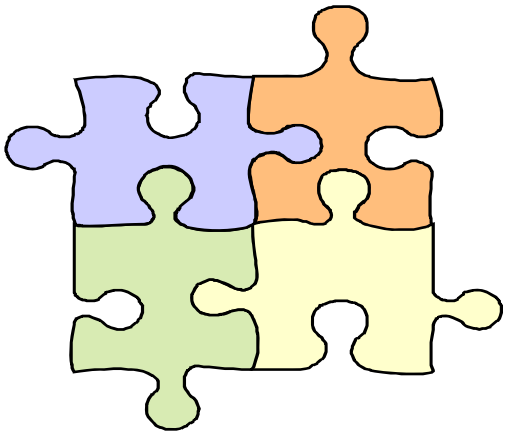
Design Patterns

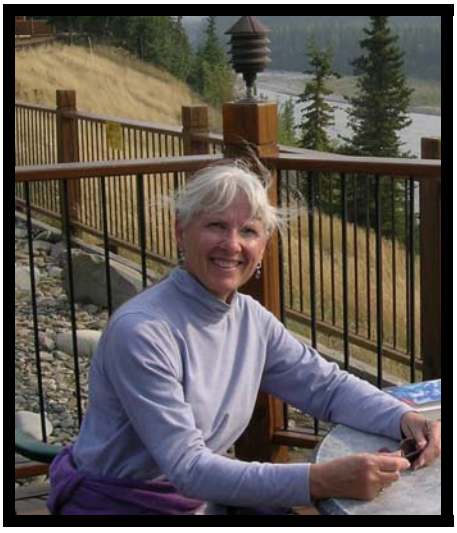


Student workbook

Design Patterns

by Marilyn McCord

(DBA ***Associated Consultants)***



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*, 2003, highlighting her father's WWII memories. During the decade of the 1980s she worked for Texas Instruments in Dallas.

Email Contact: <mccord@gobrainstorm.net>

Contents

Chapter 1 – Course Introduction

Notes

Course Objectives

● Improve **Software Architecture**

● Build **Design Pattern Vocabulary**

- ☛ Learn basic underlying object design principles (ch 3, 5)
- ☛ Learn names and intent of *all* 23 GoF design patterns (ch 6)
- ☛ Learn basic object-oriented architectural patterns and how patterns systems assist in creating overall architecture (ch 9)

□ *(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 (options for solving common problems)

● 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 relevant **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 concepts and simple familiarity with 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: Best Practices and Design Strategies, 2nd Edition,* Prentice Hall / Sun Microsystems Press, 2003, ISBN: 0-13-1422464.

An excellent presentation of best practices in J2EE; easy to read and follow. Originated via field work done by the authors while working with J2EE customers. Includes a Pet Store sample application. Lots of info – diagrams, sample code, etc. -- is available online. If you are working in J2EE, this book is essential.

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

This is XP, eXtreme Programming. 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, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1996, ISBN: 0-8053-0594-7.

Older now, but still some good points – especially on architecture (see p. 344 of this workbook). Interesting to look at the factors that drive software development process, and see where your process fits it. The most mature driver is getting the architecture right.

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

The “bible” on UML. But check the website for the latest updates.

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

✌ POSA: 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.

Cooper, James W., *The Design Patterns Java Companion,* © 1998 by James W. Cooper, IBM Thomas J. Watson Research Center. Free online book in PDF format.

Pointed out to me by a recent class interested in Java examples, available online.

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.

Freeman and Freeman, *Head First Design Patterns*, O’Reilly Media, 2004, ISBN: 0-596-00712-4.

✌ Combines the principles of our Section 3 with the GoF patterns of our Section 6, using easy-to-understand illustrations, humor, and lots of exercises. The heart monitor example evolves to a web application in fairly easy steps.

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 OptimizeIt. Several inaccuracies introduced in the rush to get this printed, but overall quite useful and there is now an errata list for Volume 1:
(see http://www.mindspring.com/%7Emgrand/volume_1_patterns_errata.html)

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 in Java.

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 out.

Shalloway, Alan, and Trott, James, *Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd Edition,* Addison-Wesley, 2004, ISBN: 0-32-1247140.

✌ Some of my students have really like this book; it has some good stuff in it. It gives details on good OO design in general and the author has an easy style which is much easier to read than GoF.

Stelting, Stephen, and Maassen, Olav, *Applied Java Patterns,* Sun Microsystems Press, Prentice Hall, 2002, ISBN: 0-13-093538-7.

✌ For the Java crowd: First three chapters parallel GoF categories of patterns, chapter 4 has System Patterns, then chapters with Java language patterns. Appendix A lists over 200 pages of full Java code for all patterns, and the code is also downloadable.

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 –

especially note Observer in Ch. 4. Puts patterns into the broader context of basic object-oriented design principles. Note: Vlissides is one of the GoF authors.

Wake, William C., *Extreme Programming Explored,* Addison-Wesley, 2002, ISBN: 9-780201-733976 (paper)

Small, easy to read, lots of code examples. Will give you a feel for the XP style and approach.

WEB SITES: (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 lea'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

http://www.patterndepot.com/put/8/JavaPatterns.htm
Location for James W. Cooper's online book with Java code for patterns

http://www.phptr.com/appliedjavapatterns

http://ww.clipcode.nte/components/snippets
Downloadable C# code snippets, included here as Appendix B.

http://www.javaworld.com
Lots of free articles, etc.

http://ambysoft.com
Scott Ambler's site

http://www.c2.com/cgi/wiki?WelcomeVisitors
The Wiki Wiki Web entry point – with a sandbox to play in

http://java.sun.com/blueprints/corej2eepatterns

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/esp.asp
.Net patterns

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

Chapter 2 – Design Pattern Overview

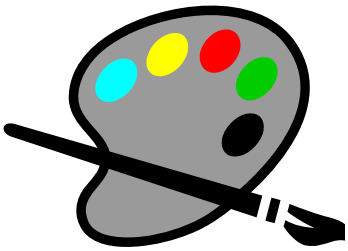
Notes

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 (rules?):

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; take small steps
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
is **S-I-M-P-L-E.**

So, why don't we achieve this?
How hard is it to write simple code?

What makes some software designs **rot**?

What can we do to help prevent software rot?
(Compare to sterile procedures, which makes surgery possible…)

What is **refactoring**?
(Compare to cleaning up the kitchen after dinner…)

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.

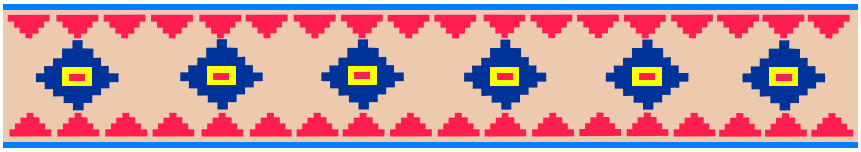
  4.

  5.

  6.

Overview of Patterns

… think at a design level rather than a code level – 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? (e.g., define a "for" loop)

- Template Brainstorm:
  (A canonical form for elements you want to see in a Pattern…)

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
  (For example, MVC is both architecture and a mechanism)

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*

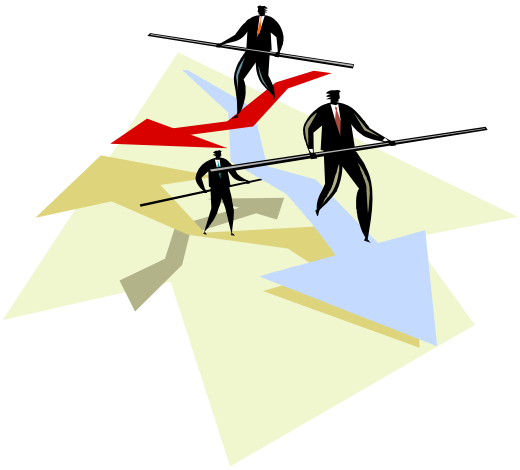
Patterns realize an Equilibrium

- balance among pattern forces and constraints
- invariants typify principles/philosophy for domain;
  minimize conflict within solution space
- rationale for each step/rule in 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**
  (What is encapsulated? What is abstracted?)
- **Patterns Exhibit Openness and Variability**
  (Many implementations; extendibility)
- **Patterns Have Generativity and Composability**
  (Provide context for next patterns; infinite variety)
- **Patterns realize an Equilibrium**
- **Patterns imply broader architectural issues –**
  More people-oriented things like *maintenance, reuse, encapsulation, variation,* etc.

►*What do these bullets really mean?*
(What about an automatic "pattern generator" tool?)

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.

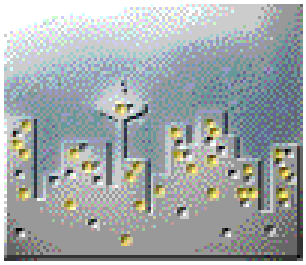
"It would not be reasonable to design a house without knowing the lay of the land, or a skyscraper without knowing the materials that could be used. The idea that we can treat concepts such as threading and distribution as mere coding details is a sure fire way to waste a lot of energy (and time, money, 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:

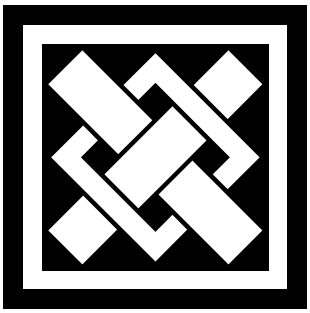
(1) highest level: architectural patterns define overall shape

(2) specific architecture related to purpose of application

(3) architecture of modules and their interconnections
(domain of design patterns)

(4) **lowest level: idioms** – patterns in language syntax (e.g., i++)

(5) 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 Riel[4] (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 stated in a sentence or two;
Patterns take more space.

Heuristics can provide the **glue** to know**:**
when to select a particular pattern, or
how to combine them during 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:
  1) Am I a special type of the thing from which I'm inheriting?
  2) Is the thing from which I'm inheriting part of me?

Multiple Inheritance

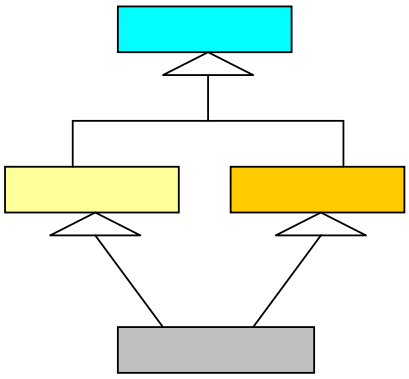
- Assume you have made a mistake and prove otherwise. (What's Java's solution to this?)

The Association Relationship

- Prefer containment to simple association if appropriate (Important in non-garbage-collecting environments)

Class-Specific Data and Behavior

- Use of class variables and methods



The "Deadly Diamond"

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 7 – Other Micro-Architecture and System Patterns

Notes

"Current architectural methods result in products that fail to meet the real demands and requirements of its users, society and its individuals, and are unsuccessful in fulfilling the quintessential purpose of all design and engineering endeavors: to improve the human condition." Christopher Alexander

Coplien talks about making a dress by specifying the route of scissors through the cloth in terms of angles and lengths of cut – vs. using a pattern. **The pattern foreshadows the product.**

Chapter Objectives

● Learn **Other Micro-Architecture and System Patterns**

1. Object Pool
2. Worker Thread
3. Dynamic Linkage
4. Cache Management
5. Type Object
6. Extension Object
7. Smart Pointers (C++)
8. Session
9. Transaction

and there are many more…

Want to be a design patterns star? User vs. Writer? Either way, study existing patterns.

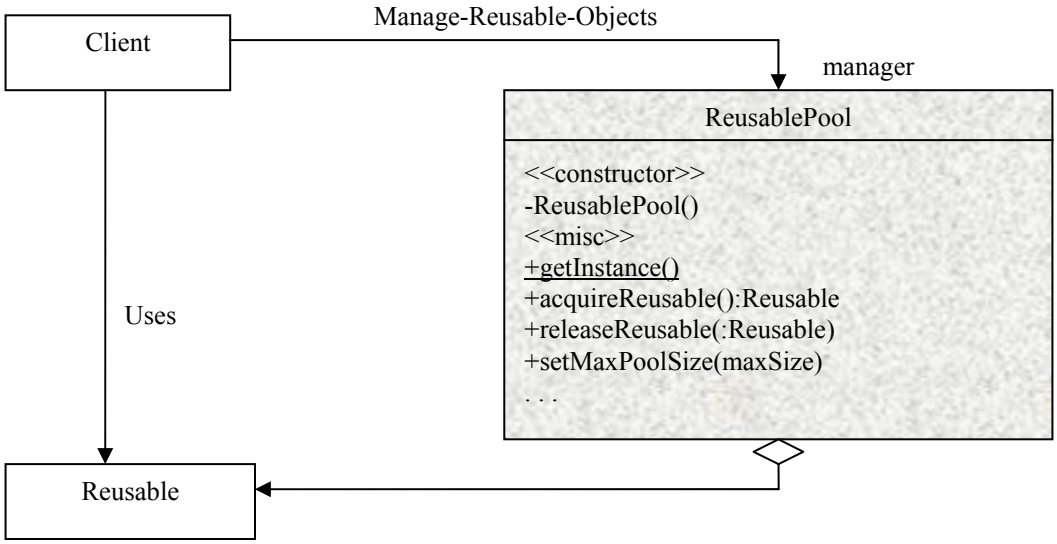
● Develop **incentive** to look for additional patterns (e.g., Successive Update, Router, Callback… all included in Sun's book, *Applied Java Patterns* – see bibliography). See also Buschmann, et al, *Pattern-Oriented Software Architecture.*

► Patterns can channel creativity, not replace it or constrain it.

Notes

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

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) – A creational pattern

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

Example: Managing connections to a database. Clients use connections to send queries and retrieve results. Avoid having each program create its own connection; each creation takes several seconds and the more connections there are to a database the longer it takes to create new connections. Each database connection needs a network connection and some platforms limit the number of network connections allowed. Solution: have a library manage database connections on behalf of 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, creating new instances for that class should be avoided.

□ Avoid creating objects by reusing objects when 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 costs.

□ Works best when demand for objects is fairly stable.

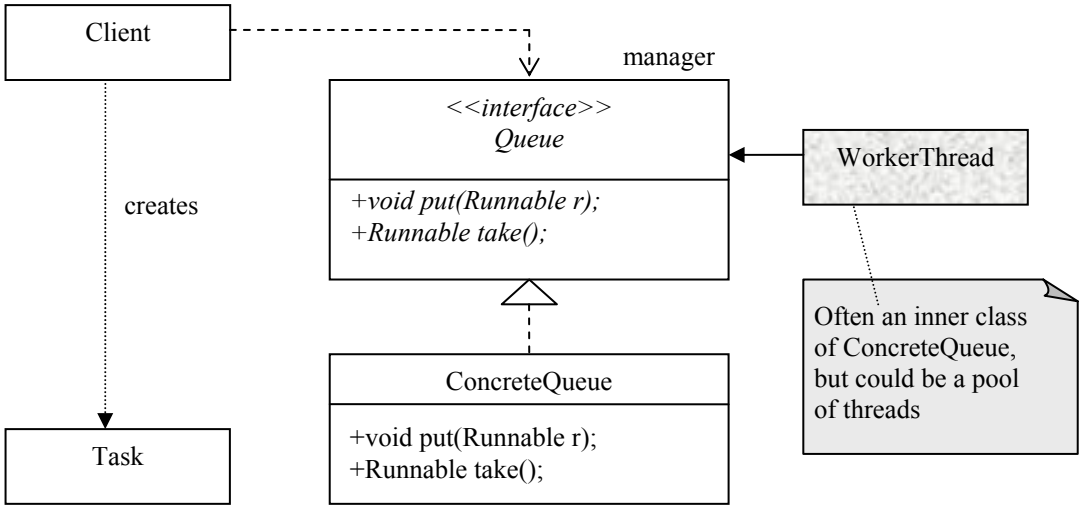
□ Keeps creation/reuse logic in separate class from class instances being managed.

□ Eliminating interactions between implementation of creation/reuse policy and implementation of managed class's functionality improves cohesion.

Notes

Comments: Creating thread instances is expensive in terms of performance. Instead create a single WorkerThread (or a pool of them) and simply give new tasks to the WorkerThread. A queue manages the wait list of tasks.

Diagram:



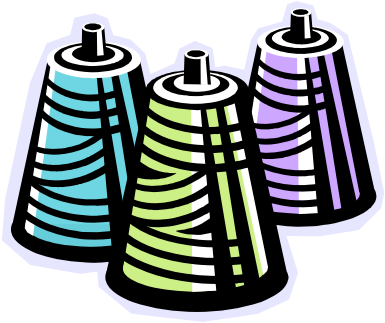
Participants: Client, Task (implements Runnable interface), Queue, ConcreteQueue, WorkerThread.

Solution: If one or more instances of a Thread are reused, you avoid creating new Thread instances for a one-shot task.

2. WorkerThread

Pattern Thumbnail: A way to improve application throughput and minimize average latency by reusing a single helper thread (or pool of threads).

Example: It requires skill to implement multithreading correctly.



Separate non-crucial tasks from the rest of your application and use Worker Thread pattern. The worker thread picks up task from queue and executes it; when that's done, it picks up next task from queue. (Compare Command)

Forces:

□ Timing of execution isn't critical (lower priority tasks),
□ Creating new thread for each task isn't efficient,
□ Want to improve throughput, introduce efficiency.

Consequences:

+ Client no longer responsible for creating threads – only for putting tasks on queue (less expensive operation), and therefore doesn't have to wait to hand-off the task.

□ Works best when number of Worker Threads matches number of possible concurrent tasks – if application has to wait for Worker Thread to become available, lose multithreading benefit.

+ Keeps creation/reuse logic in separate classes from tasks being managed.

+ Task sitting in queue and not running takes no scheduling resources; more threads mean more scheduling required and Worker Thread may simplify scheduling.

- Dependent tasks with sequential queue can produce deadlock.

- Too many tasks for number of available threads = clogged system.

Variations:

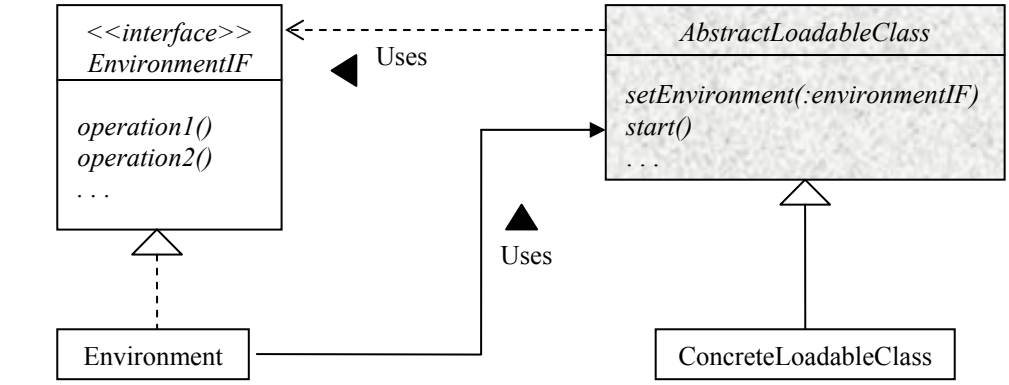
□ ThreadPool to manage worker thread class instances. Could implement a timing strategy with queue requests dropped after certain time period, and client needs to retry.

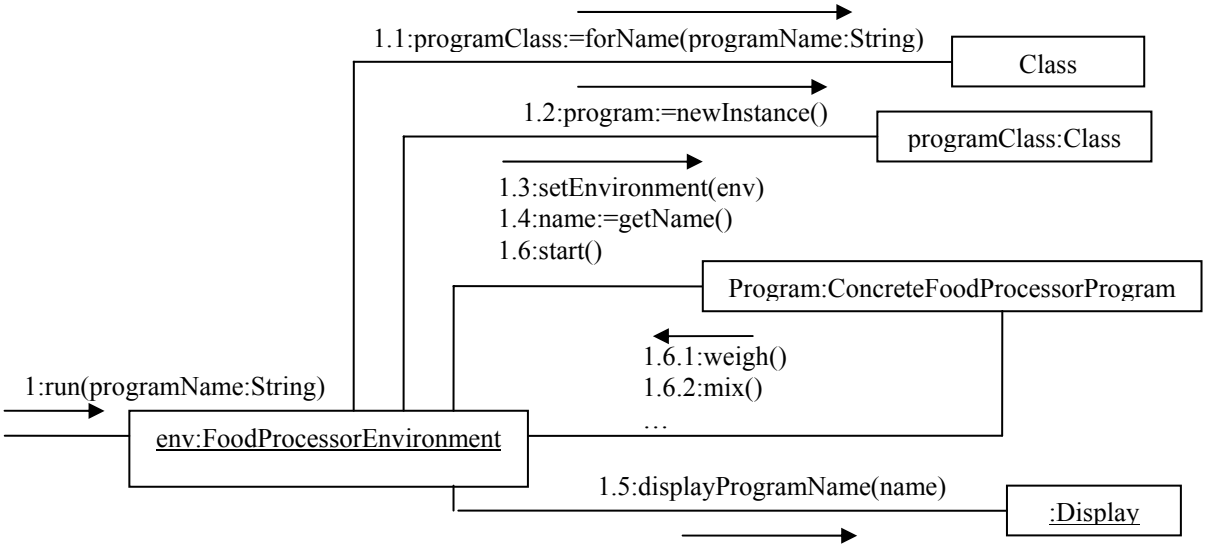
□ Create a SmartQueue that understands how tasks work together and implements varying strategies – but could become maintenance nightmare.

Notes

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

Diagram:





Participants: EnvironmentIF, Environment, AbstractLoadableClass, ConcreteLoadableClass

Usage: Web browsers use 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.

JDBC uses DriverManager to load appropriate database Driver.

3. Dynamic Linkage

Pattern Thumbnail: Allow program, upon request, to load and use arbitrary classes implementing 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:

□ Program must be able to load/use arbitrary classes it has no prior knowledge of.

□ Instance of loaded class must be able to call back to program that loaded it.

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

Implementation: Pattern requires that environment knows about *AbstractLoadableClass* class and that loaded class knows about *EnvironmentIF* interface.

When less structure is needed other mechanisms for interoperation are possible – JavaBeans uses combination of reflection classes and naming conventions to allow other classes to infer how to interact with a bean.

Also, Environment class must know name of class it wants to load. In the example above, the CD-ROM could contain directory of programs displayed as a menu, allowing user selection. Other applications might hardwire names.

To avoid incompatible versions of supporting classes, ensure all supporting classes loaded **im**plicitly as well as any **ex**plicitly dynamically loaded class are not used by any other explicitly dynamically loaded class. Implement this by using different ClassLoader for each dynamically loaded class.

Consequences:

□ Subclasses of *AbstractLoadableClass* can be dynamically loaded.

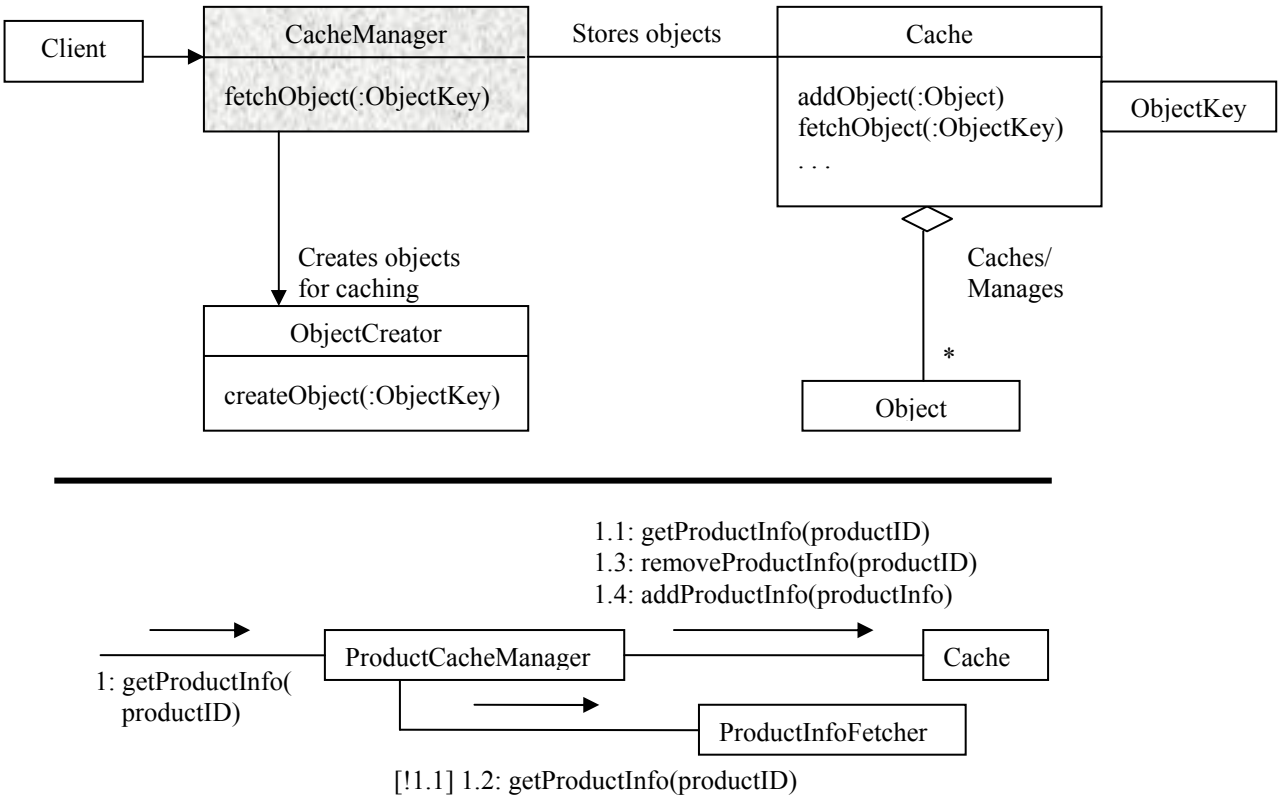
□ Operating environment and loaded classes do not need specific foreknowledge of each other.

□ Dynamic linkage increases total amount of loading time. However, it has effect of spreading out, over time, loading overhead and can make an interactive program *seem* more responsive.

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:



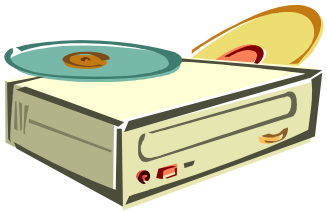
Participants: ObjectKey, CacheManager, ObjectCreator, Cache

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.

4. Cache Management

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

Example: Writing a program for fetching formation about products in a catalog; fetching all information for a product can take several seconds as it is gathered from multiple sources. Once info for product is secured, want it available for quick access if needed again soon.



Forces:

□ Need to access an object that takes long time to construct.

□ When number of such objects is small enough to fit in local memory, keeping them there provides best results.

□ May be necessary to set upper bound on number of cached objects with 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:

□ Impact of Cache Management on rest of program is minimal.

□ Positive: Program spends less time creating expensive objects.

□ Negative: Cache may become inconsistent with original data source.

read consistency means objects fetched from cache always reflect updates to information in original;

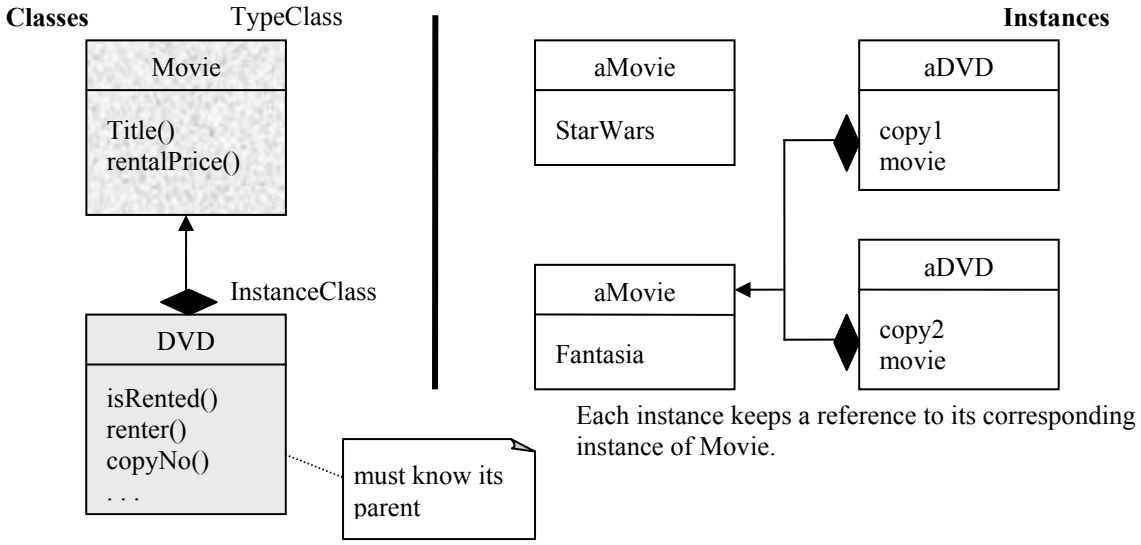
write consistency means original object source always reflects updates to cache.

□ Some synchronization mechanism 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:



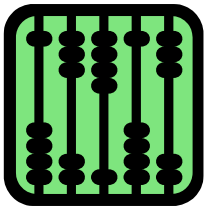
Participants: TypeClass, TypeObject, InstanceClass, InstanceObject

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 created.
2. Object's behavior can be implemented in its class or delegated to its TypeObject.
3. Messages an object understands are defined by its class, not its TypeObject.
4. New InstanceObject created by sending request to appropriate TypeObject instance (different).
5. Object could have multiple TypeObjects, but this is not common.
6. Object could dynamically change class (easier to change the reference than to mutate to new class).
7. Possible to subclass either InstanceClass or TypeClass – e.g., Videodisk instance could reference same movie as Videodisk instance;
   3 videotapes and 2 videodisks could all share same movie.

5. **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 DVDs. You don't want to store redundant information for each copy (instance). The total number of DVDs is not known ahead and changes often anyway.

Forces:

□ Instances of class need to be grouped together by common attributes and/or behavior.

□ Class needs subclass for each group in order to implement that group's common attributes/behavior.

□ Class requires large number of subclasses and/or total variety of subclasses required is unknown.

□ Want to create new groupings at runtime that were not predicted during design.

□ Want ability to change object's subclass after it has been instantiated without having to mutate it to new class.

□ Want 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:

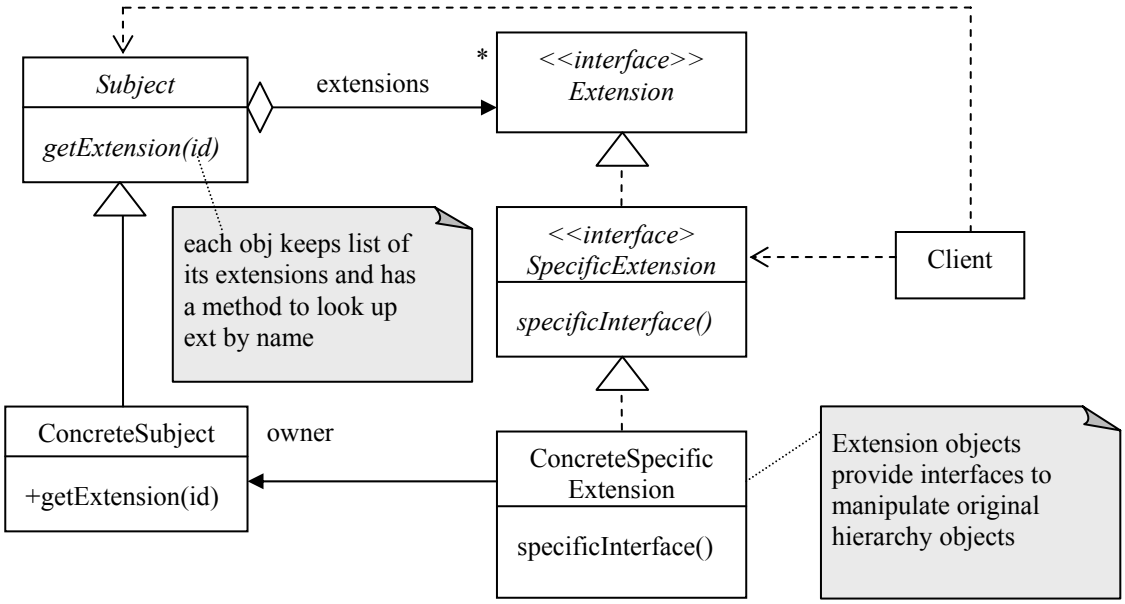
Positive: Runtime class creation, avoiding subclass explosion, hiding separation of instance and type, dynamic type changes (NewRelease can change to GeneralRelease), independent subclassing, multiple type objects.

Negative: Design complexity, implementation complexity, reference management required by 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.

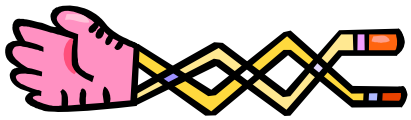
Usage: The QueryInterface in COM (Component Object Model) interfaces behave as extensions.

Software Communications Architecture (SCA) is a component-based object model used for architecting software-defined radios. At the heart of the design of this architecture is a need to add or extend existing hierarchies without modifying the hierarchies – a real-world application of the Extension Object/Interface Pattern. "Using the Extension Object Pattern, these radio components truly move into the realm of being software ICs just like their hardware counterparts. Their 'ports' correspond to the pins of an IC and the guts of the Resource correspond to the business logic of the integrated circuit." Instead of `getExtension(id)` they use `getPort(id).` See Linda Rising's discussion at <http://www.smallmemory.com/almanac/>

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.

6. **Extension Object** (aka: Facet)



Pattern Thumbnail: Add interfaces to a class and let clients choose and access interfaces they need (even dynamically); sort of like Strategy. (Also relate to Java's Reflection pkg.)

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. Now 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 separate abstract class.

Example 2: A bill of materials system (BOM) wants each hierarchy object to create XML representation of self (or CSV—comma separated values). Adding toXML() violates SRP and would be a real pain.

Forces:

□ Want to add new/unforeseen interfaces to existing classes and don't want to impact clients that don't need this new interface.

□ Clients perceive different roles for same abstraction and number of such roles is open-ended.

□ 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?

How do ISP, SRP, and OCP apply?

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.

Related Patterns: Visitor, Decorator, Adapter.

► Complex, but powerful pattern from Erich Gamma. See PLoPD3, pp 79-88.

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;}
    // . . .    etc.
};
```

```
// example of problem
    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;
        };
```

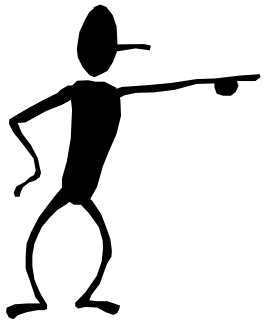
You can write:

```
    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) { // if not assigning to self
            delete ptr;
            ptr = rhs.ptr;
            rhs.ptr = NULL;
                // set pointer to NULL on copy
    }
    return *this;
}; // operator= definition
```

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



7. Smart Pointer (C++)

Pattern Thumbnail: Using **Proxy**, smart pointers implement same interface as regular pointers. Key is overloading of dereferencing operator. By wrapping an instance of a class in a template class, and returning the address of the wrapped instance from 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:

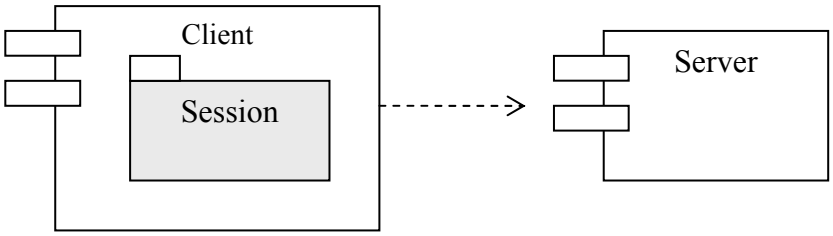
Positive: Pointers which handle common C++ bugs can save in aspirin. Smart pointers can mean fewer bugs and can do such tasks as automatic initialization, removal of dangling pointers (pointers to objects already deleted), creating new copies, transfer of ownership, reference counting, reference linking,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

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

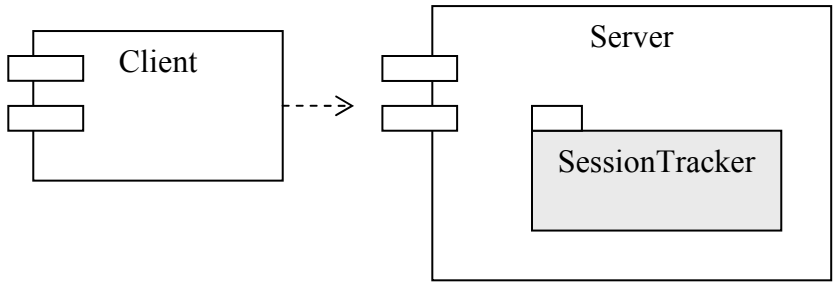
Notes

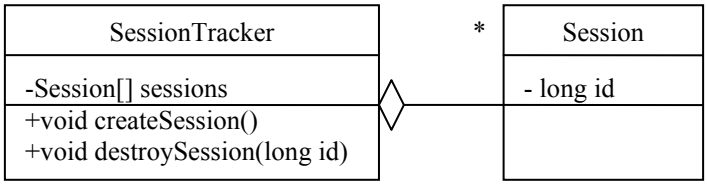
Diagram:

Client-matching session



Server-maintained session





8. Session

Pattern Thumbnail: Provide a way for servers in distributed systems to distinguish among clients and allow applications to associate state with the communication.

Example: A ChatServer application with multiple clients – using sockets – requests from clients to server can be made sequential and the server is aware of previous calls. A shopping cart application.

Question: What is the difference between *stateful* and *stateless* communication in distributed systems? What are some situations in which each approach makes sense.

What about HTTP on the internet – is it stateful or stateless?

How valuable would a stateless e-commerce application be?

Implementation Issues:

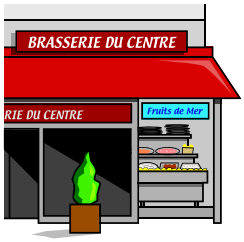
□ Managing session identity.

□ Where to keep information on state – client side (cookies) or server side (HttpSession in Java).

Consequences:

Positive: Maintaining client information during a series of communications, even when changes take place in stages. Can differentiate among clients.

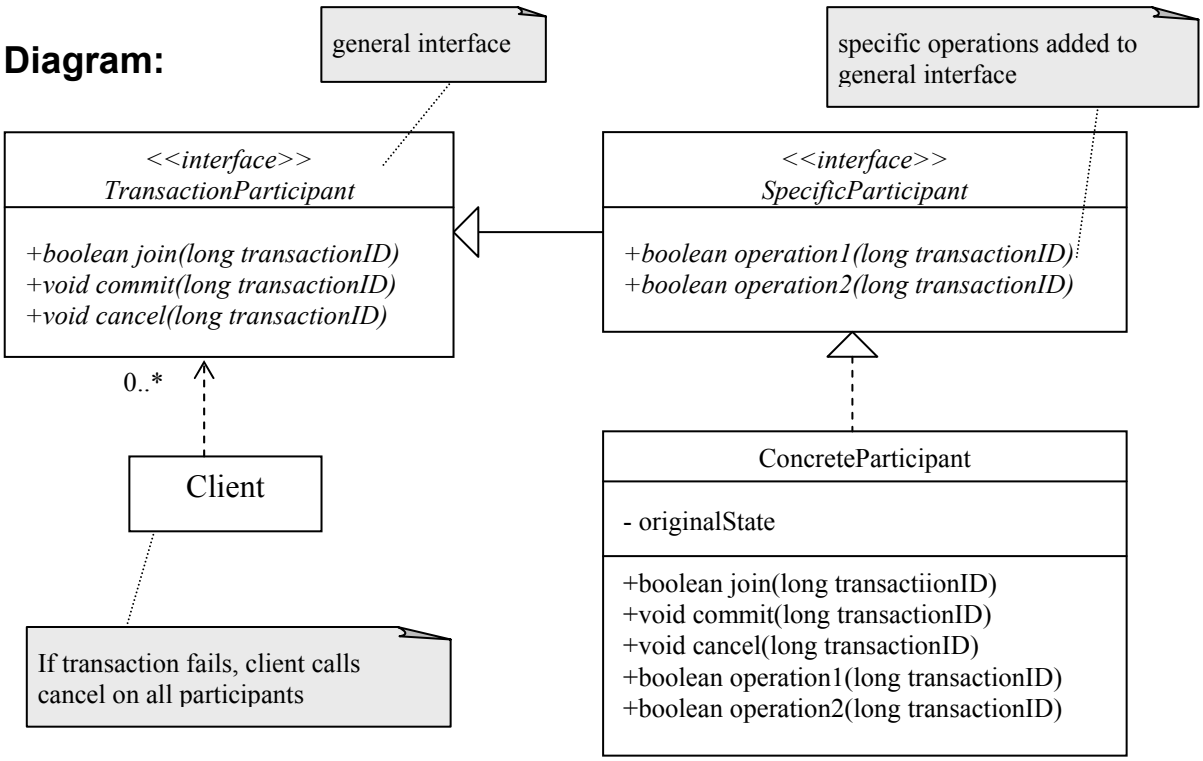
Negative: Increased workload on server, increased complexity required for client identity and storing/retrieving associated information.



► How would customers buying coffee at Starbucks be an example of this pattern? Client-side state scenario: Client specifies what they want. Server-side state scenario: "Oh hello, Ms. D, the usual?" Suppose there were only one server and a very indecisive customer – what would it mean to make this scenario "multithreaded?"

Notes

Diagram:

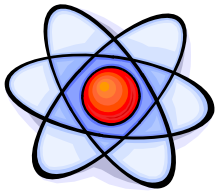


Participants: TransactionParticipant (interface defining methods to control every participant), SpecificParticipant (extension of interface to contain the business methods – which could throw exceptions as a signal of failure), ConcreteParticipant (has to keep reference to original state to be able to restore if cancel invoked), Client (acts as transaction manager; calls join on participants to start transaction and ultimately calls either cancel or commit on participants).

Normal Sequence:

1. Create a transaction ID (either as object or long)
2. Invoke join on all participants, aborting if any join fails.
3. Try the action, invoke necessary business methods, call cancel if any participant fails.
4. When action completed, call commit on all participants.

9. Transaction



Pattern Thumbnail: Group a collection of methods so either all succeed or all fail (atomic).

Forces:
□ Several methods need to be fully synchronized
□ Recovery options should be available.

Example: Transferring funds from one account to another. If transfer fails, funds magically disappear from one of your accounts – not good. And banks frown on funds magically appearing in one of your accounts…

Implementation Issues:
□ If Participant already involved in transaction and join() called again, Participant object must decide what to do. Could throw exception to caller of join and Transaction Manager could either roll-back second transaction or wait until participant available.

Consequences:
Positive: Several methods combined to act as atomic operation; application can maintain consistent state as new state is not persisted until all participants succeed.
Negative: Performance decreased.

Variations:
□ Two-phase commit – Transaction Manager assures all participants can commit before it calls commit() on them – a voting round.
□ *Optimistic* (participants can always join but may not be able to commit)
vs.
Conservative (join may fail, but when joined, participant can always commit)

► Sun's *Applied Java Patterns* has the Half-Object Plus Protocol (HOPP – p. 189, ed. 1) which splits an object for executing some methods locally, some methods remotely.

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 DVDs (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).

Notes

Chapter 11 – Selected Anti-Patterns

Notes

The AntiPatterns home page is located at the following website:

www.antipatterns.com.

Why document a bad solution?

- something about it is attractive
- the solution in the long term is bad
- the anti-pattern may suggest other patterns which may provide good solutions
- guidance for others (don't fall into the same trap!)
- you CAN refactor

Edison had how many failures before he developed a practical, incandescent electric light?

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

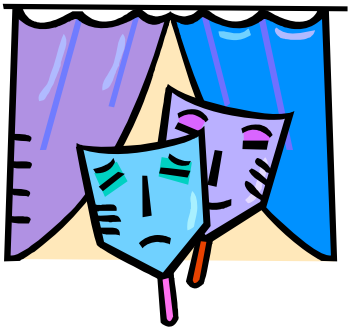
Anti-Patterns

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** = most serious **tragedy**



… the state of software engineering today is mostly a tragedy …

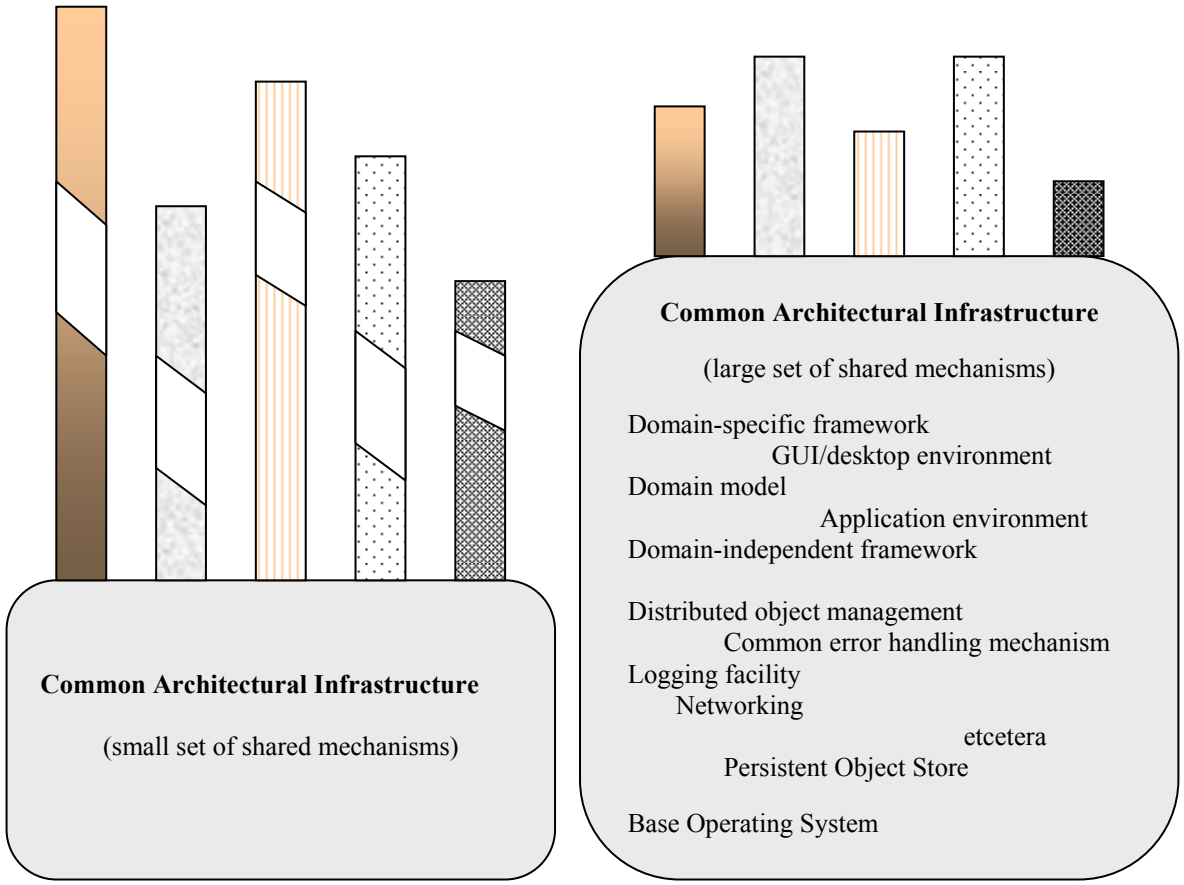
Like a small child throwing a tantrum – maybe it only works 10% of the time, but he doesn't yet have a better model…

With patterns, benefits exceed consequences.

With anti-patterns, consequences exceed benefits.

Notes

►There is a tutorial on anti-patterns on the website <www.antipatterns.com>.

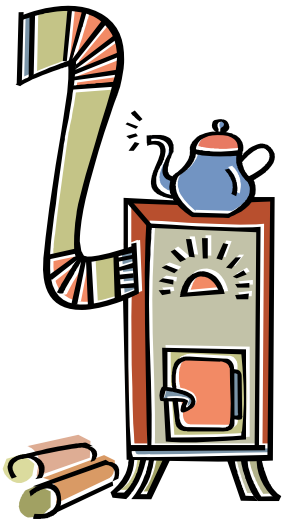


2

Note the stovepipe components on the left above… and the refactored solution on the right which brings common functionality into the architectural infrastructure. This is reuse that really counts, when the infrastructure works for all the typical applications you do…

[2] Explore more on this in Booch's *Object Solutions*, p. 43-54.

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:

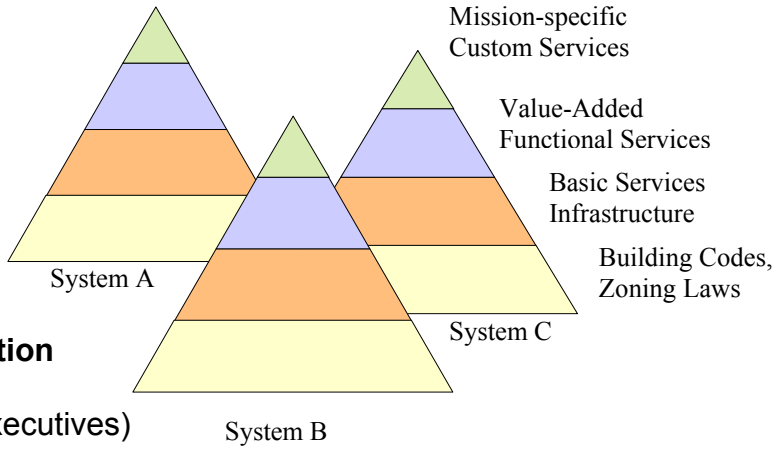
- □ The software project is way over budget;
- □ has slipped its schedule repeatedly;
- □ my users still don't get expected features;
- □ and I can't modify the system.

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:

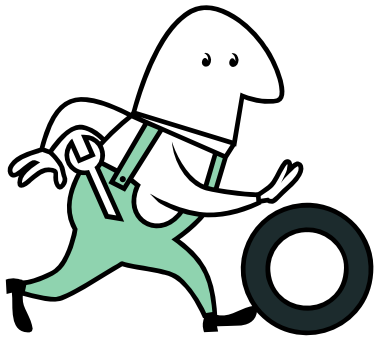
□ Incompatible terminology, approaches, technology;
□ brittle, monolithic system architectures and undocumented architectures;
□ inability to extend systems to support business needs;
□ incorrect use of a technology standard;
□ lack of reuse between systems;
□ lack of interoperability (even with same 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
(Why do we do this?!)

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: Pattern may be suitable for research environment where developers with different skills work at logistically remote sites.

► *What would **typical causes** include?*

► *Describe **symptoms** and **consequences**?*

► *What would a **refactored solution** look like?*

Notes

Golden Hammer Forces:

- **Development team committed to technologies they know**
- **Development team NOT familiar with other technologies**
- **Unfamiliar technologies are seen as risky**
- **It's easy to plan and estimate for development using the familiar technology**

Examples: Web companies keep using and maintaining their internal homegrown caching systems when open source alternatives are in use.

For more examples, see the Portland Pattern Repository's WIKI at http://c2.com/.

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: Software development team has gained competence in particular solution or vendor product (the Golden Hammer). It may be a mismatch for problem, but minimal effort devoted to exploring alternative solutions.

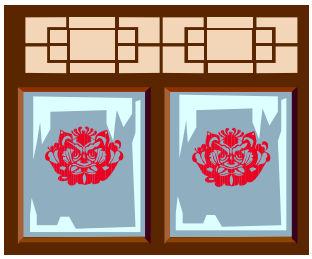
Typical Causes:

- □ Several successes have used this particular approach
- □ Large investment in training with product or technology
- □ Group isolated from industry, other companies
- □ Reliance on proprietary product features not available in other industry products
- □ "Corncob" (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." (Compare with XP)
- □ "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 pragmatic**, common-sense approach** to planning, 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** showing software component deliverables and their dates

► *Describe a **refactored solution:***

► *How is the **analysis paralysis** anti-pattern related?*

► *Compare with the approach of Extreme Programming, **XP.***

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:***

► *How might an **iterative process** help?*

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.

Business Process Management With a Business Rules Approach: Implementing the Service-Oriented Architecture, by Tom Debevoise, Oct. 2005 – which was initially billed as *Refactoring IT Business Failure Before the Money is All Gone*.

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

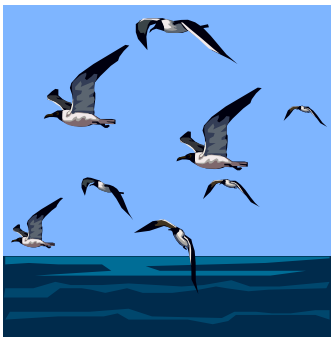
● **Throw It Over the Wall:** code is finished! (no testing, no documentation)

● **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



● **Seagull Manager:** Flies in, makes a lot of noise, craps on everything, and leaves.

● **Project Mismanagement:** "All you need in this life is ignorance and confidence; then success is sure." - Mark Twain

Notes