, 347

Beck, Kent 50

behavior 32, 34, 35, 44, 45, 52, 54, 66, 96-97, 100, 107, 109, 128, 154-155, 162-163, 175, 180-181, 202-203, 205, 243, 250, 257, 265, 266

blackboard architecture 239, 248-251

boundary 33, 62, 241

broker pattern 131, 239, 252-255, 271, 273, 301, 305, 315

C++ 26, 32, 48, 54, 102, 110, 112, 120, 184, 206-207

class

base 34, 40, 45, 54, 55, 58, 142, 149, 198, 242, 250, 287

derived 34, 40, 55

method (see method: static)

variable 35, 122, 164, 217, 230

client 40, 46, 58, 59, 65, 72, 77, 92, 96-97, 99, 112-113, 114-115, 117, 118, 120-121, 126-127, 129, 131, 132, 138-139, 142-143, 146, 149, 154-155, 164, 169, 171, 176, 181, 185, 188, 196-197, 204-205, 209, 228-229, 252, 254-255, 257, 266, 268, 270-271, 273, 280, 285, 288, 292, 296, 316-317

clone 119, 132, 133-134

CLOS (Common Lisp Object System) 266

cohesion, cohesive 33, 42, 52, 58, 73, 100, 104, 167, 197, 241

collaborate, collaboration 32, 63, 97, 125, 158, 159, 184, 220, 326

collection 31, 46, 48, 92, 100, 143, 160-161, 250, 284

Combined Method 49, 52-53

command 46-49, 151, 157, 173, 175, 176-177, 179, 180, 290-291

component 34, 40, 59, 61, 100, 129, 142-143, 148, 166, 204-205, 210, 218, 236, 237, 241, 245, 249, 253, 254-255, 256-258, 261-263, 265-267, 269-270, 293, 295, 297, 299, 301, 305, 309, 313, 315, 317, 326, 329, 339, 347

compose, composition 90, 102, 109, 143, 163, 241, 260, 262, 295, 309, 326

Composed Method 51

concrete 110, 112-113, 114, 118-119, 122, 128-129, 137, 142, 146, 148, 150, 154, 158, 160, 162-163, 164, 166, 168, 176, 180, 184-187, 198, 204

constraint 28, 34, 113, 143, 223, 266, 285

constructor 34, 111, 120-121, 123, 154, 185, 188, 231

containment 32, 34, 35, 46, 100, 144, 150, 161, 257

control, controller 33, 57, 62, 97, 100, 117, 121, 129, 131, 135, 137, 167, 169, 177, 217, 223, 249-251, 256-259, 260-263, 271, 281, 286-287, 288-289, 290-291, 293, 296-297, 298-299, 300, 303, 305, 311, 313

Copy Program example 56

CORBA (common object request broker architecture) 130, 200, 239, 254

coupling 32, 33, 47, 52, 59, 61, 93, 100, 104, 131, 139, 151, 161, 167, 242-243, 253-254, 257-259, 281, 287, 301, 305, 307, 309

delegate, delegation 26, 90-92, 96, 102, 154, 168, 289, 293, 300-301, 302, 304-305, 308, 313, 317

dependency 40-41, 56-57, 71, 75, 76, 79, 83, 93, 100, 109, 111, 113, 115, 119, 121, 127, 136, 139, 148, 149, 161, 163, 167, 168-169, 171, 173, 175, 180, 187, 202-203, 222, 237, 240, 243, 250, 253, 258, 264-265, 266, 270, 281, 283, 284, 295, 305, 306-307, 309, 312

Design by Contract 55

distribute, distribution 30, 32, 53, 167, 188, 239, 252-254, 263, 271, 273, 282, 289, 293, 301, 305, 309, 324

Don't Talk to Strangers 91, 100

Door Controller example 129, 137

Door Timer example 58, 66

Double-Checked Locking (DCL) 120, 230-231

Eiffel 49

encapsulate, encapsulation 28, 29, 33, 47, 53, 96, 97, 113, 117, 121, 142, 148-149, 155, 163, 166, 169, 173, 175, 180, 187, 192, 223, 224, 245, 265, 266, 269, 281, 293, 297, 299, 303, 305, 315

entity 33, 44, 62, 281, 302-303, 305, 306-307, 309, 311, 315, 317

extend, extension 28, 40, 44, 45, 55, 204-205, 218, 219, 263, 269

filter 92, 127, 128, 239, 244-247, 275, 281, 284-285, 286-287, 291

flexible, flexibility 26, 123, 129, 150-151, 158, 177, 191, 198, 245, 254, 271, 287, 295, 297, 311

Flowlar, Martin 51

framework 9, 34, 57, 110, 162, 205, 259, 265, 282-283, 317, 323, 324, 326-327, 339

GoF Patterns

abstract factory 26, 100, 109, 112-113, 117-118, 119, 120-121, 122, 138, 148, 159, 192, 303, 315

adapter 109, 126-127, 131, 132, 135, 154, 157, 160, 165, 255, 270, 293, 301

bridge 109, 113, 130, 148-149, 153, 243, 258

builder 109, 114-117, 118, 125

chain of responsibility 109, 142, 150-151, 153, 284, 287

command 26, 109, 150, 157, 175, 176-177, 179, 290-291

composite 109, 118, 129, 142-143, 144, 146, 150, 154, 157, 159, 160, 161, 188, 281, 294-295, 303, 306-307, 308-309

decorator 90, 109, 118, 127, **128-129**, 131, 132, 137, 151, 203, 205, 219, 285, 286-287
façade 52, 100, 109, 131, **138-139**, 141, 162, 250, 303, 304-305, 307, 308-309, 311, 313, 317
factory method 26, 100, 109, **110-111**, 112-113, 117, 119, 144, 146, 160, 164, 315
flyweight 109, **144-147**, 181, 188
interpreter 109, **188-189**
iterator 46, 49, 109, 142, **160-161**, 164, 165, 187, 188, 310-311
mediator 109, 138, 150, **166-167**, 168, 171, 253
memento 98, 109, **172-175**, 179
observer (publish-subscribe) 109, 166, **168-169**, 171, 258
prototype 26, 109, 112, **118-119**, 125
proxy 43, 52, 90, 109, **130-131**, 132, 133-136, 141, 151, 199, 207, 281, 301
singleton 26, 96, 109, 112, **120-121**, 122-124, 136, 138, 159, 181, 196, 230
state 90, 109, 144, 146-147, 167, 169, 172-175, 179, **180-181**, 183, 187, 203, 274,
strategy 90, 109, 128, 149, 151, **154-155**, 157-159, 162, 199, 203, 236, 243, 249, 251, 270, 286-287, 305, 307
template method 109, 128, 150, 154, **162-163**, 165, 179, 201, 286-287
visitor 26, 90, 109, 142, **184-187**, 188, 205, 219

GRASP patterns, **100-101**, 104

HashSet Example 124

HashTable Example **132-134**, 200

Hearsay-II Example 248, 250

hierarchy 28, 32, 34, 59, 65, 66, 111, 113, 119, 129, 143, 144, 149, 150-151, 185, 186, 188, 192, 203, 209, 250, 258, 261-262, 291, 315

Hollywood Principle 165

Immutable Pattern **94-95**

implementation 26, 28, 31, 32, 51, 97, 109, 113, 122, 129, 148-149, 155, 157, 165, 166, 169, 175, 197, 199, 202, 203, 204, 207, 216, 218, 220, 222, 224, 226, 228, 293, 301, 305, 307, 314

indirect, indirection 52, 92-93, 100, 104, 127, 207

inheritance 26, 34, 35, 55, 65-66, 90-91, 98, 102, 111, 129, 163, 191, 242, 303, 326
multiple 35, 90, 99

instance, instantiation 55, 61, 90, 93, **94-95**, 96-97, 99, 109, 110-111, 112-113, 118-119, 120-121, 122, 124, 129, 131, 132, 143, 144-145, 147, 151, 159, 160, 164, 166, 169, 171, 175, 177, 196-197, 199, 200, 202-203, 207, 217, 303, 313

intelligence 32, 249

interface 26, 32, 34, 35, 40-41, 43, 45, 49, 52, 58-59, 62, 64, 66, 76-77, **90-99**, 100, 102, 103, 111, 113, 117, 122, 126-130, 135-137, 138-139, 141, 142-143, 148-149, 153, 155, 158-159, 160-161, 164, 166, 168, 172, 175, 177, 185, 191-192, 198, 199, 204-205, 207, 210, 241, 243, 254, 256, 257, 261, 262, 266, 269, 270, 281, 287, 301, 303, 305, 310, 312-313, 316-317, 324-325, 329, 341

invariants 28, 49

Java 42, 43, 48, 90-92, 95, 98-99, 110, 112, 118, 120-124, 126, 130, 132, 135, 138, 139, 142, 146, 148, 150, 154, 160, 164, 166, 168, 171, 176, 188, 198, 199, 202, 204, 214, 216, 226, 228, 229, 230-231, 240, 266, 288, 293, 295, 297, 299, 302, 305, 309, 311
J2EE 235, 241, Chapter 10

layer, layers architecture 67, 100, 148, 239, **240-243**, 250, 264, 270-271, 280, 301, 305, 315

Law of Demeter **60-61**

Liskov, Barbara (and Liskov Substitutability Principle, LSP) **41, 54-55**

maintainable, maintenance 24, 30, 43, 45, 55, 59, 61, 115, 187, 237, 241-242, 251, 254, 261, 263, 269, 289, 293, 295, 299, 309, 315, 328, 341

4 Design Patterns

Marker Interface 89, **98-99**, 177, 179, 307

Martin Metrics (Robert Martin) 71, **83-86**

Maze Example 107, 112, 117

method 26, 32, 34-35, 46-47, 48-49, 50-51, 52-53, 55, 56, 59, 60-61, 64-66, 68, 77, 91, 92-93, 95, 97, 98-99, 100, 102-103, 110, 112, 115, 117, 119, 122, 127, 128-129, 131, 132-133, 135, 142, 144, 146, 153, 158-159, 161, 162-163, 164, 176, 184, 187, 199, 216-217, 218-219, 220-221, 223, 224-225, 227, 228, 230-231, 254, 287, 300-301, 303, 305, 311, 316-317

static 120, 121, 122

(see also Factory Method and Template Method under GoF patterns)

Meyer, Bertrand 45, 55

MFC (Microsoft Foundation Classes) 258

Microkernel architecture 239, **268-271**

Modem.java example 42

MVC (Model-View-Controller) 168, 239, **256-259**, 261, 273

object state 35, 47, 144, 147, 200, 219, 220, 221, 225, 228, 245, 261, 274, 292, 296, 298, 305, 311, 317, 324

OMG (Object Management Group) 239, 252

Open-Closed Principle, OCP 28-29, 40-41, **44-45**, 54-55, 64, 324

operation 32, 50, 92, 96-97, 99, 100, 205, 207, 219, 225, 231, 270, 313

PAC Agents (Presentation-Abstraction-Control) 239, **260-263**, 273

package 72, 104, 135, 270, 343

abstractness 85

architecture 73-81

coupling 75-79, 83

stability 79, 81, 83

parameter, parameterization 28, 110, 112, 177, 181, 183, 192, 254, 265, 326

pattern

language 320-323, 327

systems 31, 107, 135, 192, 357

template 26, 27

vs. heuristics 33

performance 231, 241, 243, 247, 252, 270, 271

pipe, pipes 226, 239, **244-247**, 275, 286-287

pluggable 112, 148, 258-259, 268, 287, 326

polymorphism, polymorphic 33, 34, 44, 97, 100, 104

pre/post-condition 49, 55

protocol 32, 167, 240, 254, 266

Publish-Subscribe (see GoF Observer Pattern)

query 46, 48-49, 173, 204, 218, 281, 311

refactor, refactoring 24, 25, 43, 51, 61, 73, 334, 338, 343, 345, 347, 349

reflection 199, 202, 239, 250, **264-267**

RMI (Remote Method Invocation) 92, 130, 301, 309

representation 115, 117, 144, 189, 249, 261, 266

ResourceAllocator example 44, 64

ResourceExample 52

responsibility 41, 42-42, 47, 48, 68, 79, 89, 96, 100-101, 109, 128, 163, 253, 261-262, 266, 282, 288-289, 292-293, 296, 298-299

reuse, reusable 30, 34, 40-41, 45, 55, 56-57, 59, 62, 72-73, 90-91, 93, 96-97, 100, 111, 147, 149, 163, 167, 196, 197, 201, 223, 225, 240-241, 243, 245, 251, 255, 287, 291, 293, 295, 297, 299, 313, 315, 325, 326, 327, 341, 357, 359

scaleability 181, 252, 271

scope, scoping 26, 34, 349

sequence diagram 63, 284, 286, 288, 290, 292, 294, 296, 298, 300, 302, 304, 306, 308, 310, 312, 314, 316, Appendix A

serialization 99, 174-175, 302

server 77, 92, 109, 130, 169, 171, 200, 229, 252, 254, 257, 268, 270-271, 273, 274, 295, 303, 309, 311, 317

signature 48, 113

Smalltalk 50, 97, 116, 258

Snapshot – see GoF Pattern, Memento

software 225, 261, 264-267, 269, 293, 325, 326, 329, 331, 336, 337, 339, 348, 359
design 25, 27, 30-31, 40-41, 43, 47, 49, 51, 52, 55, 56, 57, 101, 109, 147, 189, 203, 258, 273, 320, 329
life cycle 63
rot 25, 31

structure, structural 40, 48, 56, 58, 61, 66, 107, 109, 143, 144, 151, 161, 164, 173, 175, 184-187, 189, 197, 199, 216, 237, 239, 243, 245, 248-249, 250, 253, 258, 261, 262-263, 265, 266, 270, 282, 284, 288, 292, 294, 296, 298, 300, 302, 304, 306, 308, 310, 312, 314, 316

subclass, subclassing 51, 55, 66, 91, 102, 109, 111, 117, 119, 121, 125, 126, 129, 132, 139, 143, 148, 153, 154-155, 159, 162-163, 167, 180, 183, 187, 188, 198-199, 202-203, 205, 265

subsystem 30, 100, 138-139, 141, 237, 249

superclass 143, 146, 150, 155, 163, 180, 287

synchronize, synchronization 49, 52, 95, 201, 219, 220, 223, 226-227, 228, 230-231, 244, 254, 259, 281, 301, 303, 316-317

teams 241, 282, 293, 297, 320, 323, 324, **329**, 345

test, testing 24, 34, 44-45, 49, 60-61, 75, 96-97, 251, 255, 285, 323, 329, **330-331**, 351

thread, threading 30, 49, 53, 95, 120, 124, 169, 171, 214-231, 263, 301, 329

UML (Unified Modeling Language) 9, 10, 66, 68, 102, 107, 122, 135, 137, 153, 158-159, 171, 210, Appendix A

use case 63, 100, 282, 305, Appendix A

wrapper 61, 127, 128-129, 324

!Name That Pattern!

(The 23 GoF Patterns)

	Pattern Name	Pattern Description
1		Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
2		Attach additional responsibilities to an object dynamically; provide a flexible alternative to subclassing for extending functionality.
3		Compose objects into tree structures to represent part-whole hierarchies; let clients treat individual objects and compositions of objects uniformly.
4		Define an interface for creating an object, but let subclasses decide which class to instantiate; let a class defer instantiation to subclasses.
5		Decouple an abstraction from its implementation so that the two can vary independently.
6		Provide a unified interface to a set of interfaces in a subsystem; define a higher-level interface that makes the subsystem easier to use.
7		Represent an operation to be performed on the elements of an object structure; define a new operation without changing the classes of the elements on which it operates.
8		Ensure a class only has one instance, and provide a global point of access to it.
9		Define an object that encapsulates how a set of objects interact; promote loose coupling by keeping objects from referring to each other explicitly, and vary their interaction independently.
10		Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request; pass the request along a series of receiving objects until an object handles it.
11		Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
12		Separate the construction of a complex object from its representation so that the same construction process can create different representations.
13		Given a language, define a representation for its grammar along with a way to use the representation to interpret sentences in the language.
14		Define the skeleton of an algorithm in an operation, deferring some steps to subclasses; let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
15		Provide a surrogate or placeholder for another object to control access to it.
16		Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
17		Allow an object to alter its behavior when its internal state changes; the object will appear to change its class.
18		Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
19		Use sharing to support large numbers of fine-grained objects efficiently.
20		Convert the interface of a class into another interface clients expect; let classes work together that couldn't otherwise because of incompatible interfaces.
21		Specify the kinds of objects to create using an archetypal instance, and create new objects by copying this archetype.
22		Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
23		Define a family of algorithms, encapsulate each one, and make them interchangeable. This pattern lets the algorithm vary independently from clients that use it.

*** WORD SEARCH ***
Finding Object Design Pattern Words

T	R	U	E	L	B	A	T	U	M	M	I	G	I	N	O	T	E	L	G	N	I	S
E	D	A	C	A	F	V	E	C	A	F	R	E	T	N	I	_	R	E	K	R	A	M
M	O	D	E	L	A	L	L	W	E	L	L	A	S	N	S	T	R	A	T	E	G	Y
P	L	A	Y	E	R	S	Y	O	F	D	A	R	H	T	O	P	O	N	E	A	T	R
L	-	S	-	P	X	G	G	W	E	A	K	E	E	N	-	D	O	R	K	I	E	O
A	Y	N	O	V	E	L	A	C	E	F	R	P	D	A	S	I	R	O	L	I	A	T
T	A	K	E	I	T	N	O	M	A	I	Y	E	-	S	T	O	R	I	E	B	N	C
E	T	A	T	S	A	R	E	I	T	T	G	C	R	A	P	O	B	O	I	L	U	A
_	F	E	A	I	A	W	P	A	O	Z	Y	H	G	N	E	I	S	S	P	A	L	F
M	O	T	H	T	E	D	N	T	Q	U	E	E	T	I	S	O	P	M	O	C	L	_
E	R	R	O	O	N	C	O	R	D	A	L	E	O	N	T	H	E	B	A	K	_	T
T	E	R	A	R	E	R	S	I	T	E	A	R	O	T	A	I	D	E	M	B	O	C
H	F	E	L	O	P	E	L	F	D	R	O	P	P	I	N	G	S	G	L	O	B	A
O	L	T	E	R	A	V	A	I	L	M	S	O	P	-	C	-	O	S	M	A	J	R
D	E	E	N	D	B	R	I	D	G	E	R	E	K	O	R	B	E	S	O	R	E	T
E	C	R	R	-	I	E	L	A	R	M	O	D	E	L	J	S	K	R	A	D	C	S
R	T	P	E	I	C	S	H	_	O	E	R	D	I	E	S	P	A	E	T	S	T	B
S	I	R	K	-	N	B	F	A	W	N	D	O	C	T	O	R	E	T	P	A	D	A
D	O	E	O	P	L	O	D	D	O	T	A	T	S	A	E	O	L	L	I	D	A	M
N	N	T	R	I	_	U	P	Q	U	O	_	D	I	D	E	X	T	I	P	A	E	C
A	S	N	C	N	O	T	E	R	U	P	C	O	L	D	R	Y	W	F	E	P	E	-
M	O	I	I	T	E	R	A	T	O	R	G	I	N	A	L	L	Y	&	S	T	U	Q
M	R	A	M	R	O	L	O	H	C	S	U	A	N	A	M	-	V	-	C	E	S	-
O	H	E	L	P	M	E	L	E	T	B	E	F	U	L	L	E	R	R	O	R	I	S
C	H	A	R	E	T	E	M	E	D	O	H	T	E	M	_	Y	R	O	T	C	A	F

Try to find all 23 GoF Patterns, all 8 POSA Patterns (Patterns-Oriented Software Architecture) as well as several other patterns – over 44 in all. Note Pipes & Filters are adjacent rather than inline, and 4 principles are listed by acronyms with dashes, such as O-C-P for Open-Closed Principle.

Which pattern is included twice to emphasize it is used for both classes and objects?

* WORD SEARCH *

Finding Object Design Pattern Words

T		E	L	B	A	T	U	M	M	I		N	O	T	E	L	G	N	I	S	
E	D	A	C	A	F		E	C	A	F	R	E	T	N	I	_	R	E	K	R	A
M					L								N	S	T	R	A	T	E	G	Y
P	L	A	Y	E	R	S	Y		D			H	A		P		N			T	R
L	-	S	-	P			W	E		E	E			O			I		L		O
A			V				C	E		R	P			I				T		T	
T			I			O			I	Y			T		I		B	N	C		
E	T	A	T	S		R			T	T	G		A		B			L	U	A	
_			I	A			A	O		H	G			I				A	L	F	
M			T			N	T			E		T	I	S	O	P	M	O	C	L	_
E	R	O	O		C	O			L				N					K	_	T	
T	E	R		R	E	R			E		R	O	T	A	I	D	E	M	B	O	C
H	F	E	L		P	E		D			P							O	B	A	
O	L	T	E		V				M	S		P	-	C	-	O			A	J	R
D	E	E	N	D	B	R	I	D	G	E	R	E	K	O	R	B		S		R	E
C	R	R	-		E			R	M				J		R			D	C	S	
T	P	E	I		S		_	E				E		P	E			T	B		
I	R	K	-		B	F			N			C		R	E	T	P	A	D	A	
D	O	E	O	P		O			T		T		E	O	L	I	D				
N	N	T	R	-					O	_			D	X	I	P	A	C			
A	N	C	N						P			L		Y		F	E	P		-	
M	I	I	T	E	R	A	T	O	R		I				&	S	T		Q		
M	A	M					O			U			M	-	V	-	C	E		-	
O	H				L			B									R		S		
C		R	E	T	E	M	E	D	O	H	T	E	M	_	Y	R	O	T	C	A	F

GoF: Template Method, Command, Singleton, Façade, Strategy, Flyweight, State, Visitor, Interpreter, Decorator, Prototype, Composite, Mediator, Chain of Responsibility, Factory Method, Abstract Factory, Builder, Proxy, Memento, Iterator, Observer, Bridge.

POSA: Layers, Pipes & Filters, Blackboard, Broker, M-V-C, P-A-C, Reflection, Microkernel.

Other patterns/principles: Immutable, Marker Interface, Delegation, Inheritance, Object Pool, Null Object, O-C-P, L-S-P, C-Q-S, D-I-P, Demeter.

Which pattern is included twice to emphasize that pattern is used for both classes and objects?
(Adapter)



7245 South Havana Street, Suite 100
Englewood, Colorado 80112-1562
ph: +1-303-302-5280 fx: +1-303-302-5281
www.itcourseware.com

