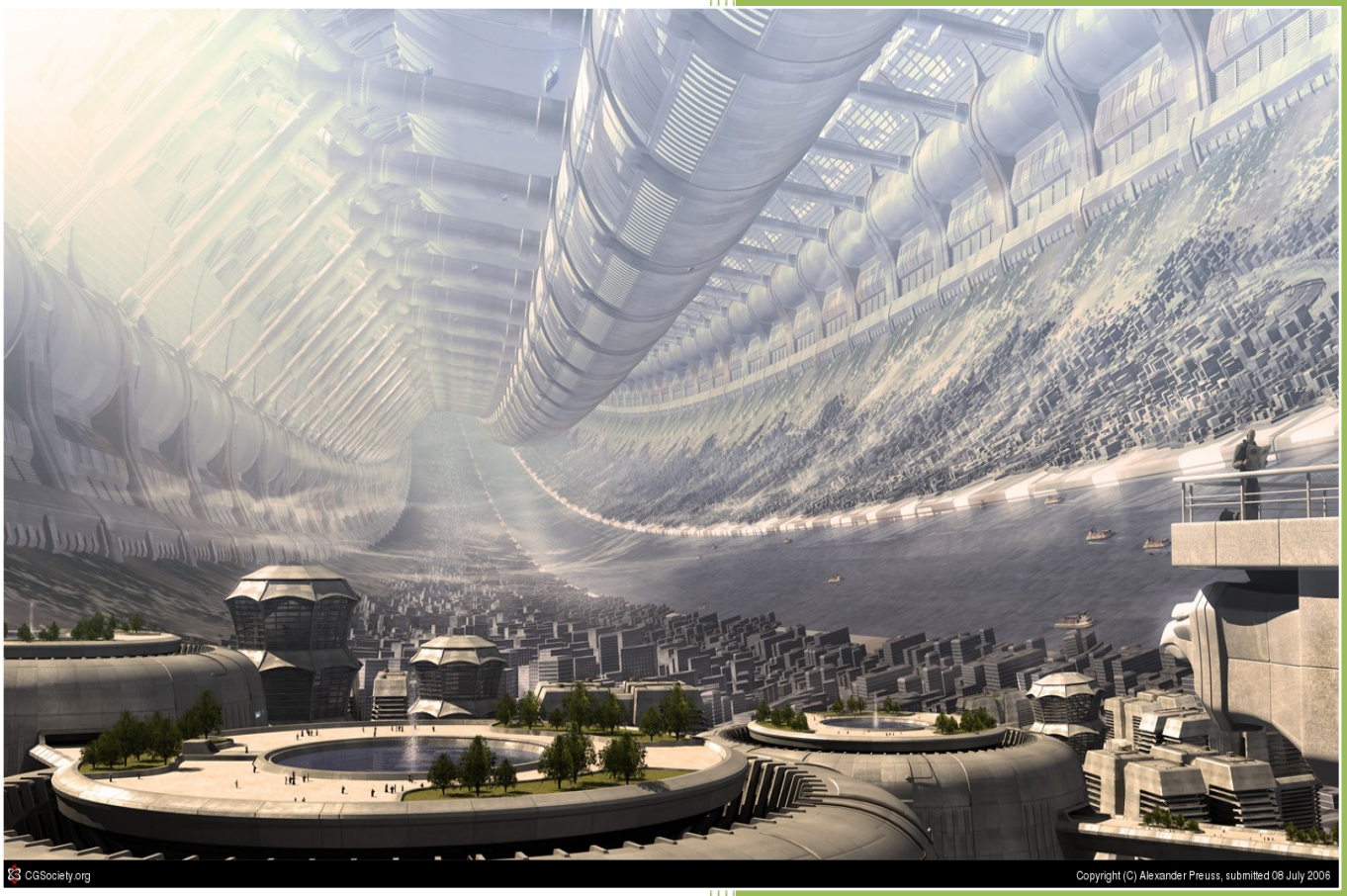
2008

Design Patterns Course



For: Varian, Inc.

Presenter: Andy Bulka

April, 2008
Version 4.2

Acknowledgements

http://www.atug.com/andypatterns

abulka@netspace.net.au

Some materials in this handout have been taken from the following sources:

Huston Design Patterns

http://home.earthlink.net/~huston2/dp/patterns.html

Data & object factory.

http://www.dofactory.com/Patterns/Patterns.aspx#list

Some diagrams from

www.javacoder.net

Table of Contents

Contents

Course Outline

Code: IM500-020

Overview

This course provides students with the necessary knowledge and skills required to understand and use the fundamental 23 design patterns as outlined in the classic book “Design Patterns”. Participants will be able to use the vocabulary of patterns to communicate to software engineers in order to design superior, maintainable, cutting edge software.

Course Outline

A brief history of design patterns - the patterns movement, how the GOF book was created, why a knowledge of design patterns is crucial to today’s software developers. Where did patterns come from, what is the role of Christopher Alexander. Who are the key patterns people around the world. How has Australia been involved in the world patterns conference scene.

Detailed presentations of 20 design patterns: including: state, strategy, adapter, mediator, template method, bridge, factory + more. “State Pattern” can get rid of lots of if/else spaghetti code. “Strategy Pattern” can make your application more pluggable. “Adapter” is one of the most important fundamental patterns. “Mediator Pattern” can reduce coupling between classes and reduce complexity. Understanding the “Template Method Pattern” requires that you know your object oriented principles and can make your code base more organized. “Bridge Pattern” is a little complex but once you have invested in it, allows you to switch implementations with a single switch. The factory patterns (“Factory Method” and “Abstract Factory”) are core patterns. Many more patterns like this are presented during this course.

How patterns are used together – most people are surprised when shown how patterns can be combined. We will look at the notion of “roles” in classes and look at diagramming techniques for showing overlapping patterns in UML diagrams.

Design Pattern Software - A presentation of a couple of UML design patterns tools for windows including Sparx Enterprise Architect and IBM Rationale Rose.

Discussion - Current research directions and the future of patterns (both theoretical and tools). Classic and cutting edge books and ideas are discussed (time permitting).

Outcome and Benefits

After completing this course, students will know how to:

- Describe the history of patterns development.
- Appreciate the benefits of a patterns approach to programming design.
- Describe the detailed design, purpose and behaviour of 20 of the original GOF design patterns.
- Communicate with other developers using the language of patterns.
- Use design patterns to improve the design of new and existing software.
- Implement various patterns in various programming languages.
- Combine different patterns so that they work together in a software design
- Understand some of the design patterns contained in such frameworks as.NET and Java
- Choose from a variety of UML tools which automate the application of patterns

Prerequisites:

Attendees should have:

- An Understanding of object oriented programming techniques

- Experience with an object-oriented programming language such as Java or C#
- Some exposure to UML diagramming models.

Course Detail

Introduction to Patterns What are Patterns? Pattern Purposes The GOF patterns Pattern classifications The Use of UML Relationships between patterns When to use patterns & when not to	**Patterns: General Issues** Reading a pattern Intent, applicability, forces of patterns Solutions & Consequences Implementation issues - language specific Anti-patterns Re-factoring of patterns	
Creational Patterns Abstract Factory Builder Factory Method Prototype Singleton	**Structural Patterns** Adapter Bridge Composite Decorator Facade Flyweight Proxy Null Object	**Behavioral Patterns** Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor
Architectural Patterns Layers Blackboard Reactor Model-View-Controller	**Patterns in Frameworks** Java .NET	**Advanced** Blending Patterns Automating patterns - tools The future of patterns

About the Trainer:

Andy Bulka is Technical Director and Chief Software Architect at Austhink Software www.austhink.com. He has been programming for over 25 years, and has applied design patterns to numerous real world applications including windows desktop applications, web sites and even a commercial computer game.

Andy has been teaching design patterns for four years, running three day workshops covering all the GOF patterns. He is an active member of Melbourne Patterns User Group giving regular presentations and talks. Andy was Local Conference Chair at the Australian Koala Plop Patterns Conference 2002, and is author of several papers on Design Patterns and Software development – see www.atug.com/andypatterns.

Design Patterns – Learning Order

Taken from A Learning Guide To Design Patterns
by Joshua Kerievsky,
http://www.industriallogic.com/papers/learning.html

● Creational
■ Structural
◆ Behavioral
Key

● Factory Method **Session 1**
Begin with *Factory Method*. This pattern is used by a number of patterns in the book and throughout the patterns literature.

◆ Strategy **Session 2**
Strategy is used frequently throughout the book, and an early knowledge of it helps in understanding other patterns.

■ Decorator **Session 3**
For an early dose of elegance, nothing is better than the *Decorator*. The discussion of "skin" vs. "guts" is a great way to differentiate Decorator from the previous pattern, Strategy.

■ Composite **Session 4**
The *Composite* pattern appears everywhere and is often used with Iterator, Chain of Responsibility, Interpreter, and Visitor patterns.

◆ Iterator **Session 5**
Reenforce the reader's understanding of Composite by studying *Iterator*.

◆ Template Method **Session 6**
The author's footnote to Iterator explains that a method called "Traverse" in the Iterator example code is an example of a *Template Method*. This pattern also reenforces Strategy and Factory Method.

● Abstract Factory **Session 7**
The reader now returns to the second-easiest creational pattern, the *Abstract Factory*. This pattern also helps reenforce Factory Method.

● Builder **Session 8**
The reader now may compare another creational pattern, the *Builder*, with the Abstract Factory.

● Singleton **Session 9**
Singleton is often used to model Abstract Factories, as the "Related Patterns" section of Singleton describes.

■ Proxy **Session 10**
The reader now has a chance to learn how *Proxy* is used to control access to an object. This pattern leads directly into the next pattern, Adapter.

■ **Adapter** **Session 11**
The *Adapter* pattern may be compared with what the reader understands about Decorator, Proxy, and later, Bridge.

■ **Bridge** **Session 12**
Finally, the reader learns how the *Bridge* pattern differs from both the Adapter and Proxy patterns.

◆ **Mediator** **Session 13**
Now the reader learns the *Mediator* pattern, in preparation for understanding Observer and the Model-View-Controller design.

◆ **Observer** **Session 14**
Discover how the Mediator is used by the *Observer* to implement the classic Model-View-Controller design.

◆ **Chain of Responsibility** **Session 15**
After exploring how messages are passed using the Observer and Mediator patterns, the reader now may contrast how messages are handled by the *Chain of Responsibility* pattern.

◆ **Memento** **Session 16**
The reader now moves on to *Memento*. This pattern leads directly into a discussion of undo and redo, which is related to the next pattern, Command.

◆ **Command** **Session 17**
The *Command* pattern is used in a number of ways, one of which relates to the previous pattern, Mediator.

● **Prototype** **Session 18**
Perhaps the most complex creational pattern, *Prototype* is often used with the Command pattern.

◆ **State** **Session 19**
The reader may now study *State* to understand another way an object's behavior changes.

◆ **Visitor** **Session 20**
Visitor is often combined with the Composite and/or Iterator patterns.

■ **Flyweight** **Session 21**
The *Flyweight* pattern is one of the more complex patterns. An examples use of this pattern is described in the next pattern, Interpreter.

◆ **Interpreter** **Session 22**
The *Interpreter* pattern is complex. It makes reference to and helps reenforce one's understanding of Flyweight and Visitor.

■ **Facade** **Session 23**

The final pattern to read is *Facade*. Facade is relatively straightforward and follows nicely after Interpreter since the example code is similar in theme to example code in the Interpreter.

Opening Questions For A Study Group

● Factory Method **Session 1**

How does *Factory Method* promote loosely coupled code?

◆ Strategy **Session 2**

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?

■ Decorator **Session 3**

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?

■ Composite **Session 4**

Part 1: How does the Composite pattern 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?

◆ Iterator **Session 5**

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?

◆ Template Method **Session 6**

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?

● Abstract Factory **Session 7**

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?

Consider the MazeFactory example. 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 & 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?

Ofcourse, nothing would prevent you from setting the color and size of the Room object *after* is 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 accomodate different numbers of parameters used by MakeRoom to both create and configure Room objects?

● Builder **Session 8**

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 sub-class of the Builder?

● Singleton **Session 9**

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?

■ Proxy **Session 10**

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

■ Adapter **Session 11**

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

■ Bridge **Session 12**

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

◆ Mediator **Session 13**

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

◆ Observer **Session 14**

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 Objserver 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 is pattern? Consider an Unregister() message being sent to a subject, just before the subject sends a Notify() message to the ChangeManager (or Controller).

◆ Chain of Responsibility **Session 15**

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?

◆ Memento **Session 16**

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* infact 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?

◆ Command **Session 17**

In the Motivation section of the Command pattern, an application's menu system is described. An application has a Menu, which in turn has MenuItems, 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 such that new comamnds could easily be written that would also have access to the information they need?

● Prototype **Session 18**

Part 1: When should this creational pattern be used over the other creational patterns?

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

◆ State **Session 19**

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

◆ Visitor **Session 20**

One issue with the Visitor pattern involves cyclicality. When you add a new Visitor, you must make changes to existing code. How would you work around this possible problem?

■ Flyweight **Session 21**

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?

◆ Interpreter **Session 22**

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?

■ Facade **Session 23**

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

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

Introduction

From Wikipedia http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems.

Not all software patterns are design patterns. Design patterns deal specifically with problems at the level of software *design*. Other kinds of patterns, such as architectural patterns, describe problems and solutions that have alternative scopes.

History

Patterns originated as an architectural concept by Christopher Alexander (1977/79). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.[1][2] In the following years, Beck, Cunningham and others followed up on this work.

Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 (Gamma et al). That same year, the first Pattern Languages of Programs Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns. The scope of the term remained a matter of dispute into the next decade.

Although the practical application of design patterns is a phenomenon, formalization of the concept of a design pattern languished for several years.[3]

Uses

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns

helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

Design patterns are composed of several sections (see Documentation below). Of particular interest are the Structure, Participants, and Collaboration sections. These sections describe a *design motif*: a prototypical *micro-architecture* that developers copy and adapt to their particular designs to solve the recurrent problem described by the design pattern. A micro-architecture is a set of program constituents (e.g., classes, methods...) and their relationships. Developers use the design pattern by introducing in their designs this prototypical micro-architecture, which means that micro-architectures in their designs will have structure and organization similar to the chosen design motif.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Caveats

In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which might complicate the resulting designs and hurt application performance.

Classification

Design patterns can be classified in terms of the underlying problem they solve. Examples of problem-based pattern classifications include:

- **Fundamental patterns**
  - Delegation pattern: an object outwardly expresses certain behaviour but in reality delegates responsibility
  - Functional design: assures that each modular part of a computer program has only one responsibility and performs that with minimum side effects
  - Interface pattern: method for structuring programs so that they're simpler to understand

- Proxy pattern: an object functions as an interface to another, typically more complex, object
- Facade pattern: provides a simplified interface to a larger body of code, such as a class library.
- Composite pattern: defines Composite object (e.g. a shape) designed as a composition of one-or-more similar objects (other kinds of shapes/geometries), all exhibiting similar functionality. The Composite object then exposes properties and methods for child objects manipulation as if it were a simple object.

Creational patterns which deal with the creation of objects. Abstract Factory and Factory Method are creation patterns

- Abstract factory pattern: centralize decision of what factory to instantiate
- Factory method pattern: centralize creation of an object of a specific type choosing one of several implementations
- Builder pattern: separate the construction of a complex object from its representation so that the same construction process can create different representations
- Lazy initialization pattern: tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed
- Object pool: avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype pattern: used when the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) is prohibitively expensive for a given application
- Singleton pattern: restrict instantiation of a class to one object

Structural patterns that ease the design by identifying a simple way to realize relationships between entities. Adapter is a structural pattern

- Adapter pattern: 'adapts' one interface for a class into one that a client expects
- Aggregate pattern: a version of the Composite pattern with methods for aggregation of children

- Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently
- Composite pattern: a tree structure of objects where every object has the same interface
- Decorator pattern: add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- Extensibility pattern: aka. Framework - hide complex code behind a simple interface
- Facade pattern: create a simplified interface of an existing interface to ease usage for common tasks
- Flyweight pattern: a high quantity of objects share a common properties object to save space
- Proxy pattern: a class functioning as an interface to another thing
- Pipes and filters: a chain of processes where the output of each process is the input of the next
- Private class data pattern: restrict accessor/mutator access

Behavioral patterns that identify common communication patterns between objects and realize these patterns. Interpreter is a behavioral pattern

- Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects
- Command pattern: Command objects encapsulate an action and its parameters
- Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems
- Iterator pattern: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
- Mediator pattern: Provides a unified interface to a set of interfaces in a subsystem
- Memento pattern: Provides the ability to restore an object to its previous state (rollback)
- Null Object pattern: Designed to act as a default value of an object
- Observer pattern: aka Publish/Subscribe or Event Listener. Objects register to observe an event which may be raised by another object
- State pattern: A clean way for an object to partially change its type at runtime

- Strategy pattern: Algorithms can be selected on the fly
- Specification pattern: Recombinable Business logic in a boolean fashion
- Template method pattern: Describes the program skeleton of a program
- Visitor pattern: A way to separate an algorithm from an object
- Single-serving visitor pattern: Optimise the implementation of a visitor that is allocated, used only once, and then deleted
- Hierarchical visitor pattern: Provide a way to visit every node in a hierarchical data structure such as a tree.

Concurrency patterns

- Active Object
- Balking pattern
- Double checked locking pattern
- Guarded suspension
- Leaders/followers pattern
- Monitor Object
- Read write lock pattern
- Scheduler pattern
- Thread pool pattern
- Thread-Specific Storage
- Reactor pattern

Documentation Format for Patterns

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.[4] There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors. However, according to Martin Fowler certain pattern forms have become more well-known than others, and consequently become common starting points for new pattern writing efforts.[5] One example of a commonly used documentation format is the one used by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (collectively known as the Gang of Four) in their book Design Patterns. It contains the following sections:

Pattern Name and Classification: A descriptive and unique name that helps in identifying and referring to the pattern.

Intent: A description of the goal behind the pattern and the reason for using it.

Also Known As: Other names for the pattern.

Motivation (Forces): A scenario consisting of a problem and a context in which this pattern can be used.

Applicability: Situations in which this pattern is usable; the context for the pattern.

Structure: A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.

Participants: A listing of the classes and objects used in the pattern and their roles in the design.

Collaboration: A description of how classes and objects used in the pattern interact with each other.

Consequences: A description of the results, side effects, and trade offs caused by using the pattern.

Implementation: A description of an implementation of the pattern; the solution part of the pattern.

Sample Code: An illustration of how the pattern can be used in a programming language

Known Uses: Examples of real usages of the pattern.

Related Patterns: Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Interface Pattern

Whilst not officially a “Design Pattern” – more a language idiom (there is a sometimes blurry line between the two) it is a core technique in OO and in patterns.

Interface is a special type of class which provides the programmer with a simpler or more program-specific way of accessing other classes. Since the Java programming language relies on its interfaces to expose the functionality of a class to the outside world, it is an integral part of the language (unlike C++, which has header files). However the "interface pattern" discussed here is a even more general term without the restrictions placed upon interfaces by Java, for example the delegation, composite and bridge patterns are different types of interface patterns, however they do also have an implementation, so they would not conform to what Java calls an interface.

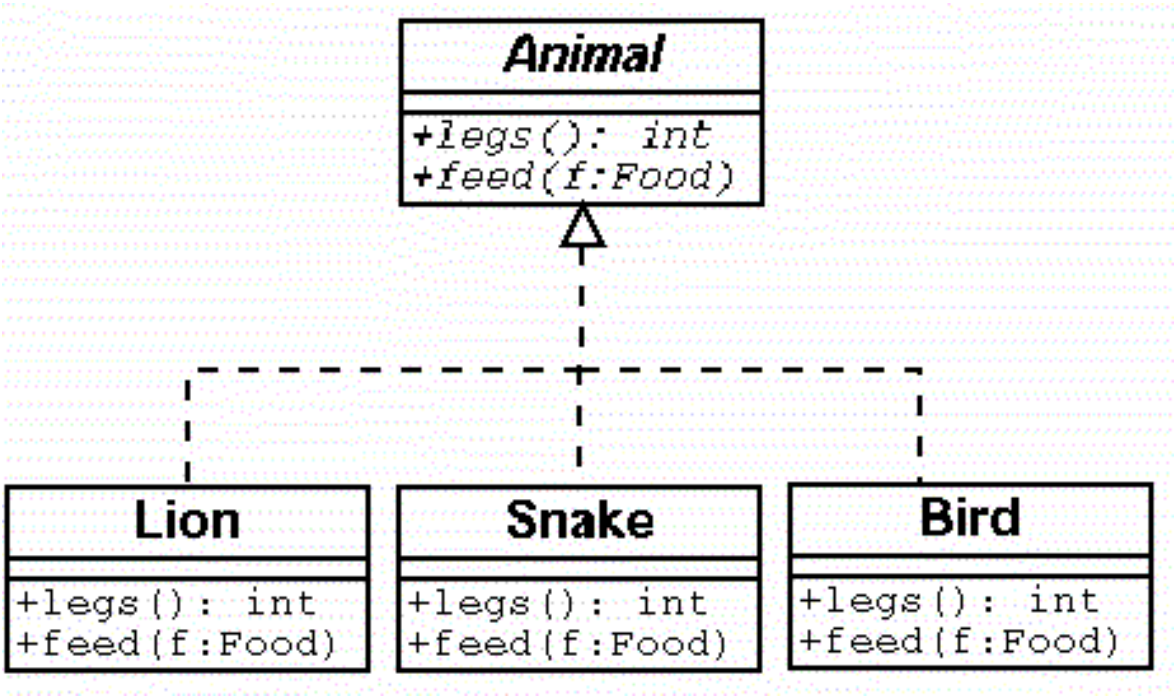
By programming “to an interface” rather than to a specific implementation of that interface, you can substitute different instances of the interface with minimal impact on its client classes.

Java vs. C++ Interfaces

> *Java elevates the notion of interface to be a separate construct, expressly separating interface—what an object must do—from implementation—how an object fulfills this commitment. Java interfaces allow several classes to provide the same functionality, and they open the possibility that a class can implement more than one interface.*
>
> *Design Patterns (Gamma et al. 1990) frequently mentions the use of abstract classes but does not describe the use of interfaces. The reason is that the languages C++ and Smalltalk, which Design Patterns uses for its examples, do not have an interface construct. This omission has a minor impact on the utility of the book for Java developers, because Java interfaces are quite similar to abstract classes.*
>
> From Design Patterns Java™ Workbook - Steven John Metsker

Benefit of the interface pattern:

> *You want to keep client classes sufficiently independent of specific data-and-service-providing classes so you can substitute another data-and-service-providing class in its place with minimal impact on its client classes. You accomplish this by having other classes access the data and services through an interface.*
>
> From Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, second edition by Mark Grand.


Animal
+legs(): int
+feed(f:Food)
Lion
+legs(): int
+feed(f:Food)
Snake
+legs(): int
+feed(f:Food)
Bird
+legs(): int
+feed(f:Food)

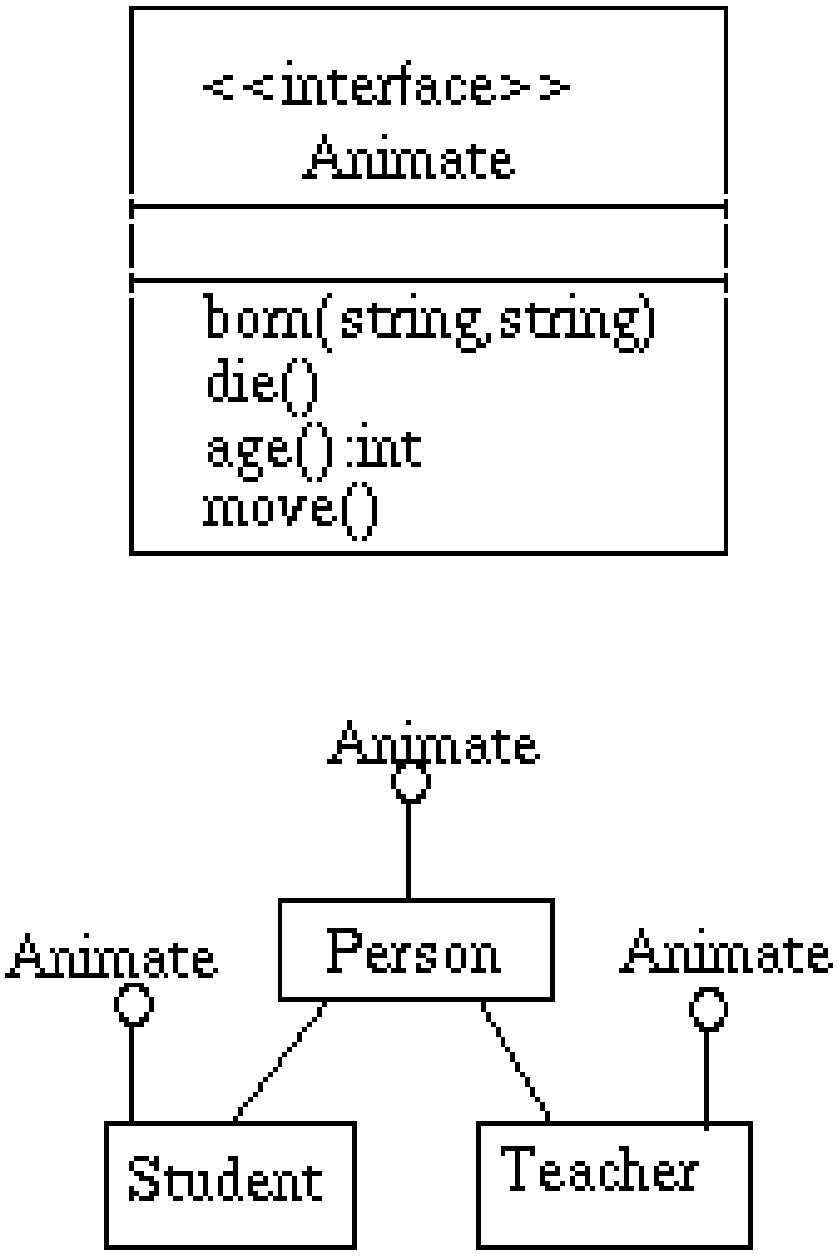


Figure 1 - We can document an Animate interface defining things that happen to Animate objects. We can then specify that People, Student, and Teacher, all implement this interface.

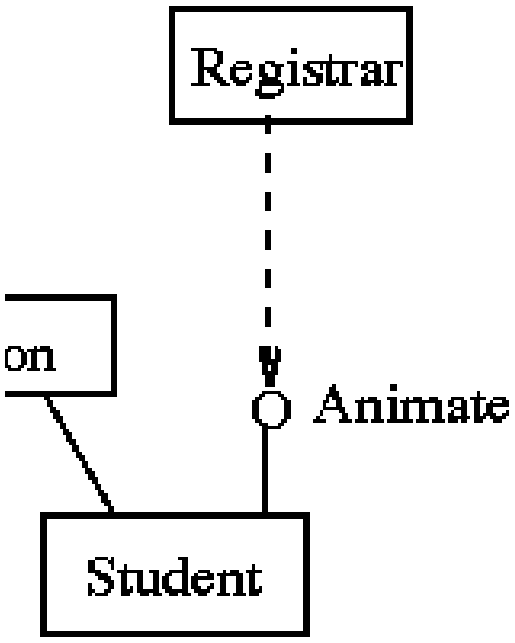


Figure 2 - There is also a notation for showing that a class is a client of an interface. For example we might show that a class of Registrars use the 'Animate' interface of a student -- probably to add and delete Student records.

From http://www.csci.csusb.edu/dick/samples/uml1b.html

The usage (dotted) arrow shows how one class uses features listed in the interface to the used class.

Related patterns are

- **Delegation**
  - The Delegation and Interface patterns are often used together.
- **Adapter**
  - The Adapter pattern allows objects that expect another object to implement a particular interface to work with objects that don't implement the expected interface.
- **Strategy**
  - The Strategy pattern uses the Interface pattern.

Notes:

Factory Method

Allow subclasses to choose which type of object to create. Instances created via call to a 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.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Participants

The classes and/or objects participating in this pattern are:

- Product (Page)
  - defines the interface of objects the factory method creates
- ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)
  - implements the Product interface
- Creator (Document)
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- ConcreteCreator (Report, Resume)
  - overrides the factory method to return an instance of a ConcreteProduct.

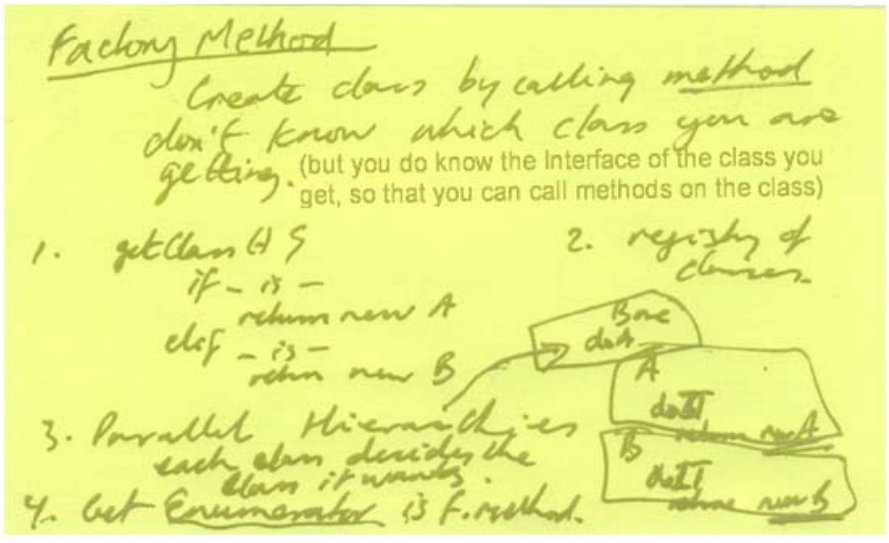
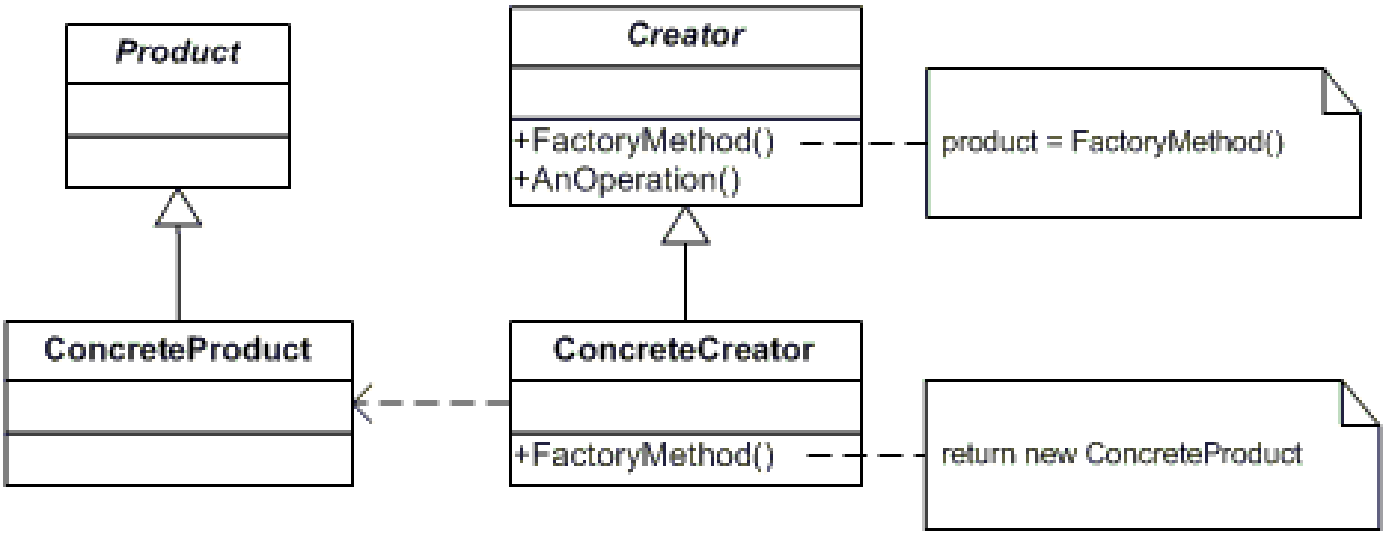


Figure 3 - A summary of the basic ideas in factory method


Product
Creator
+FactoryMethod()
+AnOperation()
product = FactoryMethod()
ConcreteProduct
ConcreteCreator
+FactoryMethod()
return new ConcreteProduct

Core Ideas

We create a class by calling a **method**, and we don't know exactly which class we are going to get. Though typically we *do know* what interface or abstract class the instance supports (so that we know how to 'drive' it).

GetEnumerator() is a factory method. It returns an interator object – but the type we get depends on the collection we are iterating over.

Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation. [GOF, p136]

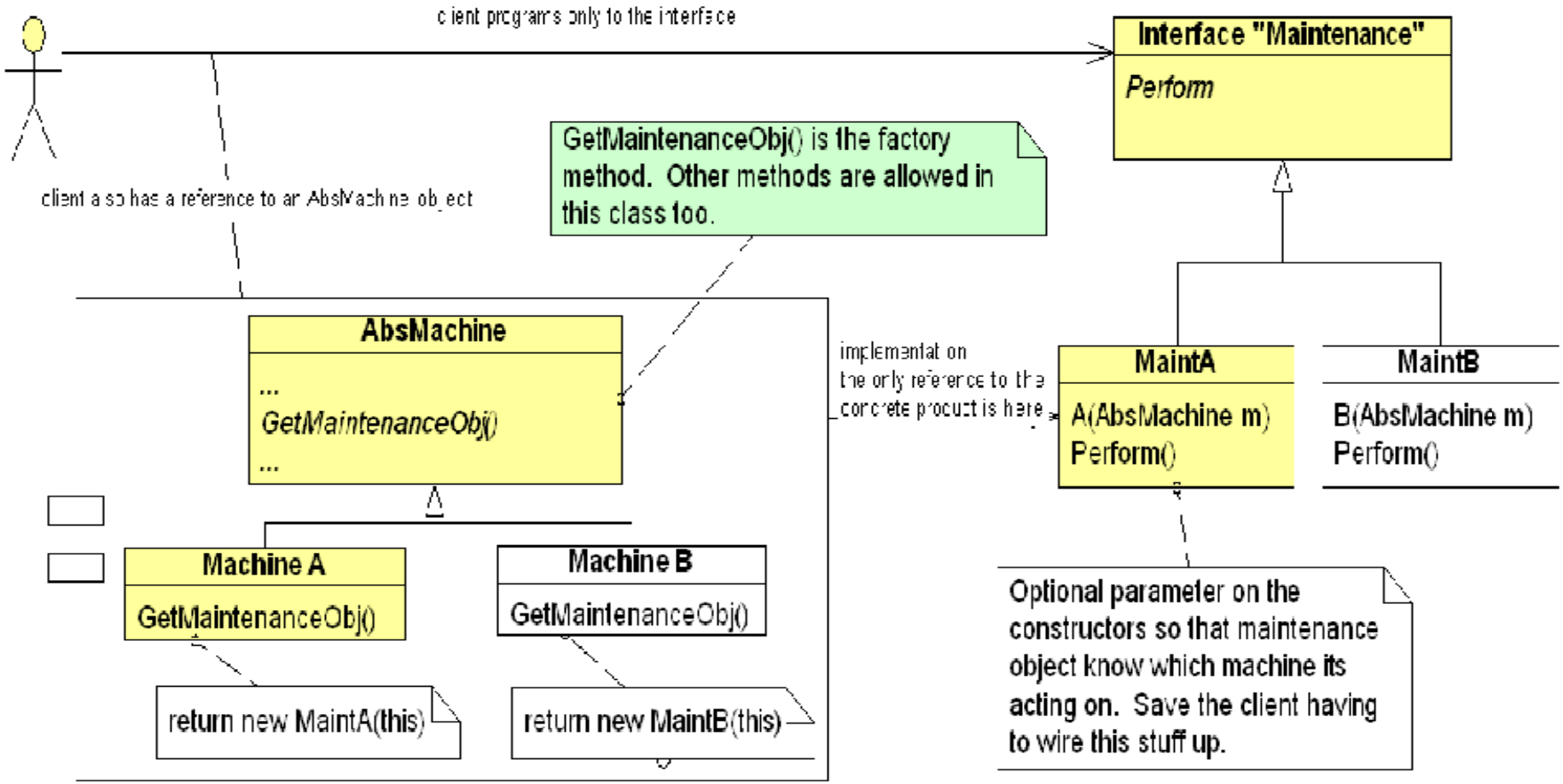
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation). [GOF, p136]

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Notes on the diagram opposite:

- Subclasses override the factory method method and return the appropriate product relating to their subtype, using code.
- Client refers to the abstract product interface only.


client programs only to the interface
Interface "Maintenance"
Perform
client also has a reference to an AbsMachine object
GetMaintenanceObj() is the factory method. Other methods are allowed in this class too.
AbsMachine
...
GetMaintenanceObj()
...
implementation
the only reference to the concrete product is here
MaintA
A(AbsMachine m)
Perform()
MaintB
B(AbsMachine m)
Perform()
Machine A
GetMaintenanceObj()
Machine B
GetMaintenanceObj()
return new MaintA(this)
return new MaintB(this)
Optional parameter on the constructors so that maintenance object know which machine its acting on. Save the client having to wire this stuff up.

Rules of thumb

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Methods are usually called within Template Methods. [GOF, p116]

Factory Method: creation through inheritance. Prototype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

Code Ideas

www.dofactory.com's structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

www.dofactory.com's real-world code demonstrates the Factory method offering flexibility in creating different documents. The derived Document classes Report and Resume instantiate extended versions of the Document class. Here, the Factory Method is called in the constructor of the Document base class.

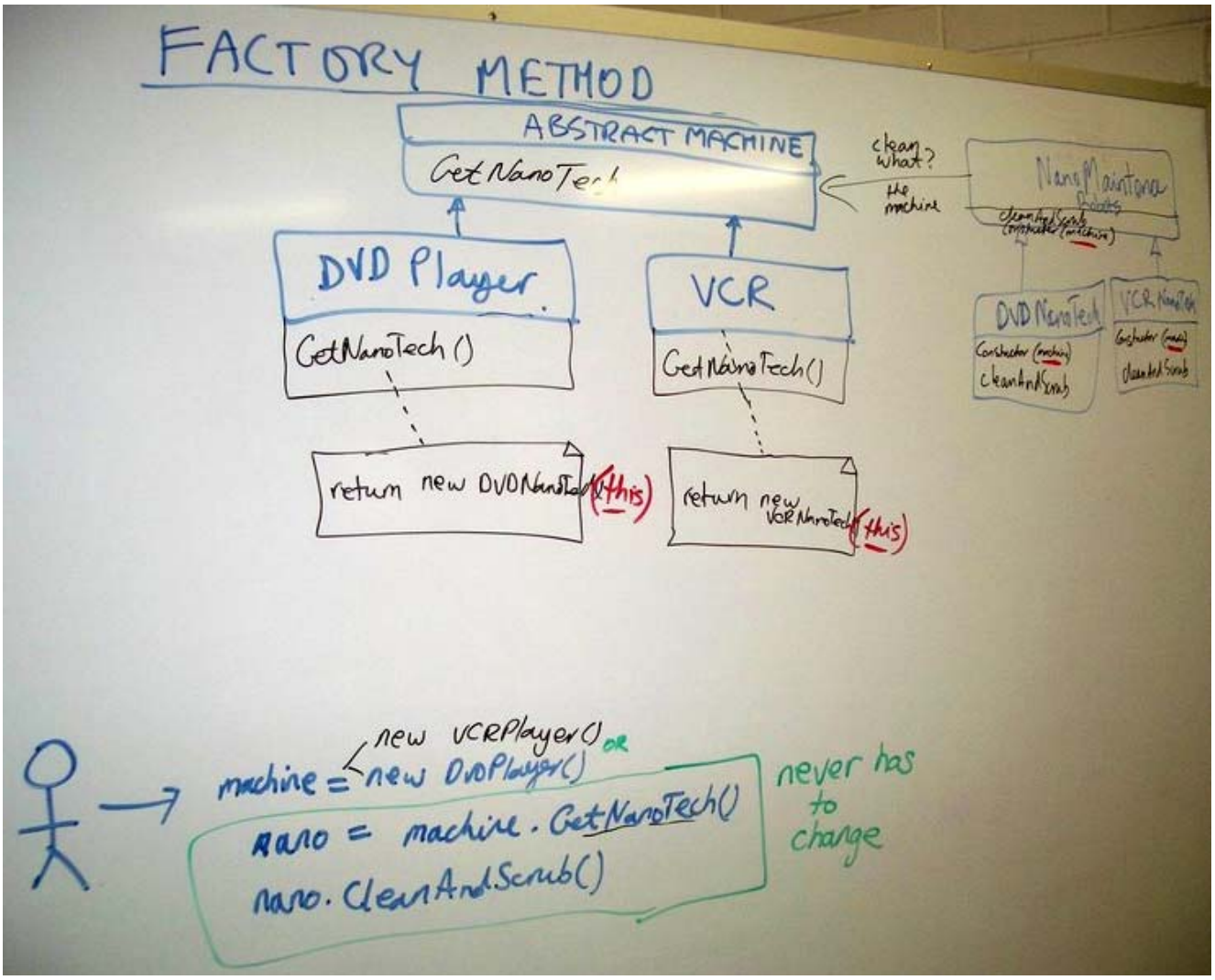


Figure 4 - How Factory Method allows a class to instantiate the appropriate "helper" class relevant to it - without the client code caring

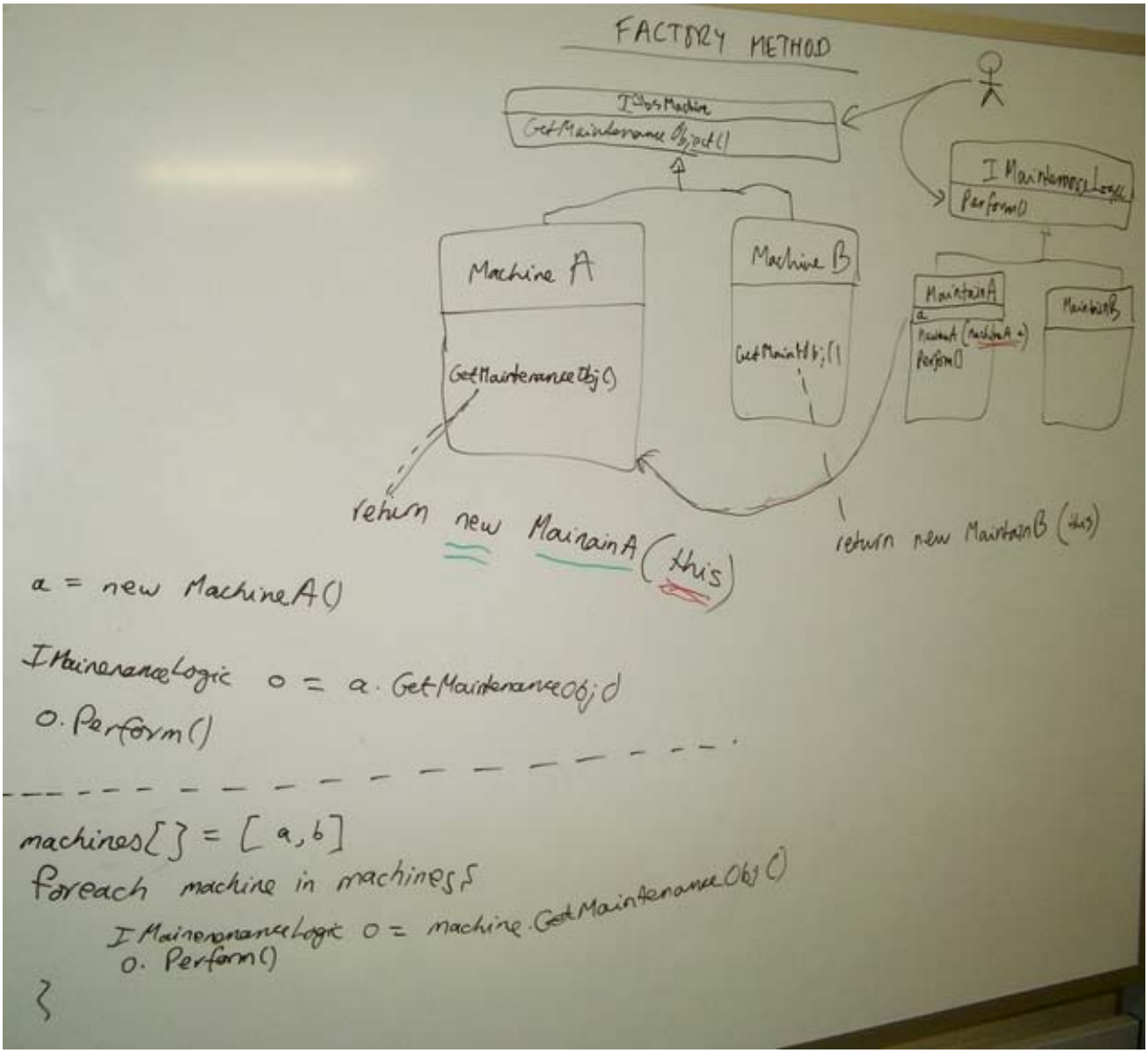


Figure 5 - Each "Machine" needs a different "Maintenance Class" – using Factory Method allows this to happen.

The road to Factory Method – Article

Taken from the article by Andy Bulka at:

http://www.atug.com/andypatterns/the_road_to_bridge.htm

This story leads us from the idea of what a 'factory' is trying to achieve, through some basic alternate implementations of factories, and then to the 'factory method' style implementation presented by GOF (the original design patterns book).

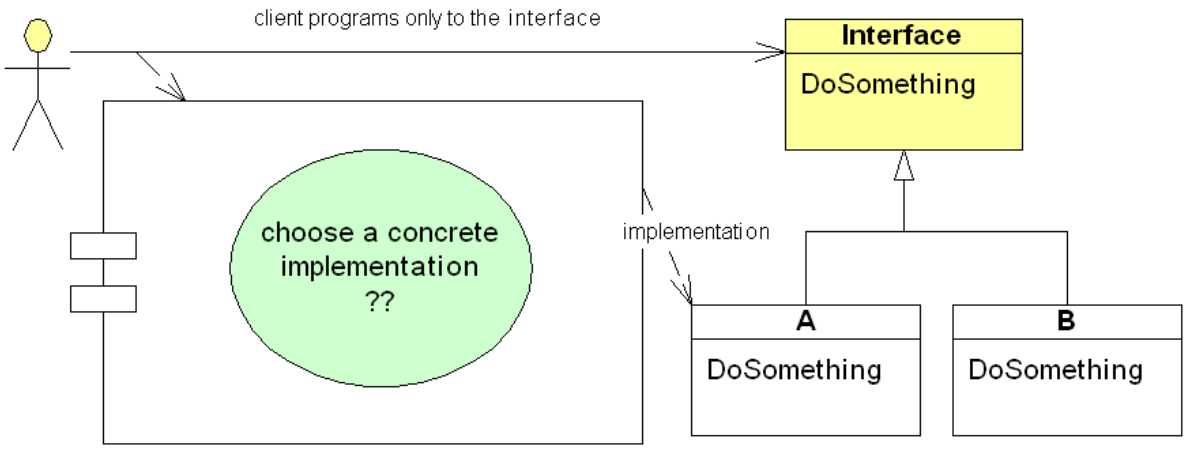
Note that design patterns are mean to be 'ideas' and not to be tied to particular implementations. As such, any 'method' that when called, returns an instance of a class is a 'factory method'.

The idea of a Factory

Create A or B at runtime by asking another class to create the concrete object for us. Pass in the flag to the factory or let the factory decide for itself which implementation we want.

Factory class is the only class to refer to concrete products. The client refers to the interface/abstract class only.

We are still talking directly to the concrete object (either an A or a B).



There are a number of factory method variants:

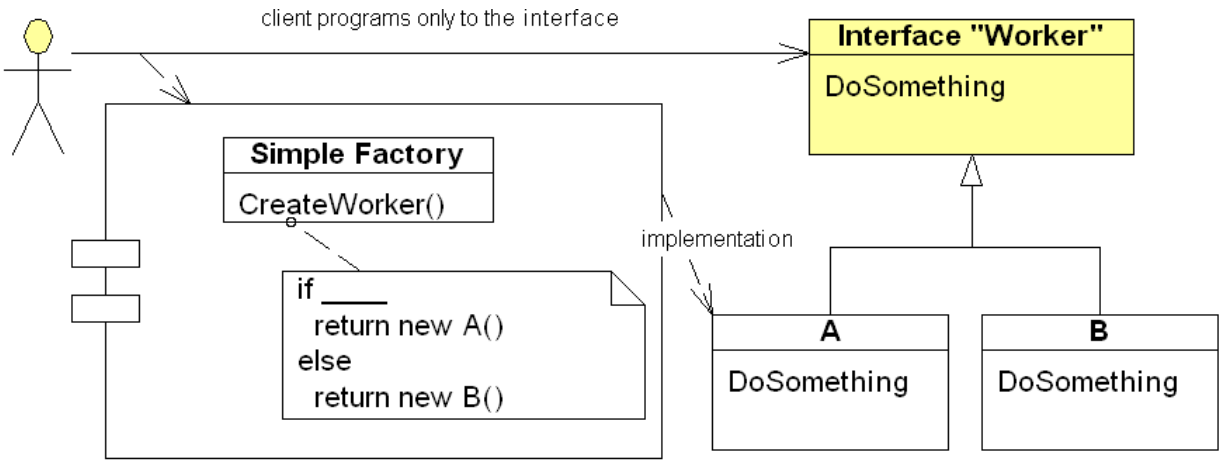
Simple Super Dumb Factory

Encapsulates the "dynamic run time choice" solution discussed in the beginning of this talk. Benefit is that the conditional logic containing the if statement is hidden and possibly centralized in a factory class.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via conditional code.

```
Factory f = new SimpleFactory()

Worker o = f.CreateWorker()

o.DoSomething()    // We don't know if its an A or a B.
                   // Everything works ok.
```



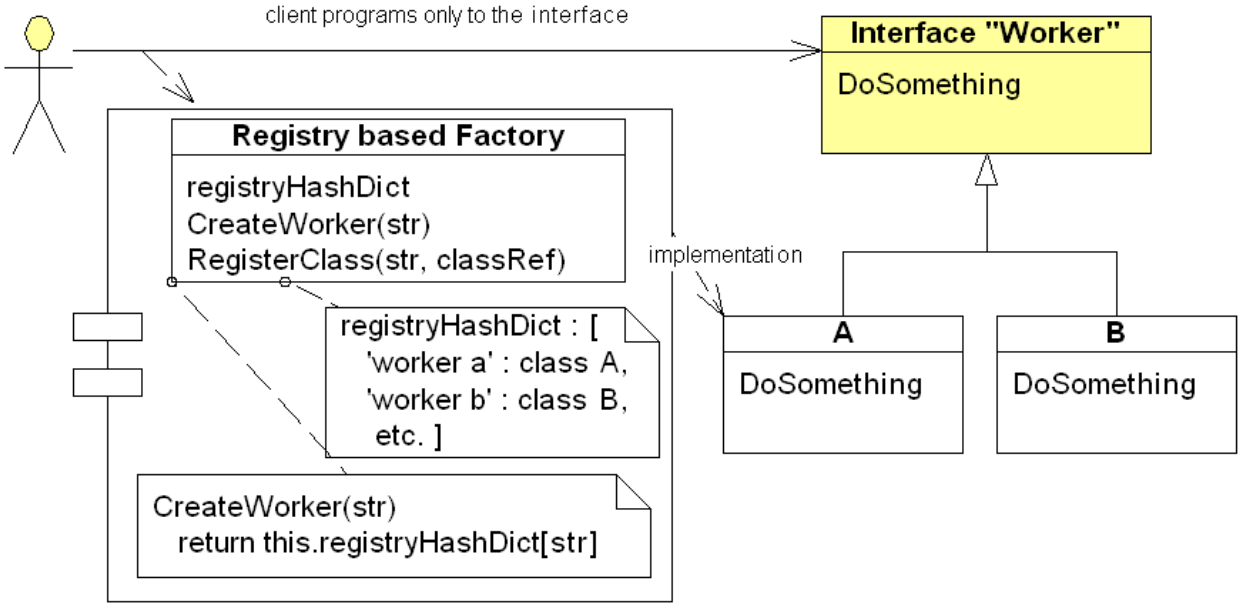
Registry Based Factory

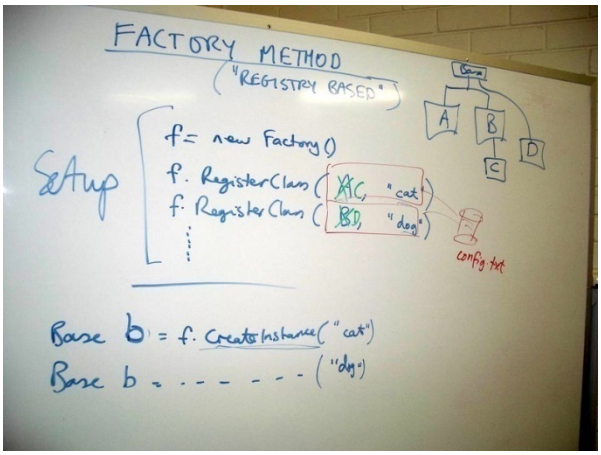
Maintains a registry of mappings between strings (or any type of key e.g. objects, class references, numbers etc.) and class references. Benefit: more generalized, no if statements.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class. *Question: Is there a need to **cast** anything in this implementation?*

The choice is made via registry key.

```
key = 'worker a'   // in setup code somewhere

Factory f = new RegistryFactory()

f.RegisterClass('worker a', class A) // more initial "setup code"

Worker o = f.CreateWorker(key)

o.DoSomething()    // We don't know if its an A or a B.
                   // Everything works ok.
```





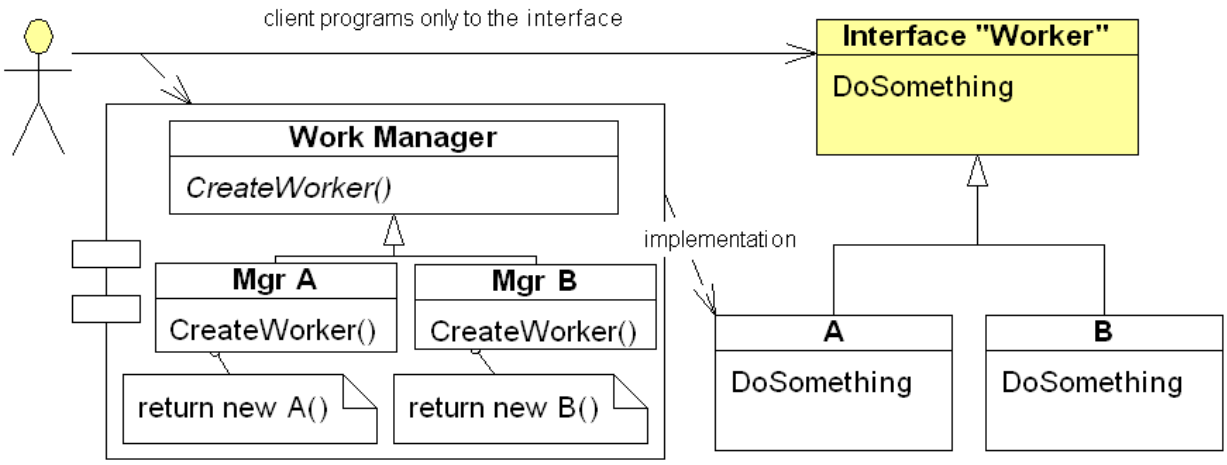
GOF Factory Method

Assumes the *client already has* an instance of some class which needs either a A or B version of a worker class.

Each alternative instance of the existing class overrides a create method differently, each instantiating a different concrete product - typically one matching their own functionality. Benefit: no class reference language facilities required.

Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.

The choice is made via polymorphic override.



Note that the choice as to which Work Manager (MgrA or MgrB) to instantiate in the first place is going to be an issue, but is not the point of this example. The point is that once you have a particular brand of work manager, then you will get a related brand of of worker via the suitably overriden CreateWorker factory method.

```
WorkManager f = new MgrA()      // done somewhere in setup

Worker o = f.CreateWorker()    // don't need to pass in key anymore

o.DoSomething()    // We don't know if its an A or a B.
                   // but we do know worker is of interface "Worker"
```

There will be parallel hierarchies, e.g. the WorkManager and the Worker hierarchies closely match, with A and B versions of their subclasses. Start to think in terms of a family of classes.

Notes:

Strategy

Encapsulates an algorithm inside a method of a class and delegate to it.

Definition

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

Problem

If clients have potentially generic algorithms embedded in them, it is difficult to: reuse these algorithms, exchange algorithms, decouple different layers of functionality, and vary your choice of policy at run-time. These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions.

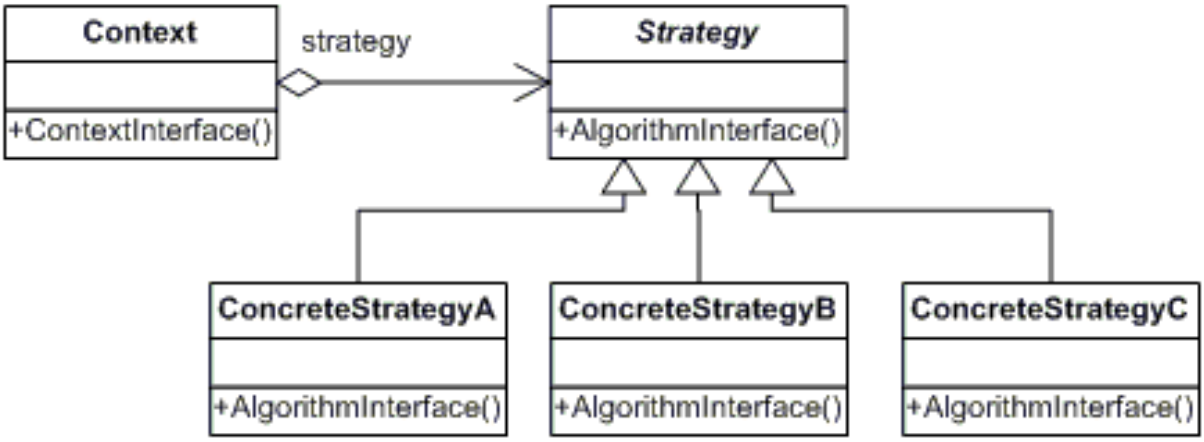
Solution

Replace the many, monolithic, conditional constructs with a Strategy inheritance hierarchy and dynamic binding.

- Identify the protocol that provides the appropriate level of abstraction, control, and interchangeability for the client.
- Specify this protocol in an abstract base class.
- Move all related conditional code into their own concrete derived classes.
- Configure the original application with an instance of the Strategy hierarchy, and delegate to that "contained" object whenever the "algorithm" is required.

Ideas

- Clients shall be decoupled from "choice of algorithm".
- Clients shall be decoupled from complex, algorithm-specific data structures.
- The choice, or implementation, of algorithm shall be configurable at run-time.
- "case" statements are a maintenance nightmare - they shall be avoided.


Context
strategy
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

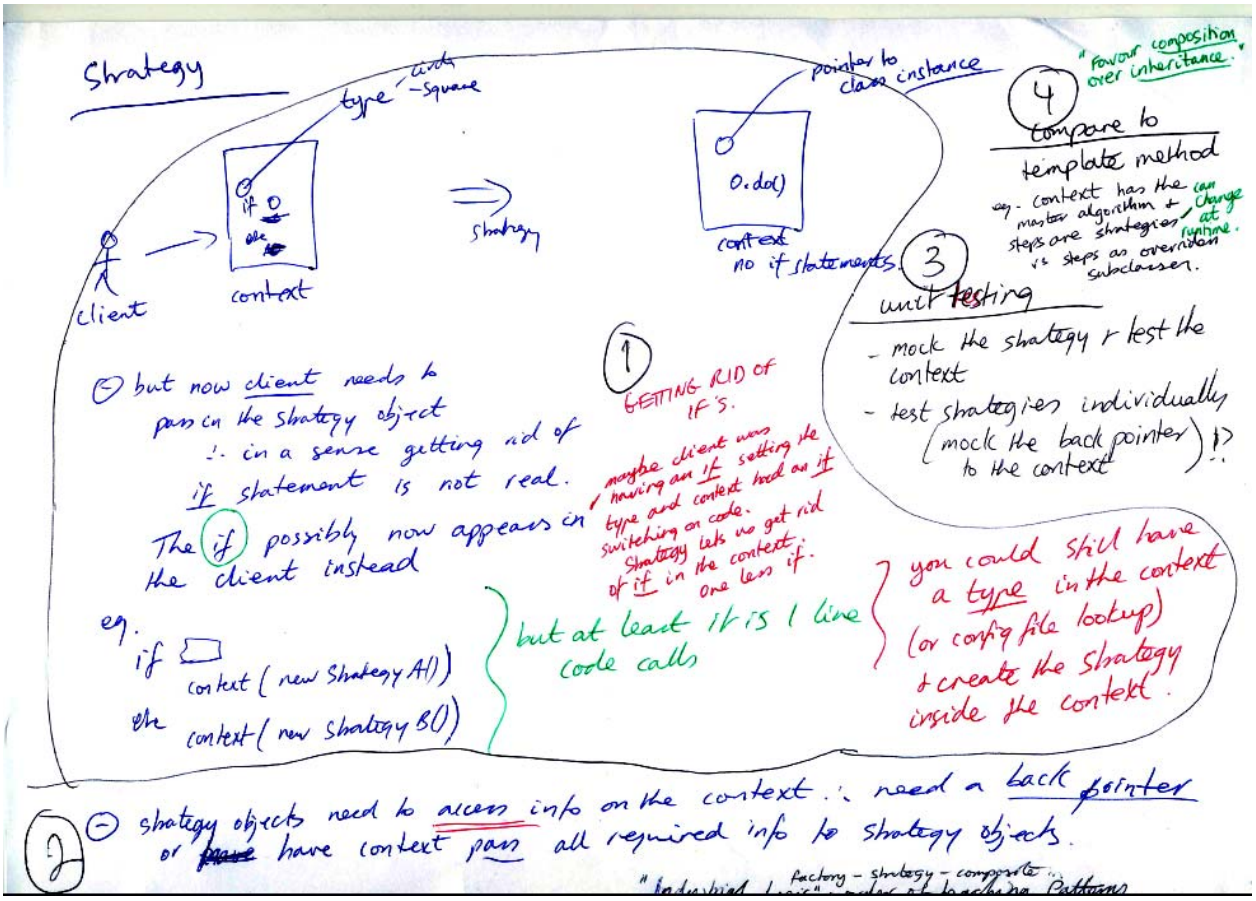


Figure 6 - The basic ideas in Strategy Pattern

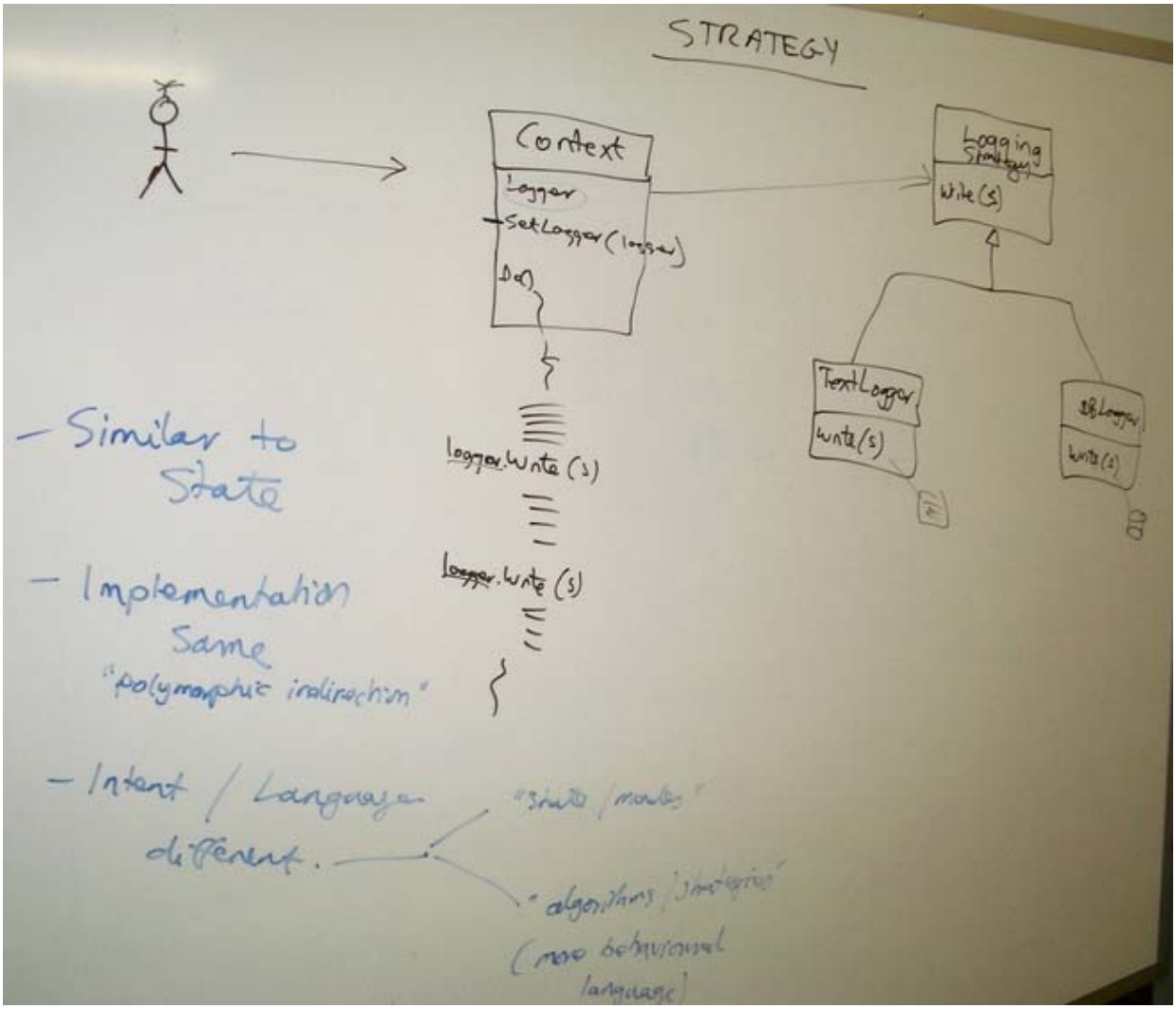


Figure 7 - Strategy Pattern – the basic idea is to delegate work to a method on another class – the "strategy". Notice the common base class, so that you can swap in different concrete strategy classes. Context (the lhs.) only talks to the Strategy Interface.

Discussion

"Strategies can provide different implementations of the same behavior. The client can choose among Strategies with different time and space trade-offs." [GOF, p318] This sounds exactly like the intent of the Bridge pattern. The apparent overlap can be explained by observing that both patterns employ the same composition-delegation structure. The distinction is in intent: Strategy is a "behavioral" pattern (the focus is on delegating responsibilities), Bridge is a "structural" pattern (the focus is on establishing relationships).

The Strategy pattern should only be used when the variation in behavior is relevant to clients. If this criteria is not satisfied, the additional abstraction and effort necessary to implement the Strategy pattern is not justified.

There are two implementations for Strategy. The first is described above. It uses abstract coupling and composition-delegation. The second uses C++ templates. It is demonstrated in the second example that

follows, and requires the client to "bind" the choice of algorithm at compile time.

Non Software Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and tlme. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Code Ideas

www.dofactory.com's structural code demonstrates the Strategy pattern which encapsulates functionality in the form of an object. This allows clients to dynamically change algorithmic strategies.

www.dofactory.com's real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

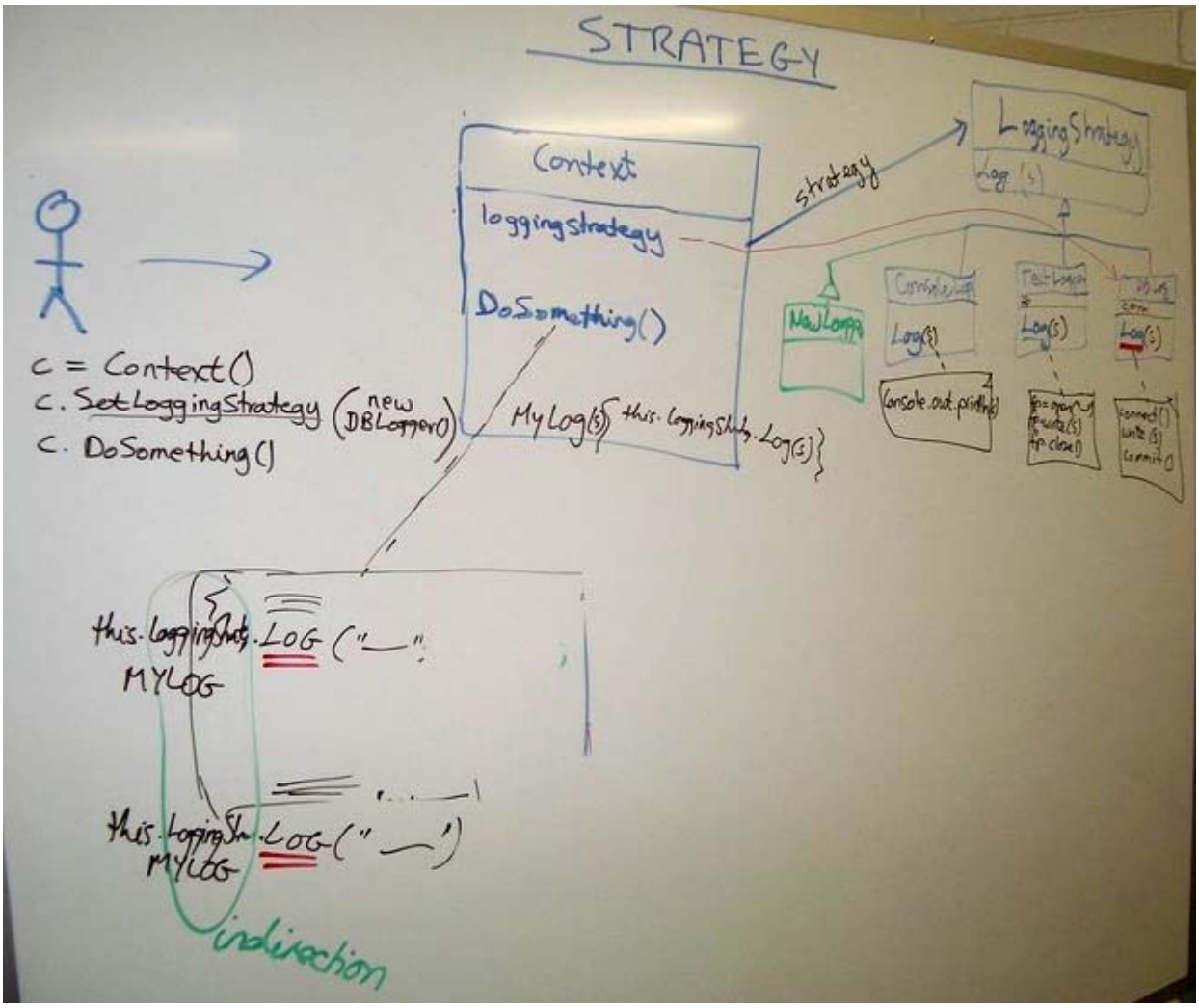


Figure 8 - A logging "strategy"

Example: Real world example of strategy being injected into java GUI

When a frame is created, it's provided with a BorderLayout layout manager. This strategy allows programmers to add controls to the Northern, Eastern, Western, Southern, and Central regions of the frame. If this isn't acceptable, a programmer can specify a different layout manager using the setLayout() method.

For example, a GridLayout strategy allows programmers to partition a frame into any number of rows partitioned into any number of columns. Here we partition our frame into a single row consisting of two columns:

```
Frame gui = new Frame();
gui.setLayout(new GridLayout(1, 2));
gui.add(new Button("ON"));
gui.add(new Button("OFF"));
```

A FlowLayout strategy attempts to horizontally center all of the controls in the frame, creating new rows as needed. Again, we use the setLayout() method to change the layout strategy:

```
Frame gui = new Frame();
gui.setLayout(new FlowLayout());
gui.add(new Button("ON"));
gui.add(new Button("OFF"));
```

Participants

The classes and/or objects participating in this pattern are:

- Strategy (SortStrategy)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy (QuickSort, ShellSort, MergeSort)
  - implements the algorithm using the Strategy interface
- Context (SortedList)
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Rules of thumb

State is like Strategy except in its intent. [Coplien, Mar96, p88]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58]. They differ in intent - that is, they solve different problems.

Strategy lets you change the guts of an object. Decorator lets you change the skin. [GOF, p184]

Strategy is to algorithms as Builder is to creation. [Icon, p8-13]

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, Mar96, p88]

Strategy objects often make good Flyweights. [GOF, p323]

Strategy is like Template Method except in its granularity. [Coplien, Mar96, p88]

More points:

- See Andy's BlueJ strategy example, fairly vanilla UML example.
- You can "inject" a strategy object during construction, so have your constructor take a parameter (the strategy object).
- Sometimes it is simpler to inject a strategy function directly, without the "baggage" of a object (with a strategy method). You can do this in python since functions are first class things. In .NET you can pass around delegates. On the other hand if the strategy function relies on state information / object attributes and other private sub-methods then having a strategy as an object with a method is just fine.

Notes:

Decorator

Add responsibilities to objects dynamically, using repeated wrapping/chaining.

Intent

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

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class. Decorating may also lead to a class hierarchy combinatorial explosion.

Participants

The classes and/or objects participating in the Decorator pattern are:

- Component (LibraryItem)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent (Book, Video)
  - defines an object to which additional responsibilities can be attached.
- Decorator (Decorator)
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator (Borrowable)
  - adds responsibilities to the component.

• Components shall be extensible at run-time.

• Functionality shall be "layer-able". The client shall be capable of configuring any combination of capabilities by simply specifying the layers (or wrappers, or onion skins) to be applied.

The Basic Idea

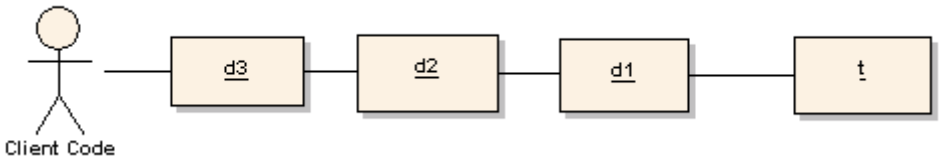
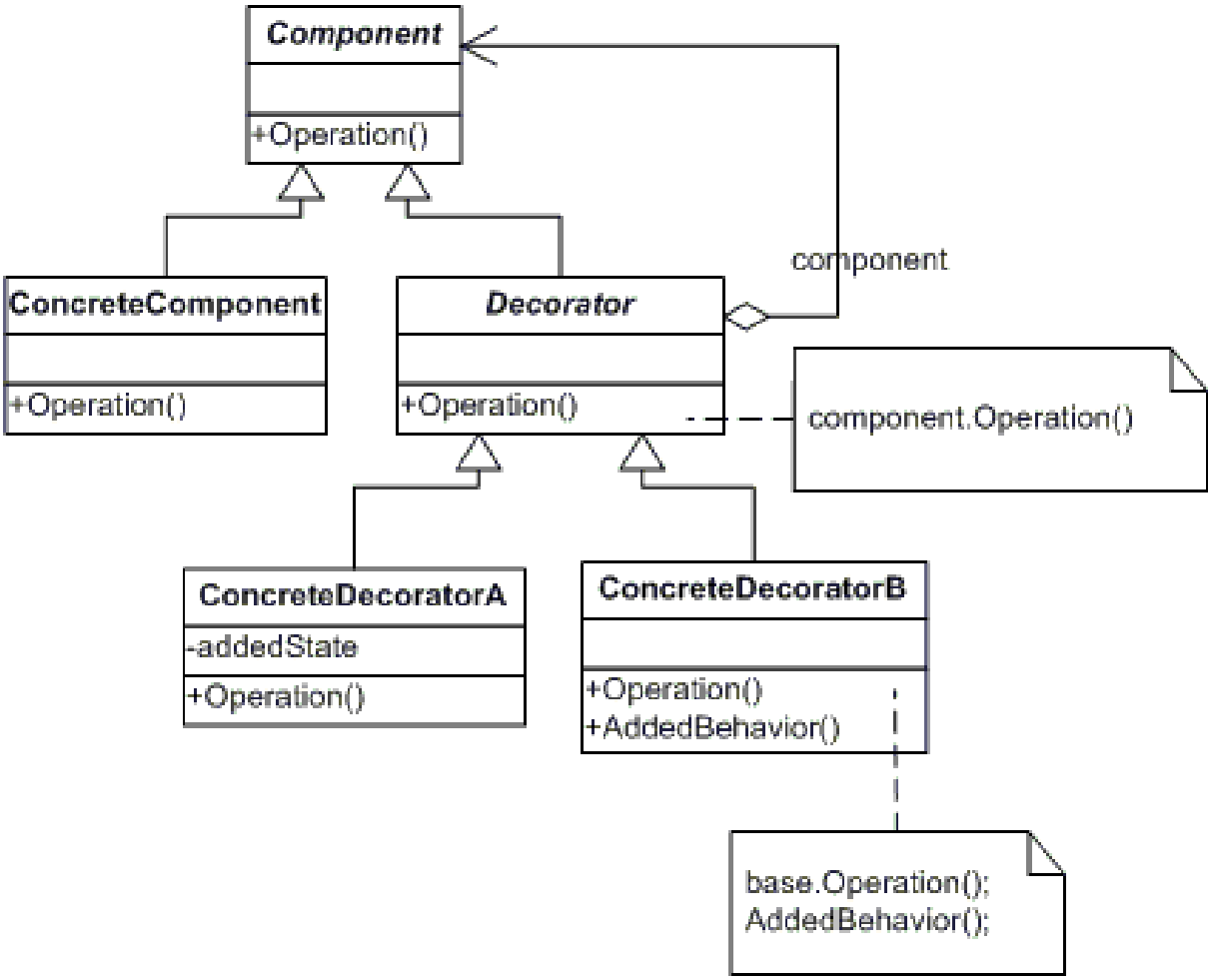


Figure 9 - Client code is used to dealing with class t. We have inserted a chain of decorators in front of object t - each object does it's thing and then calls the next object in the chain.


Component
+Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();

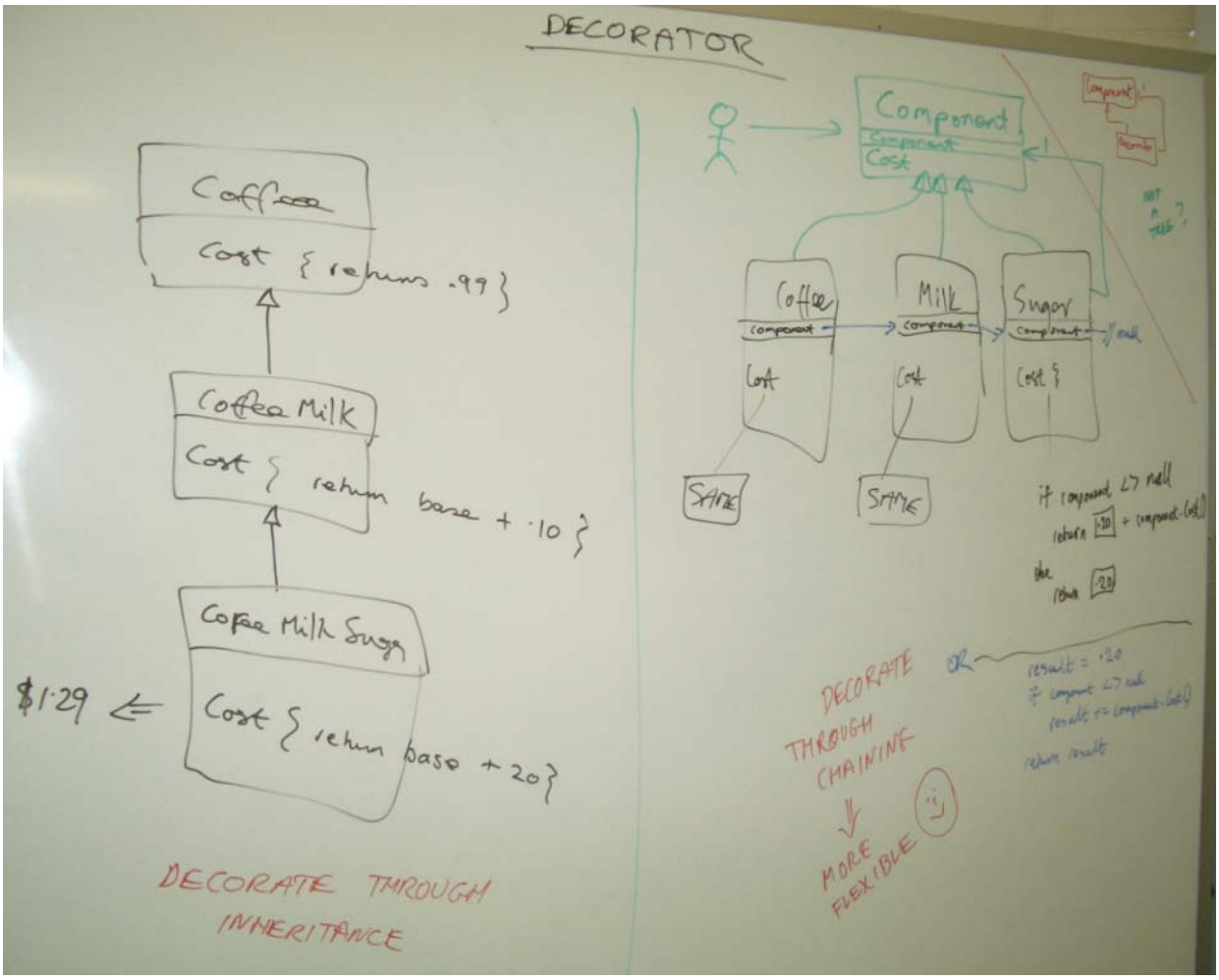


Figure 10 - Before and after Decorator. Before we have decoration through inheritance, which leads to class hierarchy combinatorial explosion. After we have decorator using chaining.

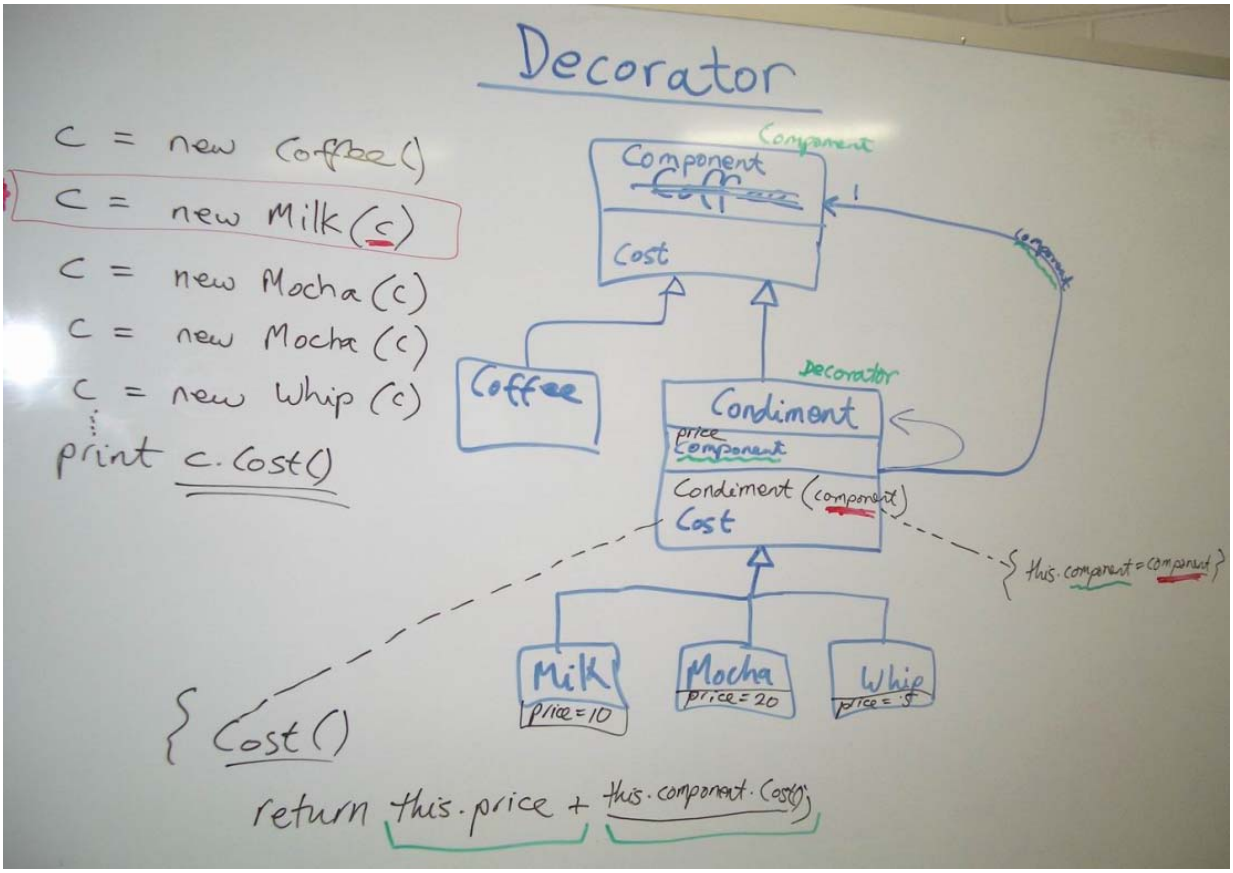


Figure 11 - Decorator UML in detail

Andy's Tips

- Adds functions by wrapping a class, again and again, if necessary.
- Add responsibilities to a class more flexibly than inheritance.
- Examples are decorated borders or decorated strems.
- Structure similar to composite. Decorators inherit from the 'composite class'.
- A common operation implemented across a hierarchy.
- Resembles proxy, forward the call to the subordinate decorator object.
- You can naively decorate (override methods and call super/base) through inheritance but this leads to class hierarchy combinatorial explosion. Decorating using chaining leads avoids the combinatorial explosion.
- A possible negative: type checking will fail i.e. the client code shouldn't check that the object it is dealing with is of a certain class, because it might be wrapped, and thus be of the decorated type.
- There are two senses in which there is 'order' in decorators.

  1. The first is the order in which decorators are chained together (this can be important e.g. in streams wrapping streams – these things may need to be done in a certain order).

2. The second sense is within the operation method that each decorator implements. Each decorator must at some point call the same operation on the next decorator in the chain. A decorator method can call the next decorator before its own code or after its own code.

Discussion

Suppose you have a user interface toolkit and you wish to make a border or scrolling feature available to clients without defining new subclasses of all existing classes. The client "attaches" the border or scrolling responsibility to only those objects requiring these capabilities.

```
Widget*  aWidget = new BorderDecorator(
                   new HorScrollDecorator(
                   new VerScrollDecorator(
                  new TextWidget( 80, 24 ))));
aWidget->draw();
```

Another example could be cascading responsibilities on to an output stream –

```
Stream*  aStream = new CompressingStream(
         new ASCII7Stream(
               new FileStream( "fileName.dat" )));
aStream->putString( "Hello world" );
```

The solution to this problem involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Code Ideas

www.dofactory.com's structural code demonstrates the Decorator pattern which dynamically adds extra functionality to an existing object.

www.dofactory.com's real-world code demonstrates the Decorator pattern in which 'borrowable' functionality is added to existing library items (books and videos).

Example1:

```
t = Thing()
d1 = Decorator(t)
d2 = Decorator(d1)
```

```
d3 = Decorator(d2)
d3.Operation()
```

Example2 - Java Coffee Code Example:

```
public class Coffee extends Component
{
        public Coffee()
        {
            this.price = 1.99;
        }

        public double Cost()
        {
                return this.price;
        }
}
public abstract class Component
{
      protected double price;

      public abstract double Cost();
}
public class Condiment extends Component
{
    private Component component;  // pointer to previous component in chain

    public Condiment( Component c)
    {
        this.component = c;
    }

    public double Cost()
    {
        return this.price + this.component.Cost();
    }
}
public class Milk extends Condiment
{
      public Milk(Component c)
      {
          super(c);
            this.price = .10;
      }
}

public class Mocha extends Condiment
{

      public Mocha(Component c)
```

```
        {
            super(c);
              this.price = .20;
        }

}
public class Whip extends Condiment
{
        public Whip(Component c)
        {
            super(c);
              this.price = .5;
        }

}
```

Notes on the code:

- The main logic is in the Condiment base class Cost() method
- Cost() is calculated by adding the current condiment price then adding the result from calling Cost() on the next condiment in the chain. You stop when you hit the Coffee's Cost() method which simply returns the coffee's price, without calling anything else in the chain.
- The condiment base class constructor is important. You wire up the chain at construction time using the constructor in the base class, passing in the next object in the chain.

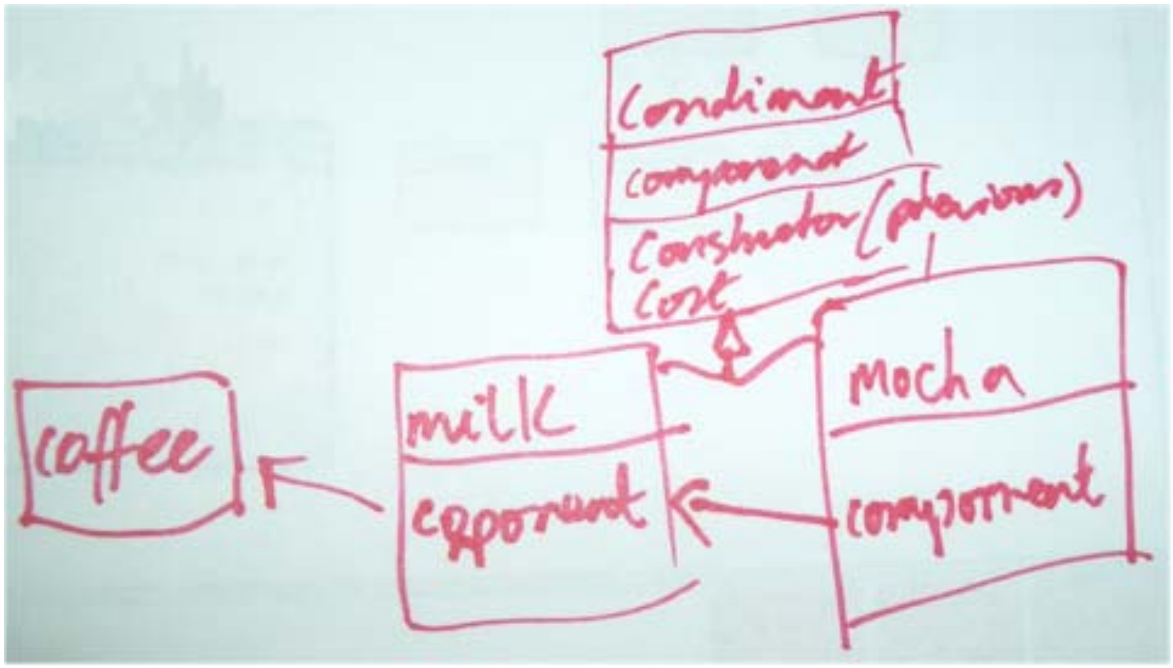


Figure 12 - How the chain is wired up. Note the constructor in the base class. Note the Coffee class points to nil.

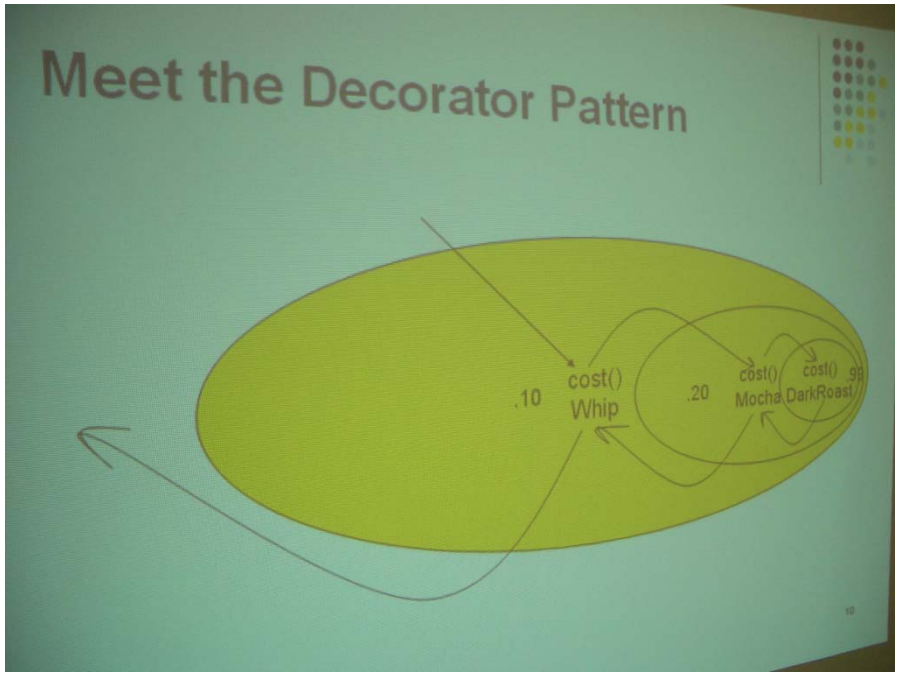


Figure 13 - Sequence diagram of how decorator works

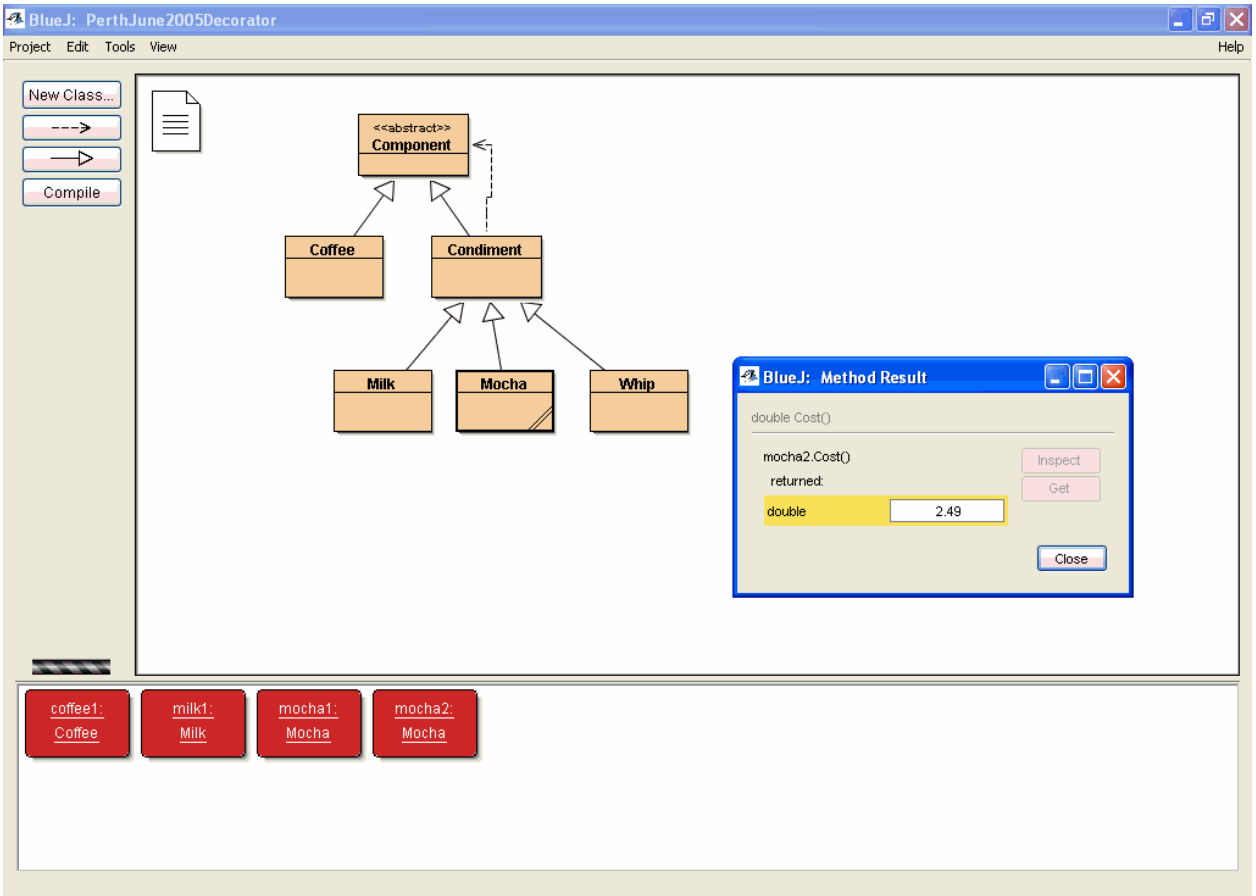


Figure 14 - Java UML implementation of coffee decorator – result (displayed in BlueJ)

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, p149]

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. [GOF, p349]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Decorator lets you change the skin of an object. Strategy lets you change the guts. [GOF, p184]

Notes:

Composite

A tree structure of simple and composite objects

Intent

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

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Discussion

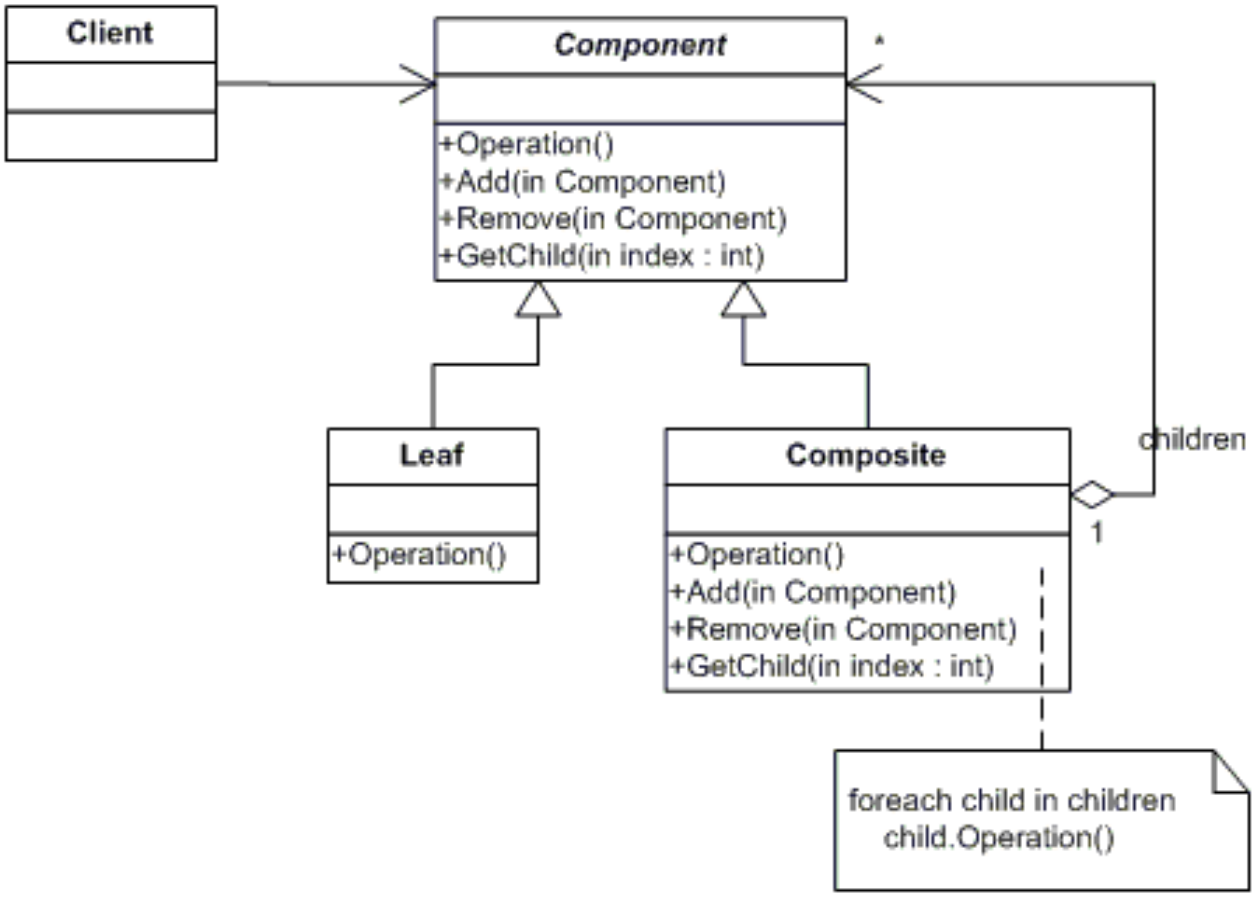
Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.

Ideas

- The system shall support recursive composition.
- The system shall support "whole-part" hierarchical assemblies of components.
- Clients shall be able to transparently interact with compositions of components (e.g. a sub-tree) and individual components (e.g. a node).


Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
children
1
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
foreach child in children
child.Operation()

Andy's Tips

- Tree structures and whole/part hierarchies let clients treat objects uniformly.
- As well as the operations

```
Add(o)
Remove(o)
GetChildren()
```

you typically have some operation method on all the classes. E.g.

```
Traverse()
```

where the implementation of the method on the composite class is different than on the leaf class. But the client code can just call the Traverse method (no knowing whether they are dealing with a composite or a leaf) and still get sensible results.

- A big implementation question with Composite Pattern is whether to implement methods like AddChild() in the base Component base class, or only on the composite class. If we want a consistent interface to the client (which deals only with components) then we should implement AddChild() in the base class. However do you then implement a vector/arraylist of children in the leaf class, or simply have the leaf class return an empty list? Should the leaf class implement AddChild(child) and if so - what does this mean - after all it's a nonsense operation for a leaf. So should AddChild() be on the base component class after all—perhaps its ok to have it there, its just that it does nothing or raises an exception if called on a leaf.

  Sometimes you might want the client to know whether they are dealing with a composite or a leaf. So you might add a IsLeaf() method. Other times you may not want to allow this sort of distinction to be able to be made. See opinions sections (next page) for more on this.

  My compromise solution is:
  - implement only guaranteed common operations on the base class
  - implement IsLeaf() on the base class and allow the client to access deeper functionality e.g. AddChild(c) safely if that's what the client code needs to do.

Ultra simple Composite implementation

An alternative implementation of composite is to simply have a class point to itself.

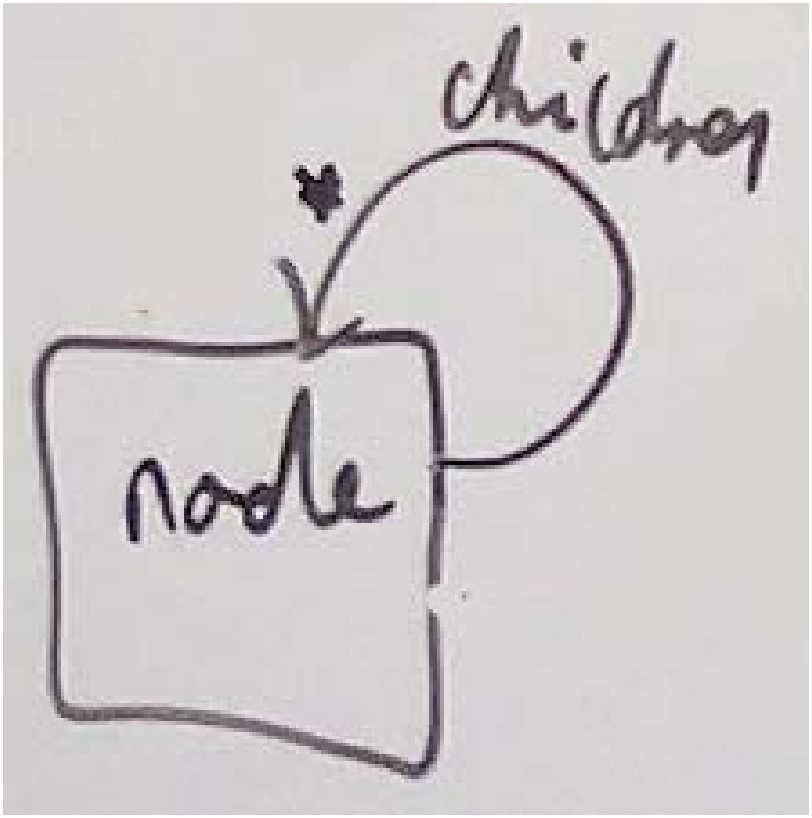


Figure 15 - Ultra simple implementation of composite pattern

This still lets you create trees. So why use the more "classic" approach of a base class and composite and leaf subclasses? Well, you may want to treat leafs and composites differently e.g. in a "file system" example, where you have a leaf as a "file" and a composite as a "directory" - you might define a "double click" method in the base class and override "double click" for files/leaves to launch an application and override "double click" for directories/composites to simply navigate into the directory and disaply its contents. Think about how "File Explorer" in an operating system works – leaves and composites (file and directories) are treated differently. If you had a simple UML implementation (as in Figure 15 above), where everything is a node, you would have to have some sort of attribute to distinguish directories from files and then further, have run time checking using if statements to distinguish files from directories – not quite as elegant as the classic GOF solution, where you can just override methods defined in a base class etc.

Participants

The classes and/or objects participating in the Composite pattern are:

- Component (DrawingElement)
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf (PrimitiveElement)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite (CompositeElement)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client (CompositeApp)
  - manipulates objects in the composition through the Component interface.

Code Ideas

www.dofactory.com's structural code demonstrates the Composite pattern which allows the creation of a tree structure in which individual nodes are accessed uniformly whether they are leaf nodes or branch (composite) nodes.

www.dofactory.com's real-world code demonstrates the Composite pattern used in building a graphical tree structure made up of primitive nodes (lines, circles, etc) and composite nodes (groups of drawing elements that make up more complex elements).

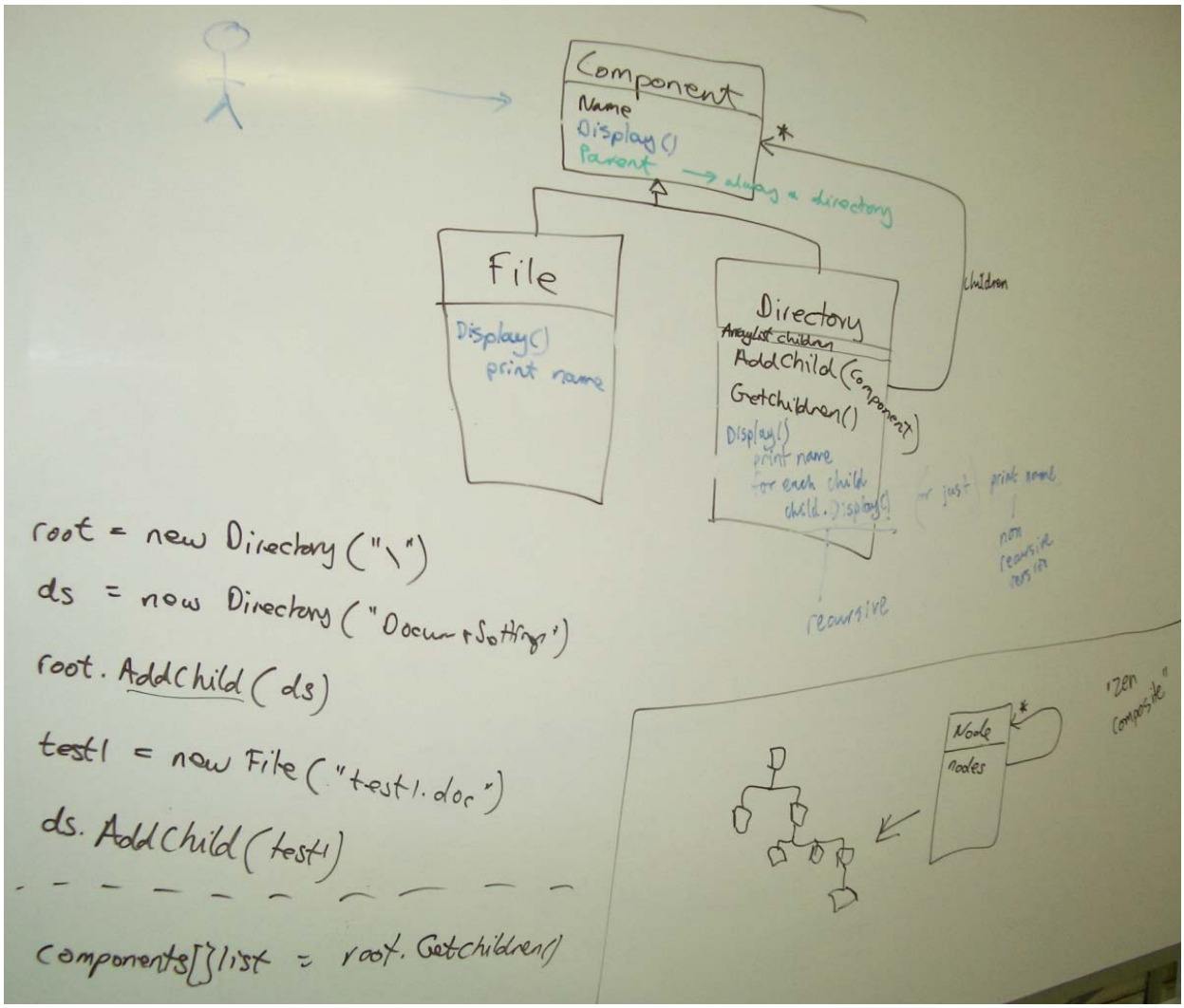


Figure 16 - Building a File System using Composite

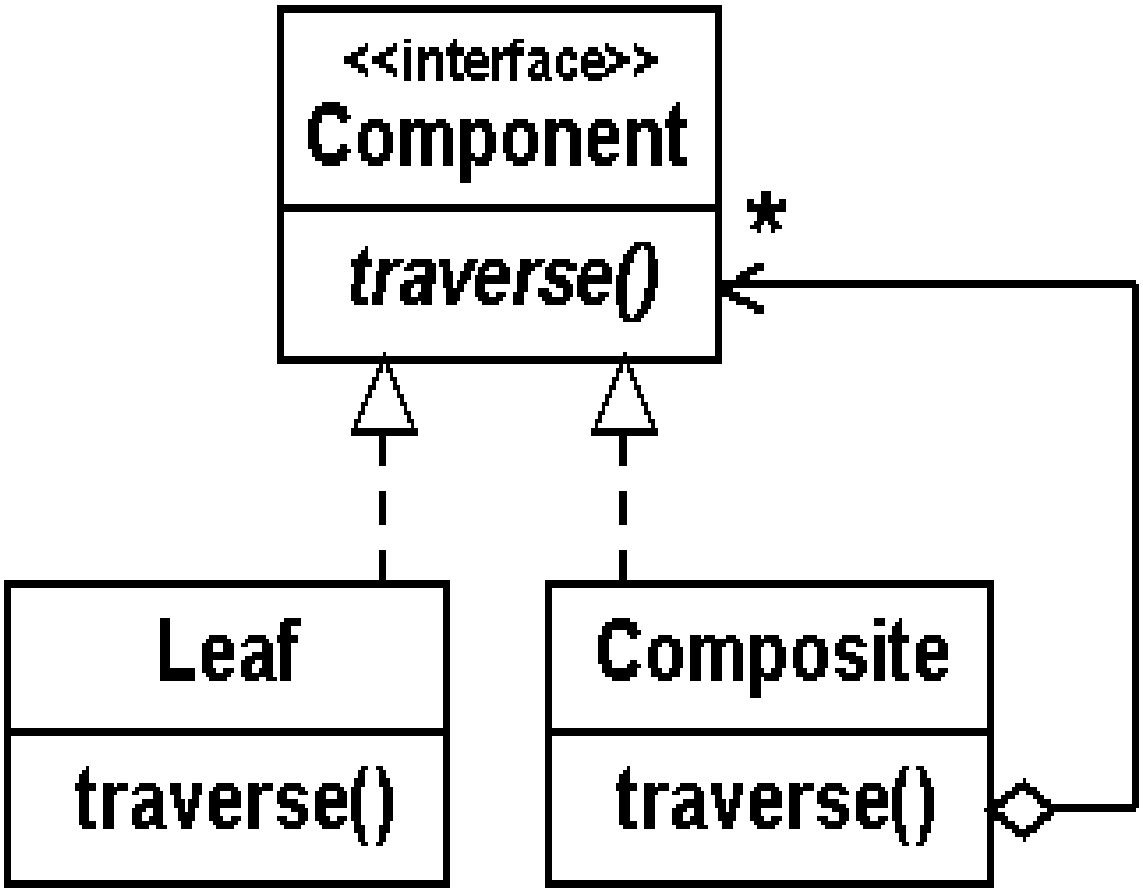
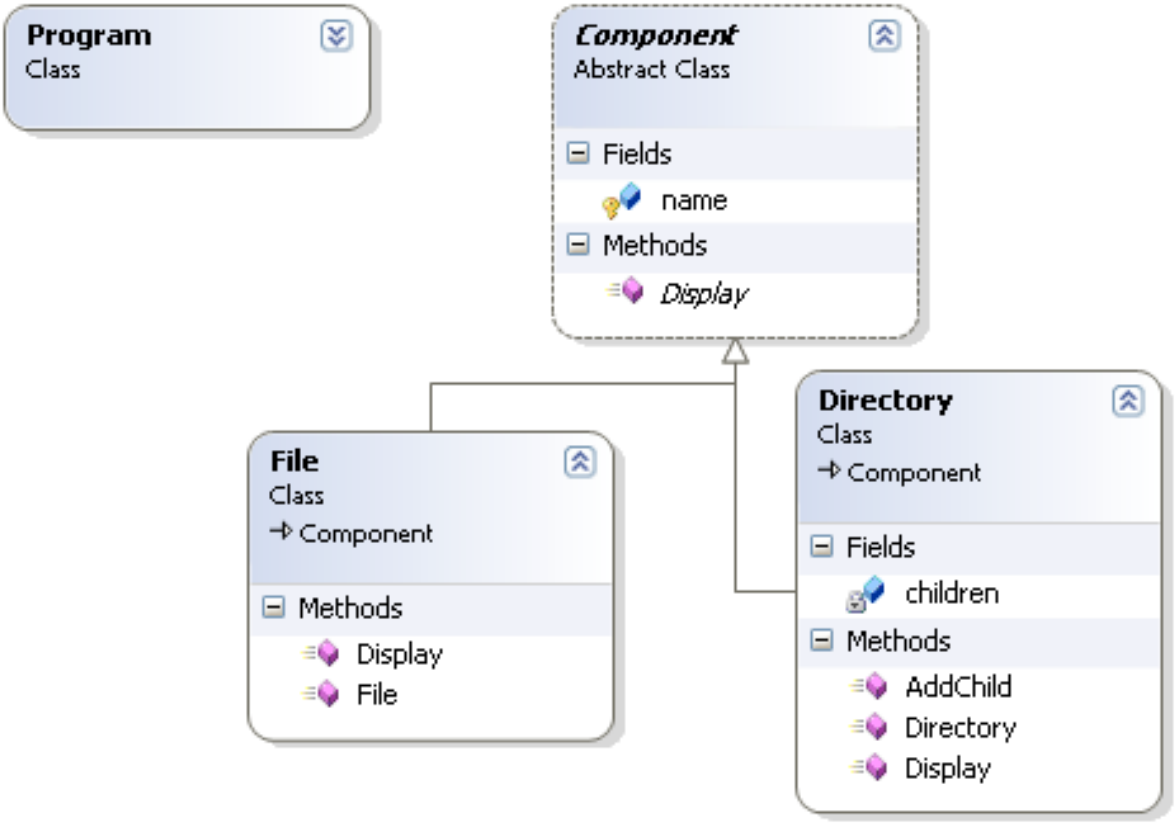


Figure 17 - UML for a simple file system

Code: Composite in Python

```
class Component:
    def __init__(self, name):
        self.name = name
    def Display(self, indent):
        print ' '*indent,
    def GetChildren(self):
        pass
    def AddChild(self, c):
        pass
class Folder(Component):
    def __init__(self, name):
        Component.__init__(self, name)
        self.children = []
    def AddChild(self,c):
        self.children.append(c)
    def Display(self, indent):
        Component.Display(self, indent)
        print self.name
        for c in self.children:
            c.Display(indent + 4)
class FSFile(Component):
    def Display(self, indent):
        Component.Display(self, indent)
        print self.name

mydocs = Folder("My Documents")
mydocs.AddChild( FSFile("todo.txt") )
mydocs.AddChild( FSFile("resume.doc") )

mypics = Folder("My Pictures")
mydocs.AddChild(mypics)

mary = FSFile("Mary.jpg")
mypics.AddChild( mary )
mypics.AddChild( FSFile("Joseph.jpg"))
mypics.AddChild( FSFile("Sam.jpg"))

mydocs.Display(0);
```

Code: Composite in C#



Component.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PerthComposite2006
{
    public abstract class Component
    {
        protected String name;

        public abstract void Display(int indent);
    }
}
```

Directory.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace PerthComposite2006
{
    public class Directory : Component
    {
        private ArrayList children = new ArrayList();

        public Directory(string name)
        {
            this.name = name;
        }

        public void AddChild(Component child)
        {
            children.Add(child);
        }

        public override void Display(int indent)
        {
            for (int i = 0; i < indent; i++)
                Console.Write(" ");
            Console.WriteLine(this.name);
            foreach (Component child in this.children)
            {
                child.Display(indent + 2);
            }
        }
    }
}
```

File.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PerthComposite2006
{
    public class File : Component
    {
        public File(string name)
        {
            this.name = name;
        }

        public override void Display(int indent)
        {
            for (int i = 0; i < indent; i++)
                Console.Write(" ");
```

```
            Console.WriteLine(this.name);
        }
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PerthComposite2006
{
    class Program
    {
        static void Main(string[] args)
        {
            Directory root = new Directory("\\");
            Directory ds = new Directory("Documents and Settings");
            Directory a = new Directory("Andy");
            Directory md = new Directory("My Documents");
            Directory mp = new Directory("My Pictures");
            root.AddChild(ds);
            ds.AddChild(a);
            a.AddChild(md);
            a.AddChild(mp);

            mp.AddChild(new File("sunset.jpg"));
            mp.AddChild(new File("ocean.jpg"));
            mp.AddChild(new File("mountains.jpg"));

            md.AddChild(new File("resume.doc"));

            root.Display(0);

            Console.ReadLine();
        }
    }
}
```

OUTPUT:

```
\
  Documents and Settings
    Andy
      My Documents
        resume.doc
      My Pictures
        sunset.jpg
        ocean.jpg
        mountains.jpg
```

Code: File System Example In Java

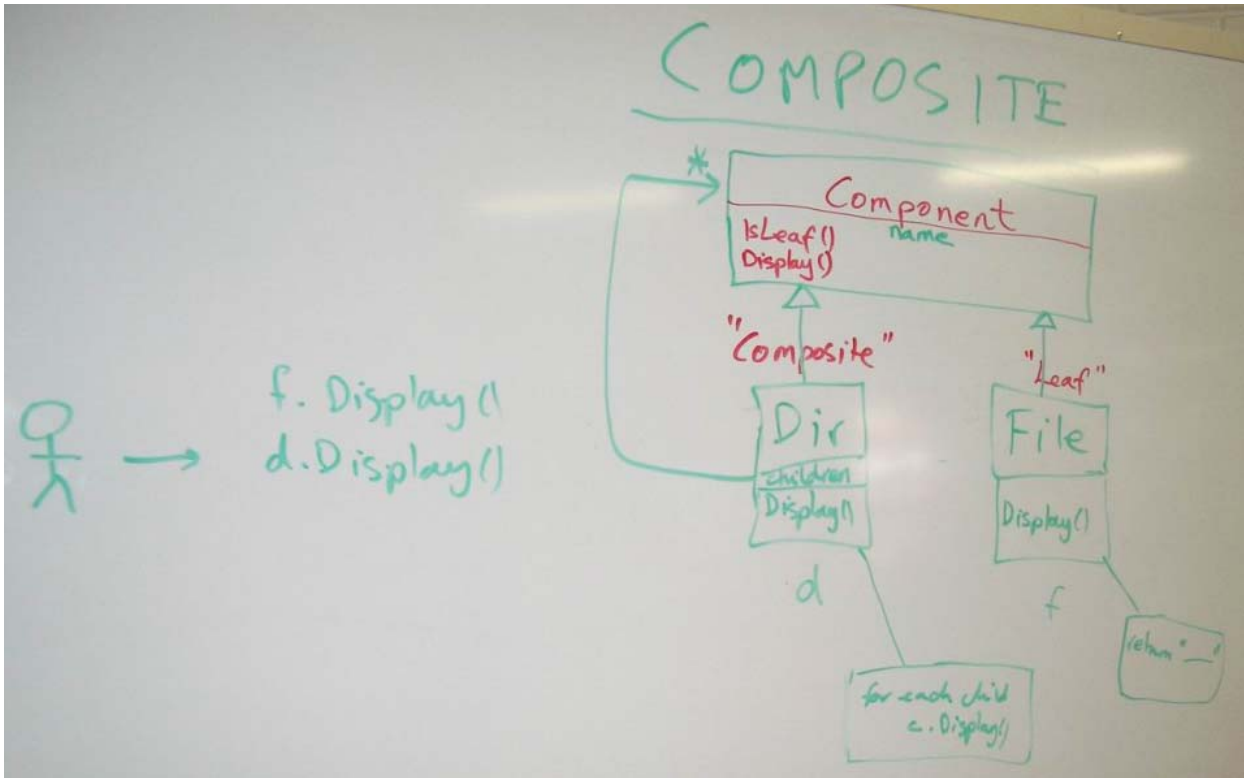


Figure 18 - File System Example In Java

```
import java.util.*;

public class Folder  extends FSComponent {
      private List children = new ArrayList();

      public Folder (String name) {
            super(name);
      }

      public void Display(int indent) {
            super.Display(indent);
            System.out.println(this.getName());
            for (Iterator i = this.GetChildren().iterator(); i.hasNext();) {
                  FSComponent c = (FSComponent) i.next();
                  c.Display(indent + 4);
            }
      }

      public void AddChild(FSComponent c) {
            this.children.add(c);
      }

      public List GetChildren() {
```

```
            return this.children;
      }

      public void RemoveChild(FSComponent c) {
            this.children.remove(c);
      }
}
```

```
import java.util.*;

public class FSComponent {
      private String name;

      public FSComponent(String name){
            this.setName(name);
      }
      public void AddChild(FSComponent c) {
      }
      public void RemoveChild(FSComponent c) {
      }

      public List GetChildren() {
            return null;
      }
      public void Display(int indent) {
            for (int i=0; i<indent; i++)
                  System.out.print(' ');
      }

      protected void setName(String name) {
            this.name = name;
      }
      protected String getName() {
            return name;
      }
}
```

```
public class FSFile extends FSComponent {
      public FSFile(String name) {
            super(name);
      }
      public void Display(int indent) {
            super.Display(indent);
            System.out.println(this.getName());
      }
}
```

```
public class MainApp {

      public static void main(String[] args) {
```

```
            Folder mydocs = new Folder("My Documents");
            mydocs.AddChild( new FSFile("todo.txt") );
            mydocs.AddChild( new FSFile("resume.doc") );

            Folder mypics = new Folder("My Pictures");
            mydocs.AddChild(mypics);

            FSFile mary = new FSFile("Mary.jpg");
            mypics.AddChild( mary );
            mypics.AddChild( new FSFile("Joseph.jpg"));
            mypics.AddChild( new FSFile("Sam.jpg"));

            mydocs.Display(0);

            /*
            mypics.RemoveChild(mary);
            mydocs.Display(0);
            */

            /*
            mydocs.RemoveChild(mypics);
            mydocs.Display(0);
            */

            Shortcut s = new Shortcut("alias to my pics", mypics);
            s.AddChild( new FSFile("Cosmo Kramer.jpg"));
            mydocs.Display(0);

            /*
             * Need better implementation of remove child
             * or better, a FSComponent.DeleteMe() method
             * so that don't need to know parent in order to delete.
             *
            Shortcut m = new Shortcut("", mary);
            mypics.RemoveChild(m);
            mydocs.Display(0);
            */
      }
}
```

```
import java.util.List;

public class Shortcut extends FSComponent {

      private FSComponent realsubject;

      public Shortcut (String name, FSComponent realsubject) {
            super(name);
            this.realsubject = realsubject;
      }

      public List GetChildren() {
            return this.realsubject.GetChildren();
      }
      public void Display(int indent) {
```

```
            this.realsubject.Display(indent);
      }
      public void AddChild(FSComponent c) {
            this.realsubject.AddChild(c);
      }
      public void RemoveChild(FSComponent c) {
            this.realsubject.RemoveChild(c);
      }
}
```

Discussion – Opinions on what Composite methods to go where

The opinions below illustrate the debate about what methods to put in the base component class vs. which to put into the subclasses.

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside. [Don Roberts]

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces. [Bill Burcham]

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use dynamic_cast to figure this out. In those cases where dynamic_cast is too expensive, I will use a Visitor. [Robert Martin]

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof. Just because Composite's structure diagram shows child management operations in the Component base class doesn't mean all implementations of the pattern must do the same. [John Vlissides]

Rules of thumb

Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes. [GOF, pp173,349]

Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Notes:

Iterator

Sequentially access the elements of a collection

Intent

Provide a way to access the elements of an aggregate object (“collection”) sequentially—without exposing the details of the underlying internal representation of the collection to the looping code.

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each consistently and transparently. E.g. one class may use an Array, another might use an ArrayList—thus the way client code would be forced to write looping code would be different in each case. It would be nice to have a consistent interface for looping through collections—a standard traversal protocol.

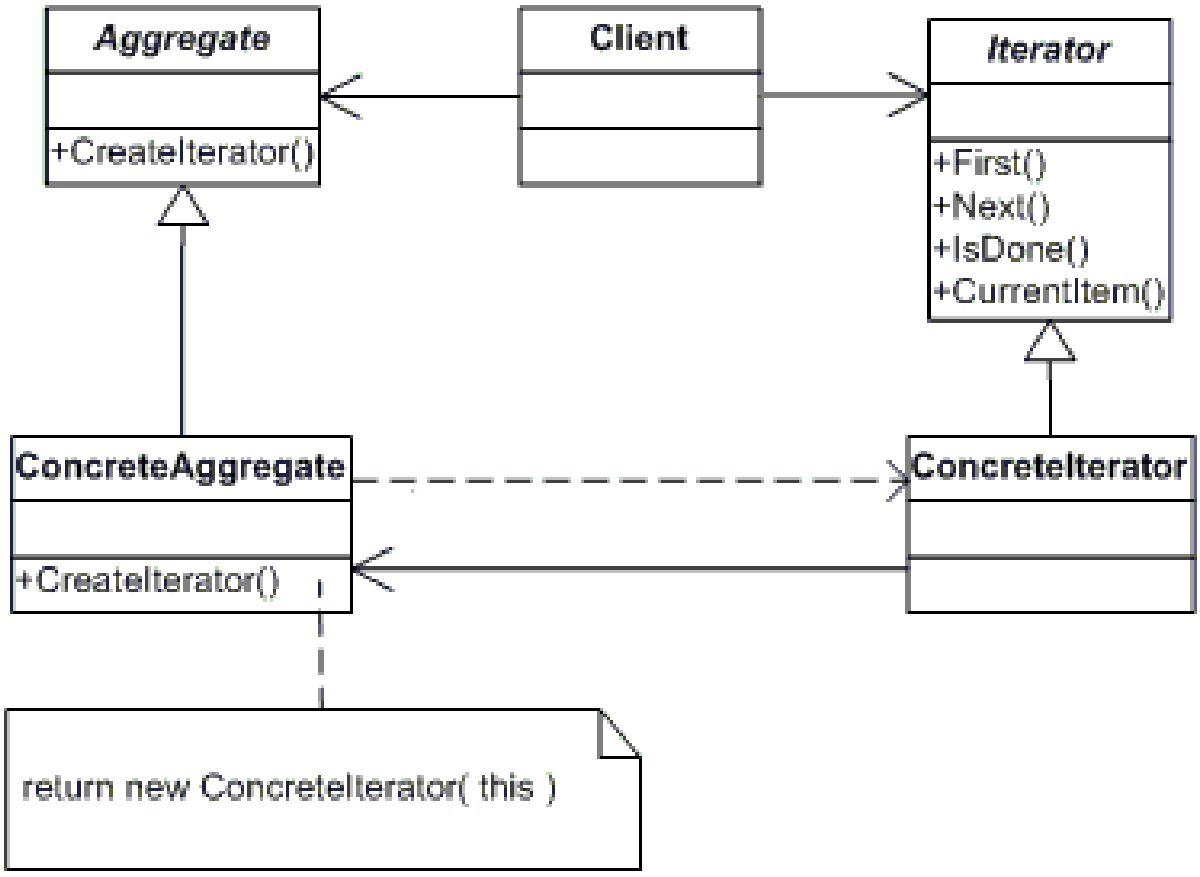
Discussion

"An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you'll require. You might also need to have more than one traversal pending on the same list." [GOF, p257] And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

Participants

The classes and/or objects participating in this pattern are:

- Iterator (AbstractIterator)
  - defines an interface for accessing and traversing elements.
- ConcreteIterator (Iterator)
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- Aggregate (AbstractCollection)
  - defines an interface for creating an Iterator object
- ConcreteAggregate (Collection)
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator


Aggregate
+CreateIterator()
Client
Iterator
+First()
+Next()
+IsDone()
+CurrentItem()
ConcreteAggregate
+CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Andy's Tips

- Iterators allow you to have multiple traversals happening at the same time because the "current position" is kept in the iterator object, and you can have as many iterator objects as you like.
- The Iterator class defines the interface for accessing and traversing elements. It could use memento to capture the state of the iteration, but usually a simple property "CurrentItem" will do.
- .NET, Java and Python and other languages have built-in support for iteration through the use of enumeration interfaces e.g. IEnumerable and built in language keyword support e.g. foreach().
- You can have different styles of iterator, iterating in different ways e.g. sequentially, depth first, breadth first etc.
- You can define special "filtering" iterators that skip certain items in your collection. This might be a cool way to organize your business logic e.g. iterate over the female customers. Perhaps chain multiple iterators—like a google search within a google search (Decorator?)
- The Aggregate (or collection e.g. ArrayList) should have a consistently named factory method called something like "CreateIterator()" which returns an iterator object—so that you always know where to get the iterator from, and also so that the built in language support (e.g. foreach) also knows how to get the iterator. The Aggregate's factory method e.g. "CreateIterator()" is typically part of an official interface e.g. IEnumerable
- Internal vs. External iterators: An external iterator is where the client code does the looping, albeit via a consistent api. With an internal iterator is you pass a method to the collection (e.g. a C# delegate, or Java object with a known method to call each time, or block of code known as a "closure") and have the collection do the iteration internally, calling the code/method/delegate with each item in the collection as a parameter. Thus the iterator is never exposed to the client.
- Traversing trees via recursion is more easily done as an internal iterator because you can take advantage of recursive programming and a call stack. You simply call a user method within the recursive algorithm.
  Doing recursive iteration using an external iterator is impossible (unless your language has a 'yield' keyword) because e.g. MoveNext() returns after each item and the call stack is lost, meaning you have to traverse the tree in a more laborious, non recursive way e.g. keeping track of pointers to children and parents etc.
- GetIterator() is an example of a factory method.

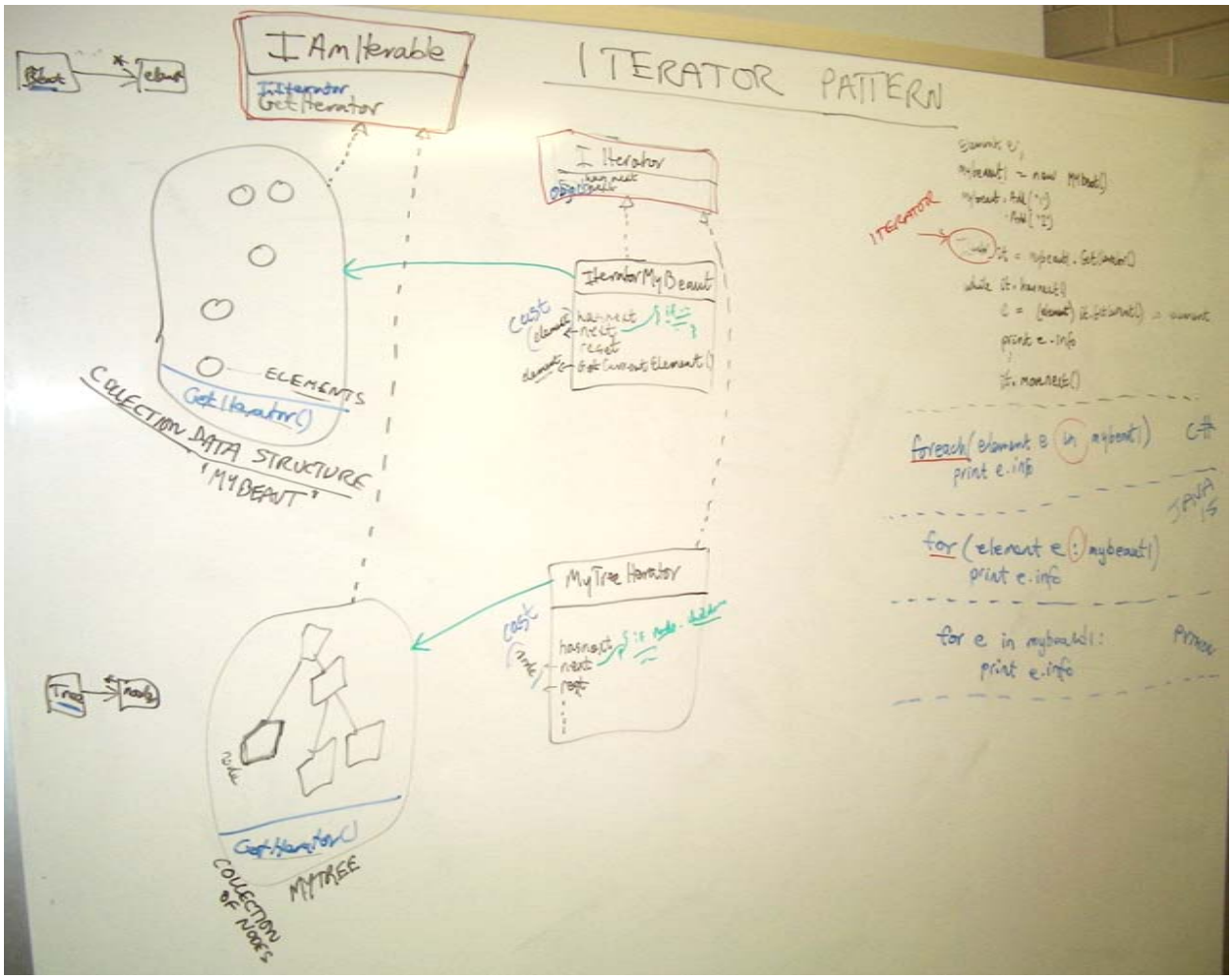


Figure 19 - Iterator pattern requires two interfaces - the collection implements one, the iterator implements the other

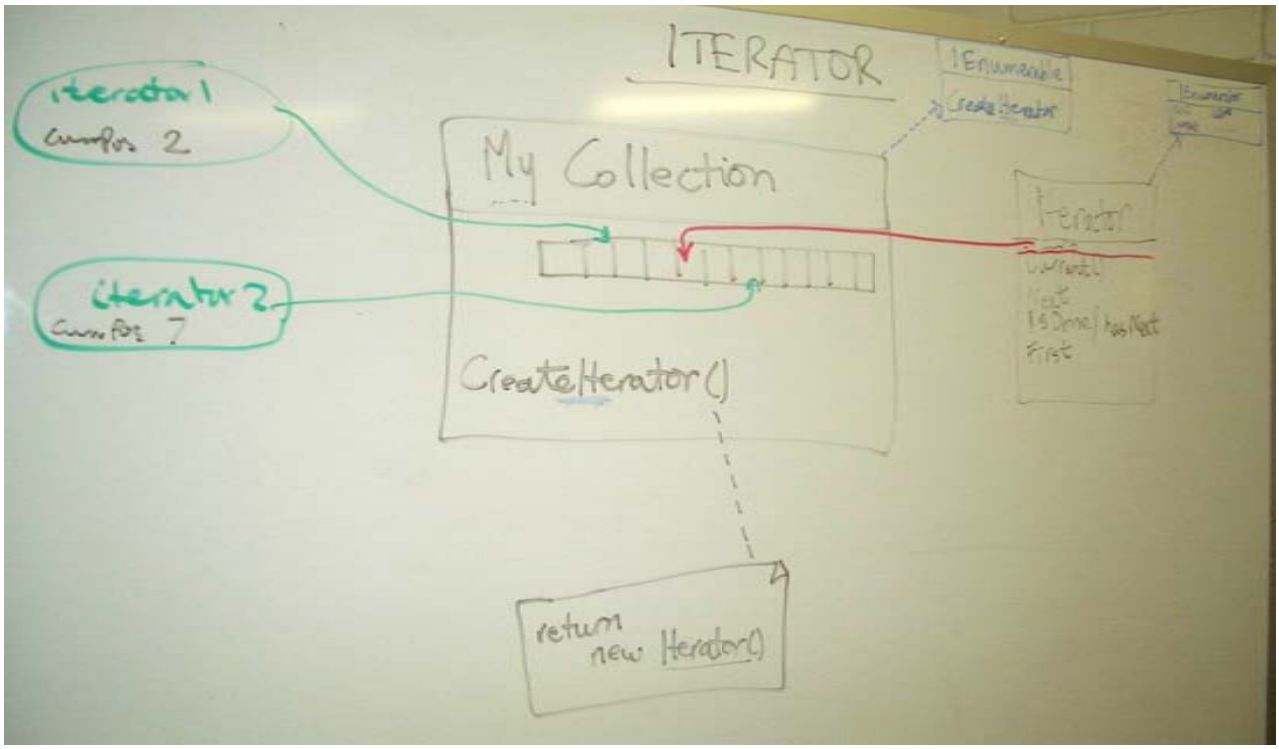


Figure 20 - Iterator Patterns supports multiple iterators through the same collection

Issues:

- Multiple Iterators through the same collection
- Changing data whilst iterating
- Range Iterator (as in python range(1,10)) iterates over a numeric sequence rather than over a “real” collection.

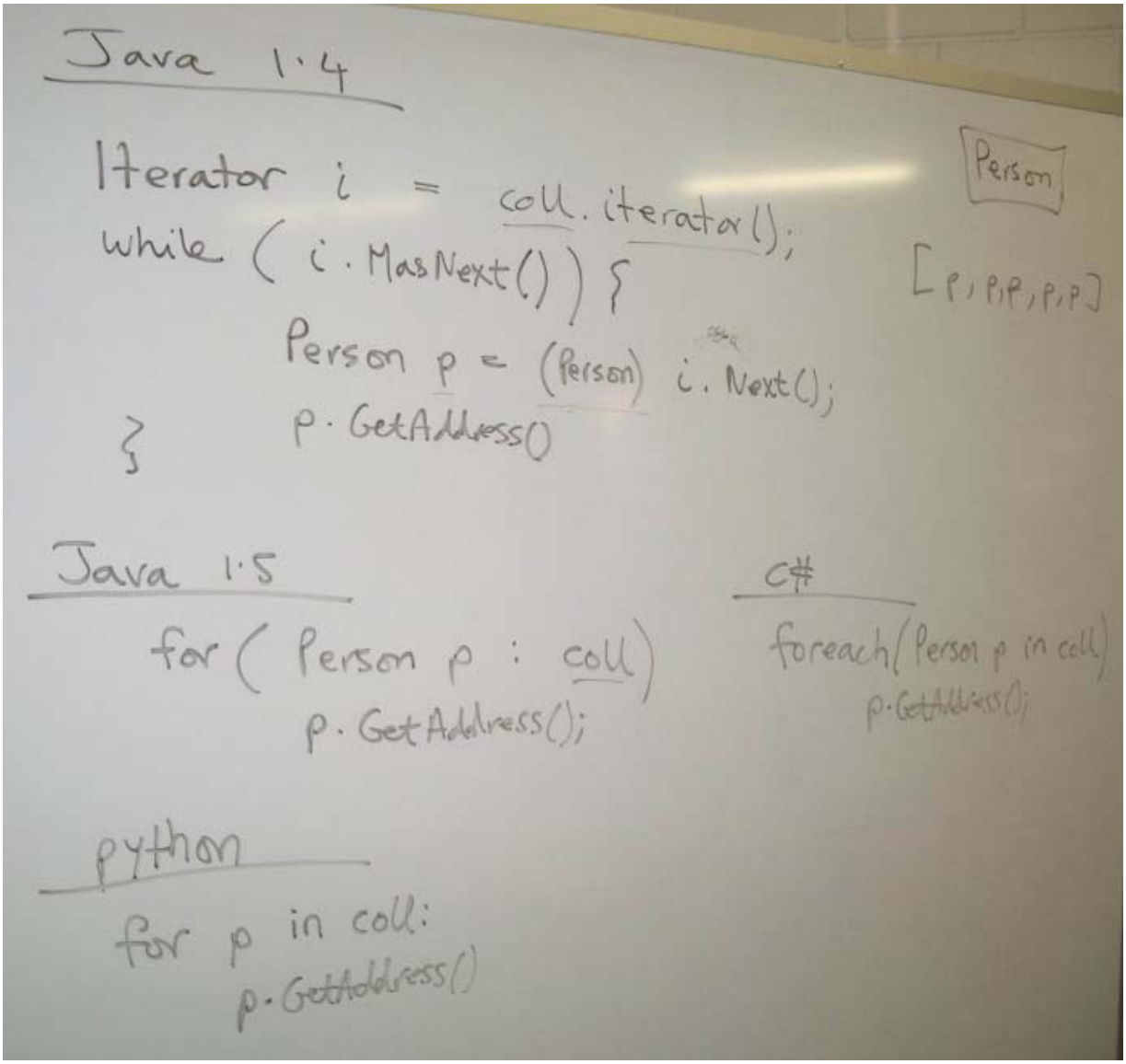


Figure 21 - Different languages have "reified" iterators into the language itself

Code Ideas

www.dofactory.com's structural code demonstrates the Iterator pattern which provides for a way to traverse (iterate) over a collection of items *without detailing the underlying structure of the collection*.

www.dofactory.com's real-world code demonstrates the Iterator pattern which is used to iterate over a collection of items and skip a specific number of items each iteration.

Code: Range Iterator

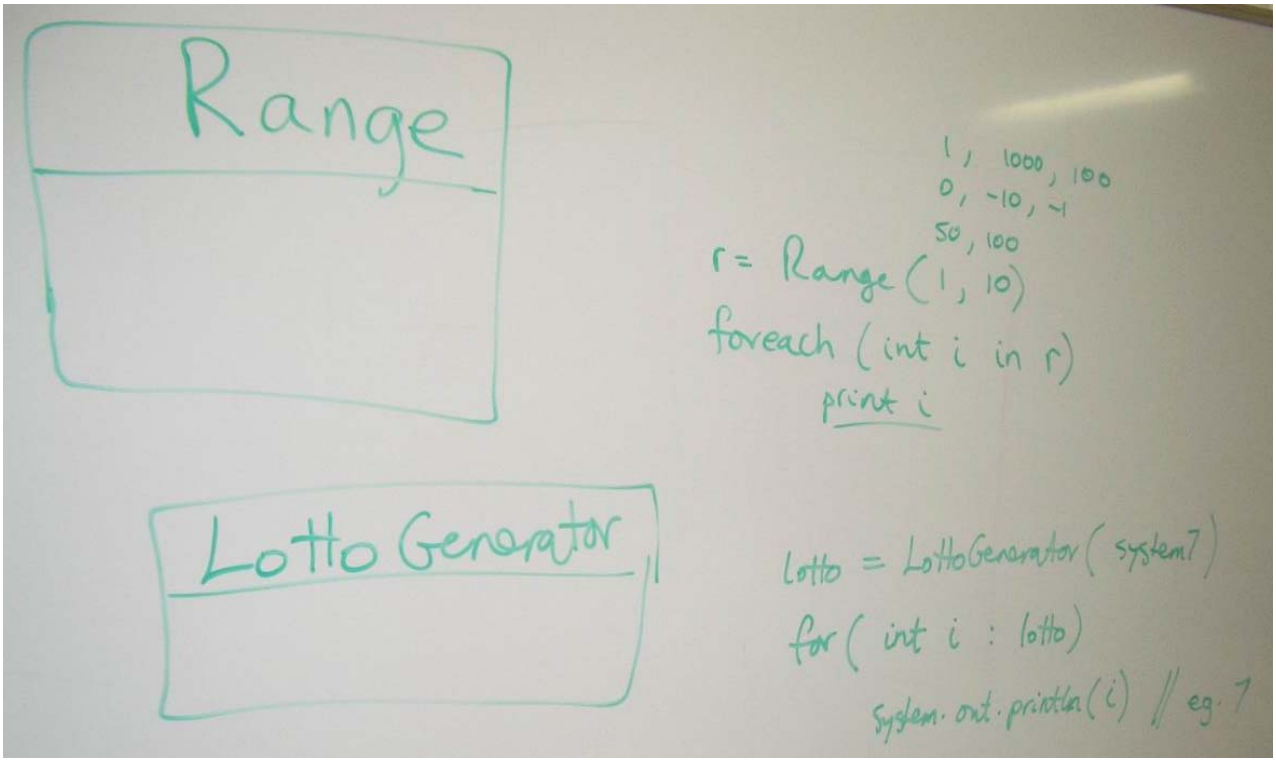
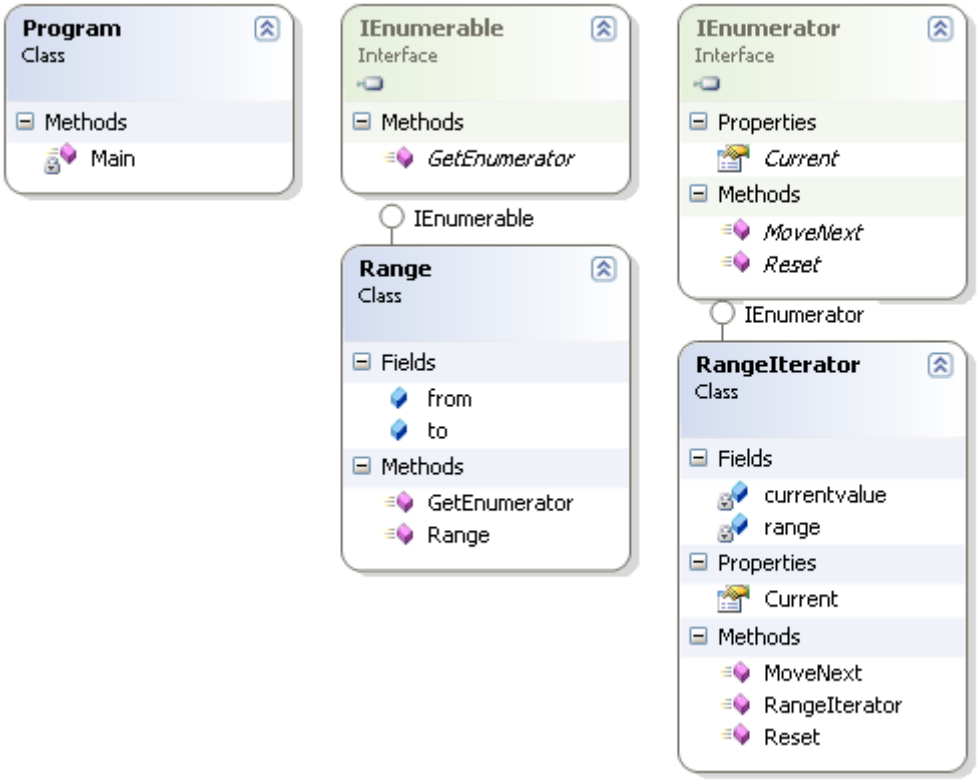


Figure 22 - A "range iterator" is a pseudo collection - elements are not real, but are usually calculated. One can forsee a Lotto Generator working in the same way, allowing the iteration over a set of lotto numbers, which are generated as you iterate.

To implement Python's range iterator feature:

```
for i in range(1,10):
    print i
```

In C#:




```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace MyRangeIterator
{
    class Program
    {
        static void Main(string[] args)
        {
            foreach(int n in new Range(1,10))
                Console.WriteLine(n);

            foreach (int n in new Range(2, 5))
                Console.WriteLine(n);

            Console.ReadLine();
        }
    }

    class Range : IEnumerable
    {
        public int from, to;

        public Range(int from, int to)
```

```
        {
            this.from = from;
            this.to = to;
        }

        public IEnumerator  GetEnumerator()
        {
              return new RangeIterator(this);
        }
    }

    class RangeIterator : IEnumerator
    {
        Range range;
        int currentvalue;

        public RangeIterator(Range range)
        {
            this.range = range;
            Reset();
        }

        #region IEnumerator Members

        public object Current
        {
            get {
                object o = currentvalue;
                return o;
            }
        }

        public bool MoveNext()
        {
            if (currentvalue < range.to) {
                currentvalue++;
                return true;
            }
            else
                return false;
        }

        public void Reset()
        {
            currentvalue = range.from-1;
        }

        #endregion
    }
}
```

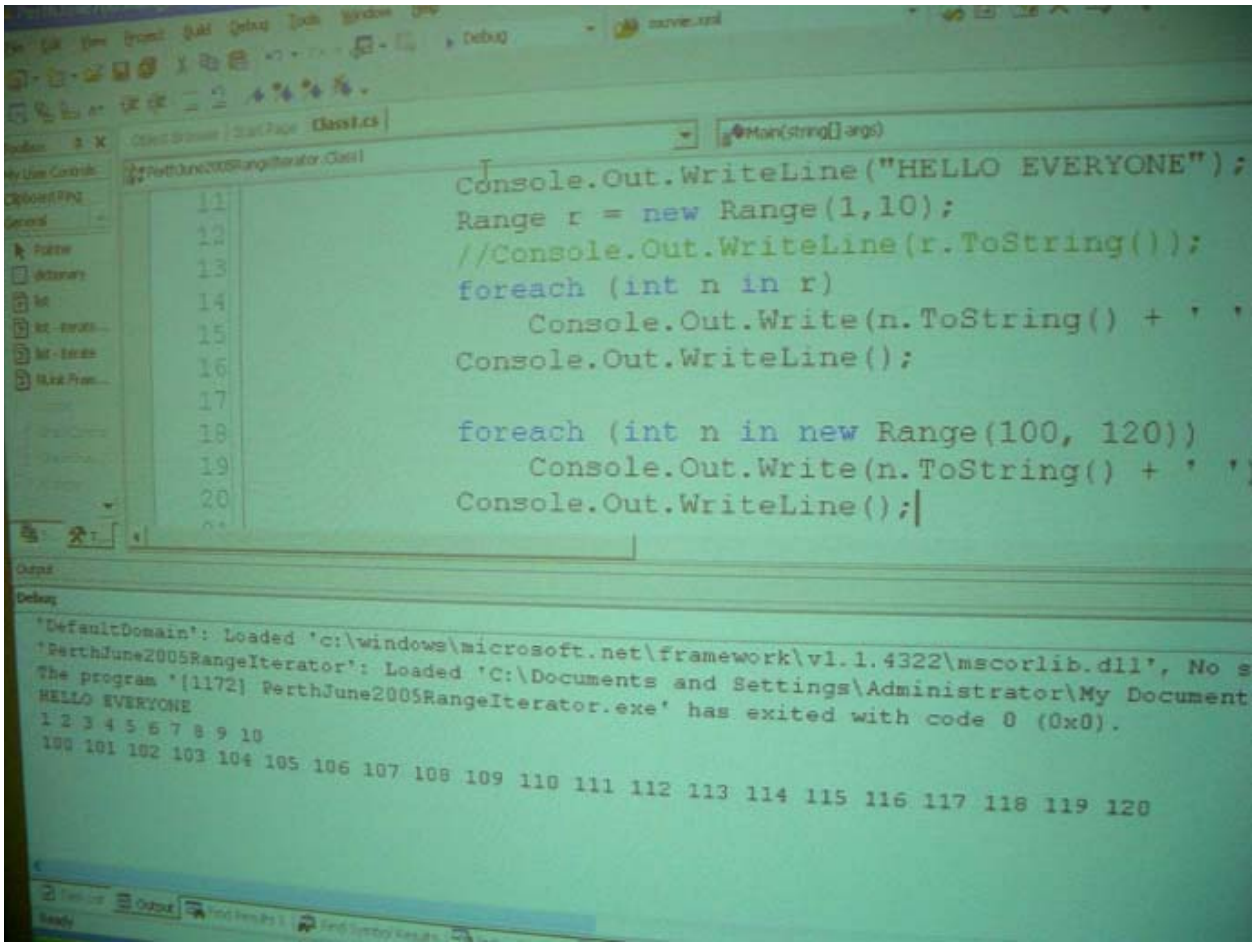


Figure 23 - C# range iterator - example output from visual studio

Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

Non Software Examples

Iterator examples included the receptionist in a doctor's waiting room. The receptionist iterates the aggregate of patients who are seated randomly around the room. The receptionist will send the "next" patient to the doctor.

The channel selector on modern day television sets is another example of an Iterator. Rather than a dial containing all possible channels or one button per tuned channel, today's television sets have a "Next" and "Previous" button for tuning channels. The "Channel Surfer" doesn't need to know if Channel 3 or Channel 4 comes after Channel 2.

These non software examples demonstrate the intent of the Iterator, since the Iterator knows how the objects are ordered.

Ideas

- The system shall support accessing an aggregate component's contents without exposing its internal representation.
- The system shall support multiple simultaneous traversals of aggregate components without complicating the implementation of the aggregate itself.
- The system shall define a uniform interface for traversing dissimilar aggregate components.
- The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.

Notes:

Template Method

Define a master algorithm which calls abstract methods as steps. Defer the exact steps of an algorithm to a subclass.

Intent

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Problem

Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Discussion

The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

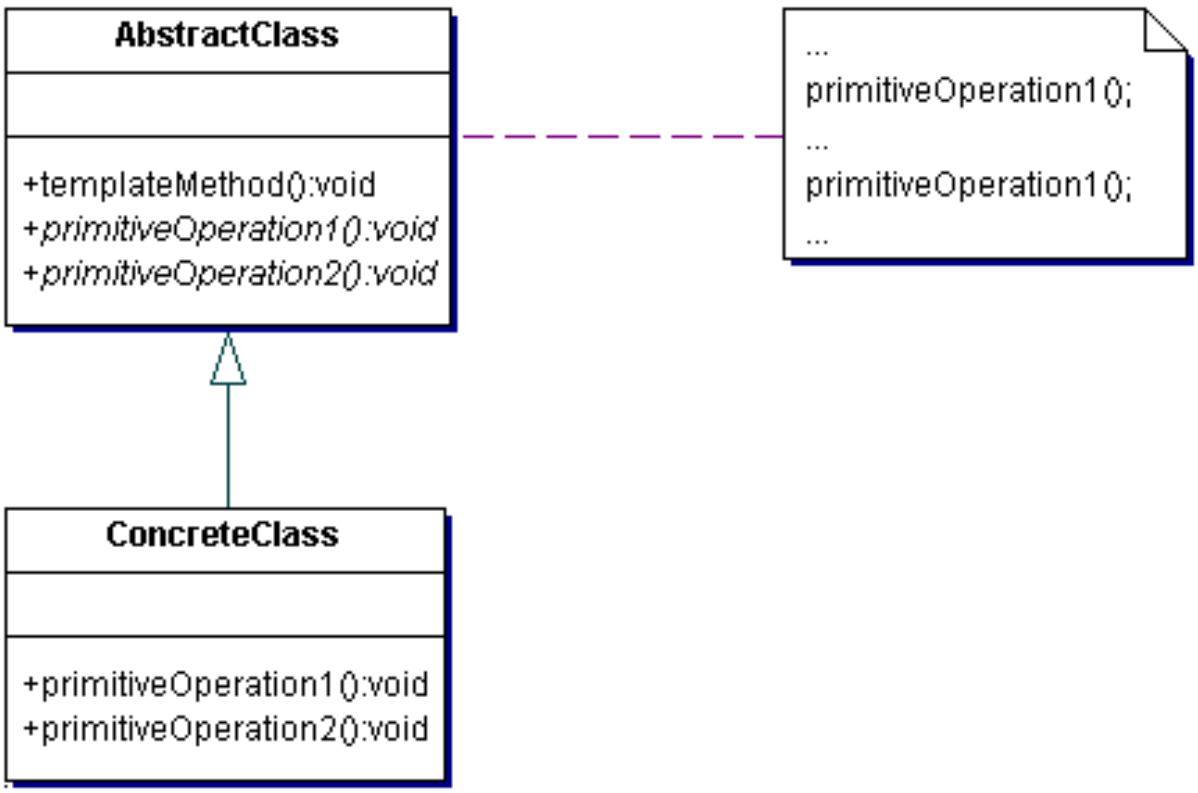
This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Participants

The classes and/or objects participating in this pattern are:

- AbstractClass (DataObject)
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  - implements a **template method** defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- ConcreteClass (CustomerDataObject)
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm


AbstractClass
+templateMethod():void
+primitiveOperation1():void
+primitiveOperation2():void
...
primitiveOperation1();
...
primitiveOperation1();
...
ConcreteClass
+primitiveOperation1():void
+primitiveOperation2():void

Ideas

- Individual steps of an algorithm shall be definable or replaceable.
- The system shall provide "hooks" wherever future functionality or extensibility may be required.
- The "invariant" parts of an algorithm shall be implemented in one place, and reused all other places.
- The "standard" parts of an algorithm shall be implemented and enforced in one place.
- Reuse shall be emphasised by architecting a "don't call us, we'll call you" framework.
- The system shall be "open for extension, but closed for modification".

Code Ideas

www.dofactory.com's real-world code demonstrates a Template method named Run() which provides a skeleton calling sequence of methods. Implementation of these steps are deferred to the CustomerDataObject subclass which implements the Connect, Select, Process, and Disconnect methods.

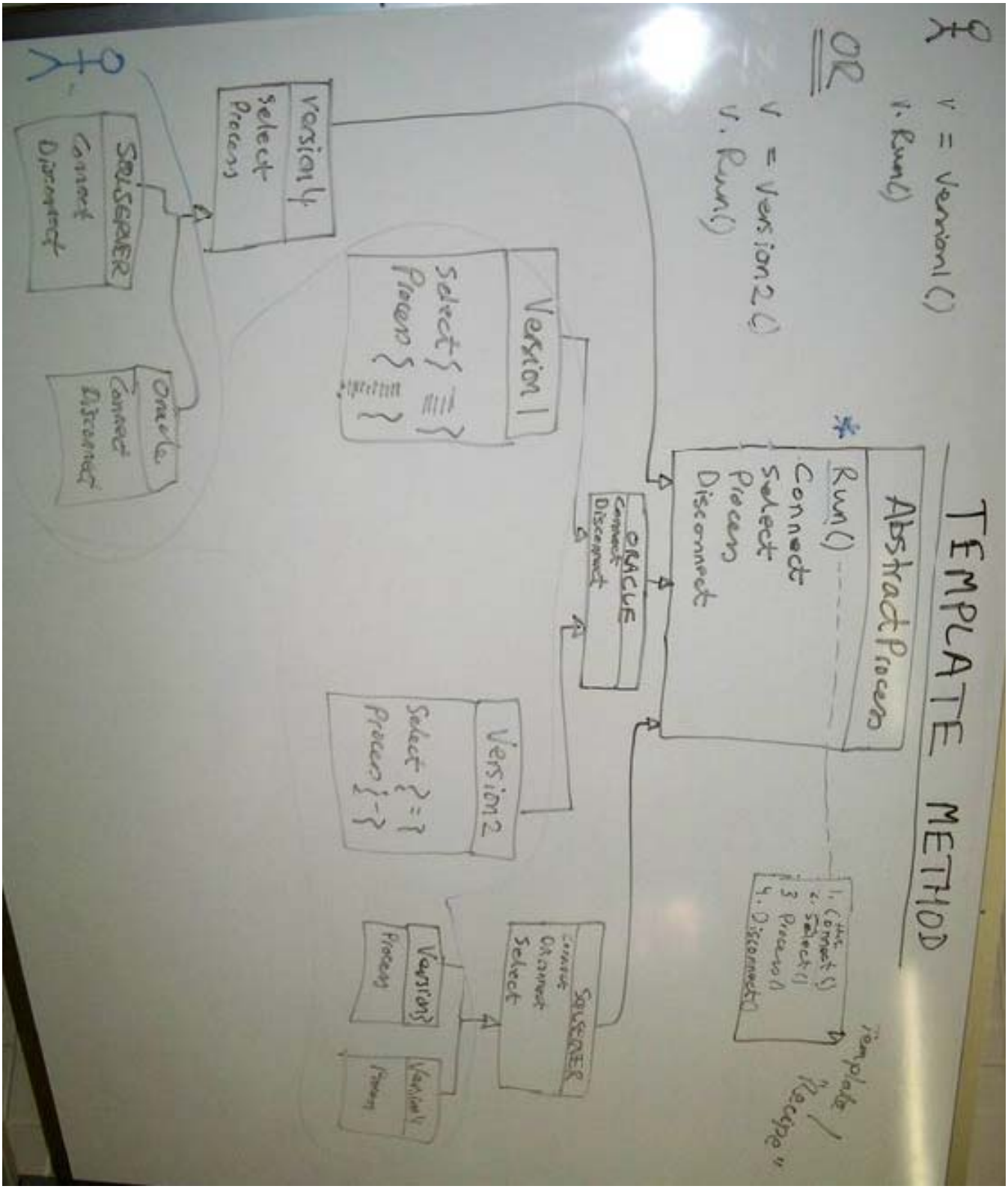


Figure 24 - Connecting to a database might usually have the same steps. Connecting to different databases will involve overriding the "connect" method.

Java Code Example:

Each game (Monopoly, Chess, Checkers) has the same basic steps.

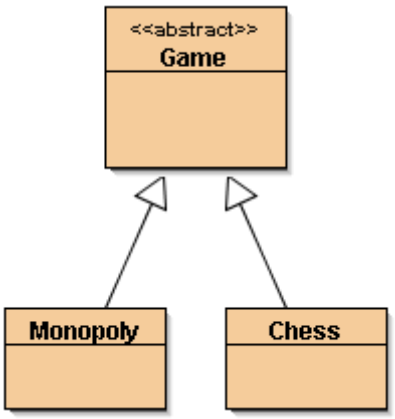


Figure 25 - UML for this example of template method pattern. The actual template "method" is Game.playOneGame()

```
/**
  * An abstract class that is common to several games in
  * which players play against the others, but only one is
  * playing at a given time.
  */

 abstract class Game {

     private int playersCount;

     abstract void initializeGame();

     abstract void makePlay(int player);

     abstract boolean endOfGame();

     abstract void printWinner();

     /* A template method : */
     final void playOneGame(int playersCount) {
         this.playersCount = playersCount;
         initializeGame();
         int j = 0;
         while (!endOfGame()){
             makePlay(j);
             j = (j + 1) % playersCount;
         }
         printWinner();
     }
 }
```

Now we can extend **this class** in order to implement actual games:

```
class Monopoly extends Game {

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ...
        return true;
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the Monopoly game. */
    // ...
}

class Chess extends Game {
    int turns = 0;

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ... hardwire 5 turns, just for this example.
        if (turns == 5)
            return true;
        else {
            turns++;
            return false;
       }
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the chess game. */
     // ...
}
```

Non Software Example

The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the later stages of construction to produce a wider variety of models. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Strategy vs. Template Method and Composition vs. Inheritance

You can compare strategy and template method pattern since both are techniques of varying algorithms or steps of an algorithm.

However things get interesting (and arguably harder to classify which pattern you are using) when you implement template method with a strategy like approach. The interesting points are:

- Granularity (overriding the entire algorithm or overriding parts of the algorithm)
- Composition vs. Inheritance (instead of overriding, delegate to a strategy)

Rules of thumb

Strategy is like Template Method except in its granularity. [Coplien, C++ Report, Mar 96, p88]

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]

Composition vs. Inheritance

Instead of overriding the steps with inheritance, you could use composition and delegate the steps. By injecting the “strategy” steps you might have greater freedom to unit test each step independently of the overall template method hierarchy.

Is this still Template Method if you are using strategy not inheritance? GOF seems to suggest that inheritance is fundamental to the template method (see “Rules of Thumb, above). Arguably the “recipe” of steps is the distinguishing feature of template method – whether that gets implemented by overriding steps or delegating those steps.

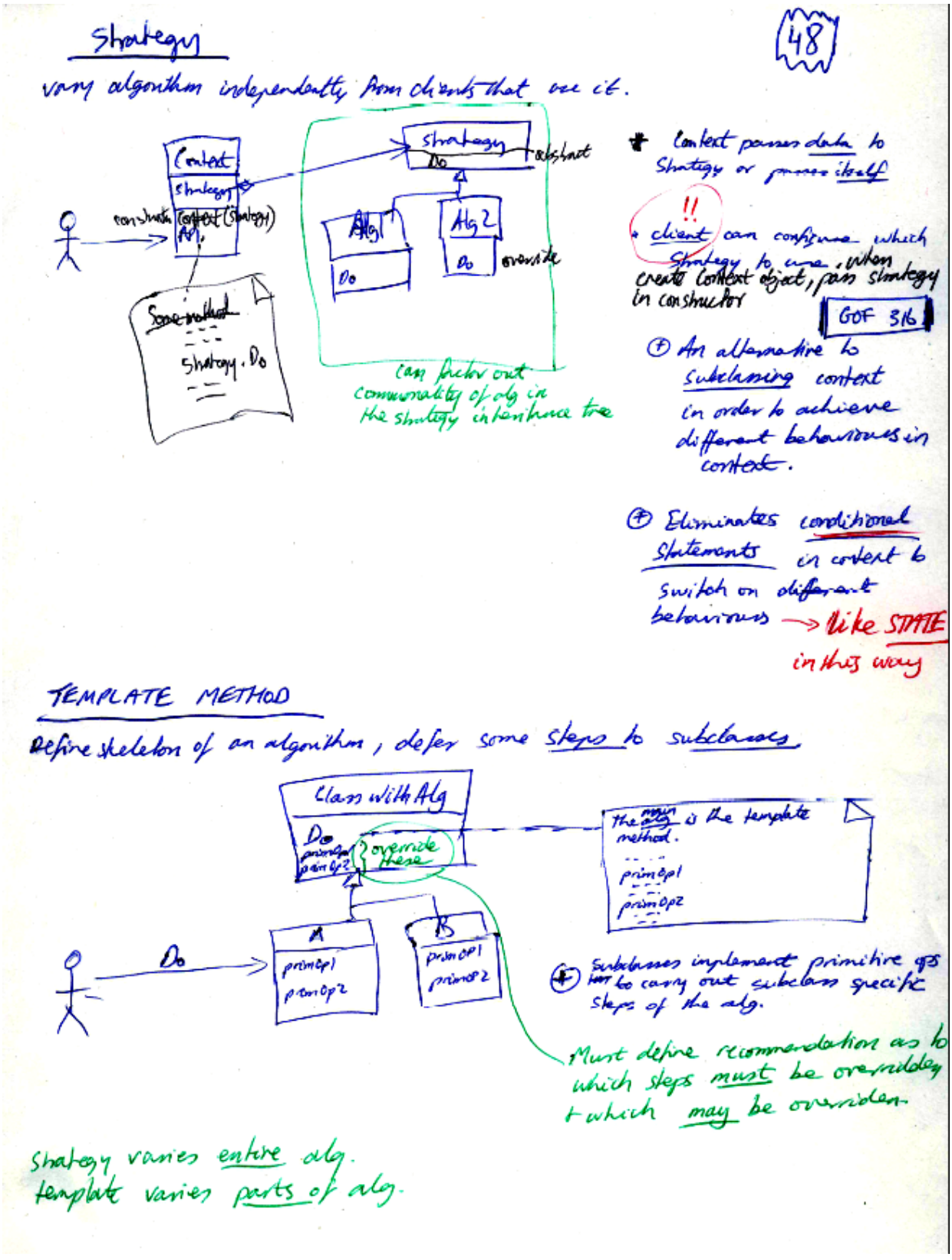


Figure 26 - Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]. Arguably you can also implement template method using delegation to a bunch of strategy methods.

Refactoring to Template Method

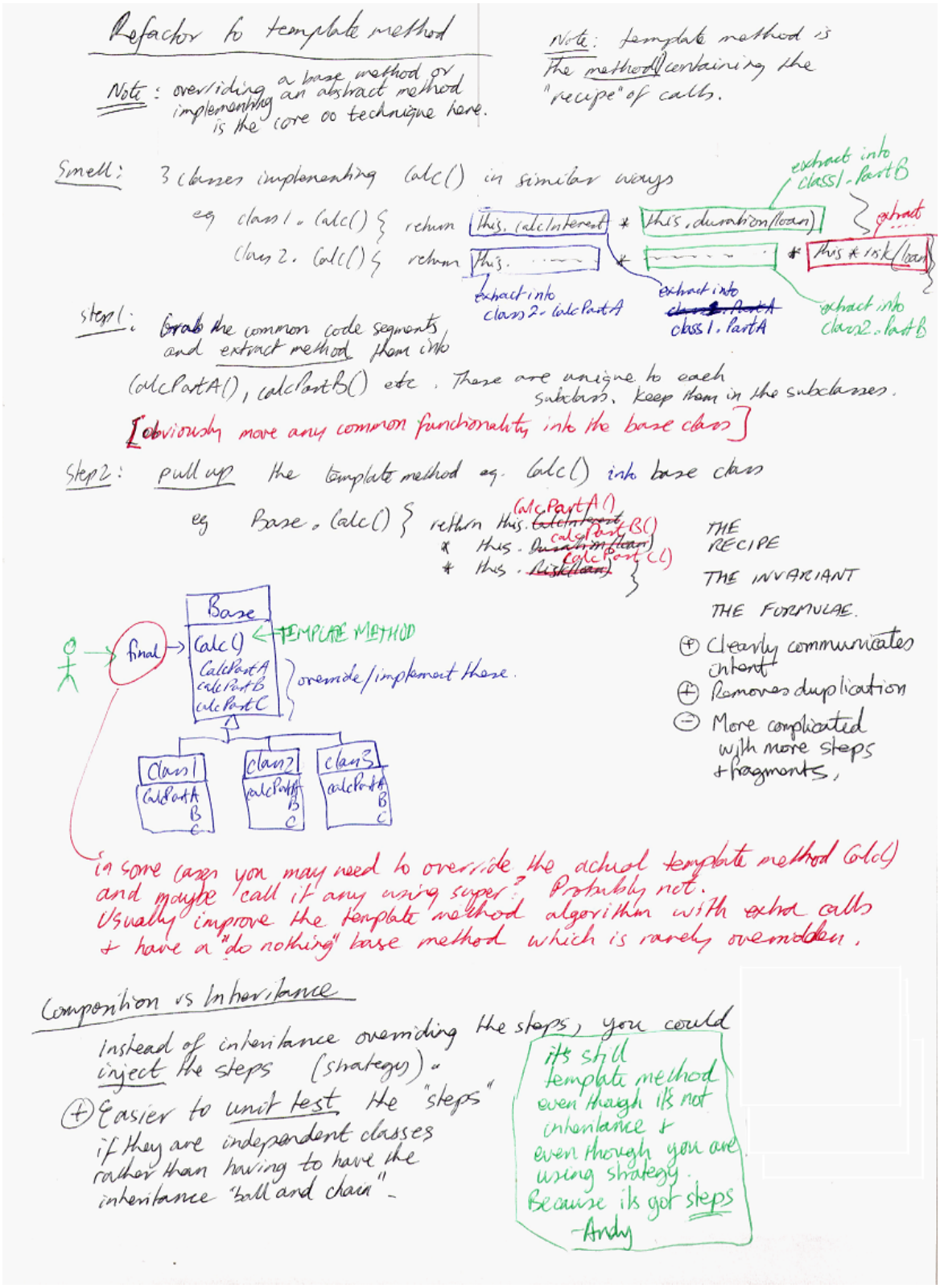


Figure 27 - Notes taken by Andy from a recent Melbourne Patterns Group session on Refactoring to Template Method.

Refactoring to Template Method - Summary

See the “Refactoring to Patterns” book by Joshua Kerievsky; Addison Wesley, 2004, ISBN 0321213351 for more detailed steps (or see Figure 27) for more info on this process.

The basic idea is to identify common code steps – by which I mean similar code - and group each such code block into an abstract “step”. The steps don’t have to contain exactly the same code, as the step is actually implemented differently by each subclass. The important thing is to identify the abstract steps as ideas.

This is arguably a great way to organize your code. Its not eliminating “code duplication” but its identifying “idea duplication” ☺

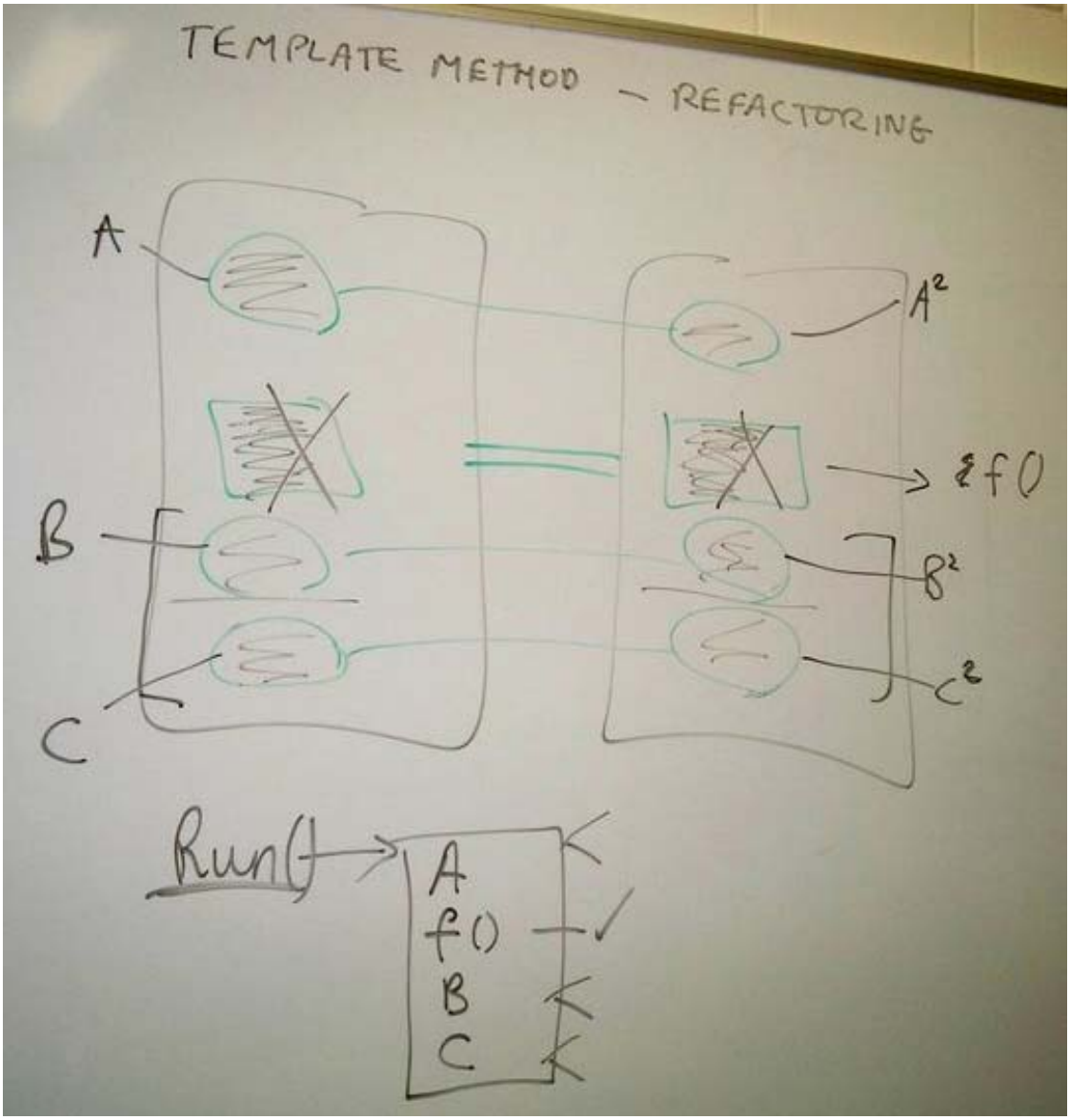


Figure 28 - Refactoring to template method. Start with a couple of classes with similar functionality and extract each common (but slightly differently implemented) step into the method containing the "recipe" of steps. Here f() is exactly the same code in all.

Notes:

Abstract Factory

Allows you to create an instance of a factory which itself can create a family of classes.

Intent

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

Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

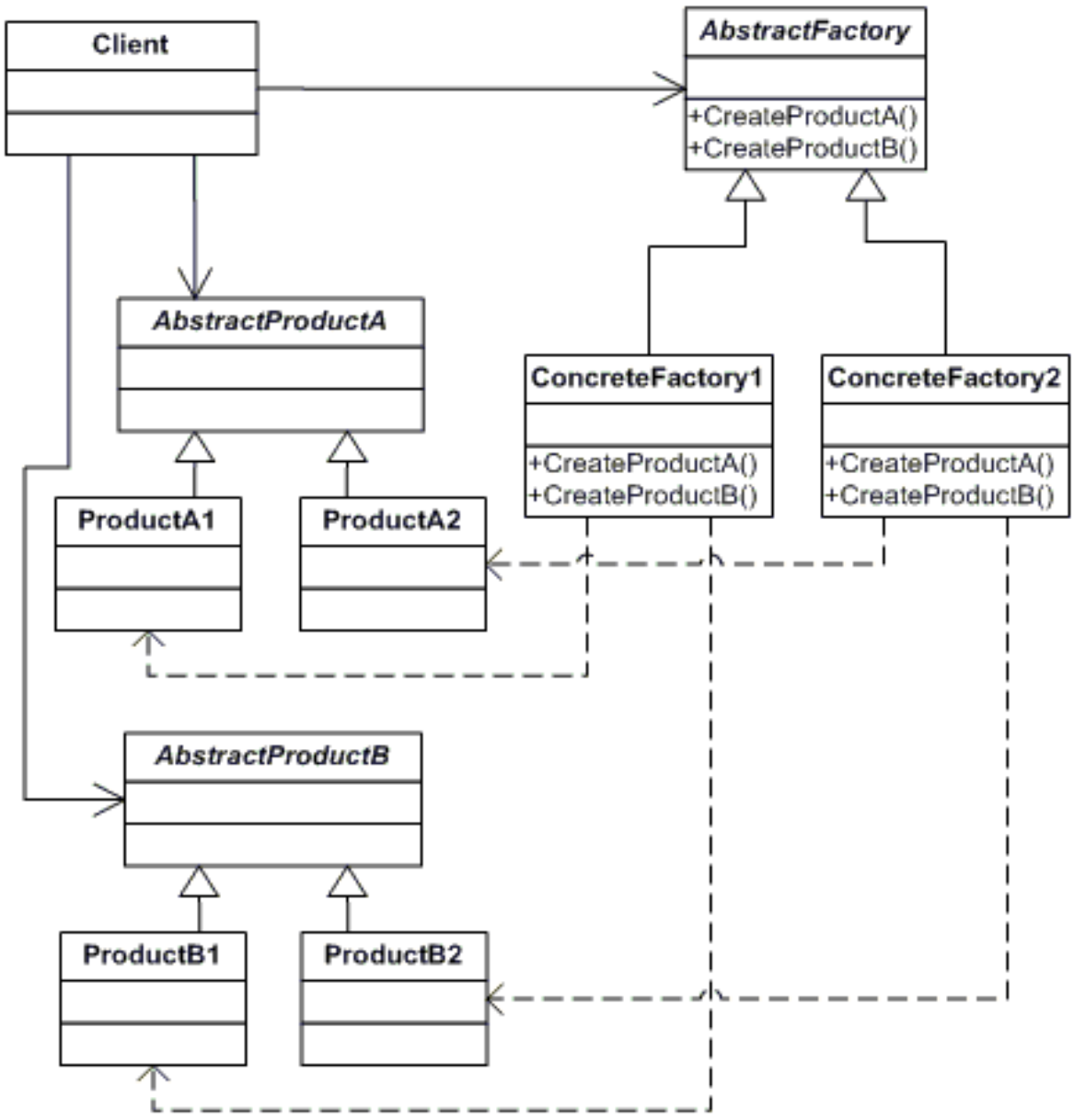
Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

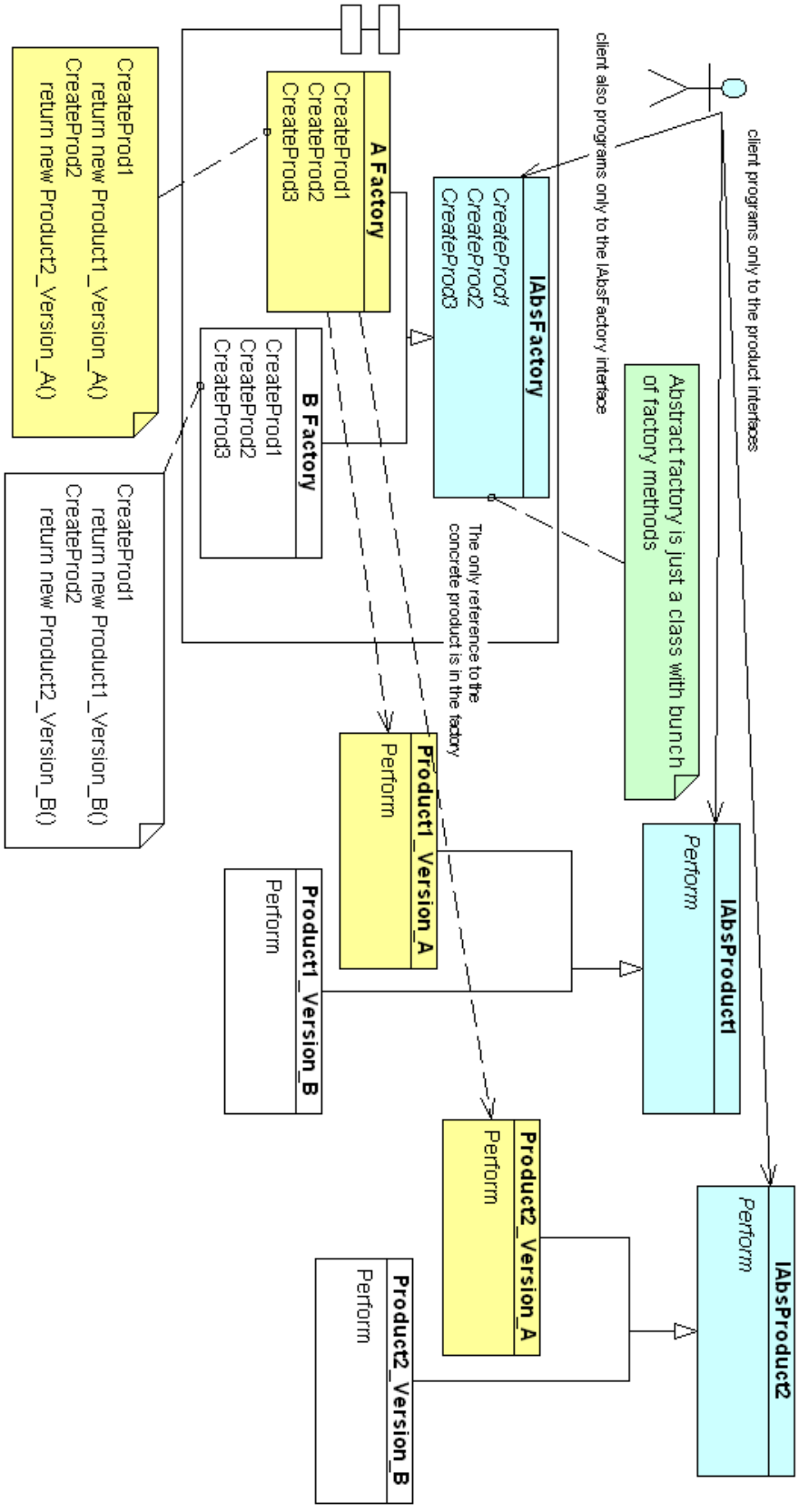
Benefit

Adding a new family of concrete types is done by modifying the client code to use a different factory, a modification which is typically one line in one file. (The different factory then creates objects of a different concrete type, but still returns a pointer of the same abstract type as before - thus insulating the client code from change.) This is significantly easier than modifying the client code to instantiate a new type, which would require changing every location in the code where a new object is created (as well as making sure that all such code locations also have knowledge of the new concrete type, by including for instance a concrete class header file).

Drawback

Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, incurs the risk of unnecessary complexity and extra work in the initial writing of code.


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2


Abstract Factory
client programs only to the product interfaces
client also programs only to the IAbsFactory interface
Abstract factory is just a class with bunch of factory methods
IAbsProduct1
Perform
IAbsProduct2
Perform
IAbsFactory
CreateProd1
CreateProd2
CreateProd3
A Factory
CreateProd1
CreateProd2
CreateProd3
B Factory
CreateProd1
CreateProd2
CreateProd3
The only reference to the concrete product is in the factory
Product1_Version_A
Perform
Product1_Version_B
Perform
Product2_Version_A
Perform
Product2_Version_B
Perform
CreateProd1
return new Product1_Version_A()
CreateProd2
return new Product2_Version_A()
CreateProd1
return new Product1_Version_B()
CreateProd2
return new Product2_Version_B()

Example – Hey I finally found a use for Abstract Factory !!

I was working on a computer game and had a problem that could only be solved by Abstract Factory. I thought I would never need this pattern – but it was a life saver! There is nothing as good as having a real need filled by a pattern – some patterns can feel a bit academic and unreal – so having a real practical application makes all the difference.

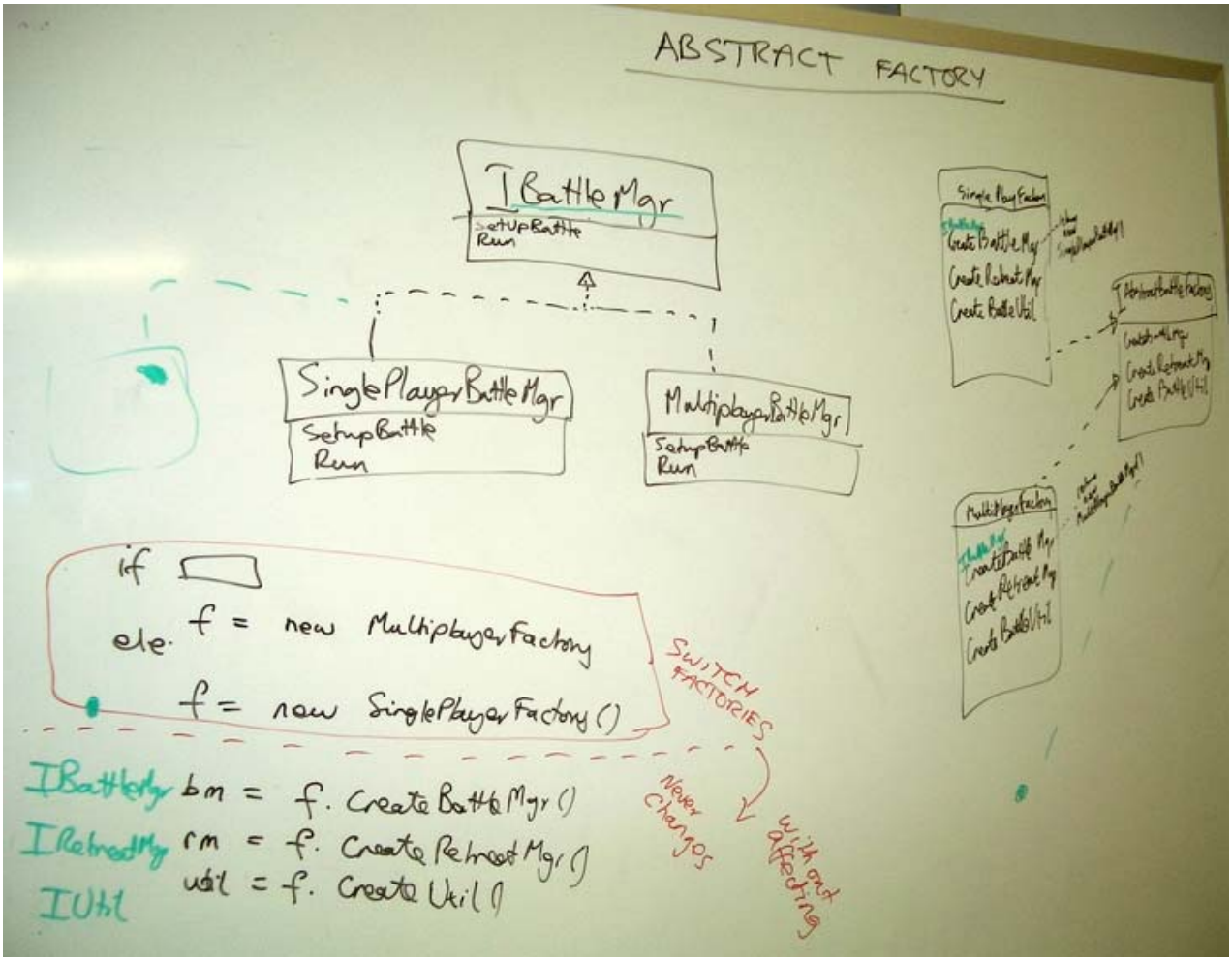


Figure 29 - Using Abstract Factory when Andy was building a computer game called "Combat Campaign". A different family of classes was required when using real battles vs. quick battles.

Andy's tips on Abstract Factory

- Abstract factory similar to factory method, in that there is something being overridden.
- Abstract factory is the same as factory method, except there is more than one Creation method. E.g. CreateWorker, CreateAdministrator, CreatePoliceman - the "concrete factory" is the class containing the factory methods dispensing these related classes.
- The abstract factory is a mere mechanism for delivering a "concrete factory " which generates "A versions" or "B versions" of a family of classes. E.g. Client wants the "windows gui" family of products/gui widgets so he needs a "windows gui factory" which has methods for creating a windows form, a windows combobox, a windows button etc. which all need to work together. Another time the client wants the "mac osx" family of products/gui widgets.
- You can add a new a new Concrete Factory (and related product classes) and voila, you can have your client code seamlessly driving a totally different implementation (product family).
- You can pull off this trick only if you follow the rule that the client code only talks to the AbstractProduct interface and to the product interfaces. Low coupling.

Client code doesn't need to change

Client code only talks to interfaces

The essential trick from the client code point of view is that the client programs against **interfaces** thus can seamlessly switch between the A or B versions of classes. Specifically, the client only talks to

IAbstractProductFactory
IProduct1
IProduct2
IProduct3
etc.

```
// Create the master factory "f" either at compile time or
// somehow at runtime by an if statement looking up some environment variable
// or even (ironically) via some other factory

IAbstractProductFactory f = new ProductFactoryVersionA()

// Client code can now create the family of related classes they need...
// via factory methods

IProduct1 p1 = f.CreateProduct1()
IProduct2 p2 = f.CreateProduct2()
IProduct3 p3 = f.CreateProduct3()

// and start using them...

p1.DoSomething();
p2.DrawLine(1,1,50,50);
```

All products p1, p2, p3 are in the above example happen to be A versions (not B versions), and compatible with each other.

Example – Client code doesn't need to change

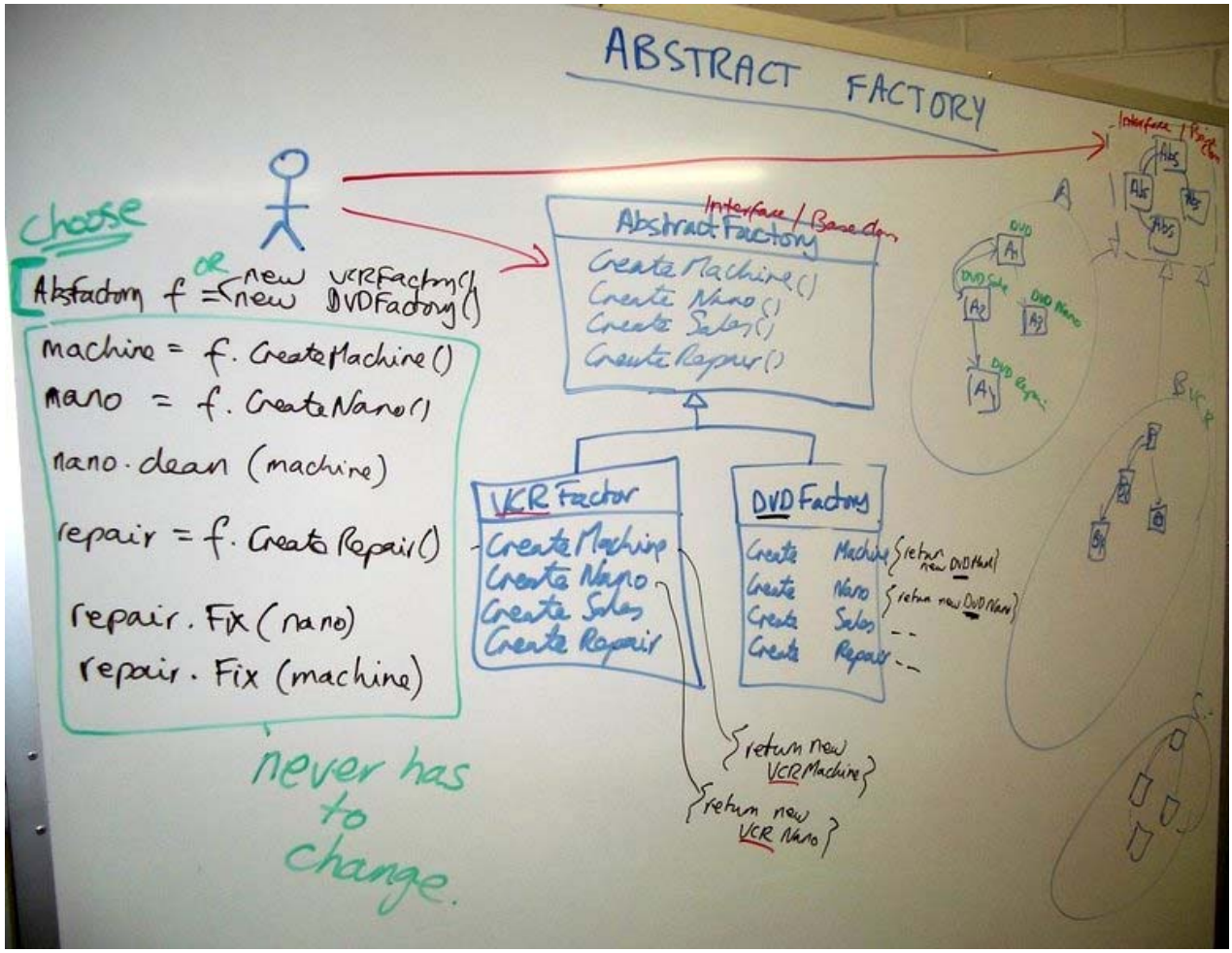


Figure 30 – A concrete factory in the Abstract Factory Pattern is a class filled with Factory Methods.

The client code has no knowledge whatsoever of the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.

What do the factory methods on a concrete Abstract Factory do?

- Probably should have the phrase 'create' somewhere in its name e.g. CreateProd1, or CreateProd2
- The factory methods on an Abstract Factory should only create 'products' - which are instances of classes
- Each returned product should be of a different class type
- The methods on the factory should not do any real work or computation

Anecdote from a Melbourne patterns group meeting…

I once saw a presentation on Abstract Factory where the factory methods on the concrete abstract factory class actually did complex computational work (each one did something different) and all returned a result class of the same type (a data set, as it happens). This is not abstract factory. The methods in the factory should not have done any real WORK just merely returned DIFFERENT 'products'. Products are a bunch of different classes which are meant to be instantiated as a family of classes, which typically work together. Thus the idea of an 'abstract factory class generating a FAMILY of DIFFERENT but relatedclasses' was entirely missing, and this it seems to me to be the essence of abstract factory.

What was being actually presented was simply a compile time choosing of one of two different implementations of an interface, which I call the Interface Pattern and discussed in the article “The road to Bridge” page 295.

Puzzle

Imagine you create an instance of a factory, either

```
factory = GetAppropriateFactory(“3D”)
```

or

```
factory = GetAppropriateFactory(“2D”)
```

and you call the following methods on it. Is one of these methods out of place?

```
factory.CreateBox()
factory.CreateCircle()
factory.DrawLine()
```

Answer: The methods on such a factory should be factory methods, which return an objects. Thus the DrawLine() method is out of place, since it does something rather than returning an object.

Implementation alternatives – registry based factories and injection frameworks

A concrete factory in the Abstract Factory Pattern is a class filled with Factory Methods. All these "create"methods are factory methods, where the methods are hard coded to return an instance of a particular class.

The GOF style abstract factory, as described in the GOF book, uses such polymorphic override. The GOF abstract factory implementation knows the family of classes you will need ("the switch") via the act of instantiatiating the required "concrete factory" subclass (with all creational methods polymorphically overridden).

Just as "factory method" can be implemented in other ways, beside the classic GOF override style (see *Simple Super Dumb Factory* on page 296 and *Registry Based Factory* on page 297), arguably, you could similarly have a have *registry based* implementation variations of abstract factory too. You could rely on some sort of generic mechanism e.g. a smart registry / injection mechanism rather than hard coding / polymorphic override in order to instantiate your family of classes.

Alternative implementation technique

For example, such a *registry based* implementation of an abstract factory could set a switch once on a **single** factory (rather than having lots of different concrete factories), and that factory object subsequently remember which family/version/style of classes to return from then on. Internally the factory could be using something less elegant than traditional GOF abstract factory e.g. messy if statements, or something more elegant e.g. grouping families of classes together by tagging them in the registry somehow. To the user of the factory (the client code), the result is the same.

```
factory.SetFamily('b version')
factory.CreateProduct1()
factory.CreateProduct2()
factory.CreateProduct3()
```

Evaluation of the alternative technique(s)

The standard GOF (Gang of Four) polymorphic override technique is actually quite an elegant implementation choice for abstract factory since we have a different concrete factory per family where we simply override the create methods to make a specific objects. To do a registry version of this would be ok as well, but a bit harder to implement due to the need to group class names together into families. Not too hard to implement though.

And arguably, dependency injection frameworks could implement the intent of abstract factory – injecting/instantiating different families of object depending on some configuration switch.

Its worth mentioning these implementation alternatives, because Design Patterns are ideas which can be implemented in a variety of ways – so lets' not get too hooked into the classic GOF example UML solution and consider alternatives too!

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Non-software Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

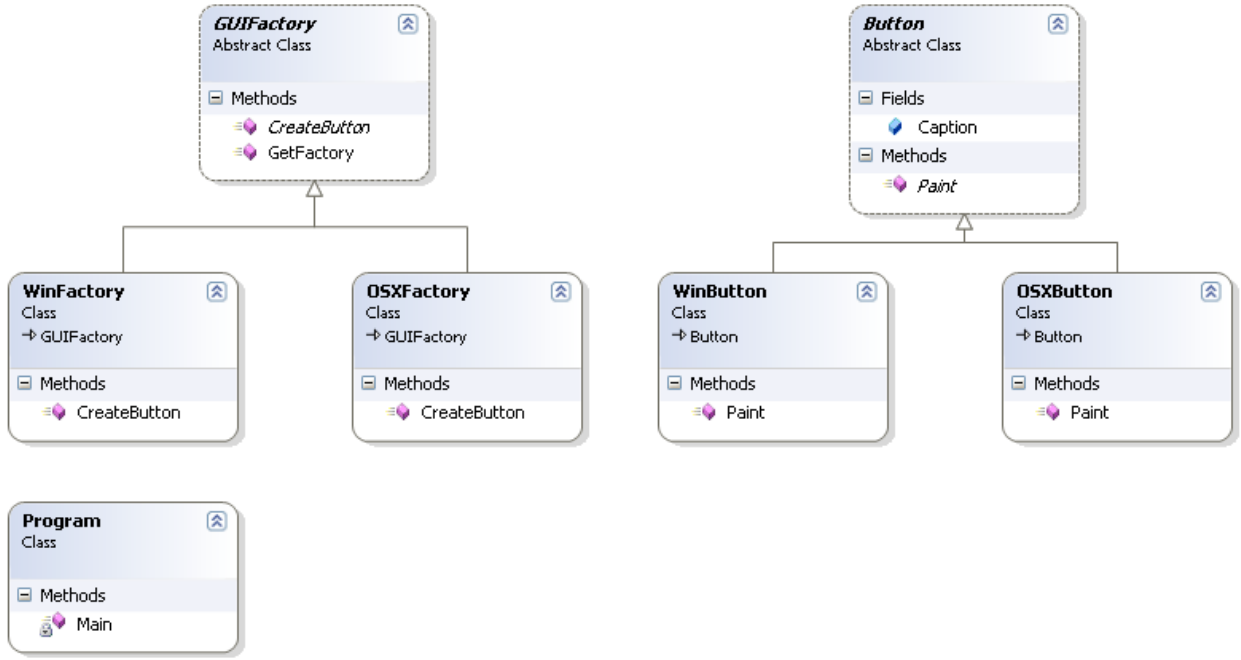
Code Ideas

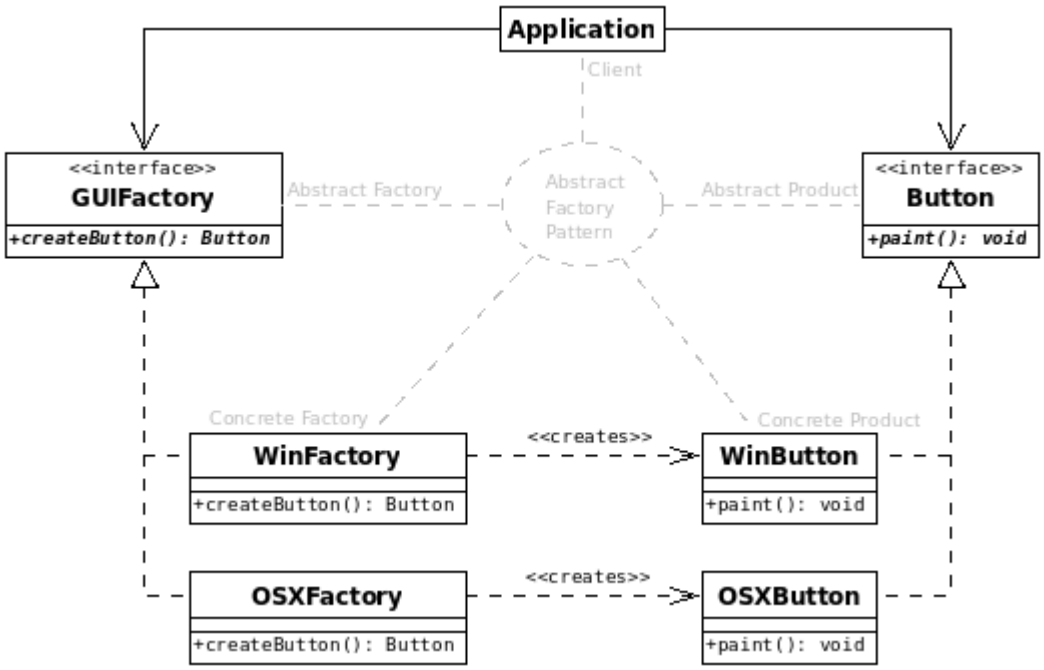
www.dofactory.com's structural code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

www.dofactory.com's real-world code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

C# Example Code:

Abstract factory is often used to switch between GUI toolkits or operating system specific libraries.






```
using System;
using System.Collections.Generic;
using System.Text;

namespace AbstractFactory01
{
```

```
/*
* Note how the abstract (base) class GUIFactory is the one
* that instantiates its own subclasses (WinFactory, OSXFactory).
* As those subclasses inherit from base and override base CreateButton method,
* it’s GUIFactory's job to disambiguate who’s overriding its CreateButton method,
* at the time of being instantiated by Client.
* Adding more factory subclasses (e.g. LnxFactory) confines change
* to GUIFactory class only
*/
abstract class GUIFactory
{
    public static GUIFactory GetFactory()
    {
        int sys = 0;  // or something like ReadFromConfigFile("OS_TYPE");
        if (sys == 0)
        {
            return new WinFactory();
        }
        else
        {
            return new OSXFactory();
        }
    }

    public abstract Button CreateButton();
}

class WinFactory : GUIFactory
{
    // Usually you would have a lot of “factory methods” here
    // each returning an appropriate instance of a class.
    public override Button CreateButton()
    {
        return new WinButton();
    }
}

class OSXFactory : GUIFactory
{
    public override Button CreateButton()
    {
        return new OSXButton();
    }
}

abstract class Button
{
    public string Caption;
    public abstract void Paint();
}

class WinButton : Button
{
    public override void Paint()
    {
        Console.WriteLine("I'm a WinButton: " + Caption);
    }
}
```

```
    class OSXButton : Button
    {
        public override void Paint()
        {
            Console.WriteLine("I'm an OSXButton: " + Caption);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            GUIFactory factory = GUIFactory.GetFactory();
            Button button = factory.CreateButton();
            button.Caption = "Play";
            button.Paint();

            Console.ReadLine();
        }
        // Output is either:
        //   "I'm a WinButton: Play"
        // or:
        //   "I'm an OSXButton: Play"

    }

}
```

Notes:

Builder

Intent

Separate the construction of a complex object from its representation so that the same parser/construction process can create different products/representations. Build valid objects from possibly unreliable streams.

Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Discussion

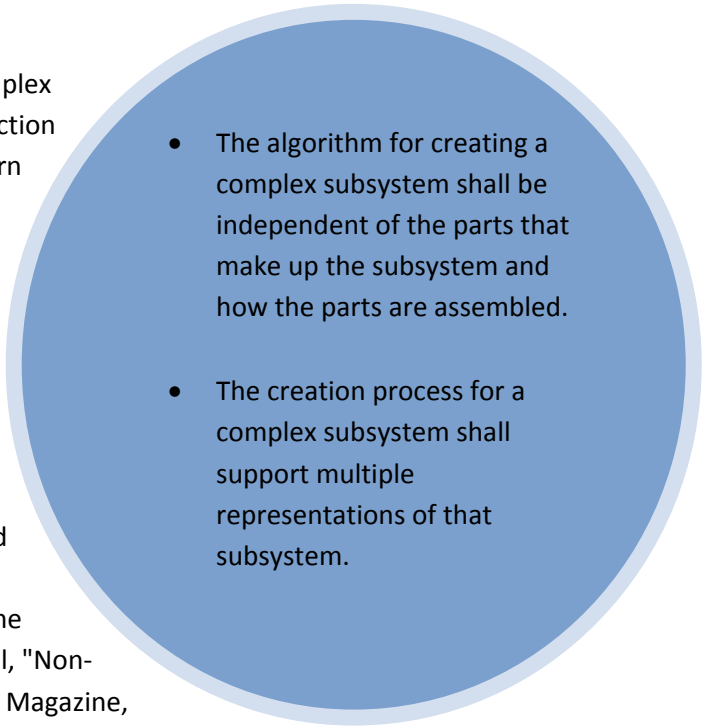
Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".

Non Software Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy car). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]



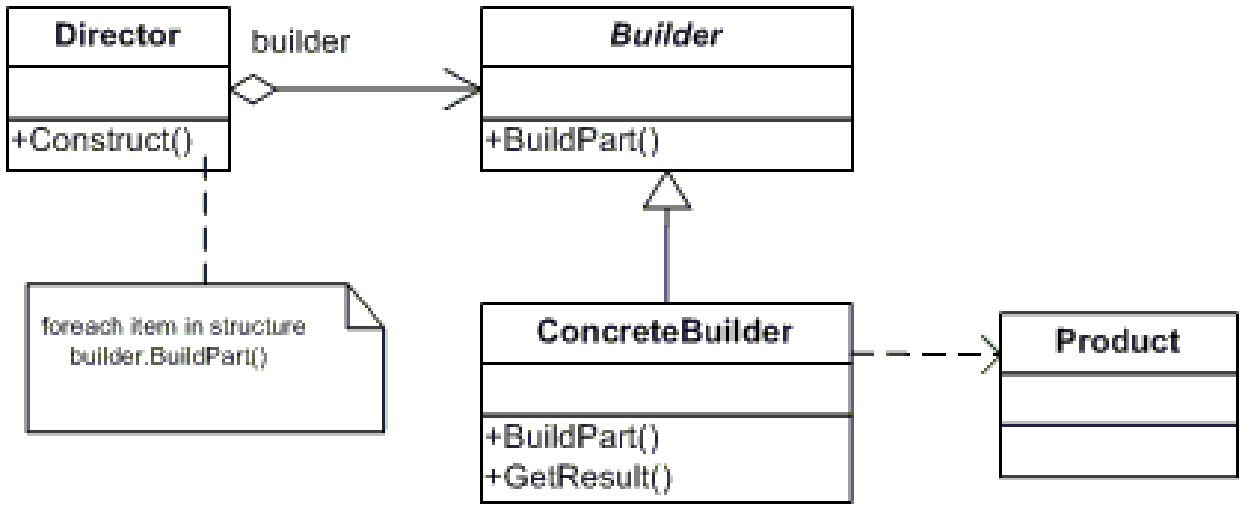


Figure 31 - UML of Builder. Note that the GetResult() occurs only on the conrete builder - not in the base Builder class.

Participants

The classes and/or objects participating in this pattern are:

- Builder (VehicleBuilder) **THE INTERFACE OF "STEPS" the parser talks to**
  - specifies an abstract interface for creating parts of a Product object
- ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder) **STUFFS VALUES into product**
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product
- Director (Shop) **DOES THE PARSING**
  - constructs an object using the Builder interface
- Product (Vehicle) **DUMB PRODUCT (in the sense that it doesn't know how to construct itself)**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Andy's Tips on Builder

- Separates the construction of a complex object from its representation i.e. The parser/director talks to a builder interface, so you can swap in different builders in order to build different products from the same "input stream" and using the exact same parser.
- It's a bit like template method in the sense there are a recipe of steps which are overridden, except the steps are delegated to another class, and the steps occur in no fixed order, depending on the parsing of a stream (usually).
- There is a distinction between "builder validation" and the "default product" that the builder builds. The product constructor should create a sensible default product. Builder validation is then extra logic that checks the validity of all builder objects that are ever produced.
- Can help avoid creating invalid/incomplete complex objects esp. when reading from streams etc. e.g. The parser/director can pass the builder lots of info and then when all of the info in, ask for the finished object, which is guaranteed to be valid (if you have a Forgiving Builder which fills in missing information for you). Overly complex business logic for validation can live in the builder rather than in the final product/object itself.

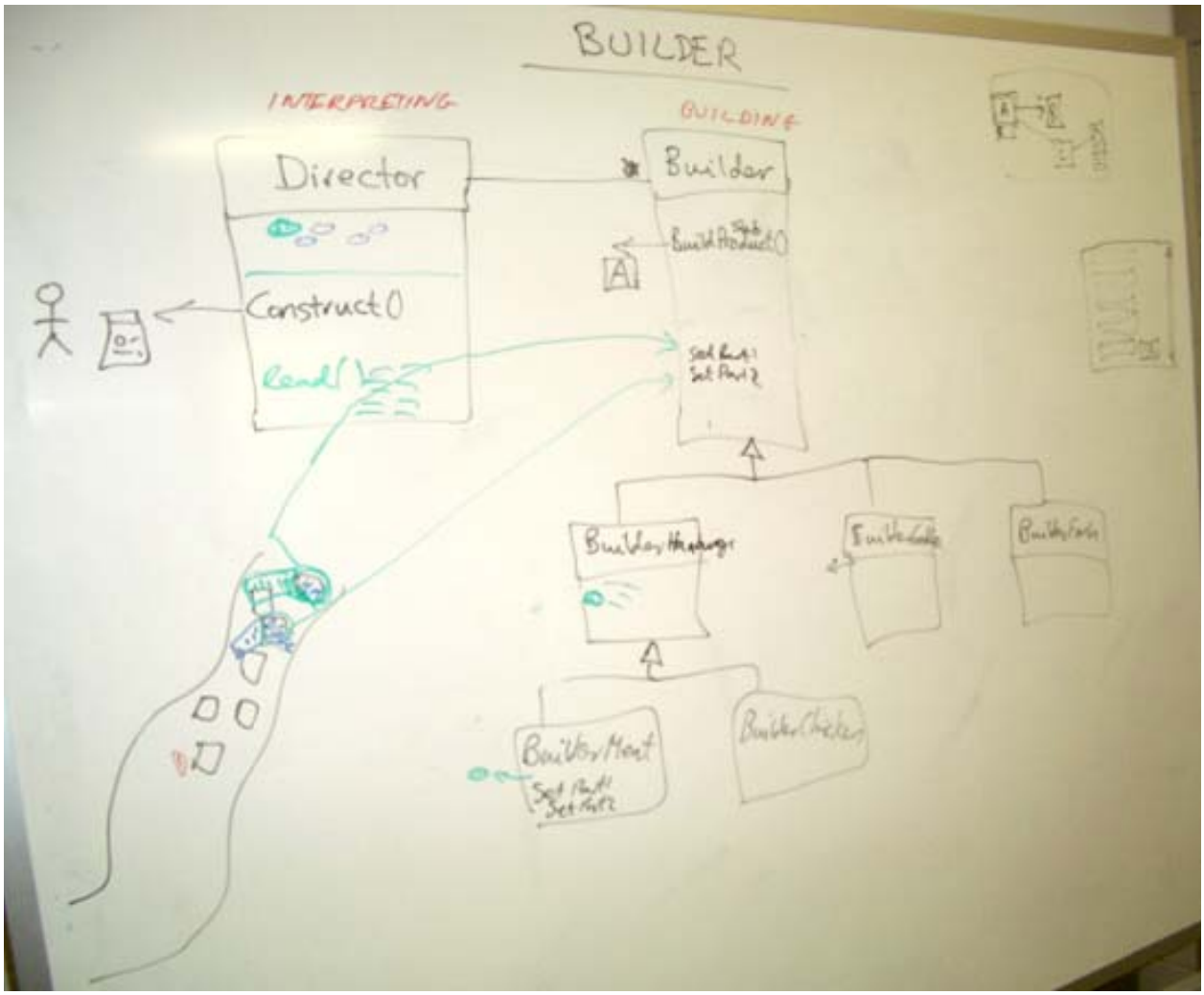


Figure 32 - Reading from an uncertain stream and building objects from it. Switch concrete Builders to get different products. The Parser (DIRECTOR) usually remains the same for all builders.

The two halves of this pattern – the director (parser) and family of builders

Here is an example of how the reader/parser/director drives the builder. Pick a different builder depending on what sort of product or representation you want. Different builders build different products.

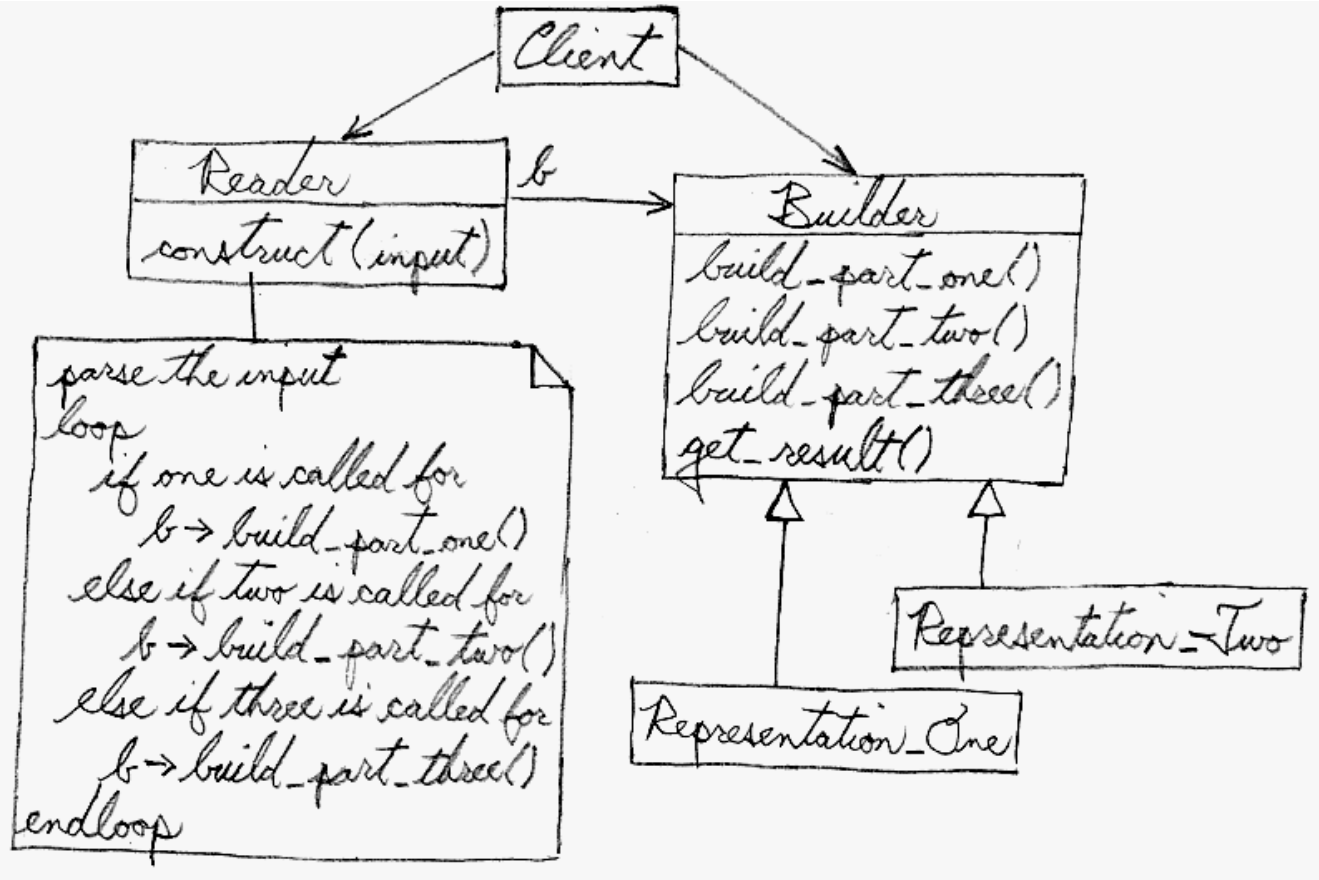


Figure 33 - The Reader (DIRECTOR) encapsulates the parsing of the common input. The Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets. Diagram courtesy www.vincehuston.org

Example: Different Concrete Builders build different products

This is a great example as it shows how the one Reader/parser/director can create different products (ascii, postscript, pdf) using different concrete builders.

The Reader/parser/director talks to the "Converter"/Builder interface.

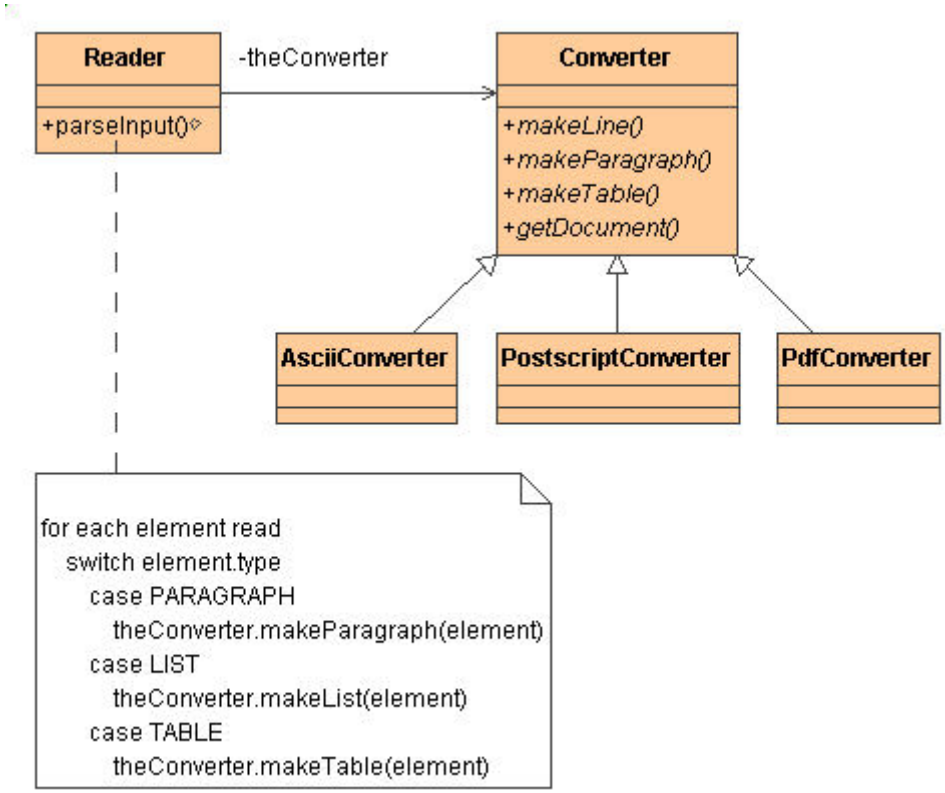


Figure 34 - Practical example where the same parser can drive the creation of different "products"

Code Ideas

www.dofactory.com's structural code demonstrates the Builder pattern in which complex objects are created in a step-by-step fashion. The construction process can create different object representations and provides a high level of control over the assembly of the objects.

www.dofactory.com's real-world code demonstates the Builder pattern in which different vehicles are assembled in a step-by-step fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

Example: C# Reservation Builder

Task: Read from a string e.g. "Date, 2/2/2002, City, Sydney" and build a Reservation object from it. Then switch in a different builder, and using the same string parser, create a different type of Reservation object, e.g. an XML style Reservation object. Repair any missing data e.g. if the string is missing the city, assume Greenwich.

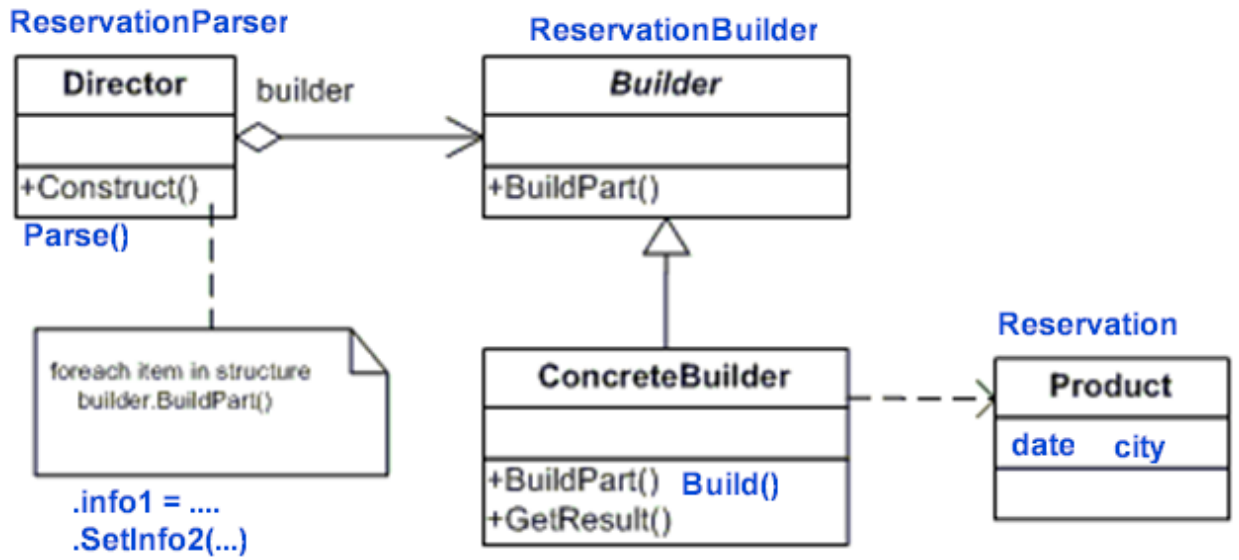


Figure 35 - A loose attempt to graft the classes in the C# example over the top of the "theoretical UML" of builder pattern

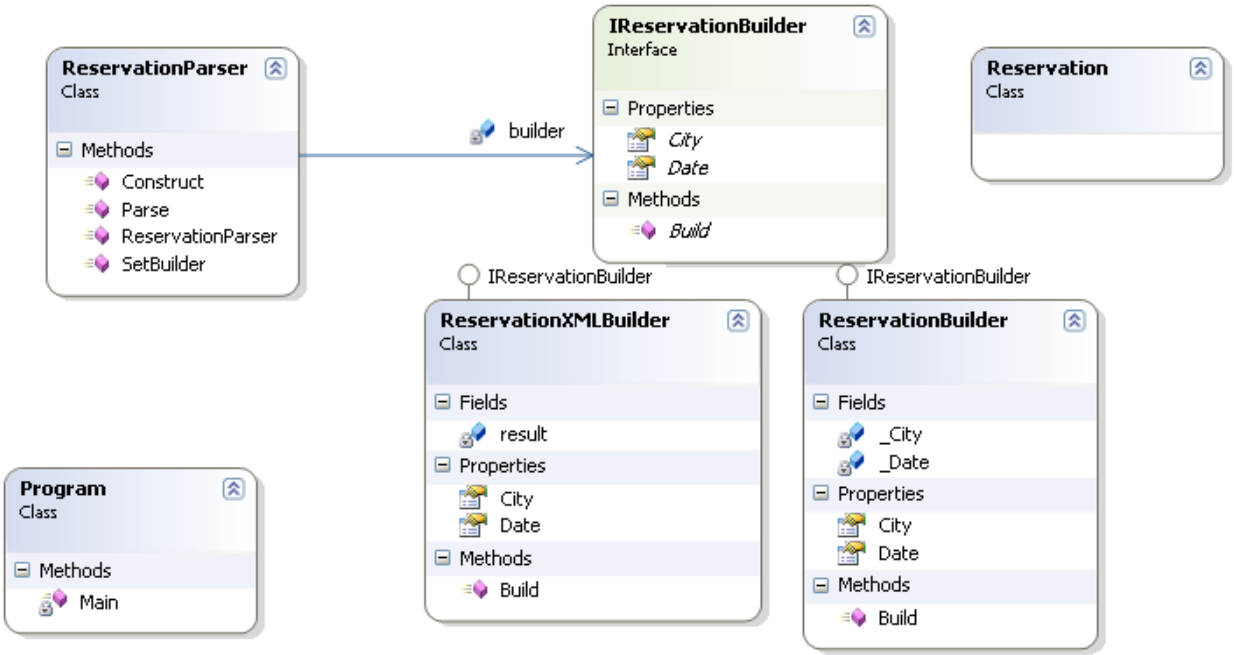


Figure 36 - Exact UML of C# example

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Text.RegularExpressions;

namespace BuilderPattern01
{
    class Program
    {
        static void Main(string[] args)
        {
            // Test the builder pattern
            ReservationBuilder b = new ReservationBuilder();
            ReservationParser p = new ReservationParser(b);
            Reservation r;

            // Example 1 GOOD
            // Parse the string and build a reservation object
            p.Parse("Date, 2/2/2002, City, Sydney");
            r = b.Build();
            Console.WriteLine(r.ToString() + "\n");

            // Example 2
            // Using the Construct() method instead, which combines
            // the parse and build into one step.
            // Also string has city info missing.  Builder should cope and repair ok.
            r = p.Construct("Date, 2/2/2003, City, "); ;   // note no CITY supplied
            Console.WriteLine(r.ToString() + "\n");

            // Example 3
            // Switch to a different builder, which builds an xml object
            // Use the same parser (DIRECTOR)
            p.SetBuilder(new ReservationXMLBuilder());
            r = p.Construct("Date, 2/2/2003, City, ");
            Console.WriteLine(r.ToString() + "\n");

        }
    }

    class Reservation {}

    class ReservationObj : Reservation
    // Product
    {
        public DateTime date;
        public String city;

        public ReservationObj(DateTime date, string city)
        {
            this.date = date;
            this.city = city;
        }

        public override string ToString()
        {
            return "Object: Reservation. date: " + this.date + " city: " + this.city;
        }

    }
```

The benefit of example 1 is that you are asking the builder for a result of a specific type – which can be different depending on the builder. You then can use the result and take advantage of the specific type of the result.

The drawback of example 2 (the all-in-one Construct() method) is that the builders all then have to return the same type e.g. **Reservation** which is too restrictive and often artificial.

Better to parse then build - two steps – this gives you more control over the second step (builder builds a specific type for you , that you know and are expecting).

```
class ReservationParser
// Director
{
    private IReservationBuilder builder;

    // Pass in the concrete builder to the parser/director
    public ReservationParser(IReservationBuilder builder)
    {
        SetBuilder(builder);
    }
    public void SetBuilder(IReservationBuilder builder)
    {
        this.builder = builder;
    }

    public void Parse(String s)
    {
        // Split data in s into separate strings
        string[] tokens = new Regex(", ").Split(s);
        string[] tokens2 = s.Split(new char[] { ' ' }); // alternative split
                                                           technique

        // Debug
        foreach (string token in tokens)
            Console.WriteLine("token=" + token);

        // Read and set values on builder OR pass values to builder.
        // Data (tokens) are in the form: "Date", date, "City", citystring
        // Loop through each pair of tokens...
        for (int i = 0; i < tokens.Length; i += 2)
        {
            string infoType = tokens[i].ToUpper().Trim();
            string infoVal = tokens[i + 1];

            // You could create the product (Reservation) here and
            // wire it up directly.  Why bother going through a intermediary
            // builder class?
            //
            // Because builder class can add value by filling in defaults,
            // and delaying creation of the product till all info is in.

            if (infoType == "CITY")
                builder.City = infoVal;
            else if (infoType == "DATE")
                builder.Date = DateTime.Parse(infoVal);
        }
    }

    public Reservation Construct(String s)
    {
        // Alternative way to drive the builder
        this.Parse(s);
        return builder.Build();
    }
}
```

```
    interface IReservationBuilder
    {
        // Reservation Build();
        string City { set; }
        DateTime Date { set; }
    }

    class ReservationBuilder : IReservationBuilder
    // Builder
    {
        private DateTime _Date;
        private String _City;

        public string City { set { _City = value; } }
        public DateTime Date { set { _Date = value; } }

        public Reservation Build()
        {
            // Always supply a city
            if (this._City == "")
                this._City = "Greenwich";

            return new ReservationObj(this._Date, this._City);
        }
    }
```

Arguably **don't** put the **Build()** method in the `IReservationBuilder` interface because then you force the Director to talk to specific builders directly and have those builders return specific types e.g. string or ReservationObj.

In other words, you want to be dealing with different products and thus you don't want to be insulated from this type information. If you insist on a Build() method in the interface then you are commited to having a generic Reservation type, which limits the specificity of what you can do with the returned product.

And… switching in a different Builder….

```
    class ReservationXMLBuilder : IReservationBuilder
    // Builder2
    {
        private string result;

        public string City { set { result += "<CITY>" + value + "</CITY>"; } }
        public DateTime Date { set { result += "<DATE>" + value + "</DATE>"; } }

        public Reservation Build()
        {
            return new ReservationXML(result);
        }
    }

    class ReservationXML : Reservation
    // Product2
    {
        public String ReservationString;

        public ReservationXML(String s)
        {
            this.ReservationString = s;
        }

        public override string ToString()
        {
            return "<RESERVATION> " + this.ReservationString + "</RESERVATION>";
        }
    }
}
```

Output:

```
token=Date
token=2/2/2002
token=City
token=Sydney
Object: Reservation. date: 2/02/2002 12:00:00 AM city: Sydney

token=Date
token=2/2/2003
token=City
token=
Object: Reservation. date: 2/02/2003 12:00:00 AM city: Greenwich

token=Date
token=2/2/2003
token=City
token=
<RESERVATION> <DATE>2/02/2003 12:00:00 AM</DATE><CITY></CITY></RESERVATION>

Press any key to continue . . .
```

Rules of thumb

Sometimes creational patterns are complementory: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134]

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]

Builder is to creation as Strategy is to algorithm. [Icon, p8-13]

Builder often builds a Composite. [GOF, p106]

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Advanced Discussions

Is the BuilderPattern a work-around for a C++ limitation?

> *Some say that the BuilderPattern is a work-around for a C++ limitation. The obvious way to achieve what the BuilderPattern achieves is to have a base-class constructor which calls a bunch of virtual methods, which can be be overridden by the sub-classes (or provided for the first time, if they were abstract on the base class). This doesn't work in C++ since function calls made within the construction phase of the base class are not polymorphic. (There may be occasions when you*

would still need to use builder classes without this limitation, and there seem to be versions of the BuilderPattern which are more complicated, but I think I have the essence of it here). The BuilderPattern is just about polymorphic construction, which is hampered if your constructors don't support polymorphism!

From http://c2.com/cgi/wiki?DesignPatternsInDynamicProgramming

There is also broader discussion on this on page with many differing opinions.

What is the difference between builder and factory pattern ?

The factor pattern defers the choice of what concrete type of object to make until run time. E.g. going to a restaurant to order the special of the day. The waiter is the interface to the factory that takes the abstractor generic message "Get me the special of the day!" and returns the concrete product (i.e Liver souffle or Chicken caramel)

The builder pattern encapsulates the logic of how to put together a complex object so that the client just requests a configuration and the builder directs the logic of building it. E.g The main contractor (builder) in building a house knows, given a floor plan, knows how to execute the sequence of operations (i,e. by delegating to subcontractors) needed to build the complex object. If that logic was not encapsulated in a builder, then the buyers would have to organize the subcontracting themselves ("Dear, shouldn't we have asked for the foundation to be laid before the roofers showed up?")

The factory is concerned with what is made, the builder with how it is made. Design patterns points out that (page 105) Abstract factory is similar to builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract factor's emphasis is on families of product objects(either simple or complex). Builder returns the product as the final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Explanation by Rod Davison - Critical Knowledge Systems Inc.

Notes:

Singleton

A class of which only a single instance can exist.

Intent

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

Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Participants

The classes and/or objects participating in the Singleton pattern are:

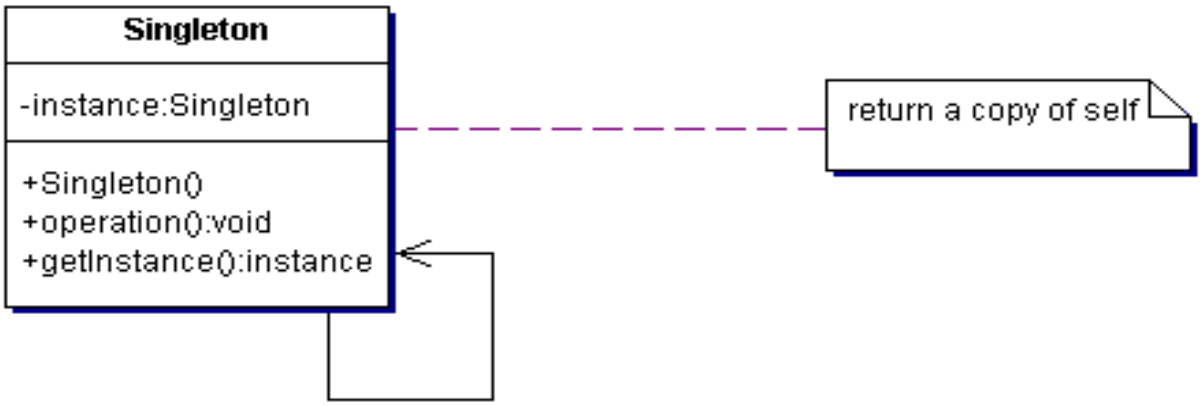
- Singleton (e.g. LoadBalancer)
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

• A single instance of a subsystem shall be enforced, and the subsystem itself shall be responsible for that enforcement.

• The single instance shall be globally accessible.

• The single instance shall be initialized only if, and when, it is accessed.

Question: Is being “globally accessible” critical to the Singleton pattern?


Singleton
-instance:Singleton
+Singleton()
+operation():void
+getInstance():instance
return a copy of self

Non Software Example

The office of the President of the United States is a Singleton. There can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

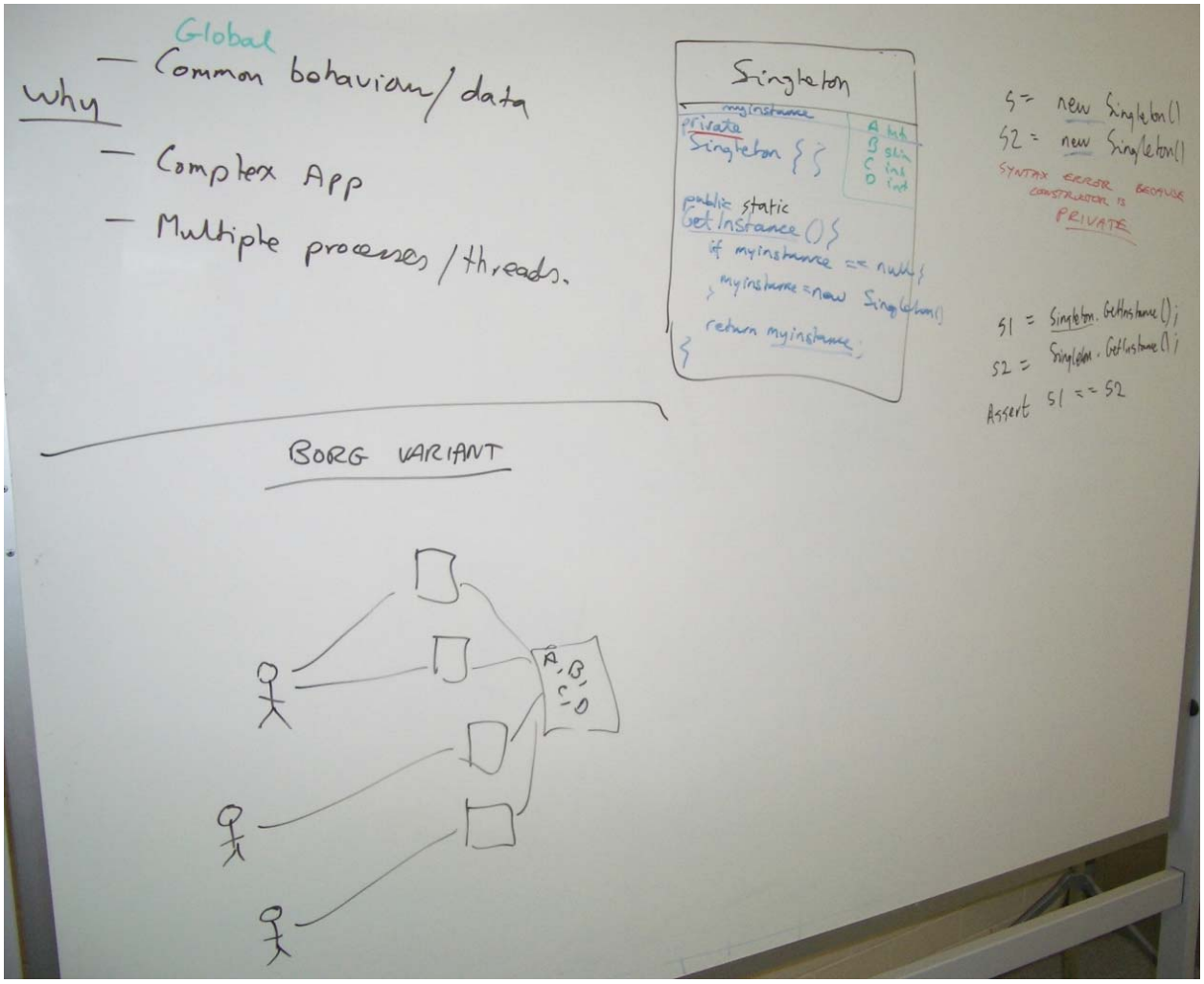


Figure 37 - Discussion on when to use Singleton, how to implement it, and finally an intriguing variant called the "Borg"

Andy's Tips

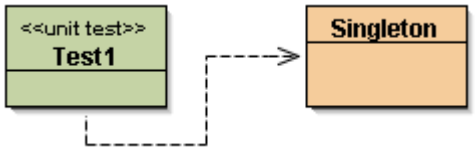
- Ensure class has only **one instance** and provide a global point of access to it.
- Can control access strictly.
- Like global without being a global variable.
- BORG pattern variant – allows many instances, but share common data. The argument here is that it's the data that's important to keep common, not how many instances of an object you have. Controversial? Easy to implement in Python – maybe not so easy in other languages, thus the need for Singleton (rather than Borg).
- Thread Safety – make GetInstance() synchronized or mutex/synch the block around the code.

Code Idea

www.dofactory.com's real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go throught the one object that has knowledge about the state of the (web) farm.

Code Example: Classic Solution

The classic solution in Java is to make the constructor private, and create a new static method called GetInstance() for returning an instance of your class. GetInstance() caches the instance so that only one instance ever gets created.




```
public class Singleton
{
    private static Singleton _instance;

    private Singleton(){}

    public static Singleton GetInstance(){
        if (_instance == null)
            _instance = new Singleton();
        return _instance;
    }
}


public class Test1 extends junit.framework.TestCase
{
      public void testClassic()
      {
        Singleton s, s1, s2;

        //s = new Singleton();    // Cannot do this anymore
        s1 = Singleton.GetInstance();
        s2 = Singleton.GetInstance();
        assertEquals(s1, s2);  // the whole point of singleton!
      }

      public void testSingularity()
      {
            assertEquals(Singleton.GetInstance(), Singleton.GetInstance());
      }
}
```

When to use Singleton

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Deleting a Singleton class/instance is a non-trivial design problem. See "To kill a Singleton" by John Vlissides (C++ Report, Jun 96, pp10-19) for a discussion.

Opinions

"Singleton is not going to solve global data problems. When I think of Singleton, I think of a pattern that allows me to ensure a class has only one instance. As far as global data, the only benefit I see is reducing the namespace." [Christopher Parrinello]

"This is not really the point, but I always teach Composite, Strategy, Template Method, and Factory Method before I teach Singleton. They are much more common, and most people are probably already using the last two. The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances." [Ralph Johnson]

Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" - even though this and that are the same thing. [Frieder Knauss]

Rules of thumb

Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, p134]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

State objects are often Singletons. [GOF, p313]

Notes:

Proxy

An object representing another object.

Intent

Provide a surrogate or placeholder for another object to control access to it.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client. Or you need to security checked before you access an object.

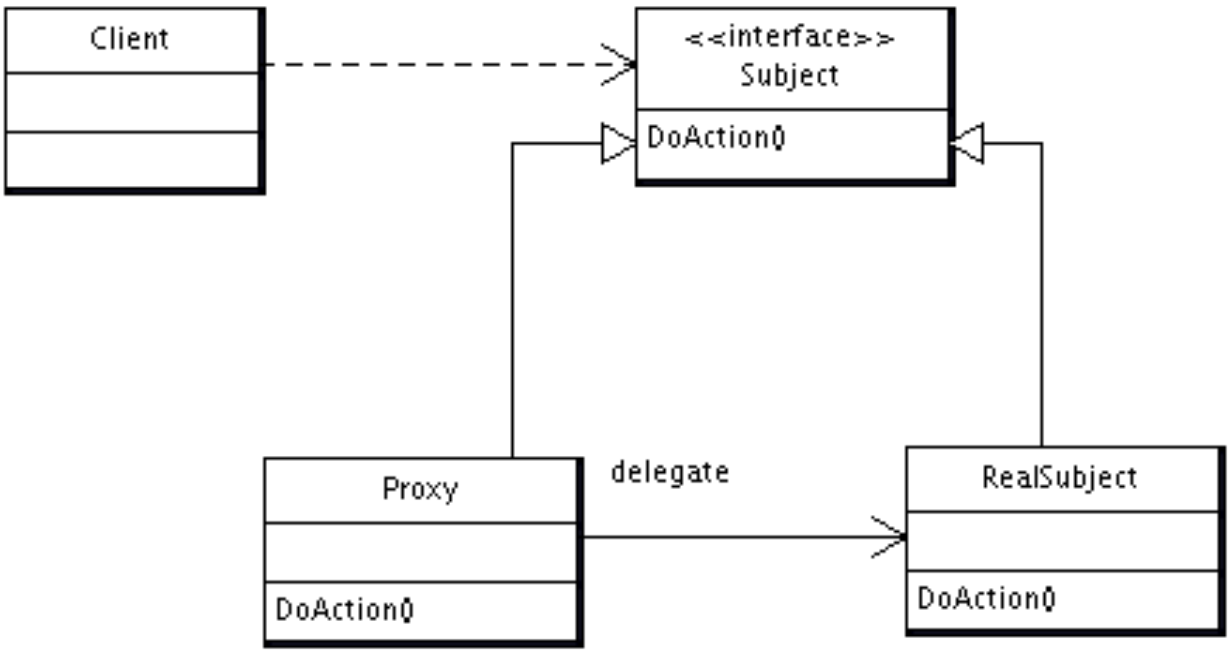
Discussion

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Participants

The classes and/or objects participating in the Proxy pattern are:

- Proxy (MathProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - **provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.**
  - controls access to the real subject and may be responsible for creating and deleting it.
- Subject (IMath)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Math)
  - defines the real object that the proxy represents.


Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

Ideas

- The system shall support "distributed processing" by providing a local representative for a component in a different address space.
- The system shall support "lazy creation" - a component is created only if, and when, the client demonstrates an interest in it.
- The system shall control access to components by interposing an intermediary that evaluates the identity and access privileges of the requestor.
- The system architecture shall be characterized by multiple dimensions of indirection that offer a locus for intelligence that would unduly complicate other components if they were obliged to support the additional responsibility.

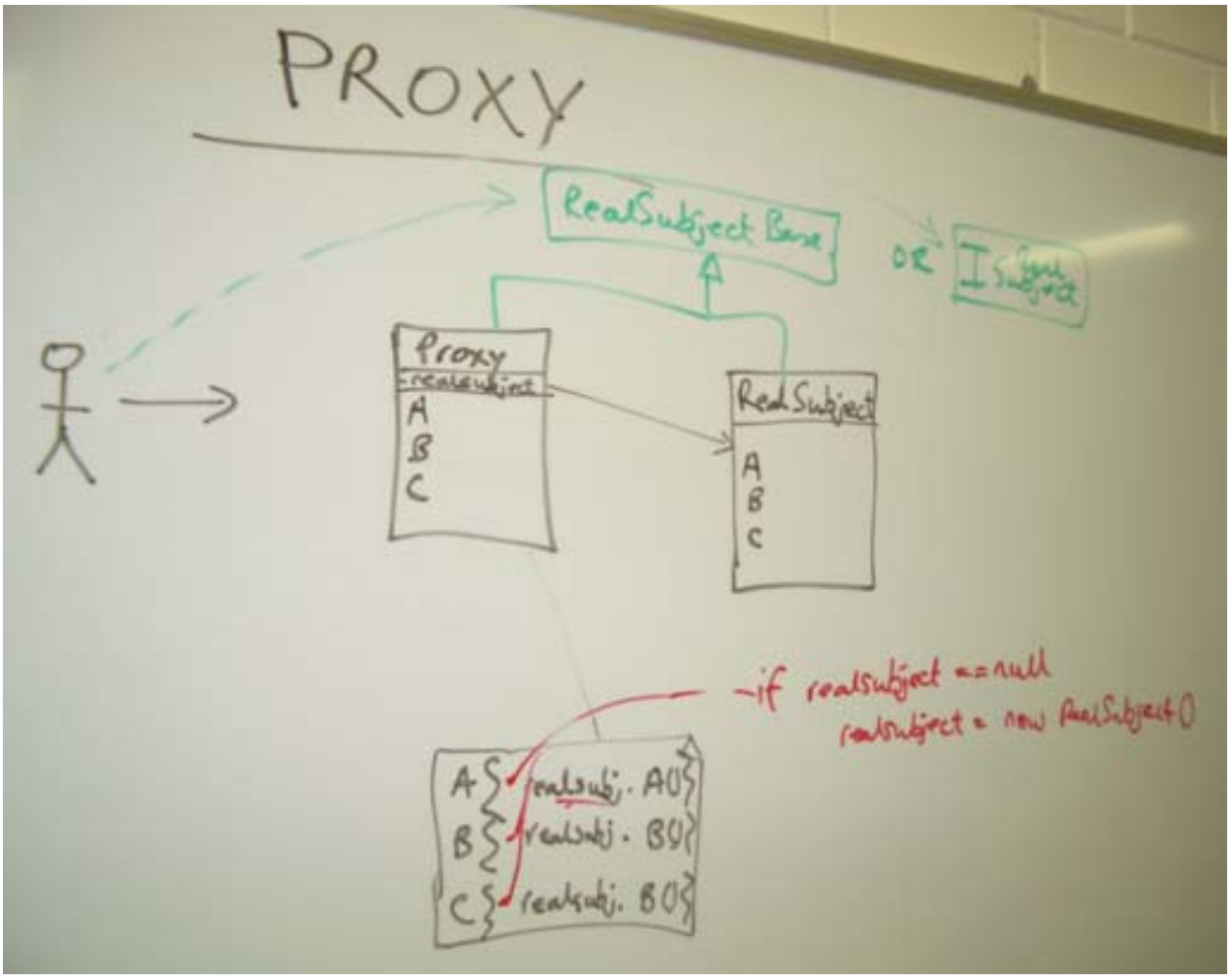


Figure 38 - Proxy and RealSubject must descend off the same base class or interface.

Andy's Tips

- Basic indirection pattern—client code talks to the proxy instead of the real thing.
- Both the proxy and the real thing either descend off the same base class or implement the same interface, so that the client can use either the proxy or the real thing without knowing. **Of course the client code should only talk to the interface or the base class.**
- Decorator wraps an object and adds functionality multiple times. Proxy wraps an object typically only once, and adds functionality only of a certain type – relating to security, cost of instantiation, lazy instantiation, remoting.

When to use Proxy?

There are four common situations in which the Proxy pattern is applicable.

- A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
- A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
- A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
- A smart proxy interposes additional actions when an object is accessed. Typical uses include:
    - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
    - Loading a persistent object into memory when it's first referenced,
    - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Types of proxy pattern include:

- **Remote proxy:** Provides a reference to an object located in a different address space on the same or different machine.
- **Virtual proxy:** Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed. (See also Lazy evaluation.)
- **Copy-on-write proxy:** Defers copying (cloning) a target object until required by client actions. This is really a special case of the "virtual proxy" pattern.
- **Protection (access) proxy:** Provides different clients with different levels of access to a target object.
- **Cache proxy:** Provides temporary storage of the results of expensive target operations so that multiple clients can share the results. (See also Memoization.)
- **Firewall proxy:** Protects targets from bad clients (or vice versa).
- **Synchronization proxy:** Provides concurrency control over an unsynchronized target object.
- **Smart reference proxy:** Provides additional actions whenever a target object is referenced, such as counting the number of references to the object.

Other responsibilites depend on the kind of proxy:

- remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
- virtual proxies may cache additional information:
    - about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
    - protection proxies check that the caller has the access permissions required to perform a request.

Code Ideas

www.dofactory.com's structural code demonstrates the Proxy pattern which provides a representative object (proxy) that controls access to another similar object.

www.dofactory.com's real-world code demonstrates the Remote Proxy pattern which provides a representative object that controls access to another object in a different AppDomain.

Example: Lazy Instatiation Proxy

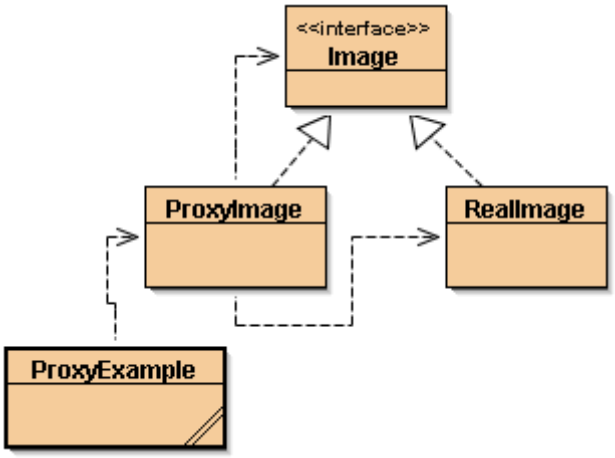
The proxy can do a lazy instantiation of the real subject, if the real subject doesn't exist—which is a nice use of a proxy. E.g. Proxy Class:

```
Proxy.SomeMethodAA() {

  // Lazy instantiation
  if this.realSubject == null
        this.realSubject == new realSubject()

  // Now do the proxy delegation work
  this.realSubject.SomeMethodAA();
}
```

Java Example: Image Proxy

You don't want to load the image off disk etc. until it is needed to be displayed. The ProxyImage class remembers the filename, and only creates an instance of the RealImage class when it is asked to displayImage().

The RealImage class however, loads the image immediately – whether it is displayed or not. Note that loadImageFromDisk() is called in the constructor - this is the heavy lifting work the proxy is trying to avoid till its absolutely necessary.




```
import java.util.*;

interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        // Potentially expensive operation
        // ...
        System.out.println("Loading   "+filename);
    }

    public void displayImage() { System.out.println("Displaying "+filename);
}
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        List<Image> images = new ArrayList<Image>();
        images.add( new ProxyImage("HiRes_10MB_Photo1") );
        images.add( new ProxyImage("HiRes_10MB_Photo2") );
```

```
        images.add( new ProxyImage("HiRes_10MB_Photo3") );

        images.get(0).displayImage(); // loading necessary
        images.get(1).displayImage(); // loading necessary
        images.get(0).displayImage(); // no loading necessary; already done
        // the third image will never be loaded - time saved!
    }
}
```

Output:

```
Loading   HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading   HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo1
```

Non Software Example

Asking technical questions of a Sales Representative. In all likelihood, the Sales Representative will forward the technical request to a Technical Representative and relay the answer to the customer.

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Notes:

Adapter

Wrap a class with another class, so that you can use a different (nicer) API to drive it.

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.

Problem

An "off the shelf" component offers compelling functionality that you would like reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

Create an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class. It effectively offers an impedence-matching facility.

Participants

The classes and/or objects participating in the Adapter pattern are:

- Target (ChemicalCompound)
  - defines the domain-specific interface that Client uses.
- Adapter (Compound)
  - adapts the interface Adaptee to the Target interface.
- Adaptee (ChemicalDatabank)
  - defines an existing interface that needs adapting.
- Client (AdapterApp)
  - collaborates with objects conforming to the Target interface.

A legacy component shall be reused in a new design, and the interface "impedance mismatch" shall be reconciled.

A reusable component shall cooperate with unrelated (or unforeseen) application components.

Structure

This diagram on the following page shows the “object adapter” implementation which uses an aggregation/delegation mechanism. There is also a “class adapter” variant. Figure 41 shows both “classic” implementations of adapter contrasted with each other.

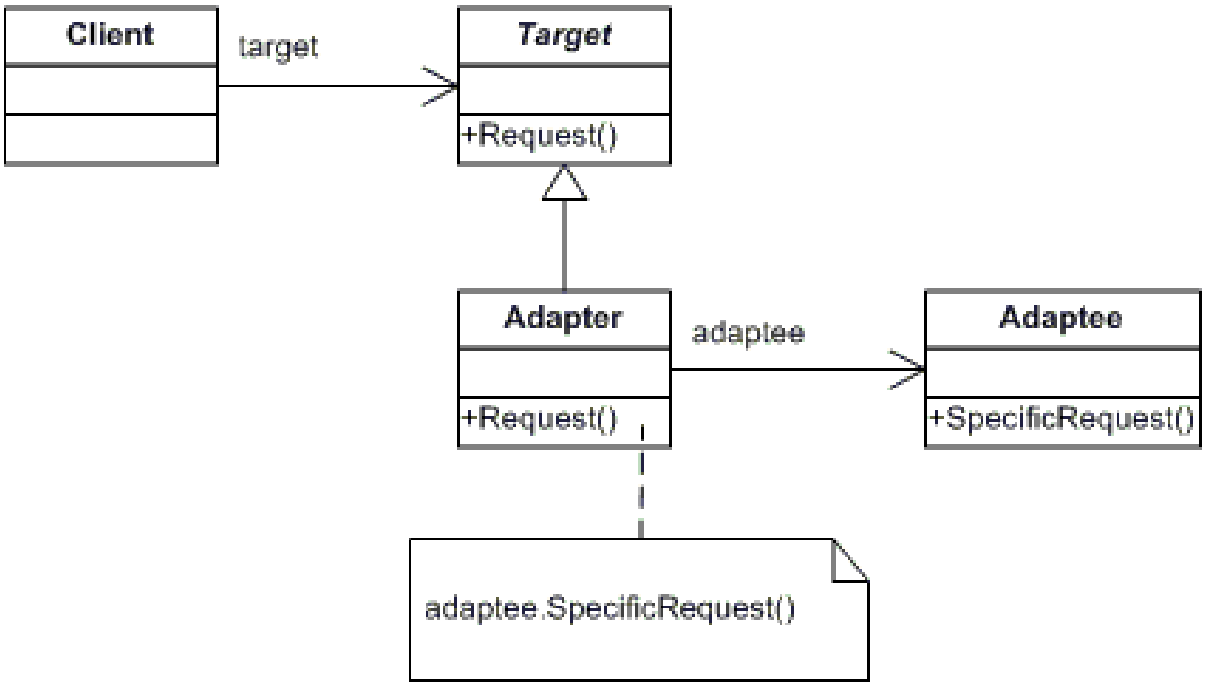


Figure 39 - UML of Adapter

Andy's Tips

The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need. However you can't change the class because it is being used by too much client code that you don't want to change. Or you can't change the class because you don't have the source code.

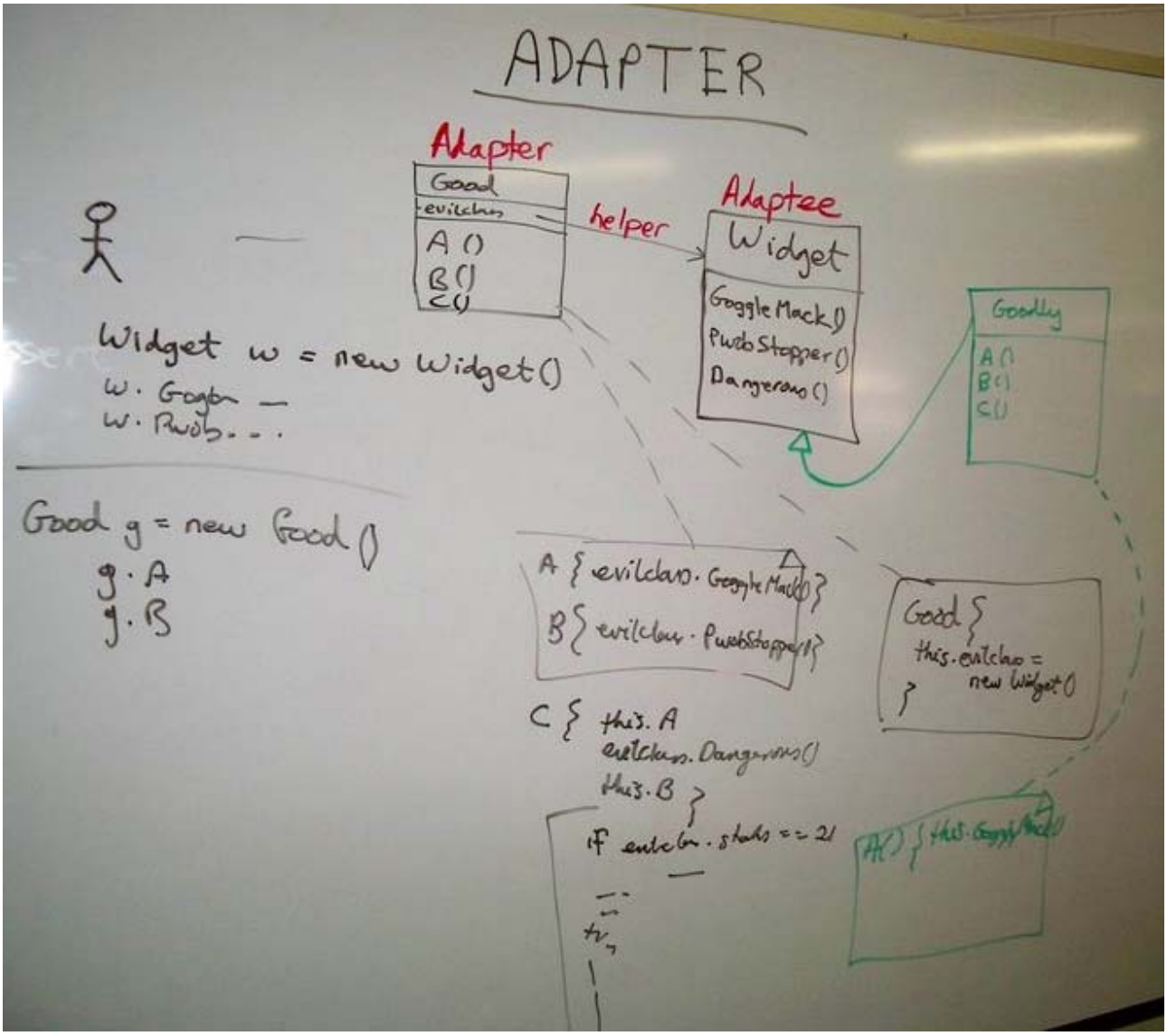


Figure 40 - Adapting the "evil class" containing bad method names like GoggleMack() and PwobStopper()

- Convert the interface of class into another interface more suitable.
- Also known as "Wrapper"
- Don't want to change the original code you are adapting:
  - e.g. because don't have the source code
  - e.g. or you don't want to affect existing legacy code which uses the code you want to adapt.

- Adds a level of indirection.
- Like strategy, but intent differs (Java Design Patterns p183).
- Can you add extra functionality into the adapting methods? Yes a little. But you *add* too much it arguably becomes decorator-like, if you *minimize* the new functionality it is just adapter.

Two classic implementations of adaptor – inherit vs. delegation

- Object Adapter vs. Class Adapter. Object Adapter adapts through delegation. Class Adapter adapts through subclassing. Sometimes known as the delegation flavour and the inheritance flavor.
- Class Adapter: Why inherit – well, target may have attributes and other methods that we still want to use. Thus in the new adapter class we get to use, the old methods are still available (both a benefit and a liability).
- Though… class adaption may not work if you are not able to inherit from the old class (e.g. the adapter already iherits from something). Typically the new adapter class inherits from the old class (the one you are adapting) and implements the new adapter interface.
- Class Adapter lets you update to a smarter API without existing client code knowing (since the old methods are still there).
- Object Adapter: Wrapper. Gets its work done by forwarding requests to the helper/adaptee. Delegation.

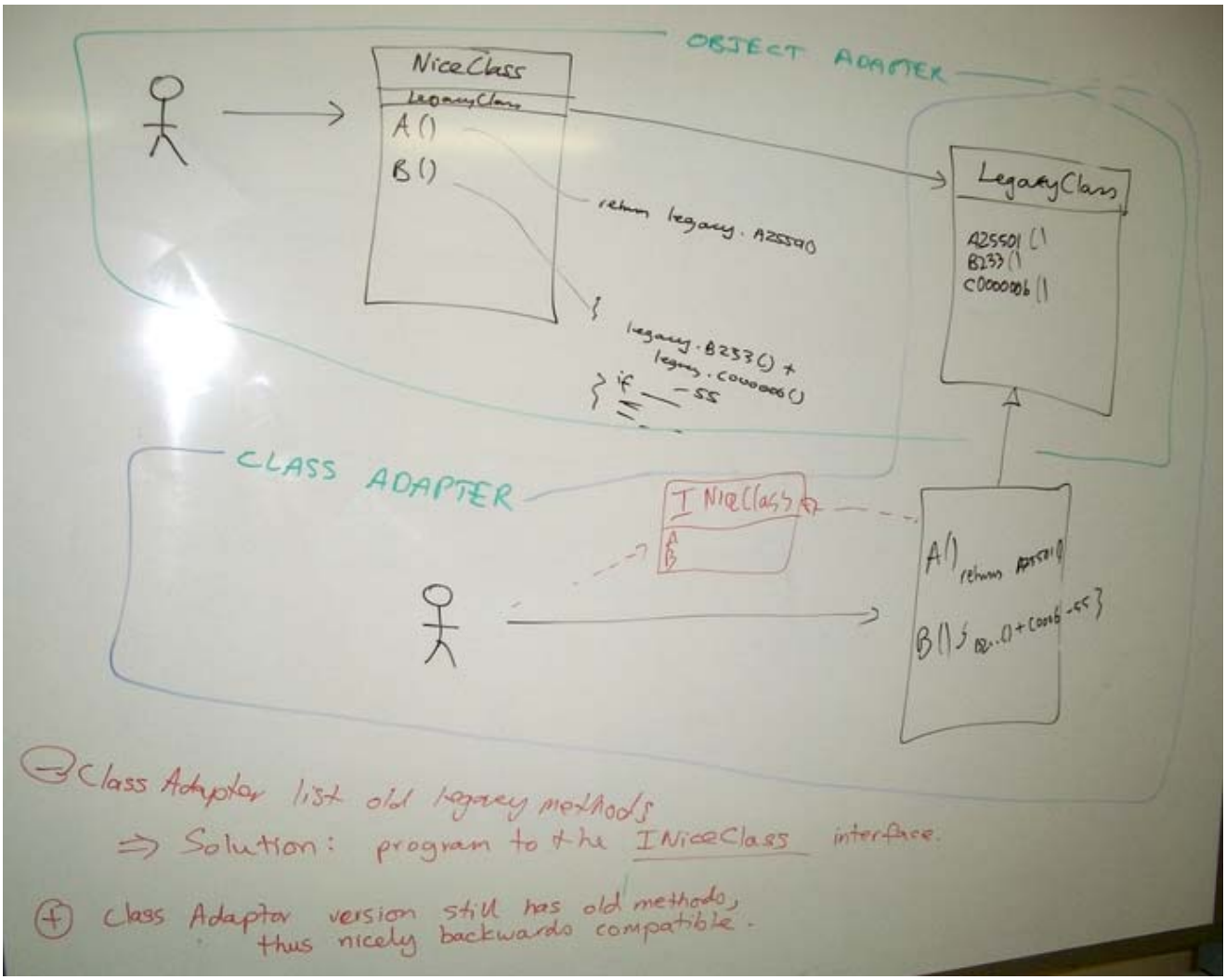


Figure 41 - class adapter vs object adapter

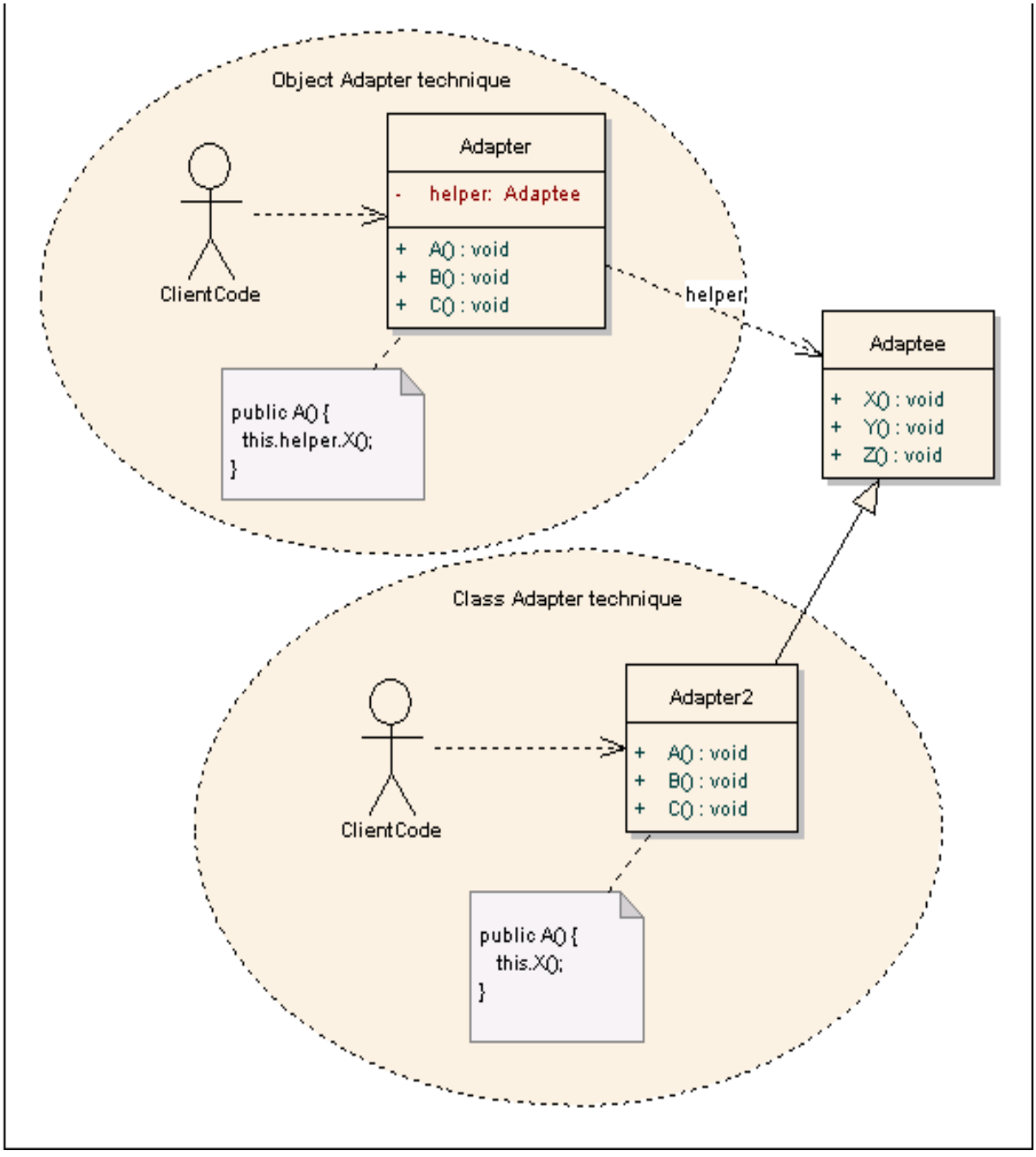
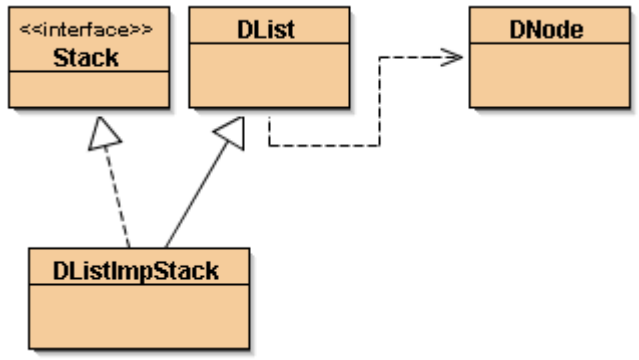


Figure 42 - Two implementation techniques for Adapter. Class adapter vs object adapter

Code Example:

Java example of adapting the DList class to the Stack interface.




```
/**
  * Java code sample
  */


 interface Stack
 {
   void push (Object o);
   Object pop ();
   Object top ();
 }



 /* DoubleLinkedList - not implemented fully though... */
 class DList
 {
   public void insert (DNode pos, Object o) { ... }
   public void remove (DNode pos) { ... }

   public void insertHead (Object o) { ... }
   public void insertTail (Object o) { ... }

   public Object removeHead () { ... }
   public Object removeTail () { ... }

   public Object getHead () { ... }
   public Object getTail () { ... }
 }
```

```
/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack
{
  public void push (Object o) {
    insertTail (o);
  }

  public Object pop () {
    return removeTail ();
  }

  public Object top () {
    return getTail ();
  }
}
```

Quiz Questions:

Question: Which version of adapter is being used in the above code example – object adapter or class adapter? **Answer:** Class adapter, since DListImpStack **extends** DList

Question: What class or interface should the client code talk to? **Answer:** Ideally the Stack interface, though legacy code could be using DList.

Non Software Example

International power adapters (the adapter) let you use your shaver (the client) with different wall sockets all around the world (the adaptees). You often need different adaptors for different countries (see discussion below on “Using a family of adapters” on page 146) – Andy

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, p161]

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, 149]

Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, pp219]

Advanced Ideas

Using a family of adapters

Here is an interesting use of adapter - a family of adapters - each concrete adapter adapts a specific class. Client uses the one interface. Is this pure adapter pattern or another pattern altogether?

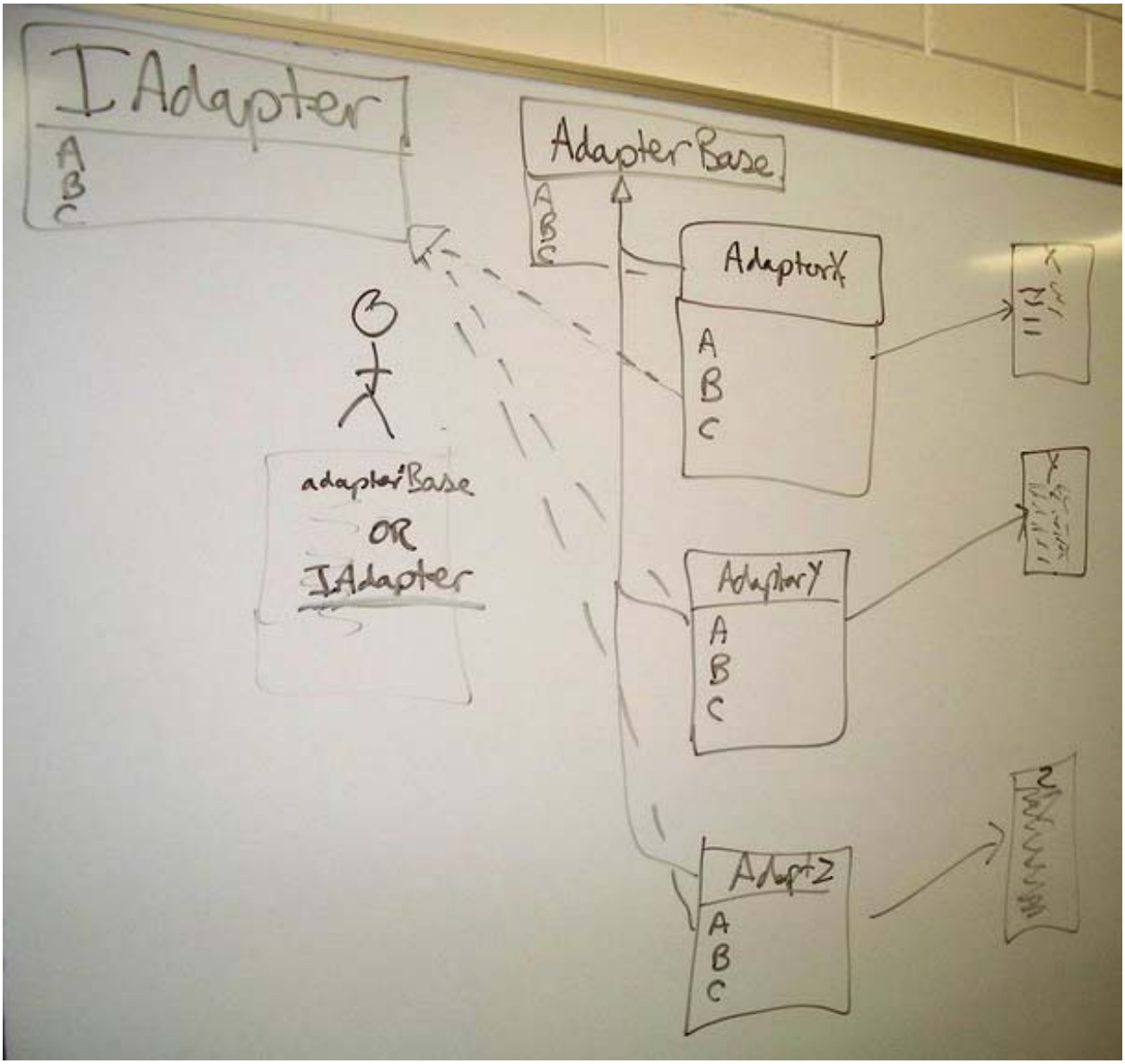


Figure 43 - Interesting use of adapter - Using a family of adapters - each concrete adapter adapts a specific class. Client uses the one interface. Is this pure adapter pattern or another pattern altogether?

Note that the use of **IAdapter** and **AdapterBase** in the above diagram are variants of the same idea – in both cases the client code talks to an abstract interface/class and is insulated from change – you can swap in different implementations without the client code knowing.

Note that the **Target** base class in the classic UML of Adapter (see Figure 39) is the same as the base class for the "family of adapters" illustrated above in Figure 43.

Architectural Idea for Adapter

Some argue that the edges of your architecture should touch your own adapters, so that you are insulated from the libraries and components you rely on. Easier said than done!

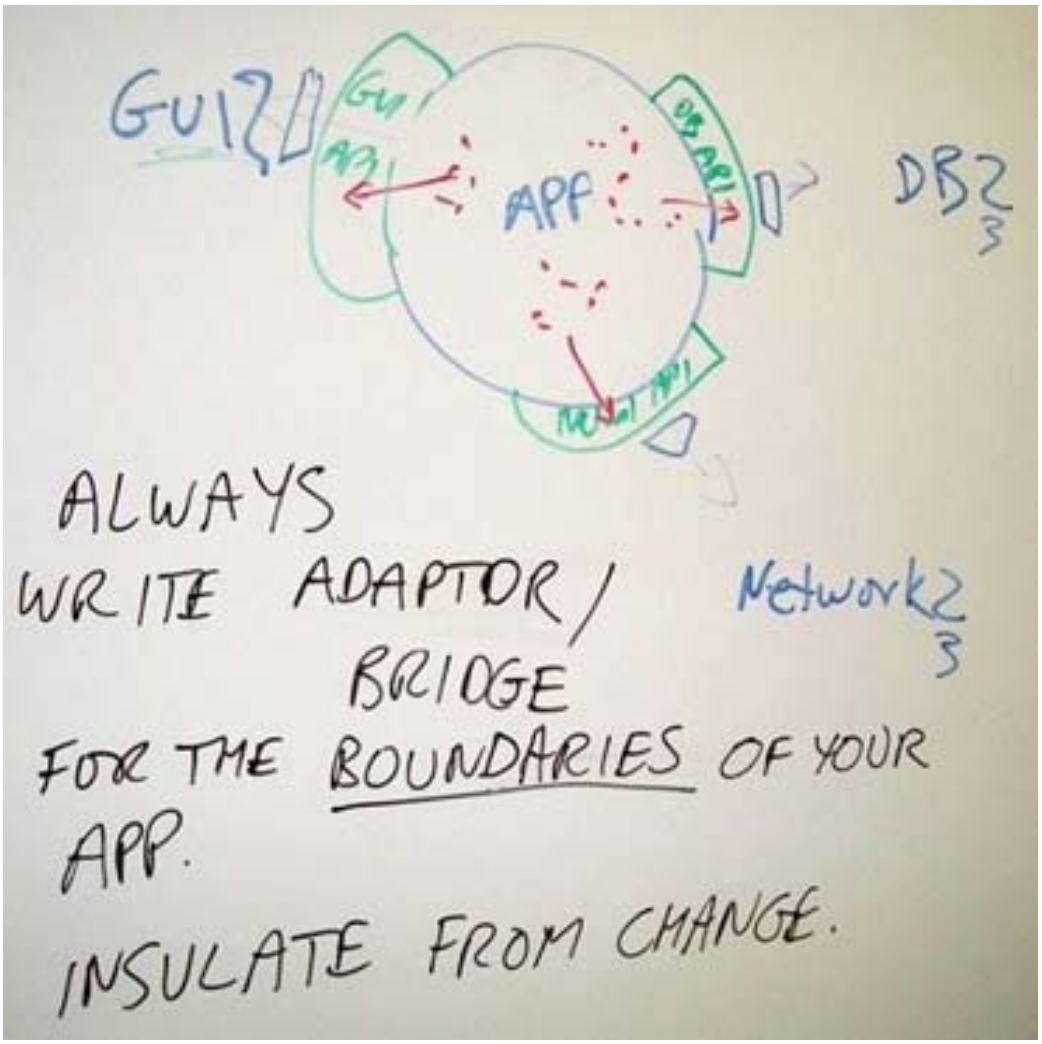


Figure 44 - Put Adapters at the edges of your architecture.

Notes:

Bridge

Separates an object's interface from its implementation. "Driver" Pattern.

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Ideas

- Component interfaces shall be decoupled from their implementation(s).
- The client shall be "insulated" from implementation changes. [Changes to implementation details should only require a re-link, not a re-compile.]
- The choice of implementation shall be configurable at run-time.
- Both the "interface" and the "implementation" shall be separately "open for extension, but closed for modification".
- "Interface" components that represent "implementation" components that are "expensive" and "equal" shall share these implementations, instead of duplicating them.

Participants

- Abstraction (BusinessObject)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- RefinedAbstraction (CustomersBusinessObject)
  - extends the interface defined by Abstraction.
- Implementor (DataObject)
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor (CustomersDataObject)
  - implements the Implementor interface and defines its concrete implementation.

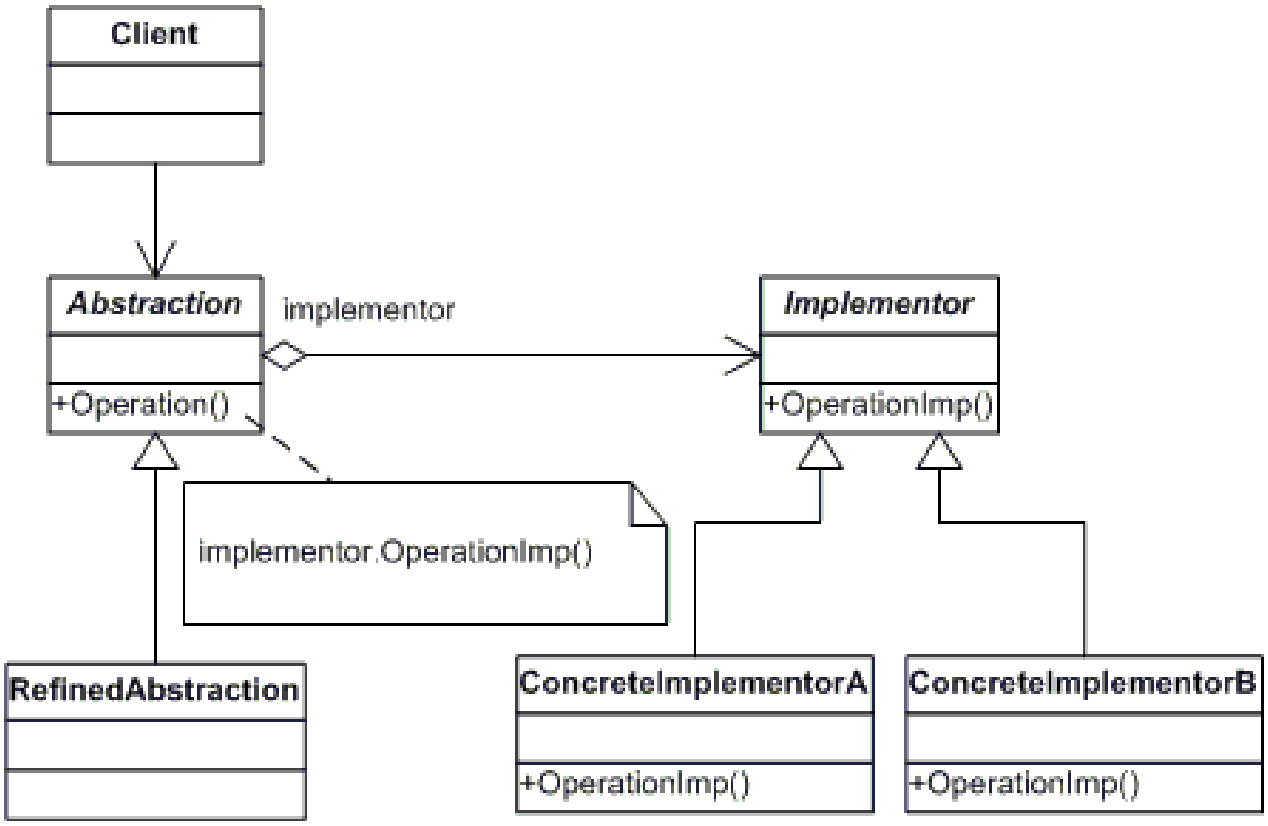


Figure 45. Pure Bridge Pattern

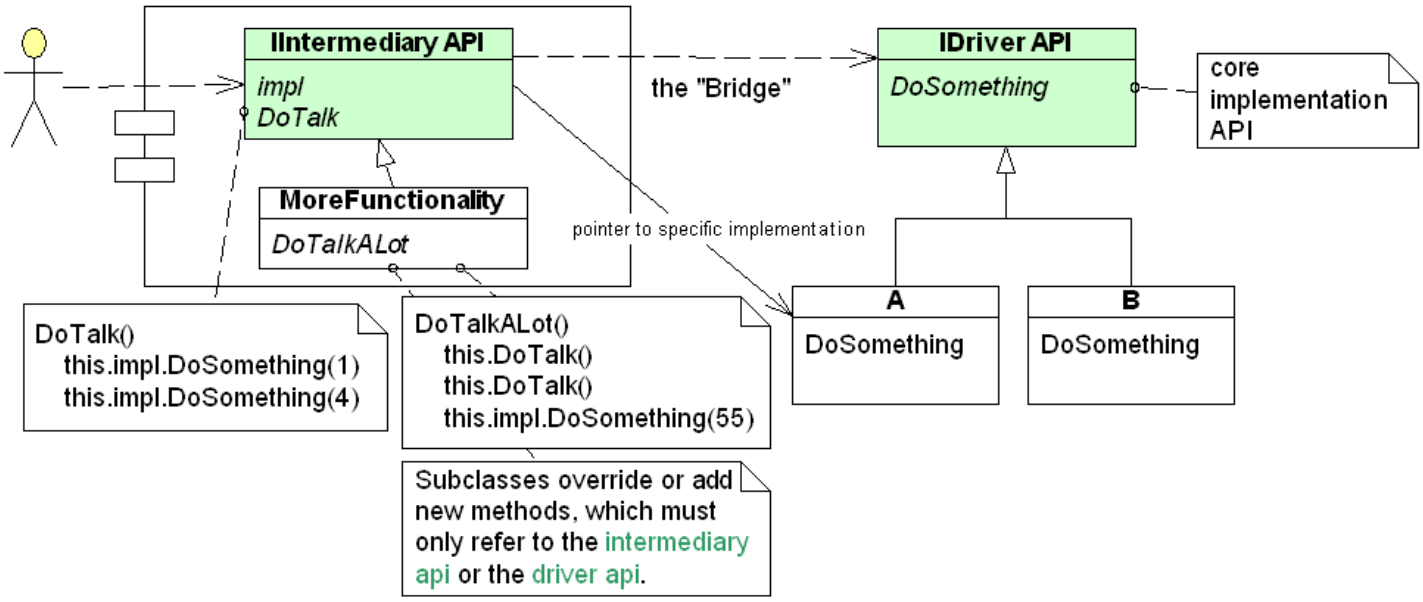


Figure 46. Bridge Pattern - slightly more detail

Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Consequences

These include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

Bridge is a synonym for the "handle/body" idiom [Coplien, C++ Report, May 95, p58]. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)." [Coplien, Advanced C++, p62]

Figure 47 - Ideas in Bridge Pattern

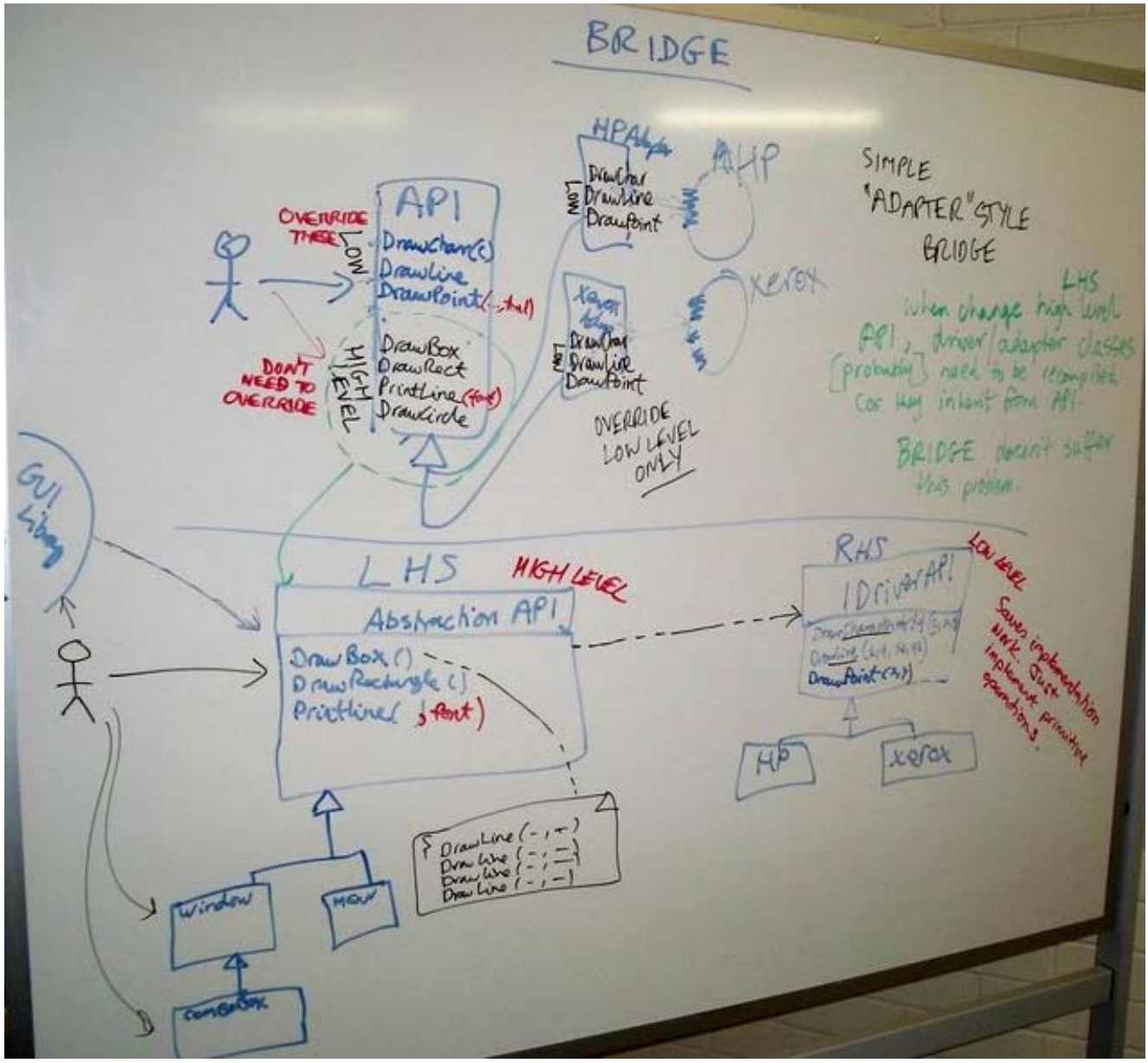


Figure 48 - More Ideas in Bridge Pattern

Naïve "BAD" initial design

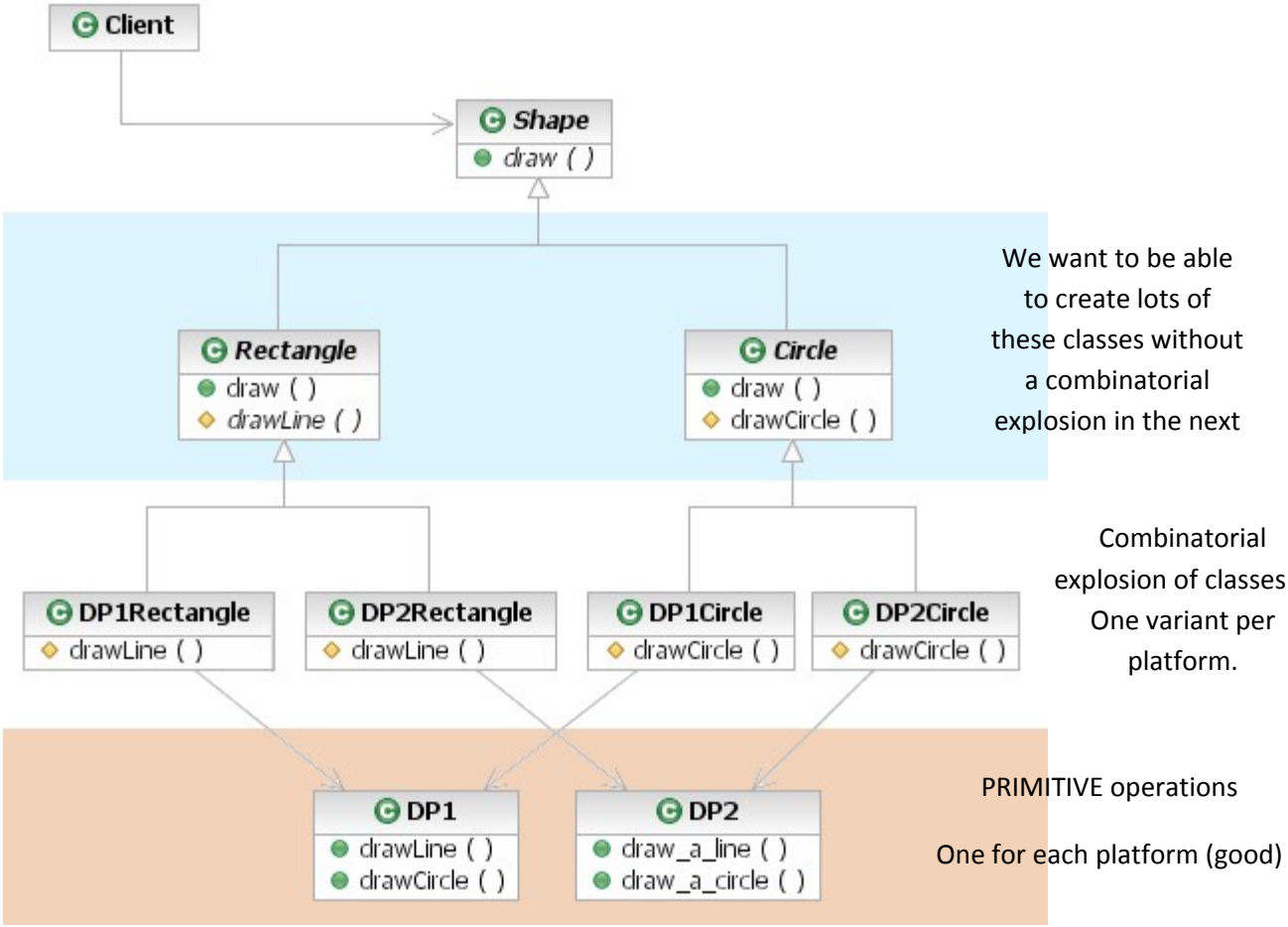


Figure 49. Naïve "BAD" initial design. (This is not the Bridge Pattern)

What are we trying to achieve?

We are trying to create a way of drawing rectangles and circles so that it works on both DP1 and DP2 where DP means "Drawing Platform" E.g. DP1 = Windows, DP2=Mac

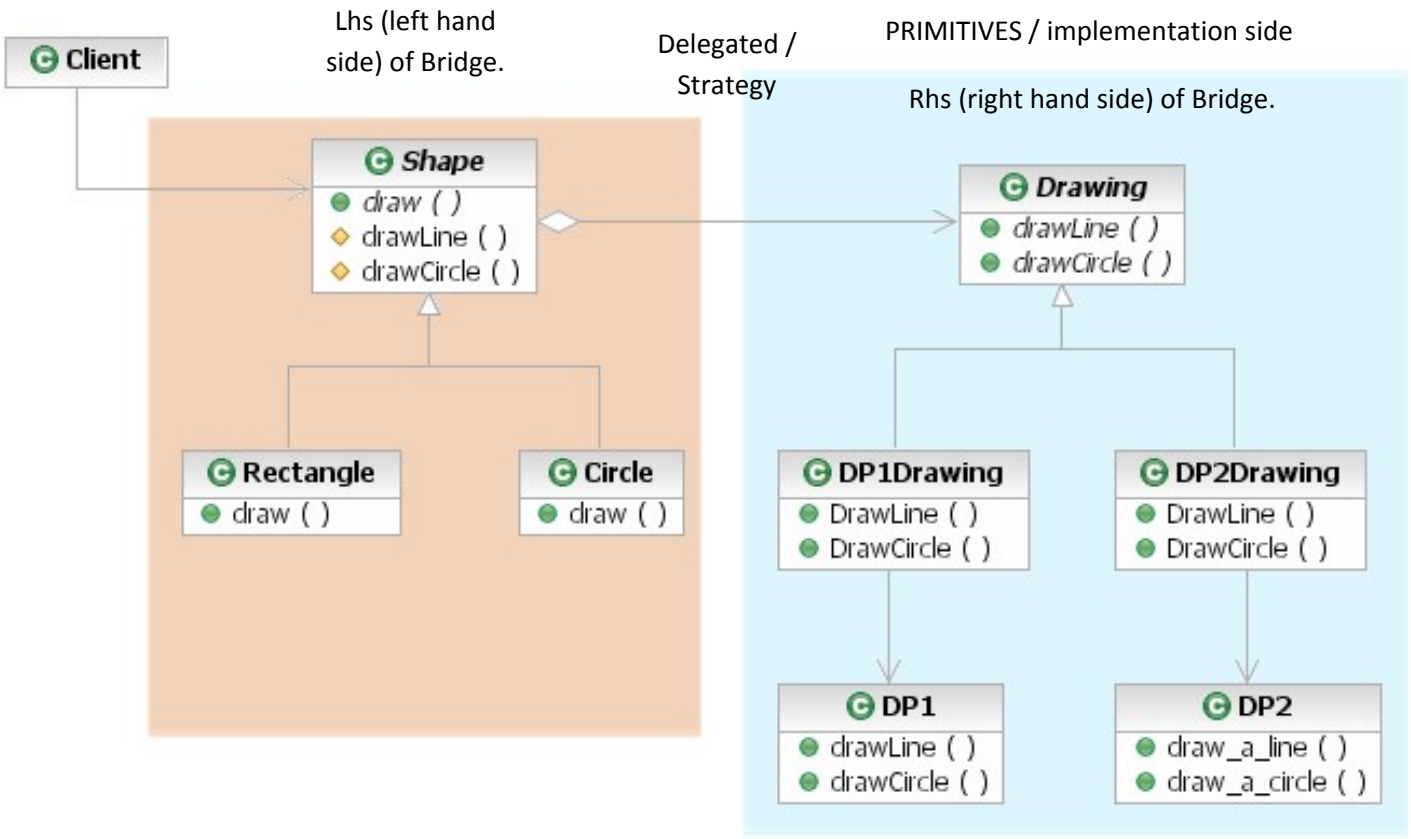
Why is this design good?

Well, at the lowest level of the UML diagram we have a nice bunch of primitive operations defined, one for each platform e.g. Mac/Windows. This seems clean. Also, we have organized our classes in an extensible way.

Why is this design bad?

- Too many classes are being created. Combinatorial explosion of classes !!
- If we want to add a new "Triangle" class, we would have to create 3 new classes e.g. "Triangle", "DP1Triangle" and "DP2Triangle"
- It would be nice if we could just create 1 new class e.g. for Triangle.

Better design—using Bridge Pattern



Why is this design better?

Easy to add a new shape e.g. Triangle. Just write it once on the lhs. and we are finished! We don't have to create a DP1Triangle and a DP2Triangle. The single Triangle class we write in terms of calls to the primitive methods on the rhs.

Imagine a combobox shape - this is quite a complex bit of programming. Being able to implement it only once, on the lhs, is a big benefit.

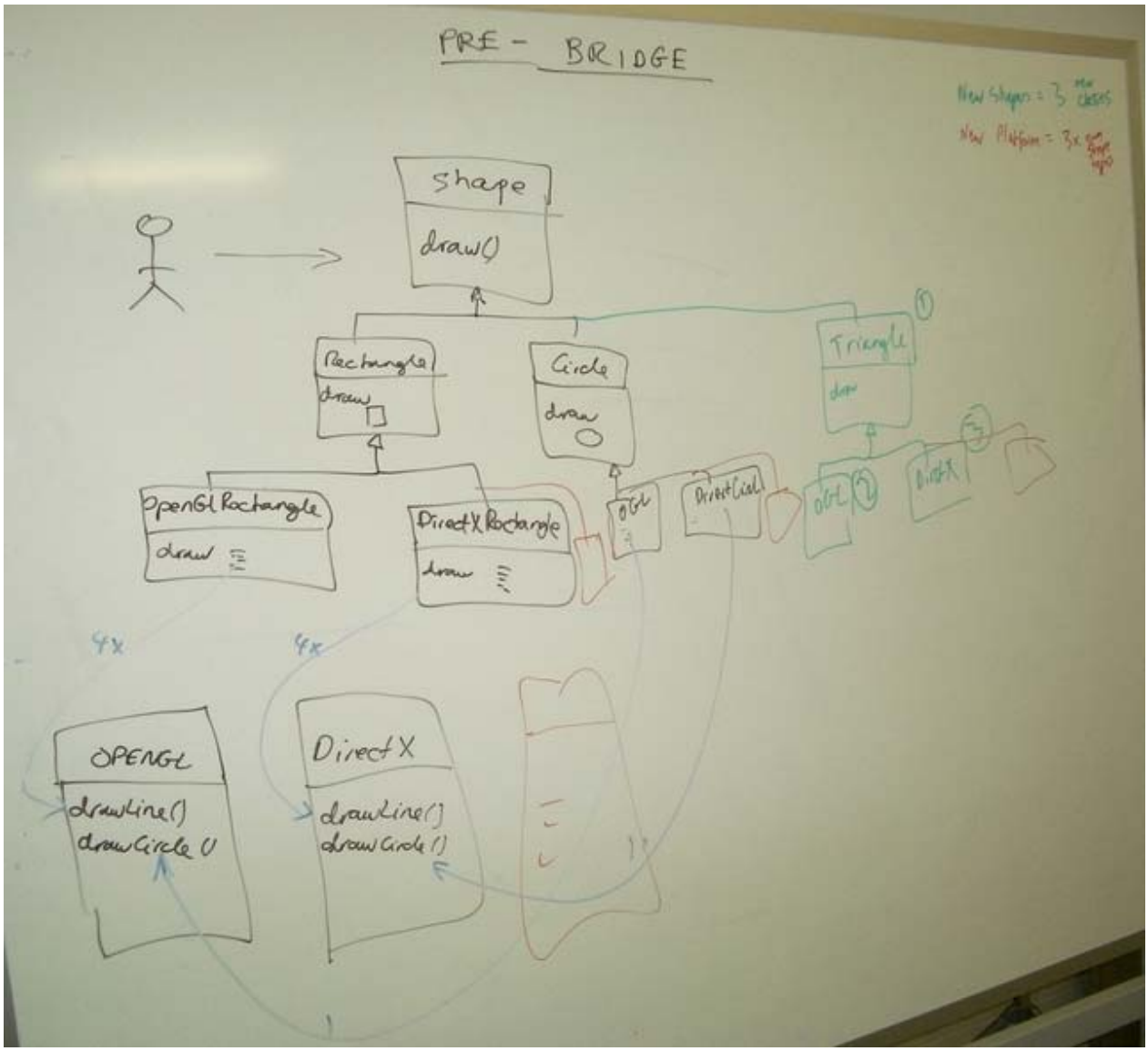


Figure 50 - Before Bridge

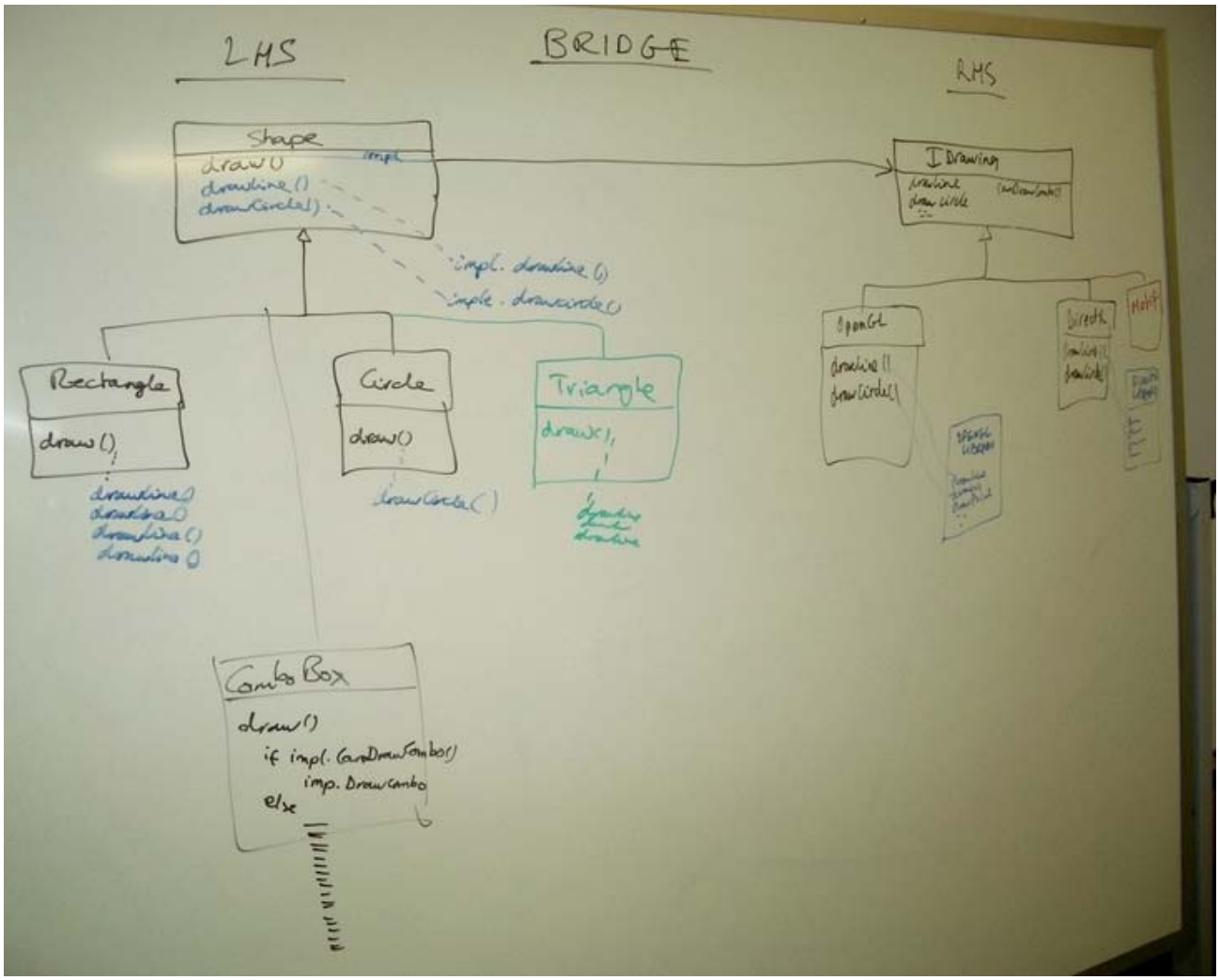


Figure 51 - After Bridge

What about the lowest common denominator problem?

How do we optimize and take advantage of a particular driver/rhs with a higher level feature? Bridge relies on a lowest common denominator API for the rhs, something all implementation platforms can easily supply.

POSSIBLE SOLUTION SUGGESTED BY ANDY: Have the rhs. implement a CanDoSmartThingBlah() method and have the lhs conditionally take advantage of it. So if the rhs. can draw a combobox shape natively then our lhs. Draw() code would:

```
if impl.CanDrawCombo()

    impl.DrawCombo()

else {

    do it the hard way
    in terms of multiple calls to rhs. primitives.
    ….

}
```

We would have to add CanDrawCombo() and DrawCombo() to the rhs. Drawing interface. Thus need to expose both in the rhs. Implementation.

```
--------------------
        IRhs
--------------------
CanDrawCombo() -> t/f
DrawCombo(…)
--------------------
```

Andy's Tips

- Bridge is closely related to the Adapter design pattern. However Adapter is usually as thin as possible on the lhs. vs. the lhs in Bridge is more complex, thick and hierarchical.
- This is a variant on accessing different behavior via an intermediary.
- Designed to separate a class's interface from its implementation, so that you can vary or replace implementations without changing client code (where 'client code' refers to any code that uses the interface in question.
- For this trick to work, the 'client code' must only talk to the interface/abstraction.
- The abstraction / interface the client uses, in turn uses the interface of the implementation. So there are two interfaces, the left hand side (LHS) and the right hand side (RHS). The implementation of the left hand side interface uses the interface of the right hand side.
- Looks like strategy, Bridge is just strategy with a oversized lhs context. Same as strategy except there is massive subclassing going on on the lhs (the 'context' side). The nature of the lhs methods are more compositional, adaptive and far reaching (not just a simply strategy

delegation)

- Abstract Factory often used to create one particular set of implementation classes on the right hand side.
- Examples are device drivers or database drivers. They force you to abstract a common model of the API. We lose some driver specific features.
- Often the LHS interface API is a higher level of abstraction e.g. “drawing a rectangle“. The RHS implementation API is often a more primitive abstraction e.g. “drawing a line“ or “drawing a pixel”. e.g.

```
DrawRect() {
    // draw a rectangle by drawing a line four times
      this.impl.DrawLine(a,n);
      this.impl.DrawLine(n,m);
      this.impl.DrawLine(m,z);
      this.impl.DrawLine(z,a);
}
```

Andy’s Tips – part II

There is massive subclassing going on in the lhs. context

The reason there is massive subclassing going on in the lhs. context is that you are wanting lots of methods and lots of functionality, lots of classes. E.g. you usually are building a complex GUI or DB subsystem, not just a single strategy.

The nature of the lhs and rhs methods

Typically rhs (implementation/driver) calls are more primitive, and one lhs method will call the rhs. many times. e.g. see the DoTalk() method, above. The lhs methods can be diverse, comprising

- lhs method simply calls rhs method. Method names can change or be the same. Simple delegation with no extra work.
- lhs methods more complex and adapt and do extra lines of code as needed
- Lots of logic in the lhs methods and may have associated helper classes. But in the end they call stuff on the intermediary api.

Insulated from change. Allow lhs and rhs to vary independently.

Client is insulated from changes. Should not talk talk to implementation, even if it is the abstract implementation interface because the abs impl. may change. If the abstract implementation interface does change then this affects only the Intermediary but not the client code. Client code should thus only talk to intermediary.

Similarly, if you change the Intermediary API, then only the client is affected - the r.h.s. (the abstract implementation interface and concrete implementations) are not affected.

In this sense the lhs and rhs can vary independently. Ok - so there are repercussions when things vary - but they are limited, as discussed above.

Rules of thumb

Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, 216]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58]. They differ in intent - that is, they solve different problems.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, C++ Report, May 95, p58]

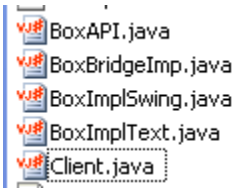
If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects. [Grand, Patterns in Java, p196]

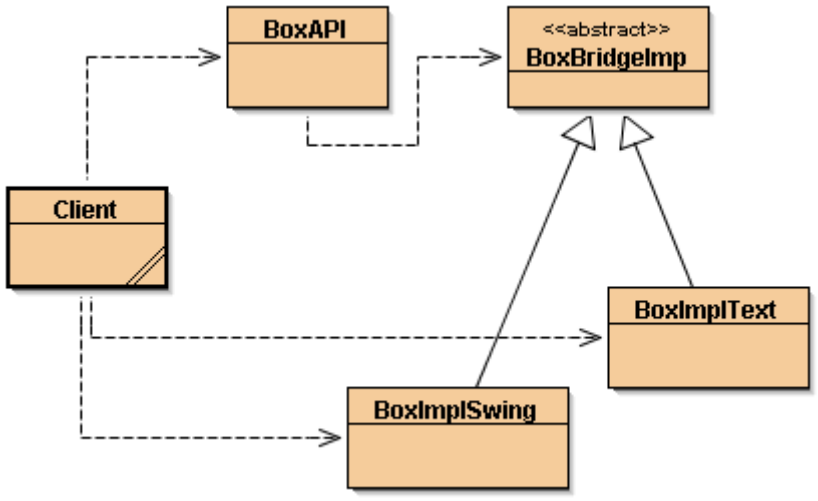
Non Software Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Code Ideas

www.dofactory.com's real-world code demonstrates the Bridge pattern in which a BusinessObject abstraction is decoupled from the implementation in DataObject. The DataObject implementations can evolve dynamically without changing any clients.






```
public class Client {

	public static void main(String[] args) {
		// Hidden - use configuration file or registry
		// or factory method.
		BoxAPI api = new BoxAPI();	// <- “left hand” side of bridge

		/* Choose implementation - or could be in a config file */
		api.SetImplementation(new BoxImplText());
		//api.SetImplementation(new BoxImplSwing());

		api.DrawBox(1,1,10,10);

		System.out.println("Done");
	}
}

public class BoxAPI{
    protected BoxBridgeImp imp;

	public void SetImplementation(BoxBridgeImp imp){
		this.imp = imp;
	}

	public void DrawBox(int fromx, int fromy, int tox, int toy) {
		this.imp.DrawLine(fromx, fromy, tox, fromy);
		this.imp.DrawLine(tox, fromy, tox, toy);
		this.imp.DrawLine(tox, toy, fromx, toy);
		this.imp.DrawLine(fromx, toy, fromx, fromy);
	}
}

public abstract class BoxBridgeImp {
	public abstract void DrawLine(int fromx, int fromy, int tox, int toy);
}

public class BoxImplText extends BoxBridgeImp {
```

```
	public void DrawLine(int fromx, int fromy, int tox, int toy) {
		// System.out.println("drawing a text line");

		// Ok we cheat a bit here and don't draw individual lines
		// instead we spit out the entire box in one hit.
		if ((fromx == 1) && (fromy == 1) ) {
			System.out.println("------------");
			System.out.println("|          |");
			System.out.println("|          |");
			System.out.println("------------");
		}
	}
}

import java.awt.Graphics;
import java.util.ArrayList;
import java.util.Iterator;

import javax.swing.JFrame;
import javax.swing.JPanel;


public class BoxImplSwing extends BoxBridgeImp {
	ArrayList drawlist = new ArrayList();
	MyFrame f = new MyFrame(drawlist);

	public BoxImplSwing() {
		f.setVisible(true);
	}

	class MyRect {
		public int fromx, fromy, tox, toy;
		int SCALE = 15;
		public MyRect(int fromx, int fromy, int tox, int toy){
			this.fromx = fromx *SCALE;
			this.fromy = fromy *SCALE;
			this.tox = tox *SCALE;
			this.toy = toy *SCALE;
		}
	 }

	 public void DrawLine(int fromx, int fromy, int tox, int toy) {
		System.out.println("drawing a swing line");
		drawlist.add(new MyRect(fromx, fromy, tox, toy));
		//f.invalidate();
	 }

	 public class MyFrame extends JFrame
	 {
		public MyFrame(ArrayList drawlist) {
			super();
			MyPanel jp = new MyPanel(drawlist);
			this.getContentPane().add(jp);
			this.setSize(300,300);
	}
	 }
```

```
        class MyPanel extends JPanel
        {
              ArrayList drawlist;

              public MyPanel(ArrayList drawlist) {
                    super();
                    this.drawlist = drawlist;
              }
              public void paint (Graphics g) {
                    for (Iterator i = this.drawlist.iterator(); i.hasNext(); )
{
                          MyRect r = (MyRect) i.next();
                          g.drawLine(r.fromx, r.fromy, r.tox, r.toy);
                    }
              }
        }

}
```

Output:

By changing this line, we get two different implementations of a box.

```
              api.SetImplementation(new BoxImplText());
```

or

```
              api.SetImplementation(new BoxImplSwing());
```

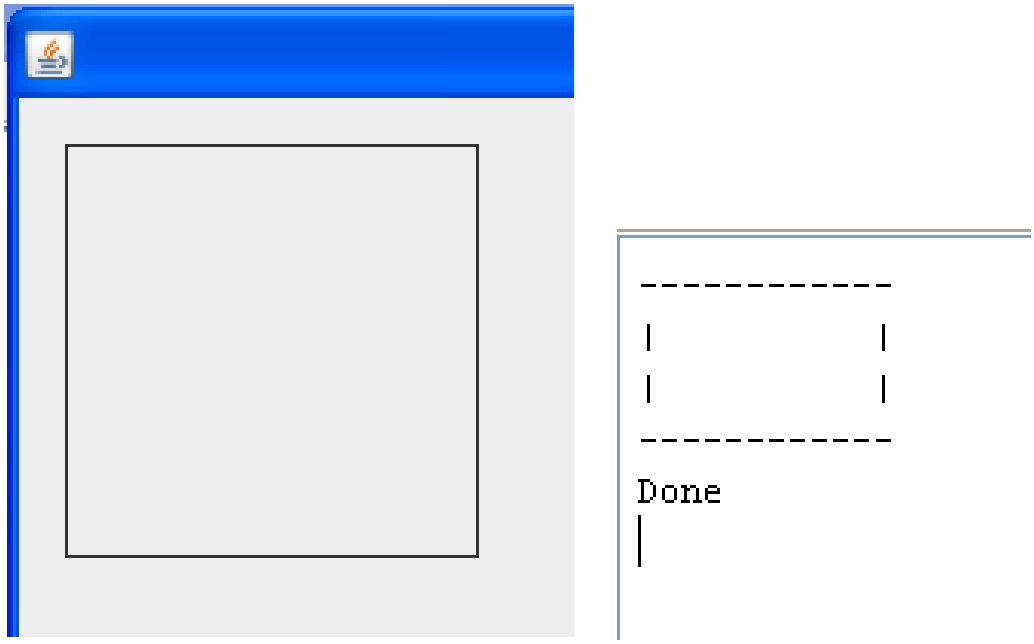


Figure 52 - Bridge pattern - drawing a box - switching between the swing and the text "driver".

A Final thought on Bridge

You could simplify Bridge and have the client code talk directly to the rhs. abstract implementation interface. You would be reverting to where we started on this "road to Bridge". Nothing wrong with that - but you would lose the 'insulation against change' that Bridge gets you. And with Bridge the lhs can have lots of complex logic and the rhs implementations need only implement the more primitive operations. That is a big win.

Notes:

Mediator

Defines centralised communication between classes. Mediator untangles the spaghetti of references between classes.

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.

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

Discussion

Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

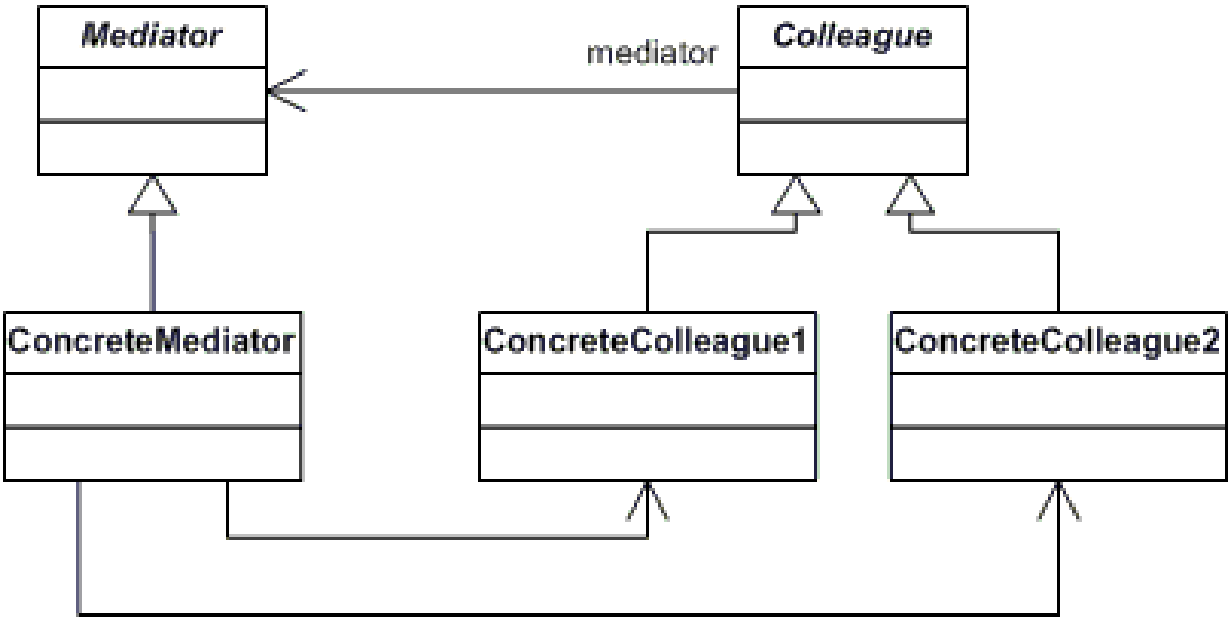
The Mediator pattern promotes a "many-to-many relationship network" to "full object status". Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.

The mediator pattern consists of a mediator class that is the only class that has detailed knowledge of the methods of other classes. Classes send messages to the mediator when needed and the mediator passes them on to any other classes that need to be informed. The mediator class promotes looser coupling between a number of other classes.

Participants

The classes and/or objects participating in this pattern are:

- Mediator (IChatroom)
  - defines an interface for communicating with Colleague objects
- ConcreteMediator (Chatroom)
  - implements cooperative behavior by coordinating Colleague objects
  - knows and maintains its colleagues
- Colleague classes (Participant)
  - each Colleague class knows its Mediator **object each colleague communicates with its mediator** *whenever it would have otherwise communicated with another colleague*.


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Andy's Tips

- Define an object that encapsulates how a set of objects interact.
- Promote loose coupling by keeping objects from referring to each other explicitly. "Decouples colleagues".
- Isolate complex inter-relationships in the one mediating class. Centralizes control. Lets you vary their interaction independently of the collegues – inside the Mediator.
- Each Colleague class knows its Mediator. Each Colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.
- Colleagues send and receive requests from a mediator object and the mediator routes requests between the appropriate colleagues.
- Can implement Mediator using the Observer pattern. Colleagues are subjects and Mediator is the one observer. E.g. Colleagues (subjects) notify the mediator (observer) when they change state or want to broadcast some information. The Mediator responds by propogating the effects of the change to the other colleagues.

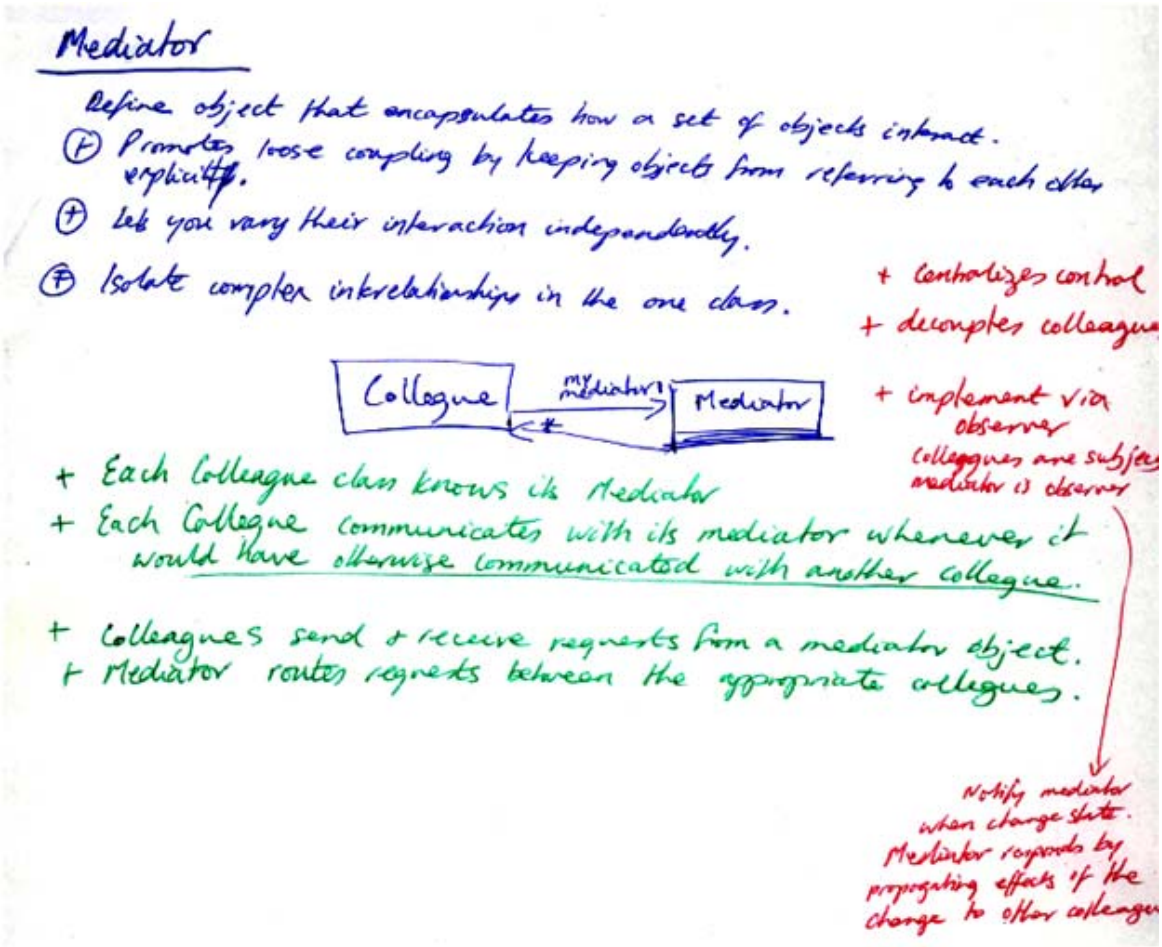


Figure 53 - Andy's Tips in handwritten form.

Ideas

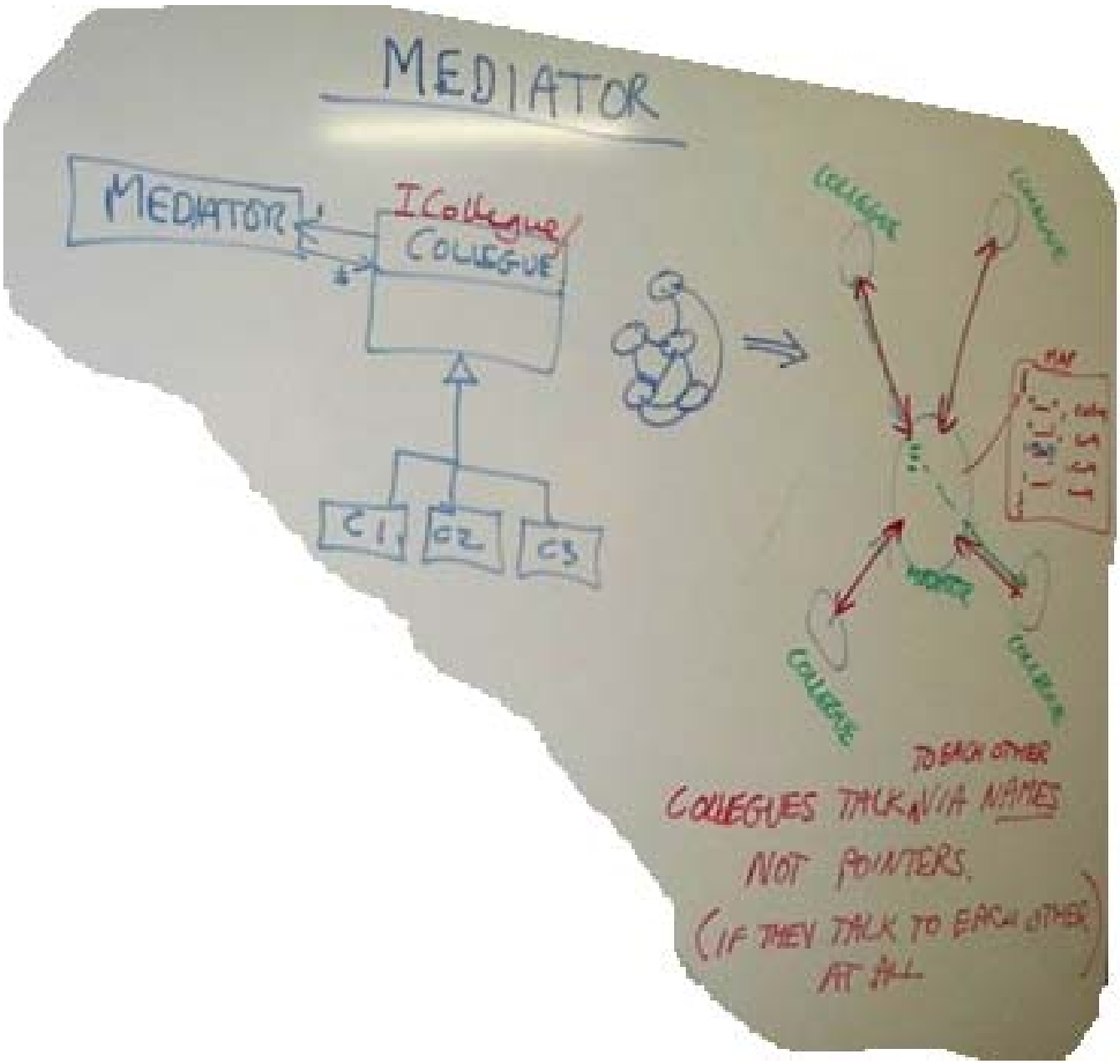


Figure 54 - Mediator UML. Also illustrates the object wiring before and after Mediator.

- The system shall encapsulate complex, many-to-many coupling between "peer" components in a separate component capable of allowing the "peers" to be: disengaged, replaced, and reused.
- The system shall support numerous many-to-many "mappings" to be defined, installed, and exchanged by the client.
- The system shall balance the distribution of intelligence emphasized by "logical" OO design with the centralization of intelligence often required by "physical" large scale design.
- "Senders" and "receivers" shall be decoupled from one another.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

Examples

Chatroom

www.dofactory.com's real-world code demonstrates the Mediator pattern facilitating loosely coupled communication between different Participants registering with a Chatroom. The Chatroom is the central hub through which all communication takes place. At this point only one-to-one communication is implemented in the Chatroom, but would be trivial to change to one-to-many.

User and group capability in an operating system

An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a centralized, flexible and non-invasive way to associate and manage users and groups. **Group/User Manager** is the mediator. **Users** and **groups** are collegues.

Non Software Example - Air Traffic Control

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

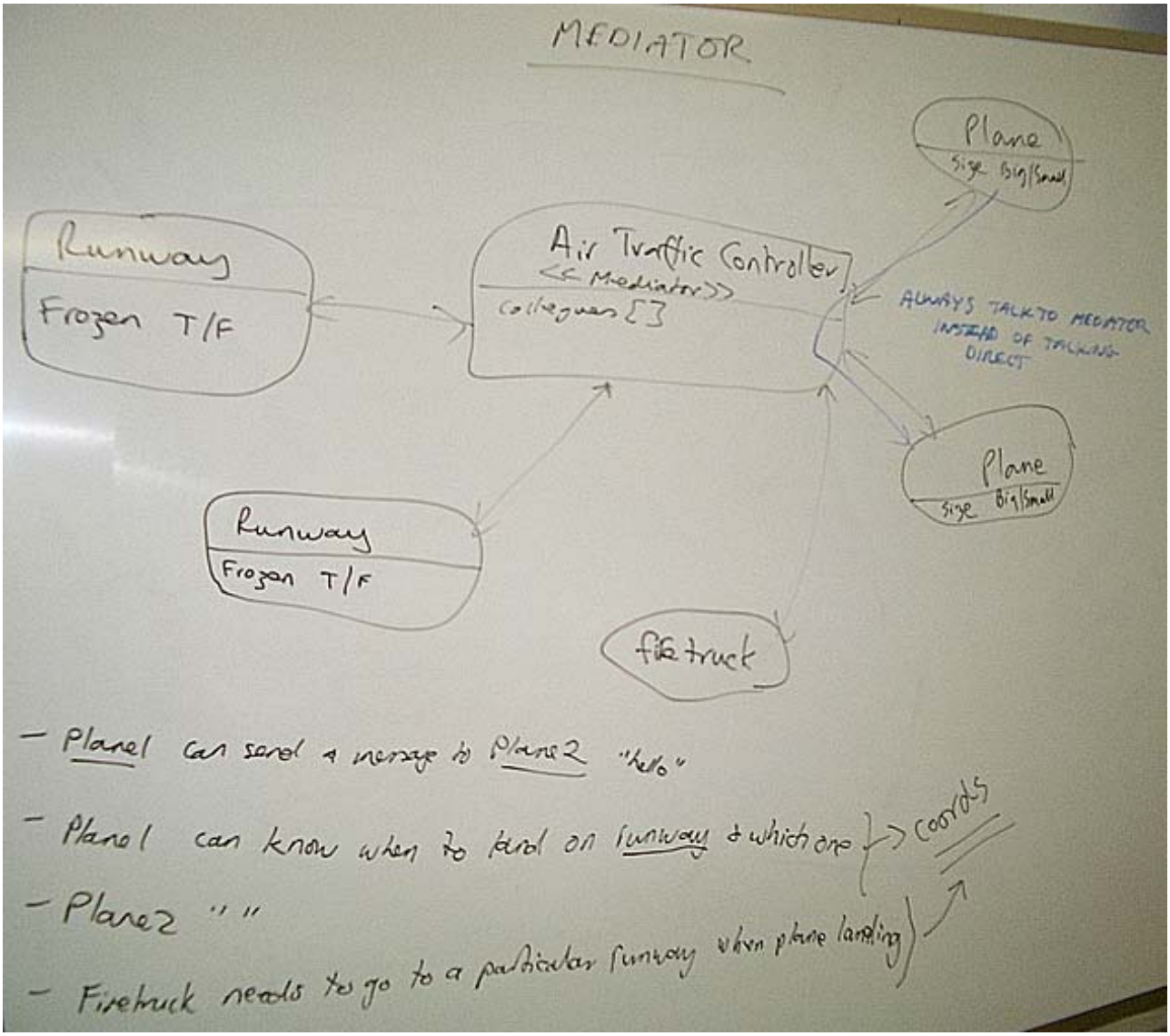


Figure 55 - . The control tower at a controlled airport is a mediator. The plane, fire truck and runway are all collegues.

Code: Air Traffic Control in Python

So let's implement the non software example in software! ☺

```
class Colleage:
    pass

class Plane(Colleage):
    def __init__(self, name):
        self.name = name

    def SendMessage(self, to_name, msg):
        self.mediator.SendMessageForMe(self.name, to_name, msg)
```

```
    def ReceiveMessage(self, from_name, msg):
        print self.name + " received message " + msg + " from " + from_name

    def LandPlaneAt(self, coords):
        print "Landing at", coords

    def RequestToLand(self):
        self.mediator.RequestToLand(self)

class Runway(Colleage):
    def __init__(self, coords):
        self.coords = coords
        self.frozen = False
        self.occupied = False

    def SetFrozenState(self, isfrozen):
        self.frozen = isfrozen

    def SetOccupied(self, isoccupied):
        self.occupied = isoccupied

class FireTruck(Colleage):
    pass

class AirTrafficController:   # Mediator
    def __init__(self):
        self.planes = []
        self.runways = []
        self.firetrucks = []

    def AddPlane(self, plane):
        self.planes.append(plane)
        plane.mediator = self

    def AddRunway(self, runway):
        self.runways.append(runway)
        runway.mediator = self

    def AddTruck(self, firetruck):
        self.firetrucks.append(firetruck)
        firetruck.mediator = self

    def GetAllPlanes(self):
        return self.planes

    def SendMessageForMe(self, from_name, to_name, msg):
        print "Controller got request to send msg " + msg
```

```
        foundPlane = False
        for plane in self.planes:
            if plane.name == to_name:
                plane.ReceiveMessage(from_name, msg)
                foundPlane = True
        if not foundPlane:
            print "COULDN'T FIND", to_name
            #raise "no plane mate"

    def RequestToLand(self, plane):
        foundRunway = False
        for runway in self.runways:
            if not runway.frozen and not runway.occupied:
                plane.LandPlaneAt(runway.coords)
                foundRunway = True
                runway.SetOccupied(isoccupied=True)
                break
        if not foundRunway:
            plane.ReceiveMessage("controller", "No runways free mate, sorry."
)


# TEST
airTrafficController = AirTrafficController()
plane1 =    Plane("Boeing1")
plane2 =    Plane("Quantas50")
runway1 =   Runway((50,60))
runway2 =   Runway((2000, 3000))
firetruck = FireTruck()

airTrafficController.AddPlane(plane1)
airTrafficController.AddPlane(plane2)
airTrafficController.AddRunway(runway1)
airTrafficController.AddRunway(runway2)
airTrafficController.AddTruck(firetruck)

plane1.SendMessage("Quantas50", "Hello")
plane1.SendMessage("Quantas5099999", "Hello")

runway1.SetFrozenState(True)
plane1.RequestToLand()
plane1.RequestToLand()
plane1.RequestToLand()

print "Done"
```

Output:

```
Controller got request to send msg Hello
Quantas50 received message Hello from Boeing1
Controller got request to send msg Hello
COULDN'T FIND Quantas5099999
Landing at (2000, 3000)
Boeing1 received message No runways free mate, sorry. from controller
Boeing1 received message No runways free mate, sorry. from controller
Done
```

Advanced: Is a GUI form a Mediator?

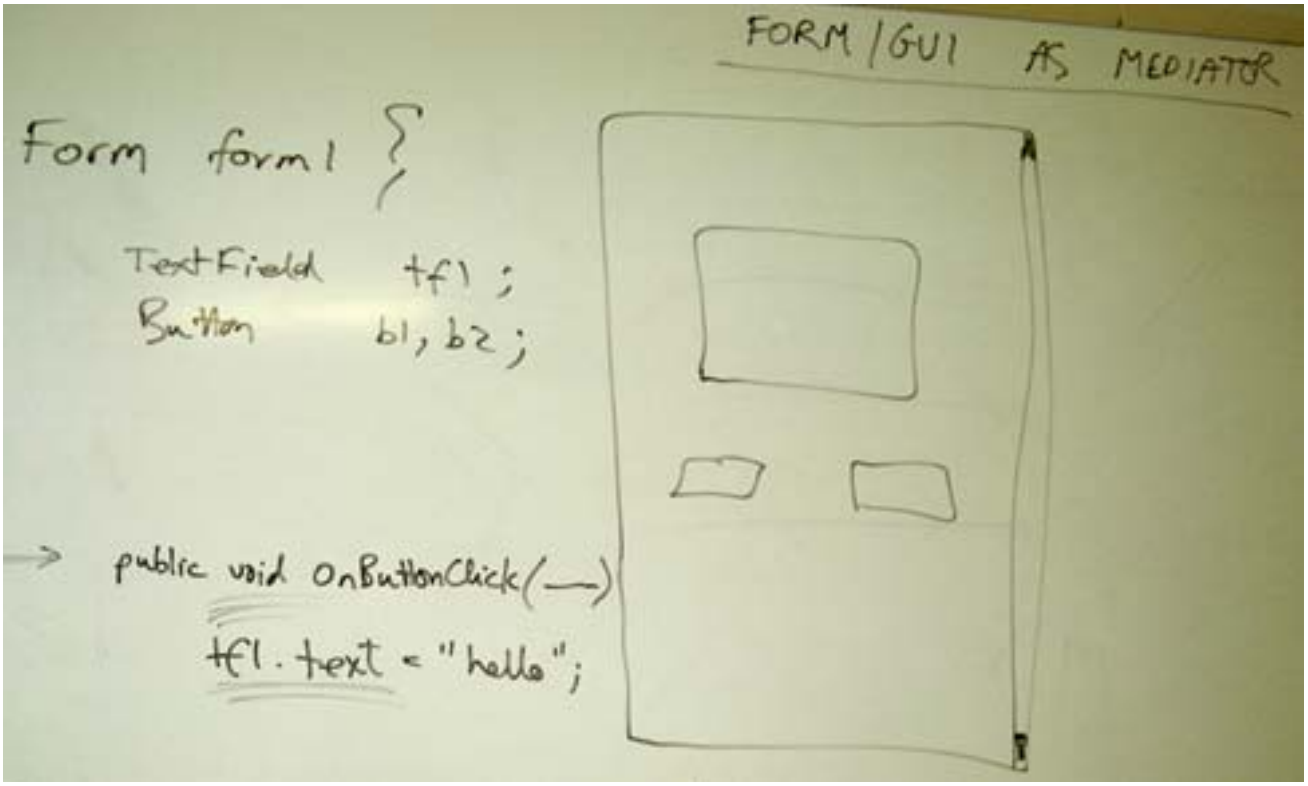


Figure 56 - A GUI form as mediator.

A GUI form (e.g. Swing form, Winform, Delphi TForm etc) could be though of as a mediator since GUI components ("colleagues") on the form don't know about each other (they are, after all, standard components). However their event handlers are housed inside the one form, thus the form (incl. event handler code) is a mediator. E.g. a button click causes text to appear in a label—the mediator makes this happen without the button GUI component knowing anything about the label GUI component.

Remember, the components on the form are “generic” and don’t know about each other. Its only the event mechanism and the code behind the events (housed in the form) that links the components together. It’s the code in the form that wires up the components to talk to each other. Thus the form is a mediator. *Note: Are we possibly dealing here with nested mediators? - with e.g. a panel component being both a colleague (within the form) and itself a mediator for its own panel contents? Or probably the organization is flat and all events go to the form level, thus the form is the only mediator.*

Advanced: Is a centralized message kernel a mediator?

In some architectures, objects want to talk to each other using events sent to a centralized message kernel. Such a centralized message kernel is a mediator.

Figure 57 - a centralized message kernel is a mediator

Related Pattern: Relationship Manager

http://www.atug.com/andypatterns/rm.htm

The Relationship Manager is a central mediating class which records all the one-to-one, one-to-many and many-to-many relationships between a group of selected classes.

Problem

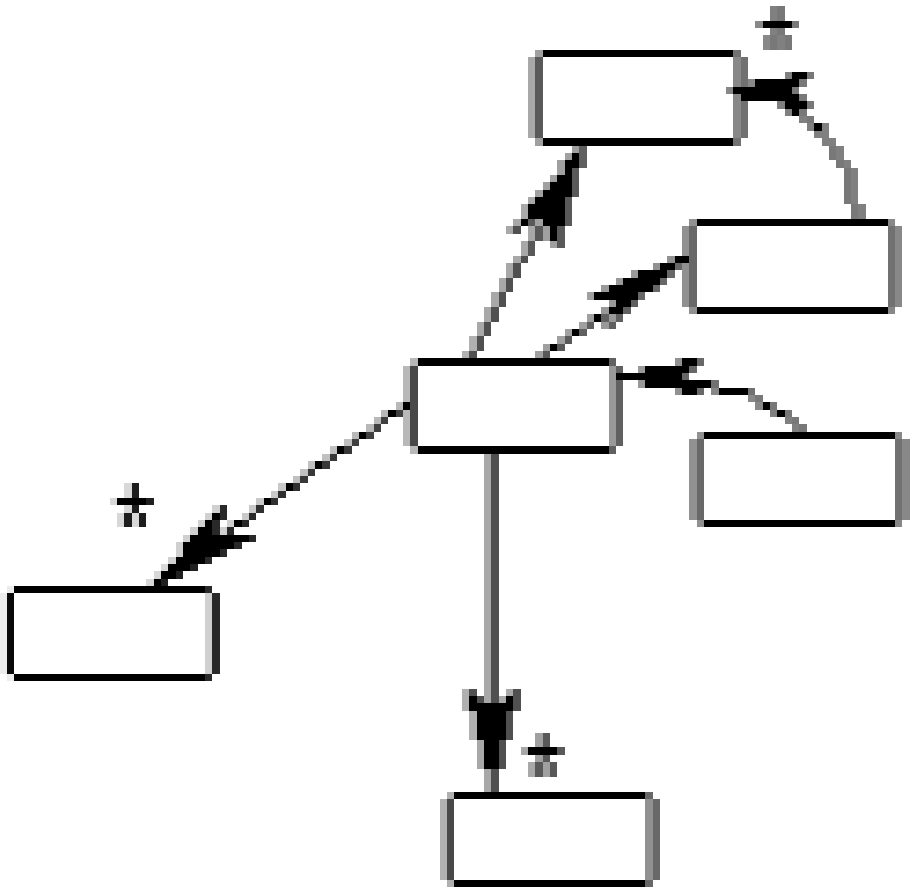


Figure 58 - How to implement a rat's nest of relationships?

- How do you robustly implement the required relationships between your classes with as little code as possible - minimising the number of wiring idioms you need to learn?
- How do you avoid tightly coupled classes full of code dedicated to maintaining relationships - tedious code which reduces the cohesiveness of your classes?
- How do you ensure that your relationship related code is exceptionally easy to read, modify and maintain by existing and especially new programmers joining the project?

Solution

Use a Mediator object - a central relationship manager to record all the one-to-one, one-to-many and many-to-many relationships between a group of selected classes. Provide a query method on this

mediating Relationship Manager. Implement all relationship code functionality by calling methods on the central Relationship Manager.

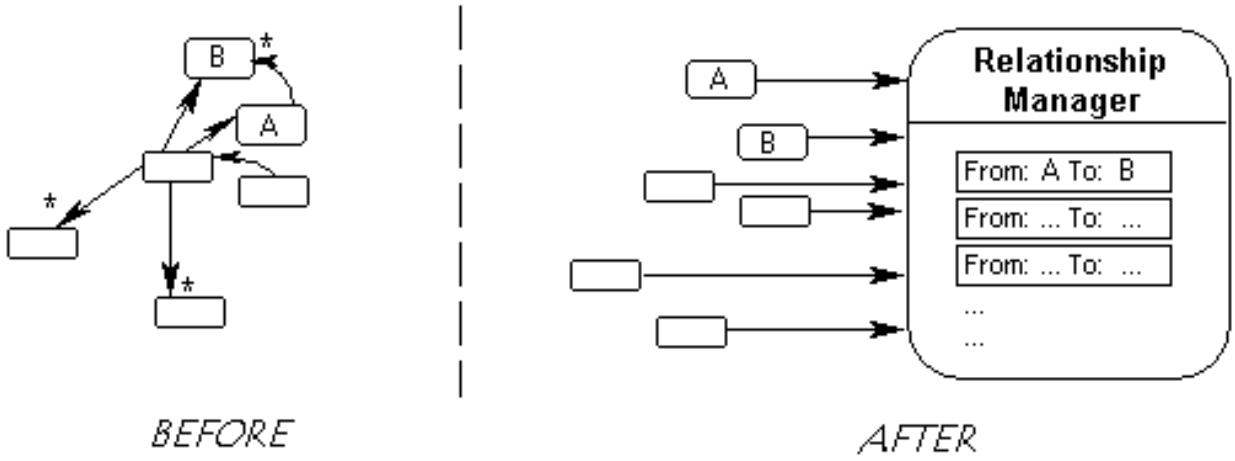


Figure 59 - Instead of refering to each other using pointers, objects refer to each other via a mediating relationship manager.

The big benefit of this solution

Your classes should now implement relationships with simple one line calls e.g.

```
class Customer(BaseBusinessObject):
    def AddOrder(self, order):
        self.RelMgr.AddRelationship(From=self, To=order, RelId=101)
    def RemoveOrder(self, order):
        self.RelMgr.RemoveRelationships(From=self, To=order,
                                                        RelId=101)
    def Orders(self):
        return self.RelMgr.FindObjects(From=self, RelId=101)


class Order(BaseBusinessObject):
    def GetCustomer(self):
        return self.RelMgr.FindObject(To=self, RelId=101)
```

Relationship Manager is an implementation choice

The decision to use a Relationship Manager to implement your business object relationships does not affect the design of the interfaces of your business classes. Your classes look and work the same e.g. `c = Customer(); o = Order(); c.AddOrder(o);`

It’s only when *implementing* a property or method that entails getting, manipulating or creating a relationship that you *implement* this functionality by calling methods on the central Relationship Manager, rather than attempting to define the attributes and code required for maintaining these relationships within business classes themselves.

The client code doesn’t care how methods on the business classes are implemented!

Notes:

Observer

A way of notifying change to a number of classes

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [GOF p. 297]

Problem

You want to achieve decoupled communication between classes, without breaking layering or encapsulation. You want to be able to notify an arbitrary and dynamically changing list of objects that something has happened.

Discussion

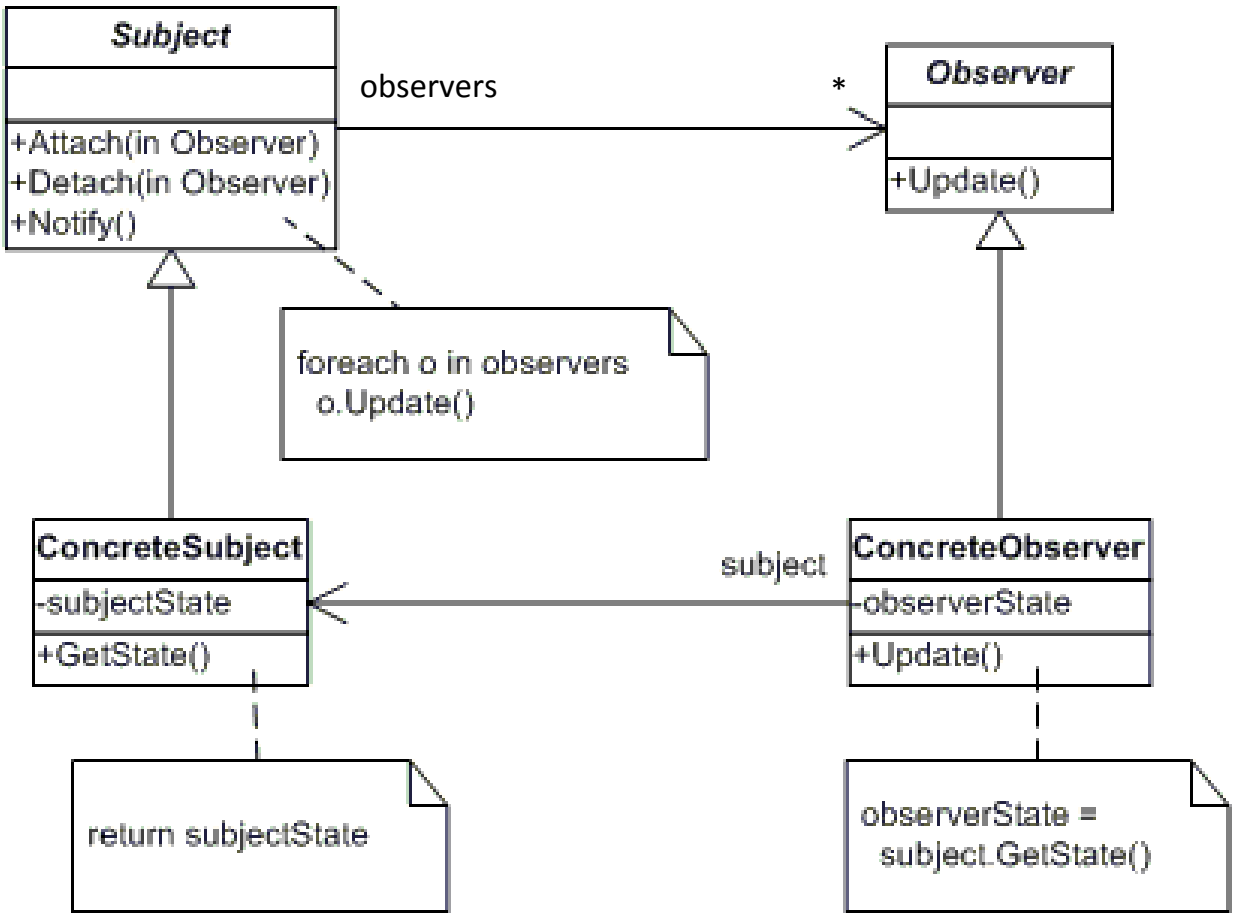
Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

This allows the number and "type" of "view" Observer objects to be configured dynamically, instead of being statically specified at compile-time.

Participants

The classes and/or objects participating in this pattern are:

- Subject (Stock)
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- ConcreteSubject (IBM)
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- Observer (IInvestor)
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteObserver (Investor)
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's state
  - implements the Observer updating interface to keep its state consistent with the subject's state.


Subject
observers
*
Observer
+Attach(in Observer)
+Detach(in Observer)
+Notify()
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Andy's Tips on Observer

- The subject's Notify() method simply loops through the list of observers and calls each observer's Update() method. This is the only knowledge the subject has about the observer. The observer, on the other hand, often knows all about the subject it is observing, including knowledge of how to extract specific data from the subject.
- The subject's Notify() method can be implemented in the base "Subject" class (as can the arraylist holding the list of observers). Thus the mechanism for being a Subject need only be written once.
- Observer is an abstract coupling—the subject doesn't know the concrete class of the observer, thus notifications can be safely done between different layers of the system e.g. between model and GUI.
- .Net has the multicast delegate mechanism for doing observer. Java has the listener pattern for doing observer. Alternatively it is reasonably easy to build your own observer implementation if you want a specific type of observer-like behaviour.
- Watch out for dangling references—there should be a notification between the subject and observer when subject or observer is deleted.
- There can often be unnecessary notifications to the observers—the naïve implementation of Observer is blind to the cost of updates/notification. A solution is to build a mechanism to switch on/off updates (only sending a single change broadcast after a series of consecutive changes has occurred).
- You can achieve the intent of the observer design pattern (of decoupled communication) by using an event. The subject defines an event and fires it on occasion. The observer installs a method in that event and thus gets notified whenever the event fires. A multicast delegate allows multiple methods to be installed / setup on an event, and is the same as the observer pattern. A non multicast event would be a weaker form of the observer pattern, since you could only have one observer at a time.
- Implementation variant: having a single Observer monitoring multiple Subjects
- Observer pattern typically used in hooking GUI to model. Model objects are the subject, GUI components (or their controllers) are the Observers.

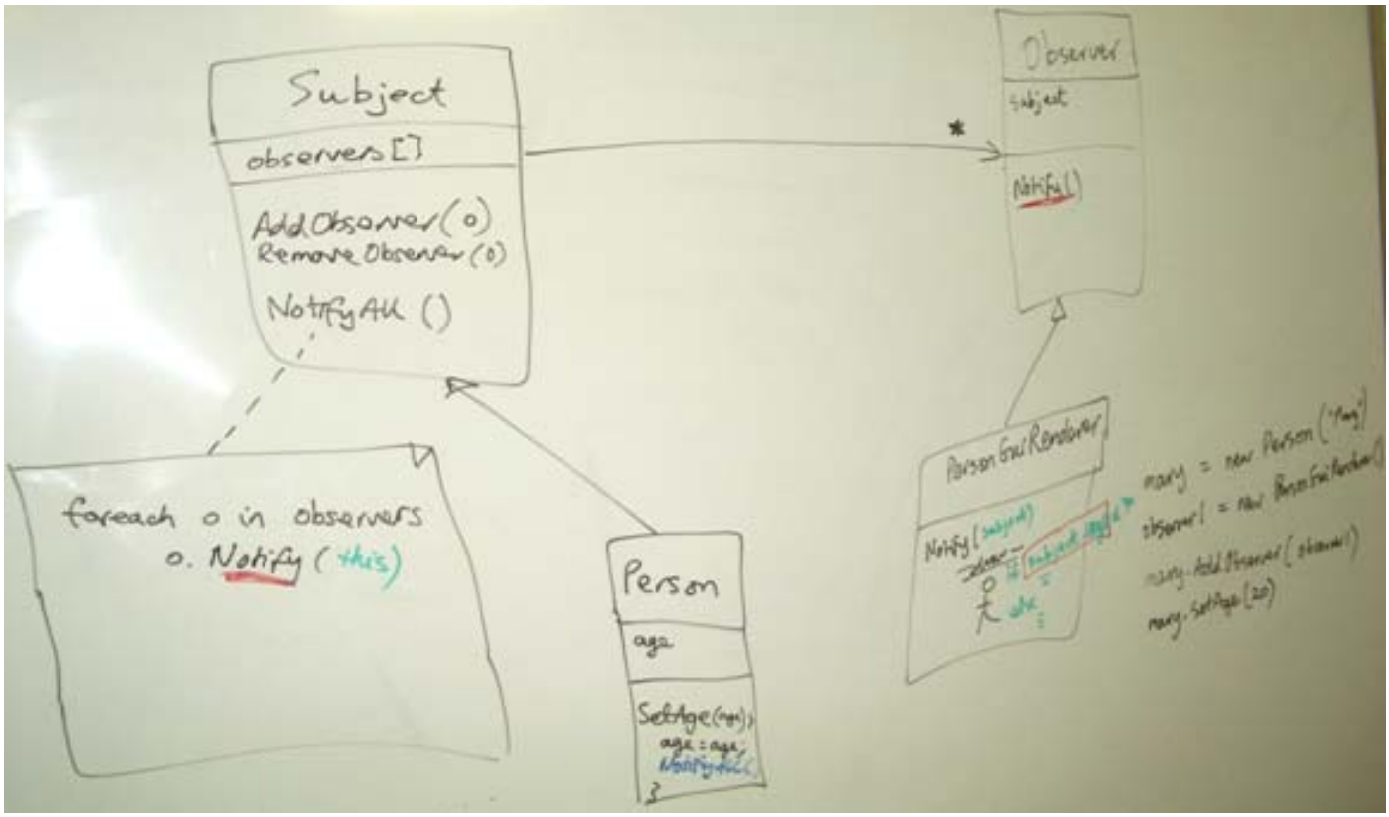


Figure 60 - Observer UML example - When the person class changes various Observers will be notified

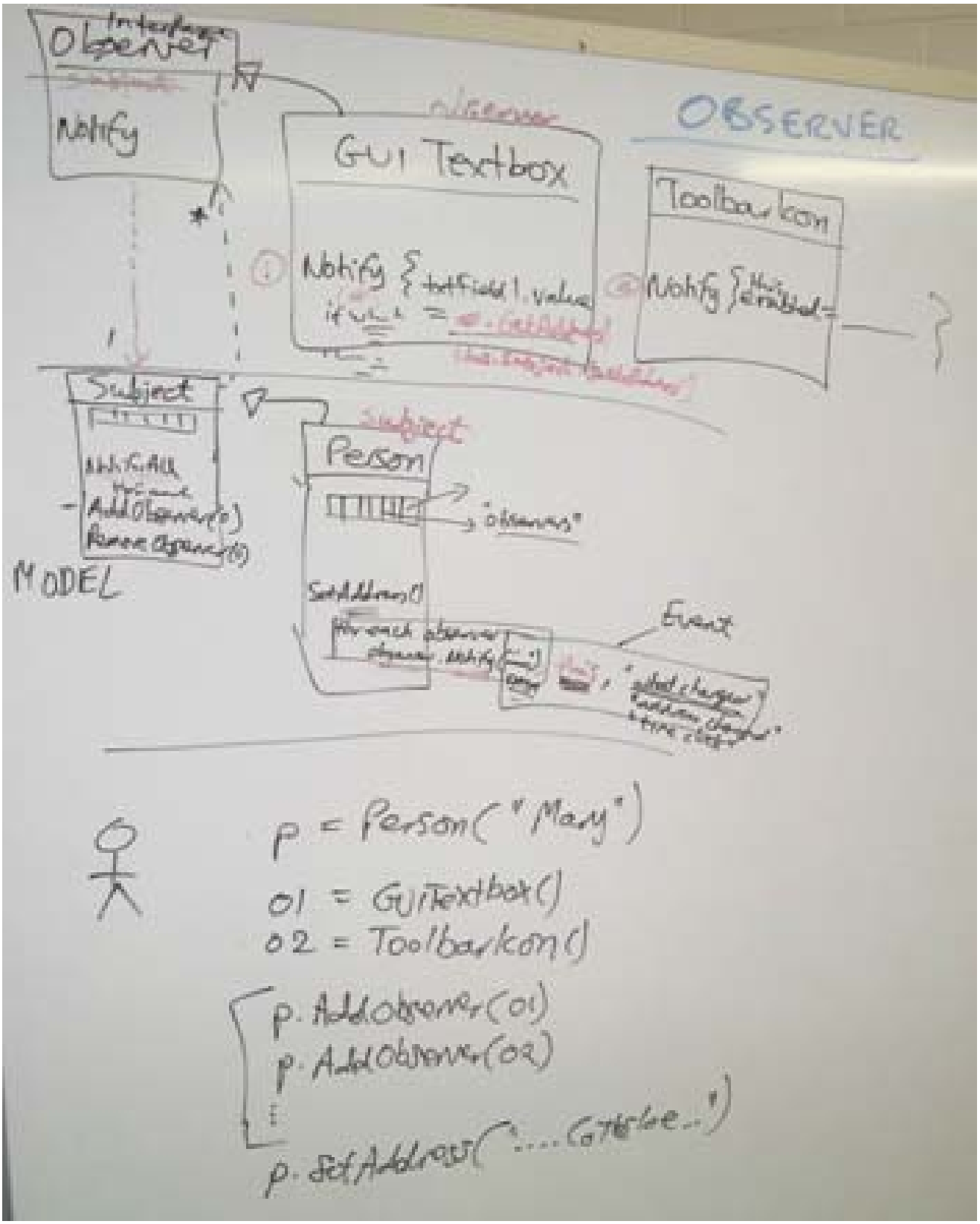


Figure 61 - Observer used to link together model and GUI

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

You can have a “push” or “pull” designs. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Non Software Examples

Consumers (observers) who send in warranty registration cards to a company (subject) will be notified in the event of a recall, new version release or any other news related to the product. This example uses explicit registration, and the collaborations may be easier to see.

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

The CNN news agency was presented as an example of the Observer pattern. The news agency acts as the subject, and the viewing public are the observers. When something in the world changes, everyone is notified. The discussion of this example centered on whether or not the subject knows the observers and whether or not registration is essential to the pattern. Turning on a television set could be considered to be registering. This broadcast method of distributed communication is common in data communications.

Code Ideas

www.dofactory.com’s structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

www.dofactory.com’s real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

Code: Observer in Python

```
class Subject:
    def __init__(self):
        self.observers = []
    def NotifyAll(self, whatchanged):
        for observer in self.observers:
            observer.Notify(subject=self, whatchanged=whatchanged)
    def AddObserver(self, observer):
        self.observers.append(observer)
        observer.subject = self

class Observer:
    def __init__(self):
        self.subject = None
    def Notify(self, subject , whatchanged):
        pass

class Person(Subject):
    def __init__(self, name=""):
        Subject.__init__(self)  # super()
        self.name = name
        self.address = ""
    def SetAddress(self, newaddress):
        self.address = newaddress
        self.NotifyAll(whatchanged='name')  # tells all observers about the c
hange
    def SetName(self, newname):
        self.name = newname
        self.NotifyAll(whatchanged='address')

class Stalker(Observer, Subject):
    def Notify(self, subject, whatchanged):
        print 'Stalker', self, ' watching', whatchanged, ' change to ' + subj
ect.name, ' subject=', subject

p = Person()
o1 = Stalker()
o2 = Stalker()
p.AddObserver(o1)
p.AddObserver(o2)
p.SetName("Mary")
p.SetAddress("12 Smith st")
```

Output:

```
Controller got request to send msg HelloStalker <__main__.Stalker instance at 0x011AC828>  watching address  change to Mary  subject=
<__main__.Person instance at 0x011B1AF8>
Stalker <__main__.Stalker instance at 0x011AC850>  watching address  change to Mary  subject= <__main__.Person instance at 0x011B1AF8>
Stalker <__main__.Stalker instance at 0x011AC828>  watching name  change to Mary  subject= <__main__.Person instance at 0x011B1AF8>
Stalker <__main__.Stalker instance at 0x011AC850>  watching name  change to Mary  subject= <__main__.Person instance at 0x011B1AF8>
```

Code: Delegates in C#

```
private void button1_Click(object sender, System.EventArgs e)
{
      Console.Out.WriteLine("hi");
      Person p = new Person("Mary");
      Stalker s1 = new Stalker("Allan");
      Stalker s2 = new Stalker("Nick");
      p.AddObserver(s1);
      p.AddObserver(s2);
      p.SetName("Maryanne");

      Console.ReadLine();
}

using System;

namespace PerthDP_Delegate01
{
      public delegate void Watcher();  // define a type

      public class Person
      {
            private Watcher observers;
            string name;

            public Person(string name)
            {
                  this.name = name;
            }
            public void AddObserver(Stalker s)
            {
                  observers += new Watcher(s.Update);
            }
            public void SetName(string newname)
            {
                  this.name = newname;
                  observers();
            }
      }

      public class Stalker {
            string name;

            public Stalker(string name)
            {
                  this.name = name;
            }
            public void Update()
            {
                  Console.Out.WriteLine(this.name + " saw name change");
            }
      }
```

```
}
```

A note on Events in C#/.NET. Events are public only and furthermore, only allow += and -= operations, not anything else.

Using Observer Architecturally

Observer is often used to connect objects between layers of an architecture, without the lower level objects knowing about the higher (e.g. GUI) level objects. This keeps the layers cleaner and decoupled.

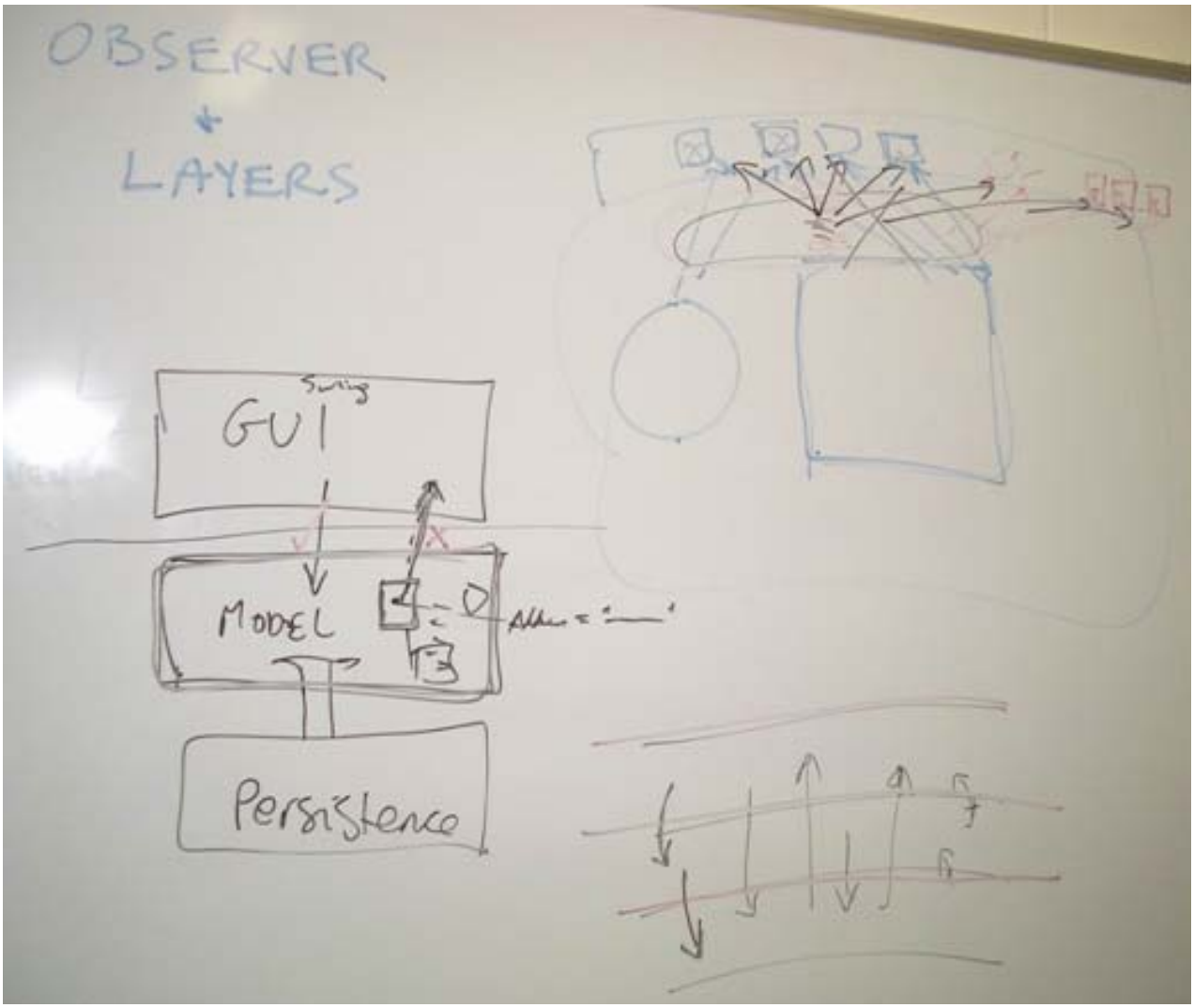


Figure 62 - Model layer talks to the GUI layer in a decoupled way using observer.

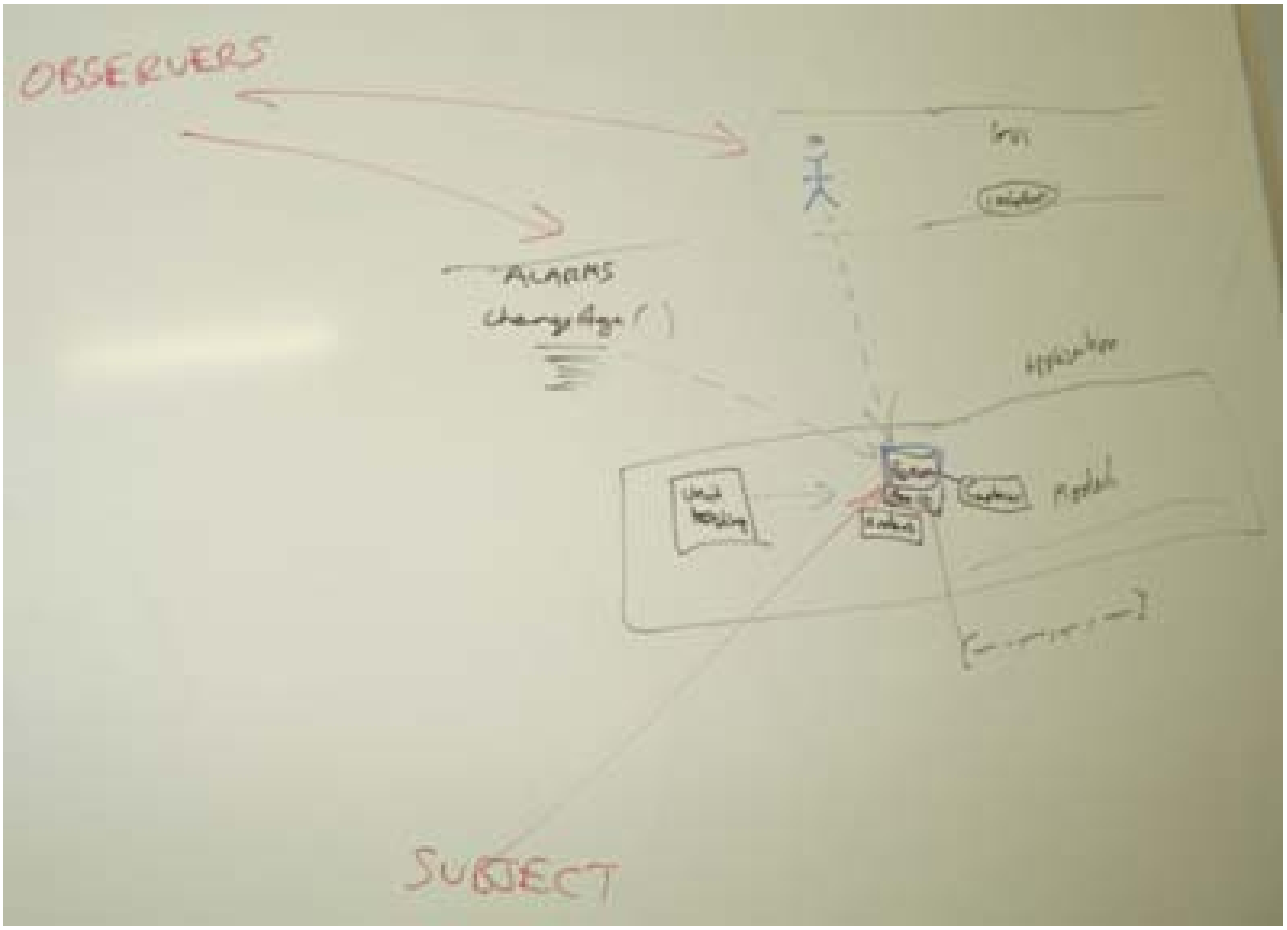


Figure 63 - Observer as used in a layered architecture

A Centralised Observer Implementation

Instead of each subject keeping a list of observers, you could store all observers to all subjects in a centralized place – a broker of sorts. GOF talk about a "Change Manager" style implementation. See also "Relationship Manager" on page 175 for an example of a centralized style implementation (albeit it is for Mediator, not Observer – however the patterns are related)

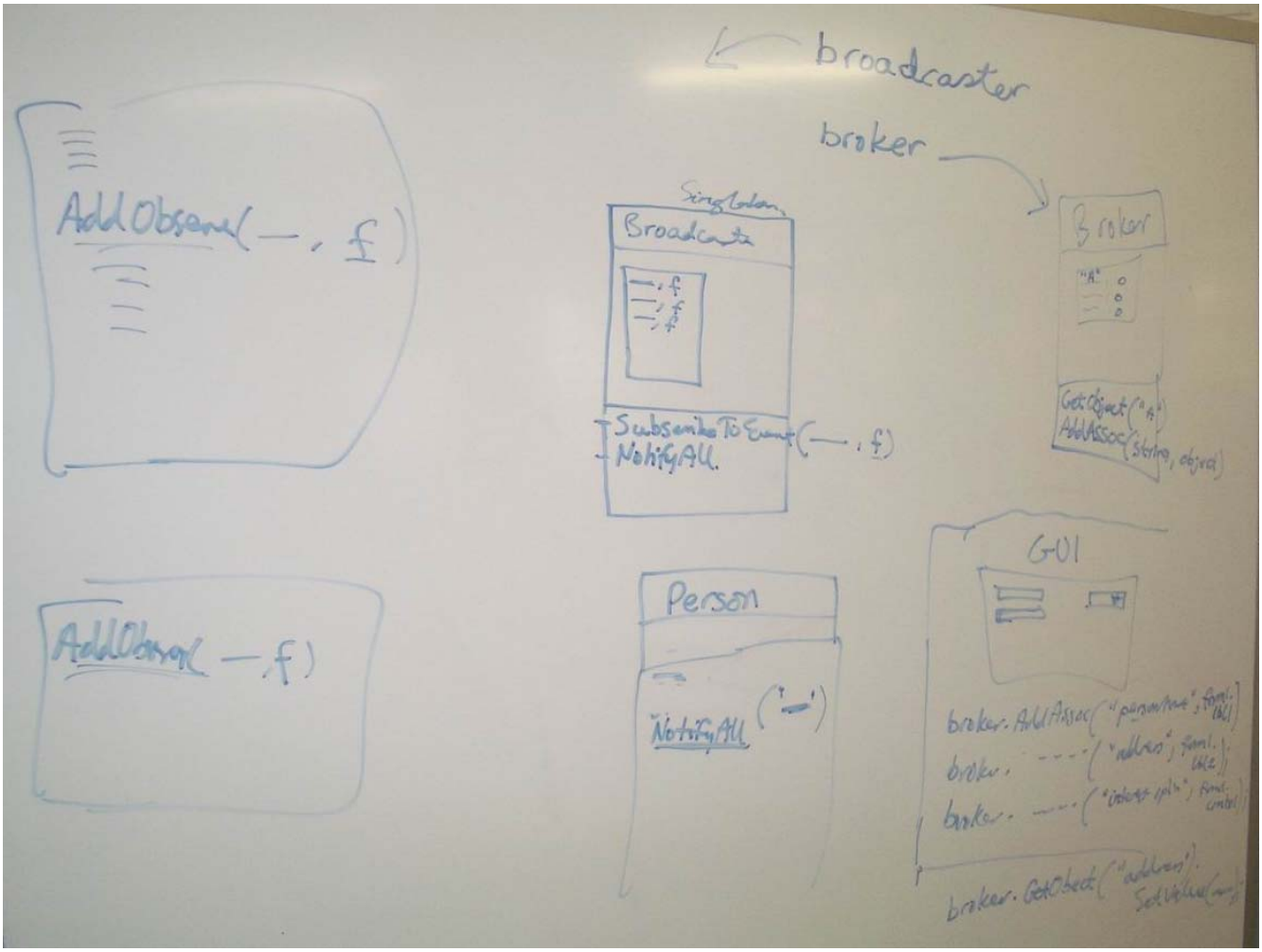


Figure 64 - Broker pattern observer variation

Notes:

Chain of Responsibility

A way of passing a request between a chain of objects

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.

Problem

There is a potentially variable number of "handler" objects and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

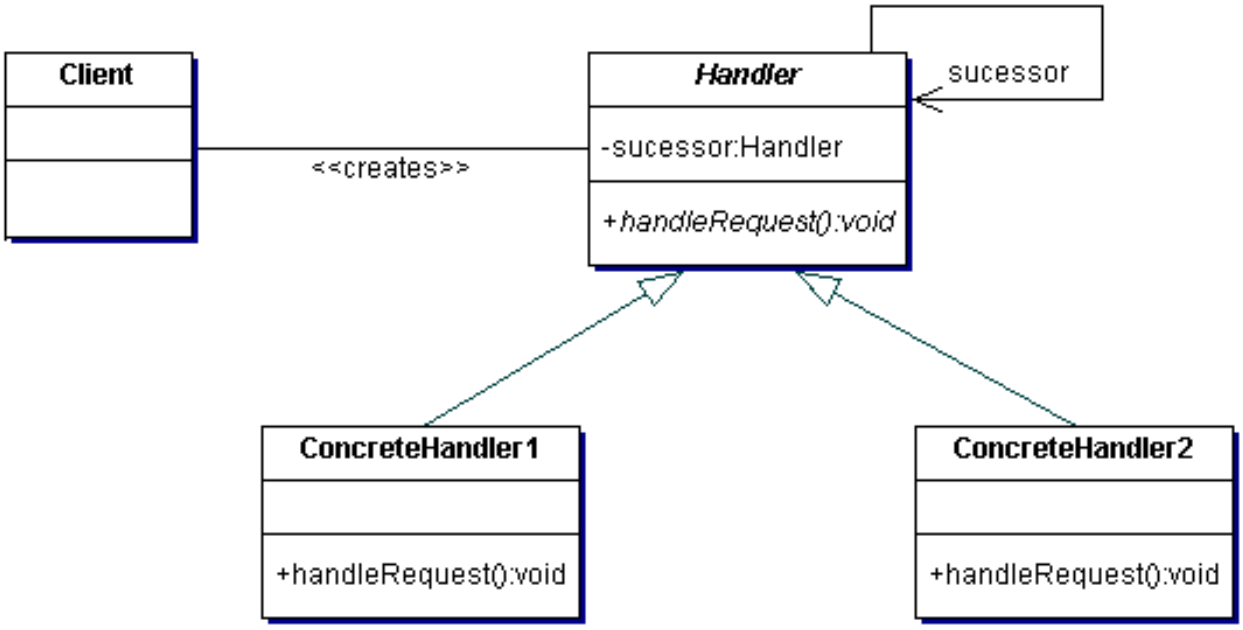
Discussion

The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a "safety net" to "catch" any requests which go unhandled.

Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.


Client
Handler
sucessor
-sucessor:Handler
+handleRequest():void
<<creates>>
ConcreteHandler1
+handleRequest():void
ConcreteHandler2
+handleRequest():void

Participants

The classes and/or objects participating in this pattern are:

- Handler (Approver)
  - defines an interface for handling the requests
  - (optional) implements the successor link
- ConcreteHandler (Director, VicePresident, President)
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- Client (ChainApp)
  - initiates the request to a ConcreteHandler object on the chain

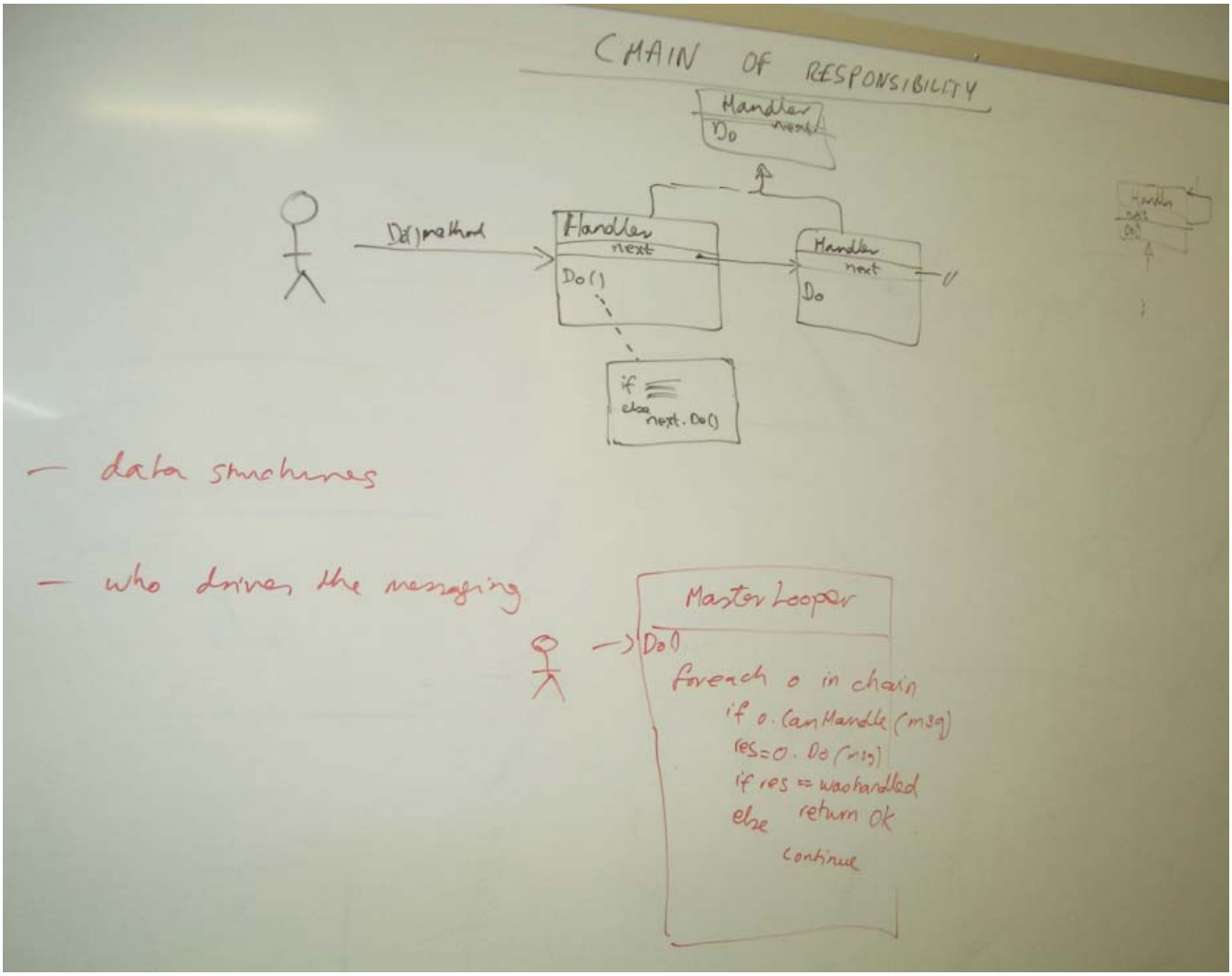


Figure 65 - UML for Chain of Responsibility

Ideas

- The system shall allow the client to issue a request to one of several "handlers" without knowing or specifying the receiver explicitly.
- The system shall allow a suite of "handlers" to be configured at run-time.
- The system shall allow the client to "launch and leave" a request with a "virtual pipeline of handlers".
- "Senders" and "receivers" shall be decoupled from one another.

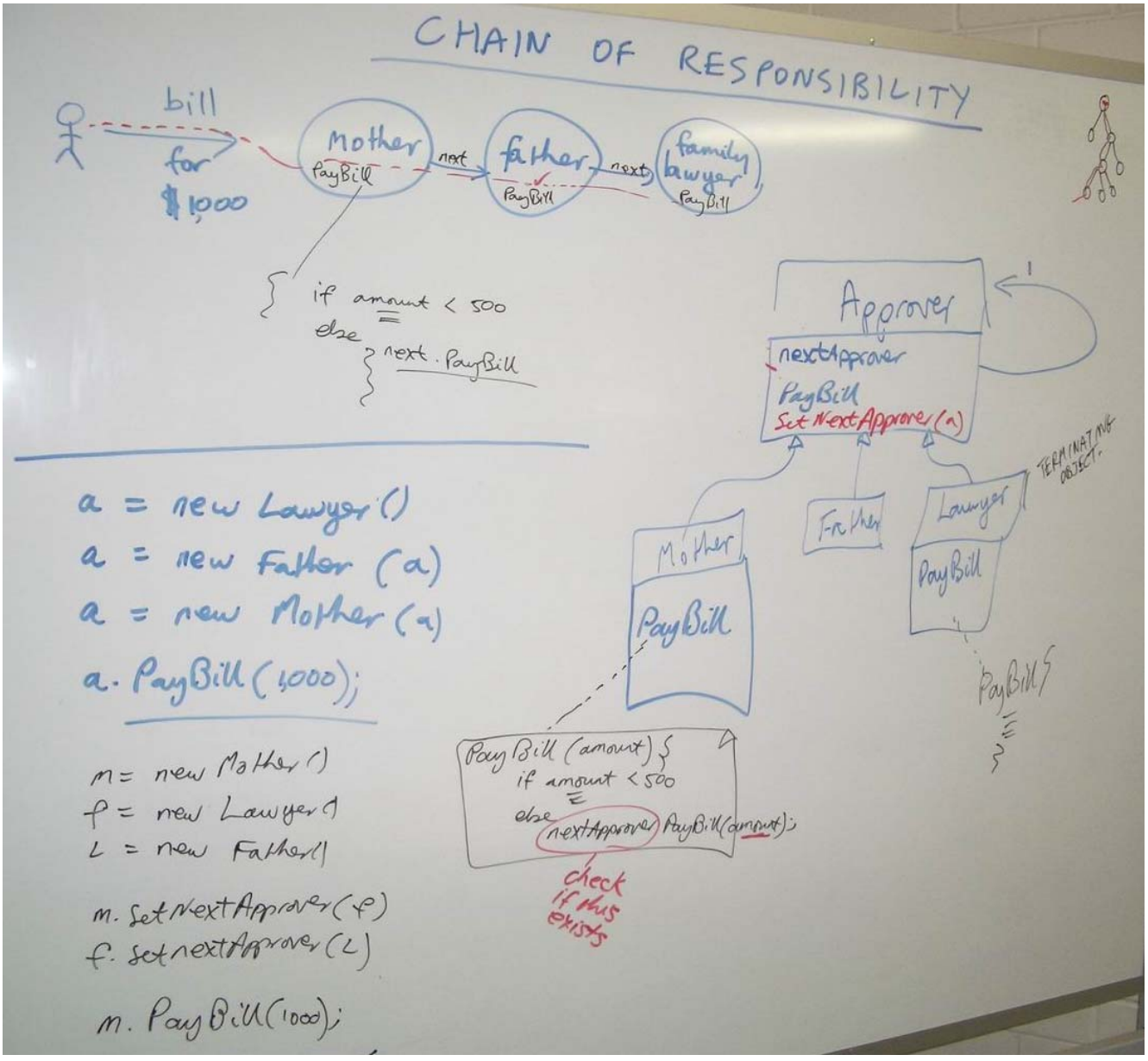


Figure 66 - Basic Idea of Chain of Responsibility

Andy's Tips

- If you find your client code is making "probing calls" before issuing the request it really wants to make, apply the chain of responsibility pattern.
- You relieve the client from having to know which object in a collection supports a given behavior.
- Can be implemented as a chain (common) or a hierarchy where messages / requests trickle up from the leaves to the parents to the grandparents etc. E.g. Toolbook's hierarchical message passing, Java 1.0 (user interface event container hierarchy).
- Consequence: reduce the coupling between classes. Objects just need to know only how to handle a request, or alternatively, forward the request to other objects.
- Requests can be forwarded by each individual object in the chain, or alternatively there can be an external loop which calls each object in the chain. External control probably better (more decoupled and less fragile). This is the distinction between internal and external iterators.
- When you have a chain of objects e.g. **[condition | action]** → **[condition | action]** and an **external iterator** loops through the chain and calls the condition() method (which returns true/false) on each object to see if the object is prepared to handle the condition, then *you don't want any complex decision logic in the external iterator itself.* Calling the condition() method on each object should be enough – all the decision logic should be encapsulated in each object's condition() method.
- To wire up your chains/hierarchies? Constructors or an AddToChain(o) method. Perhaps use an existing hierarchy or chain/arraylist?
- The inheritance chain (in OO programming) is a kind of chain of responsibility e.g. when you call super() / base() from one method it calls the same method in the parent class etc.

Code Ideas

www.dofactory.com's structural code demonstrates the Chain of Responsibility pattern in which several linked objects (the Chain) are offered the opportunity to respond to a request or hand it off to the object next in line.

www.dofactory.com's real-world code demonstrates the Chain of Responsibility pattern in which several linked managers and executives can respond to a purchase request or hand it off to a superior. Each position has can have its own set of rules which orders they can approve.

Example in Toolbook:

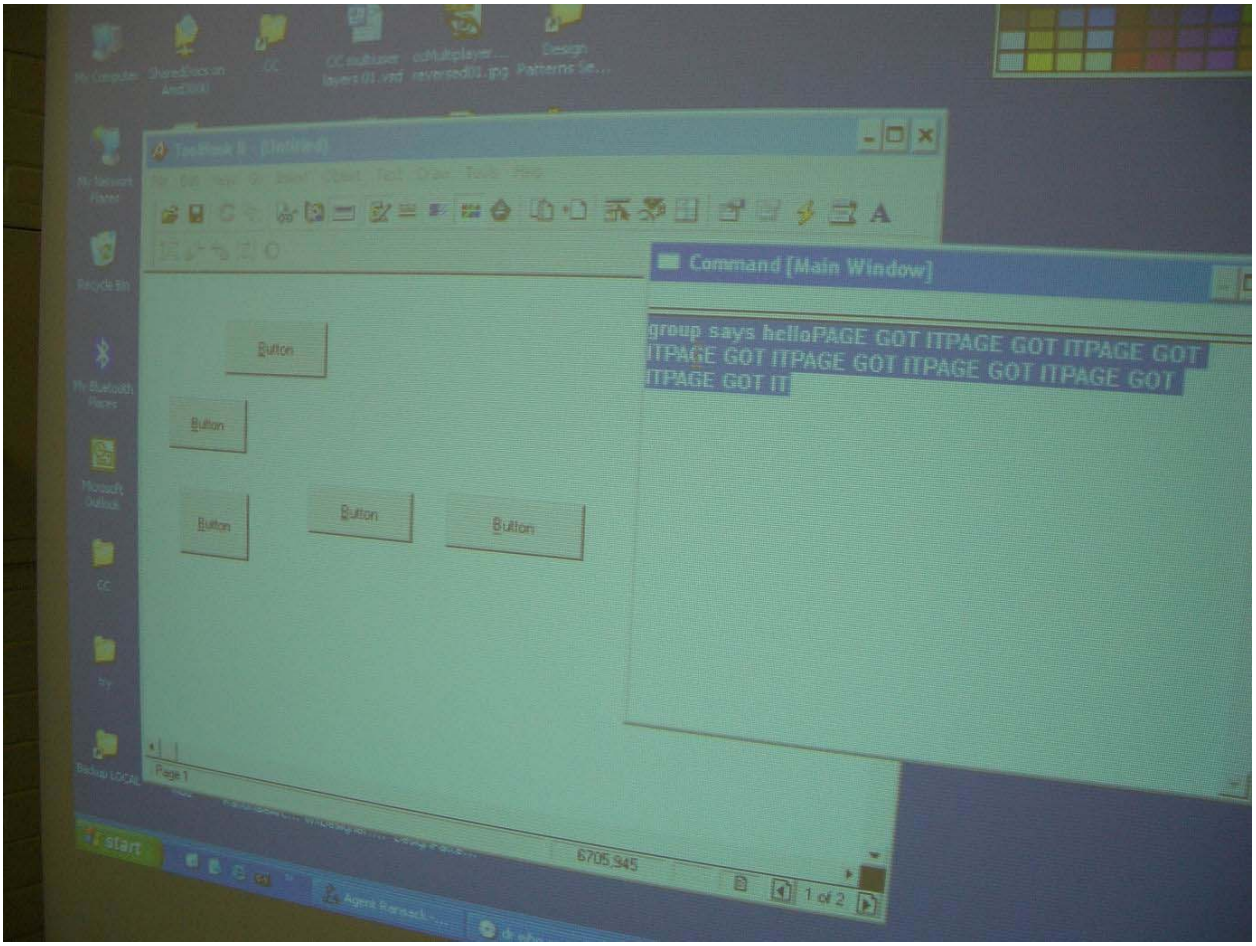


Figure 67 - Chain of Responsibility in the Toolbook Multimedia Authoring Framework

Non Software Examples

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. Mechanical coin sorting banks use the Chain of Responsibility. Rather than having a separate slot for each coin denomination coupled with a receptacle for the denomination, a single slot is used. When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Non Software example 2 – military chain of command

The chain of command in the military illustrates the Chain of Responsibility pattern. When a request is made of a private, the private can either grant the request or forward the request to his or her superior. The request will be forwarded until someone with the appropriate authority is able to act upon it.

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349]

Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor. [GOF, p232]

Notes:

Memento

Capture and restore an object's internal state

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.

Problem

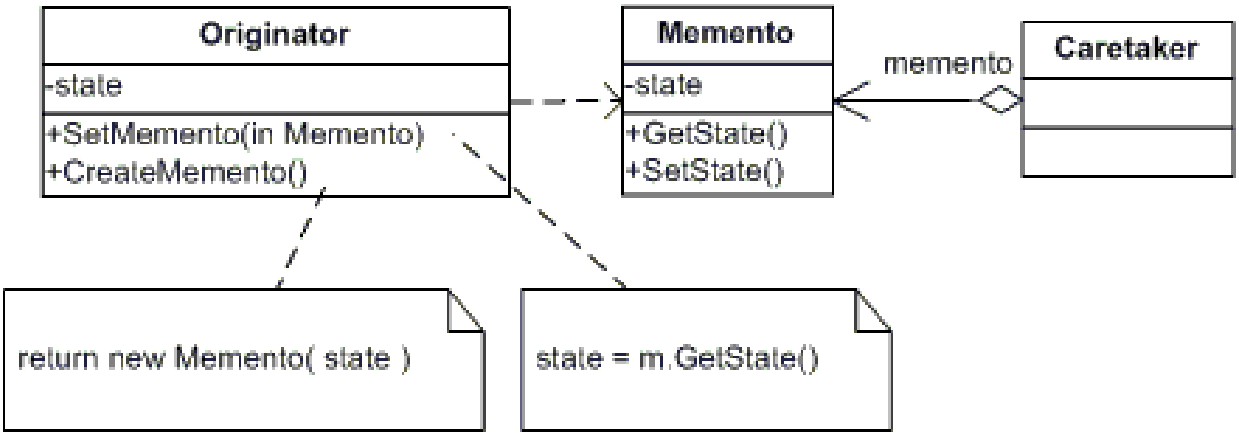
Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

Discussion

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

- *The system shall support "undo" or "rollback"*
- *The system shall support transactions*
- *The system shall support saving, or "flattening", or "streaming" components without compromising their encapsulation.*


Originator
-state
+SetMemento(in Memento)
+CreateMemento()
Memento
-state
+GetState()
+SetState()
memento
Caretaker
return new Memento(state)
state = m.GetState()

Participants

The classes and/or objects participating in this pattern are:

- Memento (Memento)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- Originator (SalesProspect)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- Caretaker (Caretaker)
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

Andy's Tips

- Capture an object's state and restore it later, without violating encapsulation
- The Originator (the class who's state you are capturing) sees a wide interface to the memento (the class containing the state). The Caretaker (or whoever else holds a reference to the memento object) sees a narrow interface.
- The Caretaker never operates or examines the contents of the memento (hence the narrow interface)
- It is not the responsibility of the memento object to have a method called say, restoreState(). It is sufficient for the memento to just have simple property accessors. It is the originator object which has all the restoration logic e.g. SetMemento(m)
- The originating class and the memento probably need to be "friends" – see Figure 69 - Memento Idea - more detail. This can be implemented using a "friends" language feature, an "internal" assembly or possibly through the use of two different interfaces.
- Be open to ways of using memento not just to store "state" but to e.g. store a sequence of commands that would recreate the state.

Rules of thumb

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, 242]

Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

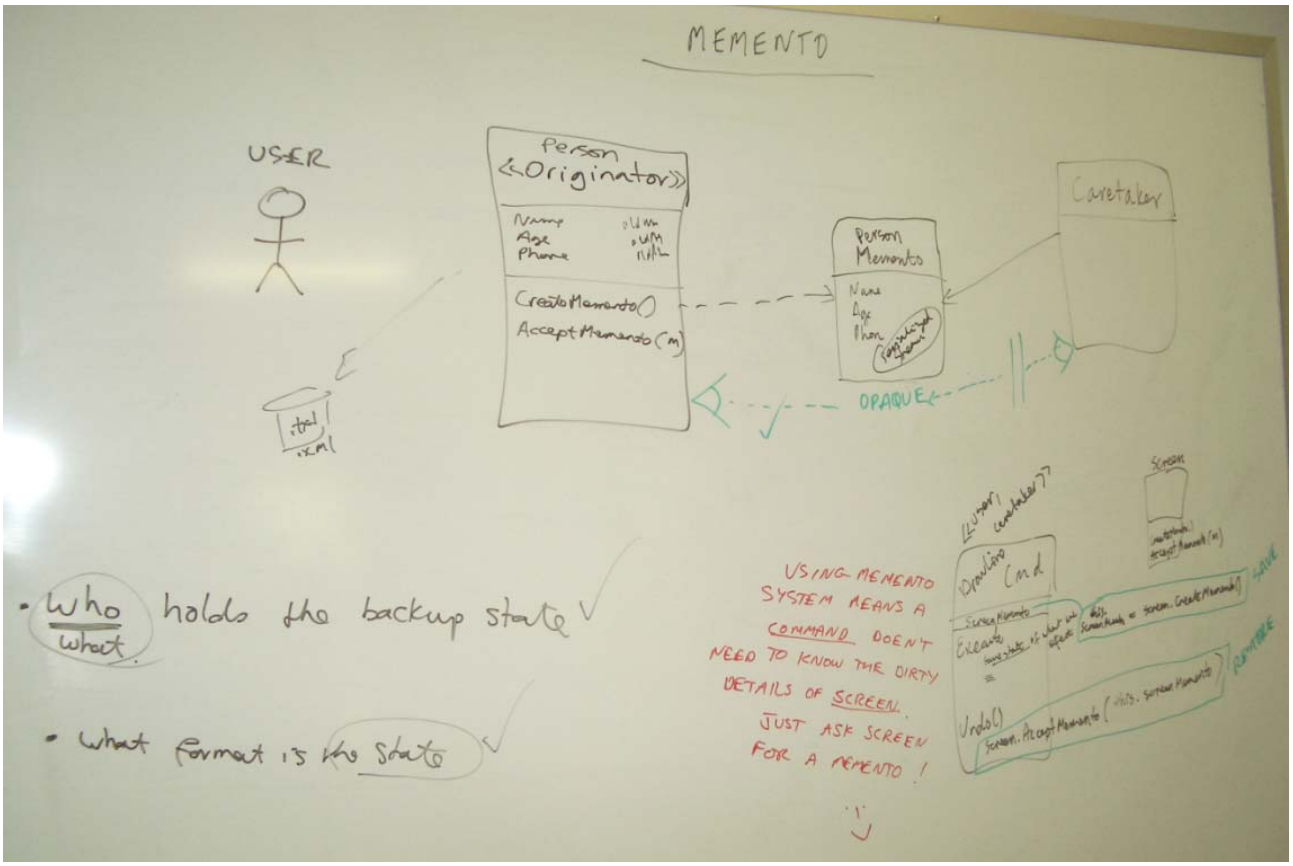


Figure 68 - Memento Idea

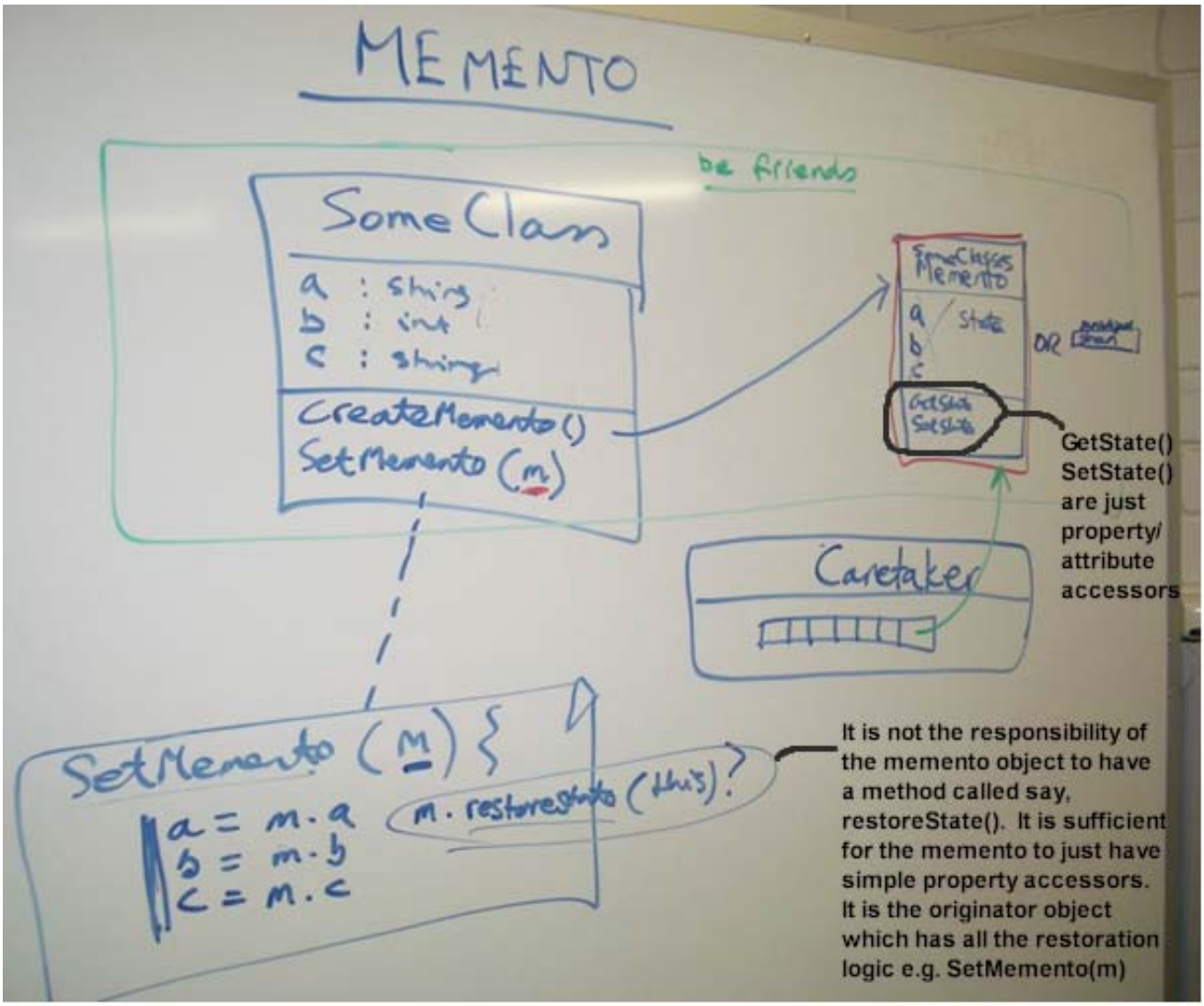


Figure 69 - Memento Idea - more detail. Note the "friends" relationship between originator and memento.

Code Ideas

www.dofactory.com's structural code demonstrates the Memento pattern which temporary saves and restores another object's internal state.

www.dofactory.com's real-world code demonstrates the Memento pattern which temporarily saves and then restores the SalesProspect's internal state.

Non Software Examples

An example of audio mixing equipment was then proposed. Since there are several switches set in different positions, it is common to photograph the equipment, so that the switches can be returned to their original positions if perturbed.

Yet another example was proposed at the poster session. When a restaurant patron enters an establishment wearing a coat, it is common for the coat to be left in the coat room while the patron dines. The patron is given a claim check, so that upon leaving the restaurant, he or she is returned to the same state: HAS_COAT. In this example, the claim check is used to ensure that the coat the patron has upon leaving the restaurant is the same coat that he or she had upon entering.

Notes:

Command

Encapsulate a command request as an object. Implement Undo/Redo.

Intent

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

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Discussion

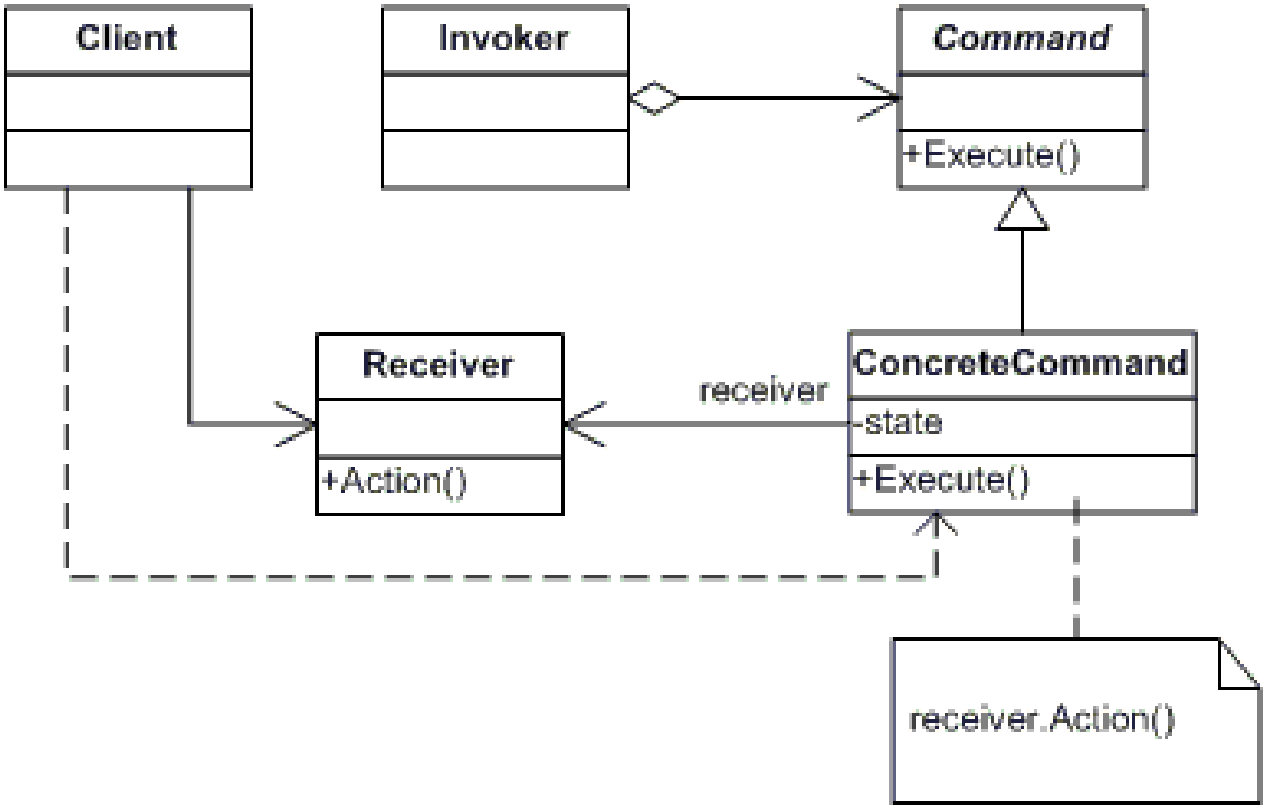
Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.

All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual Execute() method whenever the client requires the object's "service".

Sequences of Command objects can be assembled into composite (or macro) commands.

Non Software Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

Participants

The classes and/or objects participating in this pattern are:

- Command (Command)
  - declares an interface for executing an operation
- ConcreteCommand (CalculatorCommand)
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- Client (CommandApp)
  - creates a ConcreteCommand object and sets its receiver
- Invoker (User)
  - asks the command to carry out the request
- Receiver (Calculator)
  - knows how to perform the operations associated with carrying out the request.

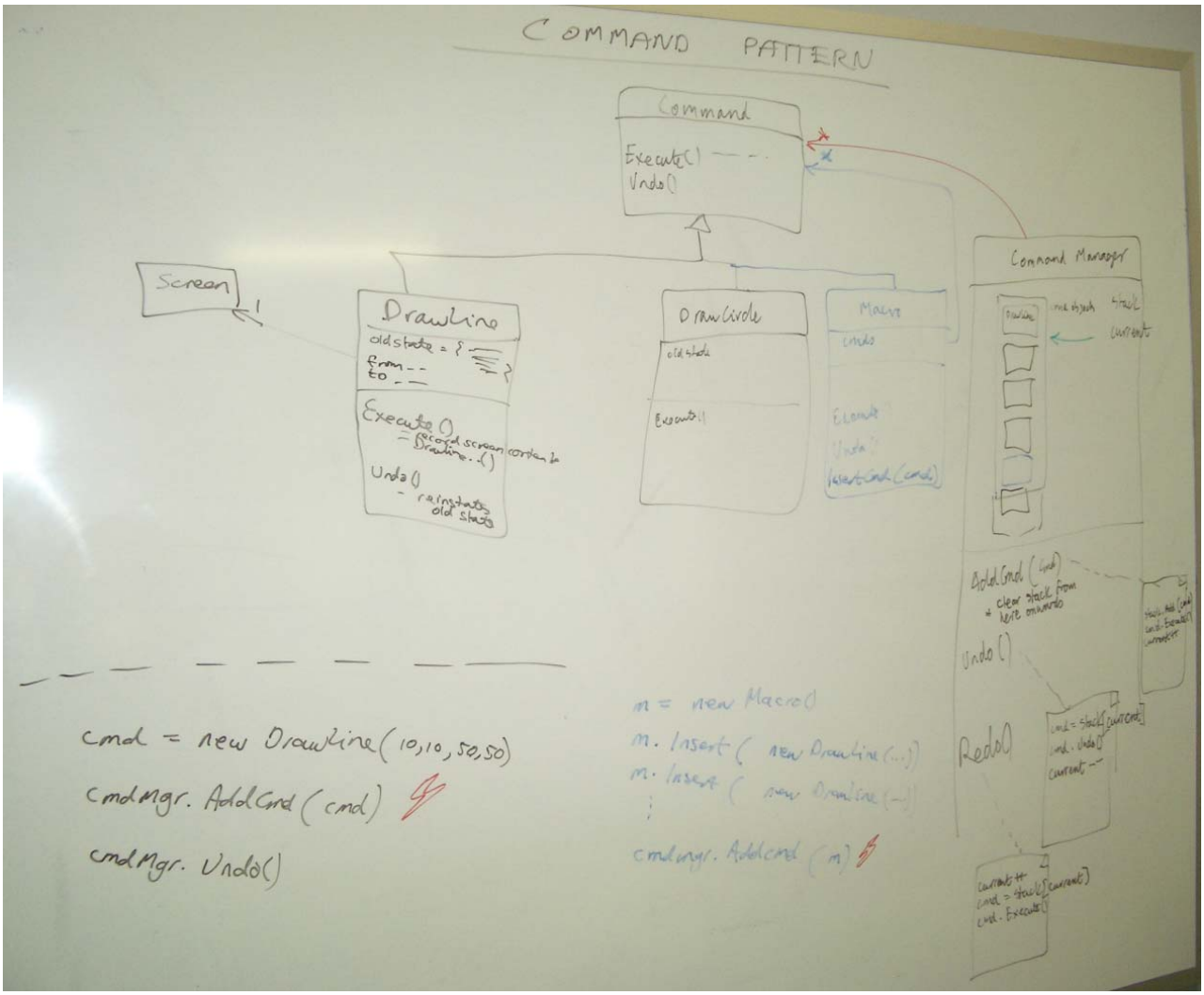


Figure 70 - Command Pattern - UML and interaction with a Command Manager

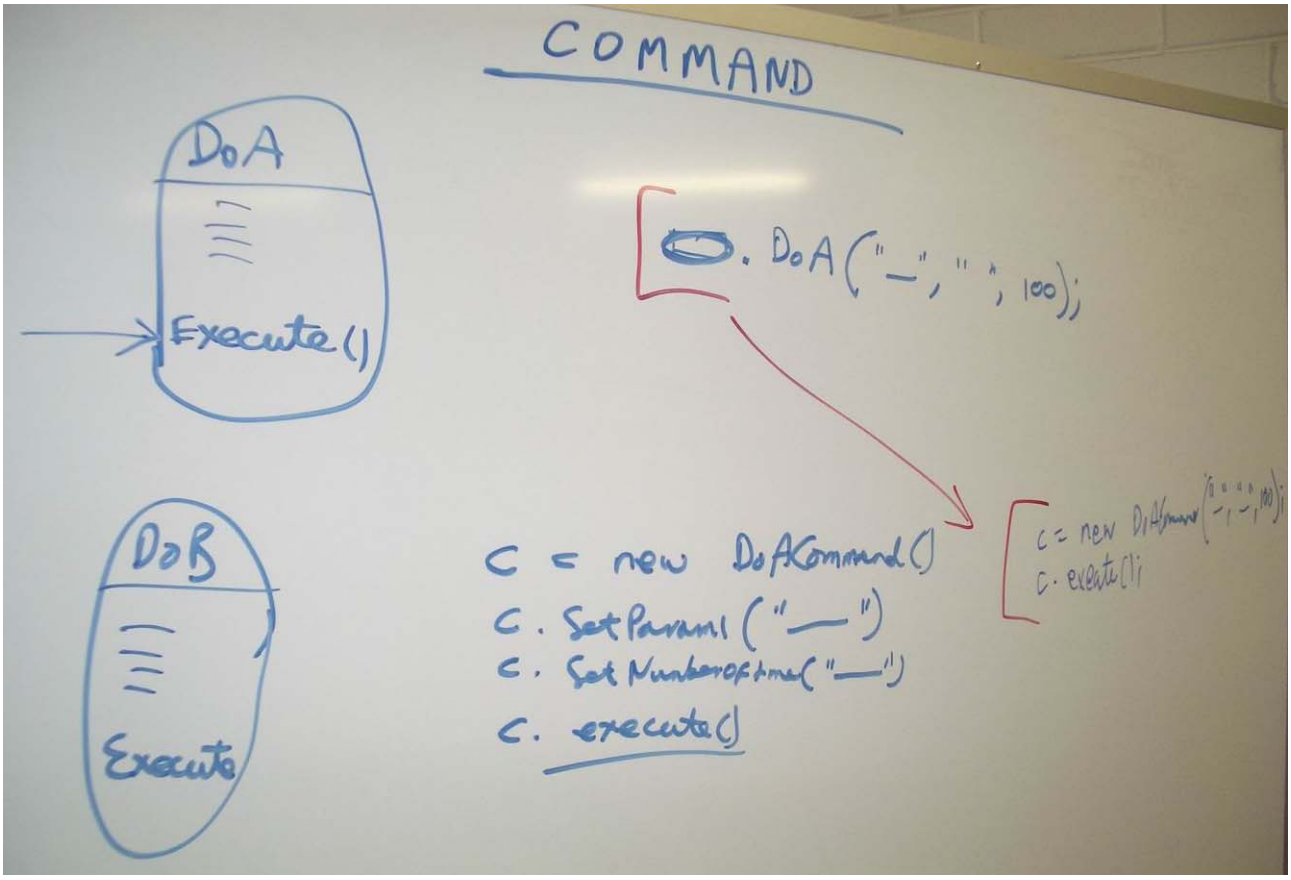


Figure 71 - Paradigm Shift - instead of calling DoA() on an object we instead create a command object and call Execute() on it.

Ideas

- The system architecture shall provide a "callback" framework.
- The system shall encapsulate "execute" requests so that they may be created, queued, and subsequently serviced. They may also be logged, archived, loaded, and re-applied.
- The system shall support hierarchical compositions of primitive "command" abstractions.
- The system shall support reuse of "command" abstractions. [Define a "command", and simultaneously attach the encapsulation to a push button, a toolbar icon, a menu item, and a keyboard accelerator - so that all users of the GUI can enjoy their preferred method of interaction.]
- The system shall support "redo"
- The system shall support transactions
- "Senders" and "receivers" shall be decoupled from one another.

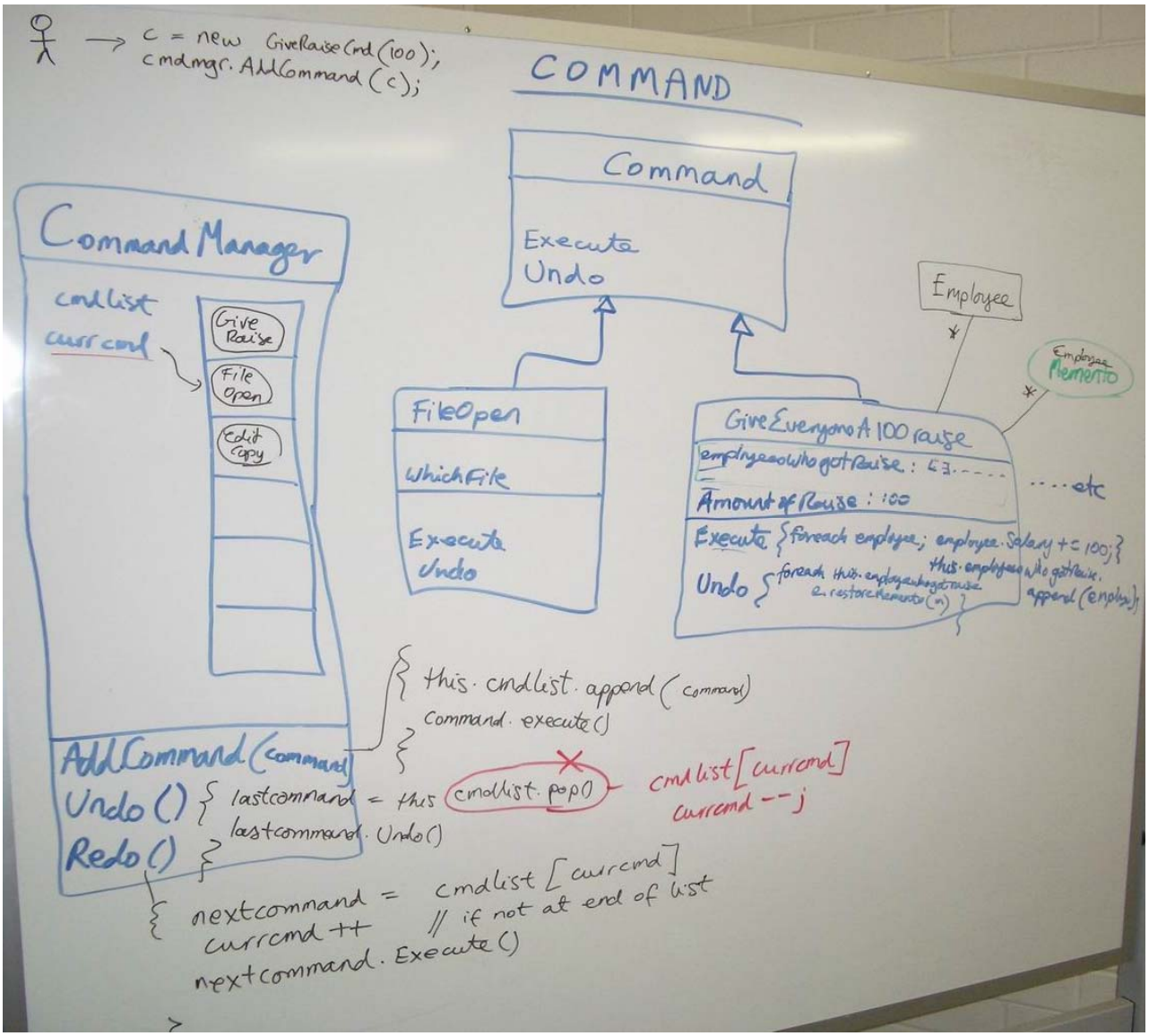


Figure 72 - How Command Manager works with Command Pattern.

Andy's Tips

- Commands are object oriented replacements for callbacks (a callback is a function that's registered somewhere to be called at a later time).
- A memento can keep the state the command requires to undo its effect.
- To undo, you can either take a snapshot of the way the system was and restore it, or alternatively, just undo the steps the command did. Depending on your situation and how much info there is in the "state", it might be easier one way or the other.
- You can use two separate lists to implement undo – a do list and an redo list. So rather than a pointer into one list, you maintain two lists.
- You could add an "action" system over the top of this command manager pattern, and wire up observers so that the state of menus etc. change (e.g. dim) as various commands are enabled or

disabled.

Code Ideas

www.dofactory.com's structural code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.

www.dofactory.com's real-world code – demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C# the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

Code Idea

You may need to pass state between commands and between macro commands esp. when one command relies on the result of an earlier command. Because commands don't take parameters when called (just the .Execute() method) getting this communication happening can be tricky. e.g.

```
MACRO
     state = new state() // object used to communicate info
     cmd1 = new createbox(state)  // command puts result into state var
     cmd2 = new setboxtext(state, text) // command looks in state var
      // for the reference to the box to set the text of.
     cmd1.Execute()
     cmd2.Execute()  // commands communicate via the state object
END MACRO
```

Rules of thumb

Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.

Chain of Responsibility can use Command to represent requests as objects. [GOF, p349].

Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

Command can use Memento to maintain the state required for an undo operation. [GOF, p242]

MacroCommands can be implemented with Composite. [GOF, p242]

A Command that must be copied before being placed on a history list acts as a Prototype. [GOF, p242]

POSA's Command Processor pattern describes the scaffolding Command needs to support full multilevel undo and redo. [Vlissides, Java Report, Nov 2000, p80]

Notes:

Prototype

A fully initialized instance to be copied or cloned

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Problem

Application "hard wires" the class of object to create in each "new" expression.

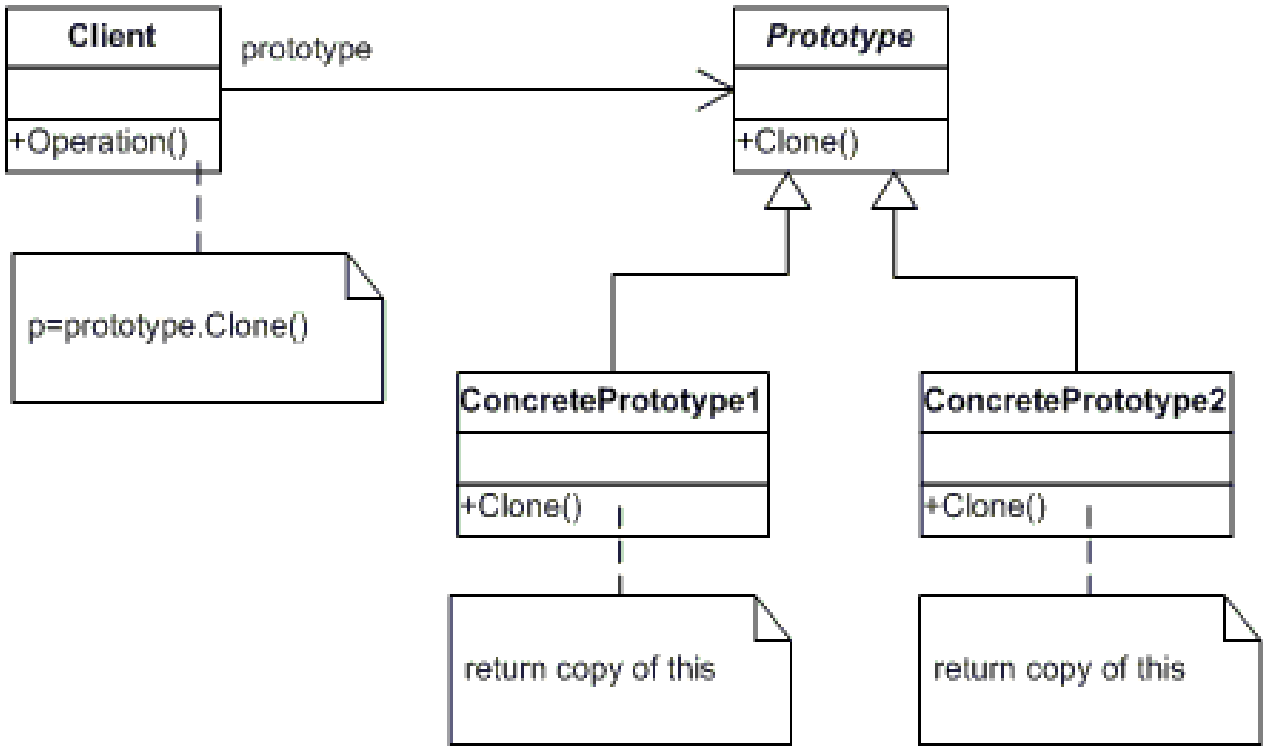
Discussion

Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

• The overhead of component creation shall be localized and minimized. Instead of creating entirely new instances of a component type, a "prototypical" instance of each type shall be designed to "clone" itself.

• Decisions about the type of component to create shall be deferred until application run-time.


Client
prototype
Prototype
+Operation()
+Clone()
p=prototype.Clone()
ConcretePrototype1
+Clone()
return copy of this
ConcretePrototype2
+Clone()
return copy of this

Participants

The classes and/or objects participating in the Prototype pattern are:

- Prototype (ColorPrototype)
  - declares an interace for cloning itself
- ConcretePrototype (Color)
  - implements an operation for cloning itself
- Client (ColorManager)
  - creates a new object by asking a prototype to clone itself

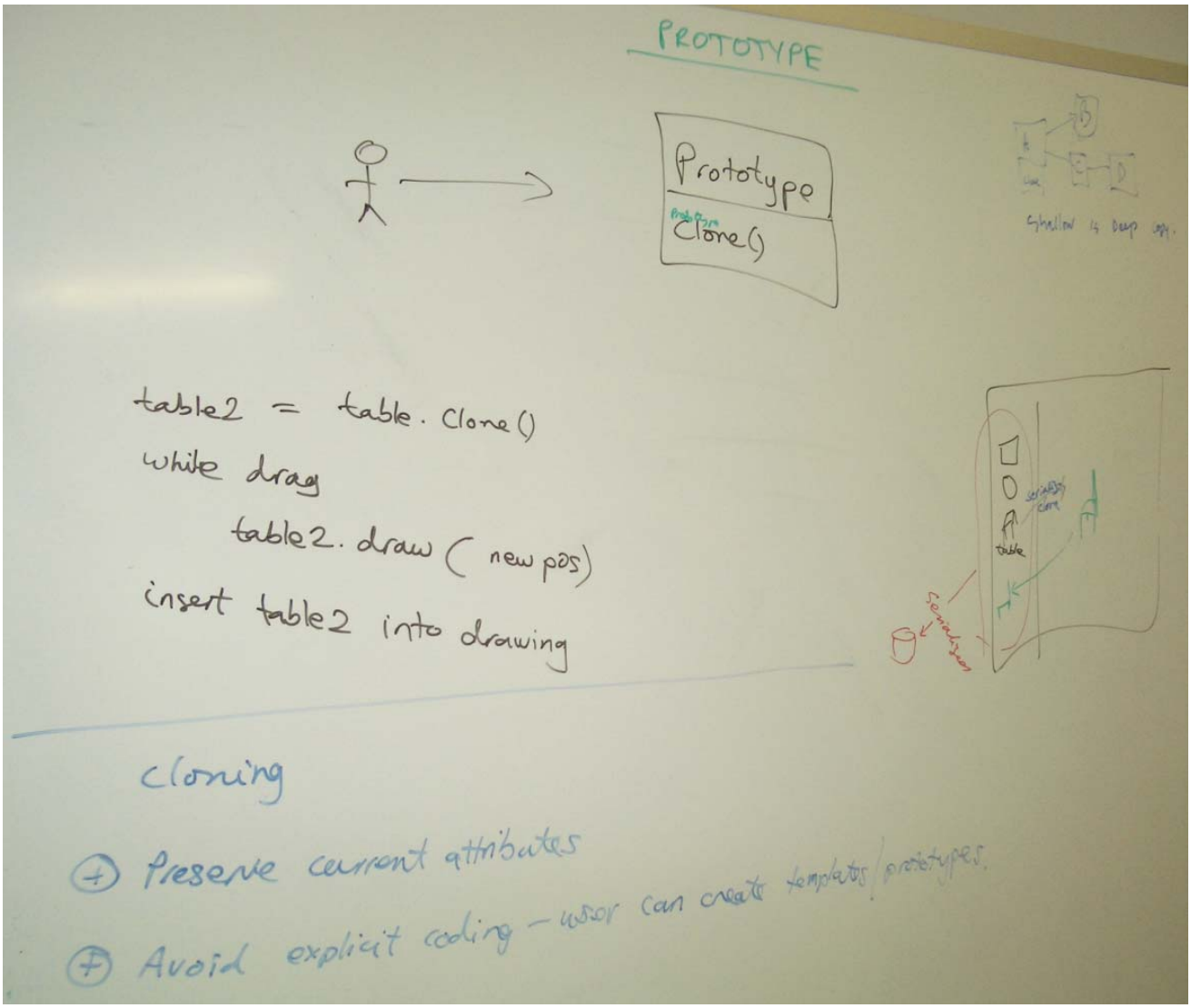


Figure 73 - Prototype Class has a Clone() method

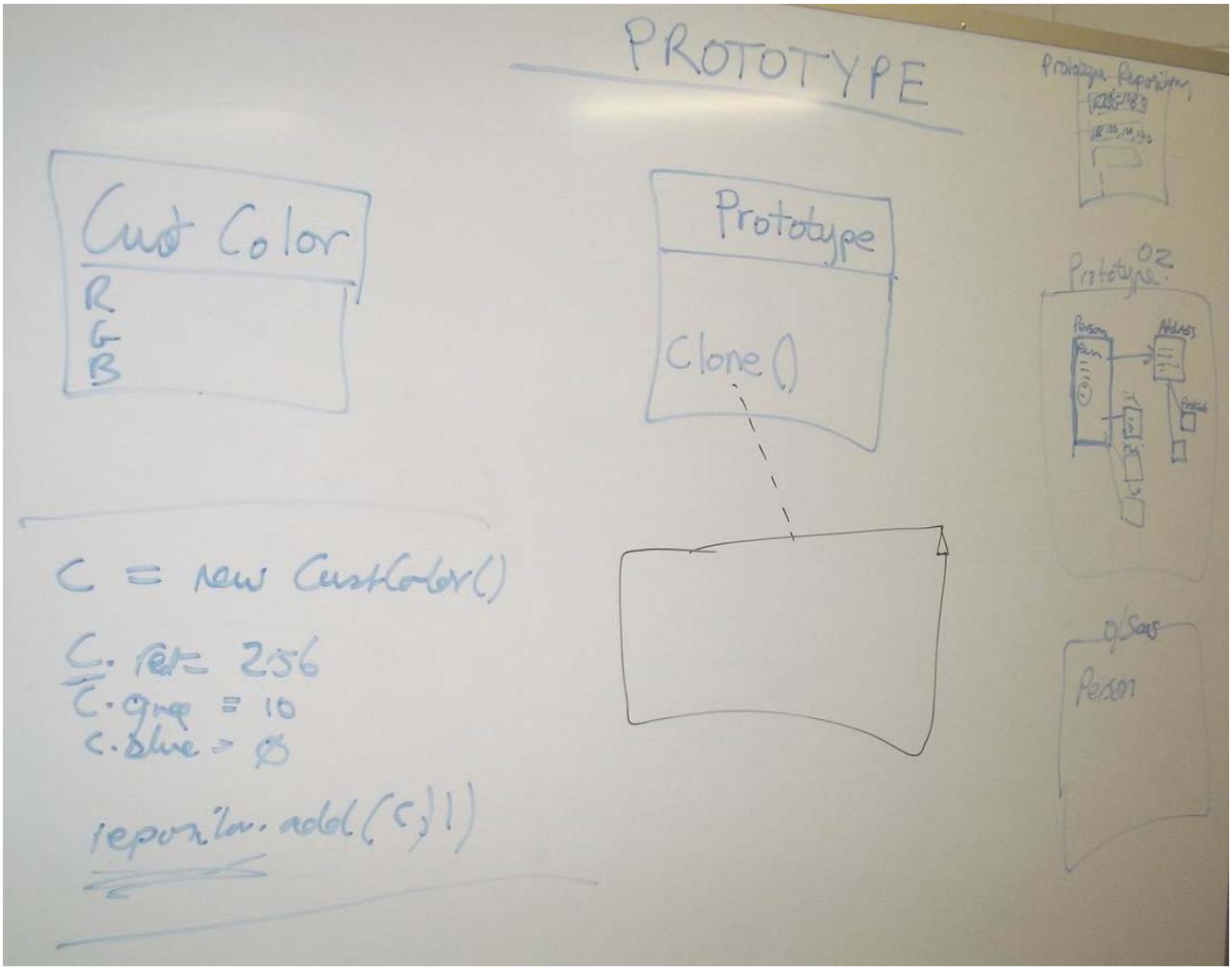


Figure 74 - A prototype could be a complex object with lots of properties - thus easier to call Clone() rather than build it from scratch.

Andy's Tips

- Java has a cloneable interface
- C# copy constructor (o) -> o'
- Deep vs. Shallow copy
- Use serialisation to deep copy. Watch out for circular references.
- Use a prototype registry (loaded from disk) from which to clone. Dynamic loading is a plus.
- Create new instances by copying a prototype – a complex object, which has been streamed/pickled to disk. Example, a graphics library, a character editor. Can dynamically define new types of 'classes' by composing a complex object, then registering it as a prototype! :-) No need for a special class or programmer intervention. E.g. www.dofactory.com has a good C# example of a custom colour repository.

- Initialisation issue – no parameters on the clone() method allowed. Thus add your own Init() or Start() method to the cone class and call it after the clone operation has completed.

Code Ideas

www.dofactory.com's structural code demonstrates the Prototype pattern in which new objects are created by copying pre-existing objects (prototypes) of the same class.

www.dofactory.com's real-world code demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.

Example

```
p2 = p.clone()
while drag
    p2.draw(newpos)
insert p2 into drawing
```

Rules of thumb

Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126]. Abstract Factory, Builder, and Prototype can use Singleton in their implementations. [GOF, p81, 134].

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Method: creation through inheritance (specifically, polymorphic override). Protoype: creation through delegation [is this right??].

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Protoype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an **"initialize" operation**. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. [GOF, p126]

Non Software Example

The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotvpe result. In other words, the cell clones itself. [Michael Duell, "Non-software examples

of software design patterns", Object Magazine, Jul 97, p54]

Notes:

State

Alter an object's behavior when its state changes. Put the “state” in a class and delegate to it.

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterised by large and numerous case statements that vector flow of control based on the state of the application.

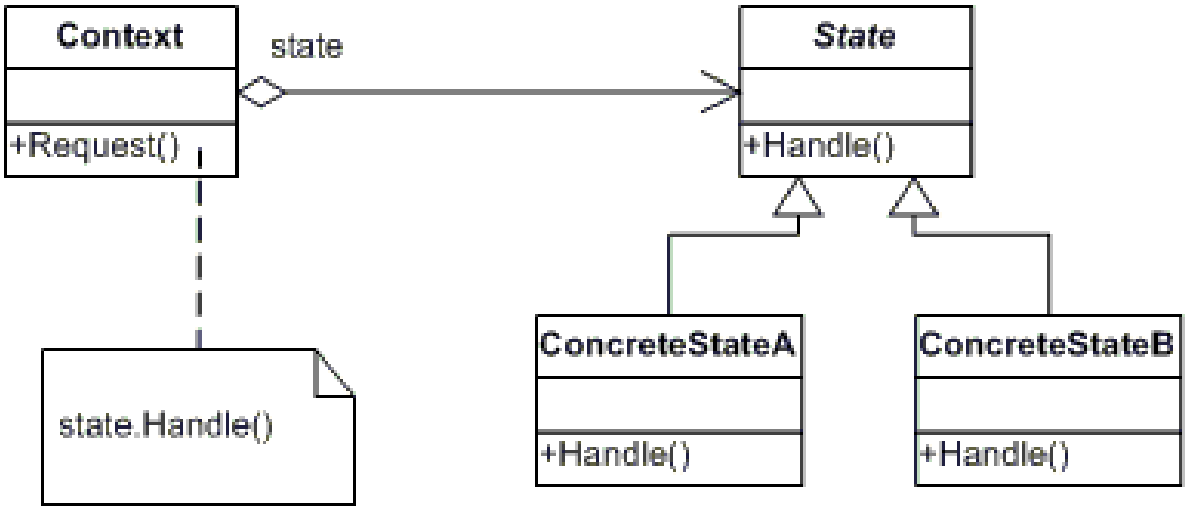
> **Problem**
> When the state space of a class, C, is modified, then all member functions of C that depend on the current state of their implicit parameter must also be modified. Failing to modify any one of these functions introduces a bug into the program. This can be especially hazardous if there are many member functions that need to be modified or if the state space changes frequently.
> **Solution**
> If the state space of C is relatively small, then we can associate with each state a distinct subclass of an abstract State base class. If x is an instance of C, then the state of x is given by an associated instance of one of these classes. Each state-dependent member function of C simply forwards client requests to the associated state object. Of course the behavior of these functions will vary according to the type of the associated state object, but that's the point. Modifying the state space of C is simply a matter of adding, removing, or modifying classes derived from the State base class. No changes to C are necessary.

Ideas

- Components shall be capable of "morphing" their behavior at run-time.
- The system architecture shall be characterized by a "finite state machine". The state machine will need to support **application logic** at each state transition, *not simply the transition itself*. [A "table-driven" approach routinely supports only the latter.]
- "case" statements are a maintenance nightmare - they shall be avoided.

Right, so changing *state* is one thing. You can do that with a simple attribute of enums. The tricky part is to change *behaviour* too.

State is like Strategy except in its intent. [Coplien, Mar96, p88]


Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Adding a new font – why we need state pattern

```
Font = all fonts currently available on a typical computer
```

we will still need to occasionally add fonts as our definition of "typical" changes.
To see why this is a problem, assume we temporarily define font as follows:

```
enum Font { Times, Roman, Helvetica, Courier };
```

Consider the implementation of the display() method:

```
void Text::display(int x, int y)
{
        switch(font)
        {
              case Helvetica:
                    // display text in Helvetica at position (x, y);
                    break;
              case Times:
                    // display text in Times at position (x, y);
                    break;
              case Roman:
                    // display text in Roman at position (x, y);
                    break;
              case Courier:
                    // display text in Courier at position (x, y);
                    break;
              default:
                    throw AppError("font unavailable");
        }
}
```

Now suppose we want to add a new font called Dingbats to our "typical" computer. Of course we need to modify the definition of Font:

```
enum Font { Times, Roman, Helvetica, Courrier, Dingbats };
```

We also need to add a new case clause to the display() method:

```
case Dingbats:
        // display text in Dingbats at position (x, y);
        break;
```

But we will also need to add this case clause to all other functions that depend on the state of a text object. There could be many functions like this. Failing to add the case clause to just one of them introduces a bug into existing code.

In situations where the state space of a class is open ended, it is sometimes useful to decouple an object from its current state. This is the idea behind the State Pattern.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

The State pattern does not specify **where the state transitions** will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses. [GOF, p308]

More Discussion

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add **actions** to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accomodating state transition actions. [GOF, p308]

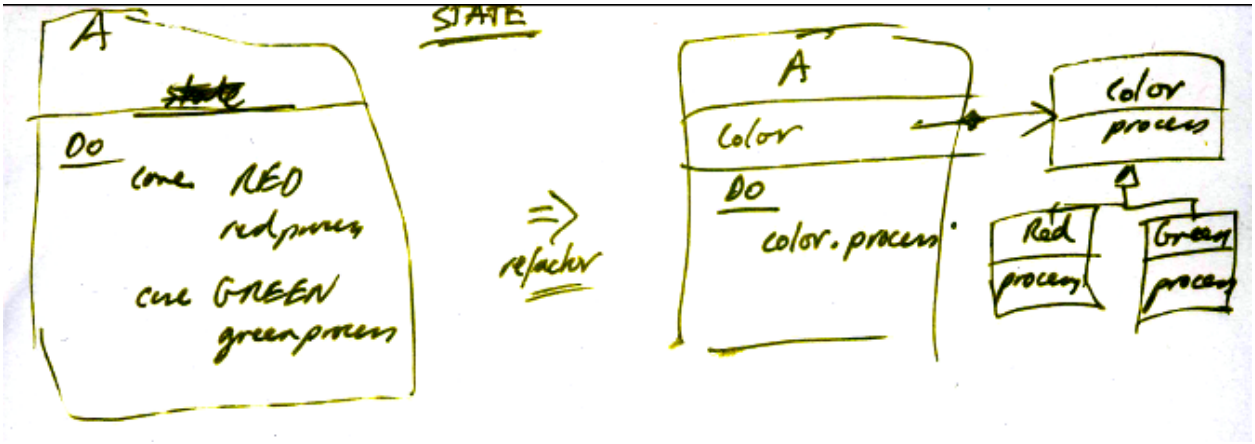
The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object **causes it to select from its "palette" of Strategy objects.** [Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 1999, p253]

Participants

The classes and/or objects participating in this pattern are:

- Context (Account)
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- State (State)
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- Concrete State (RedState, SilverState, GoldState)
  - each subclass implements a behavior associated with a state of Context

Example 1: - Representing the colour of a car using state pattern



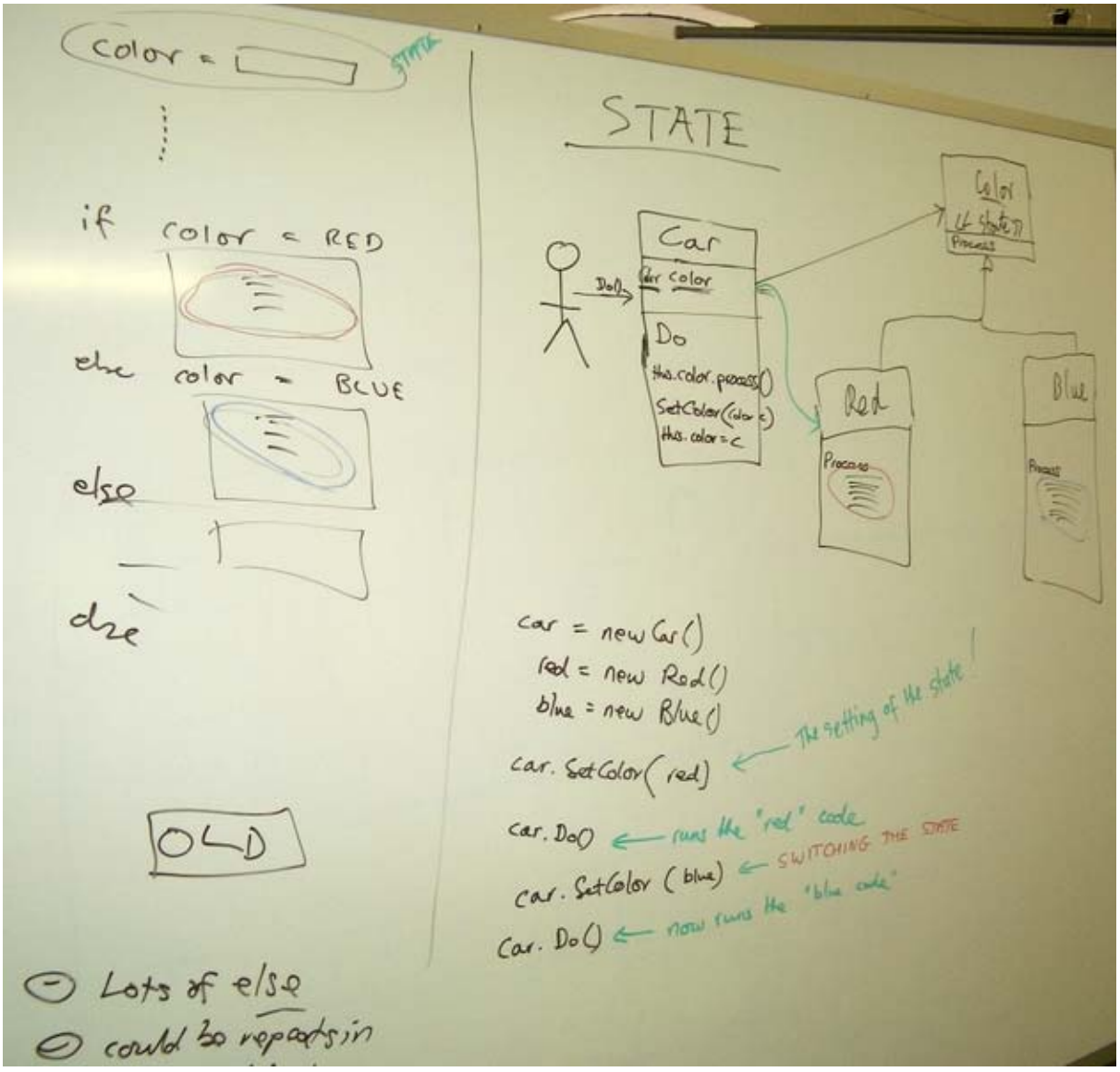


Figure 75 – Before and after - representing the colour of a car using state pattern

The representation of **state** “attributes” by using a delegated class is one thing. Putting **behavior** on those states is another important aspect of state pattern. The above car examples use the methods Process() and Do() as example behaviors (*rather artificial conjured up behaviors notwithstanding* ☺).

Example 2: Using State Pattern to model Traffic Lights – Before and After

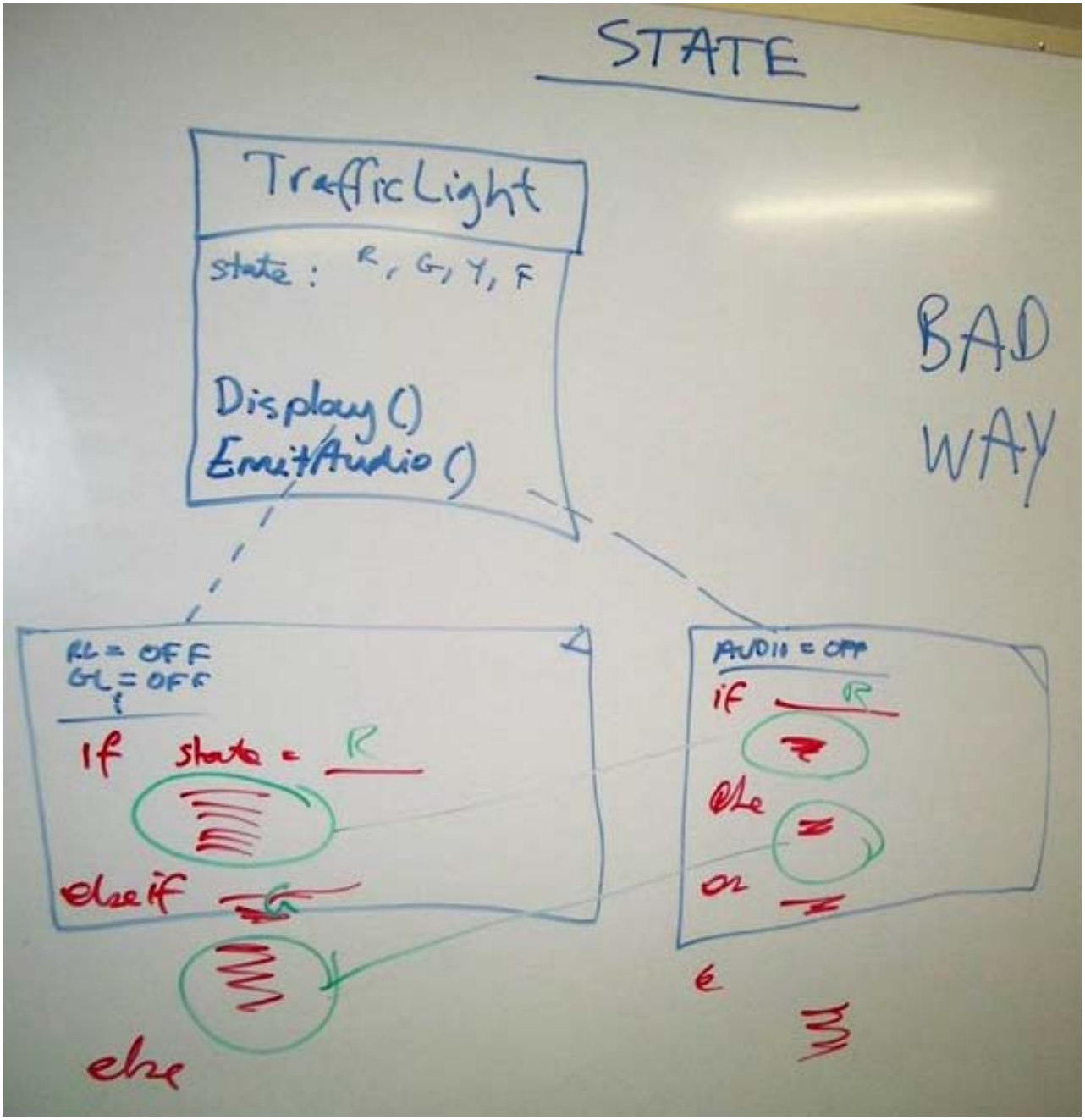


Figure 76 - So what if you want to add the new "Orange" state? You usually have to modify code in a number of places.

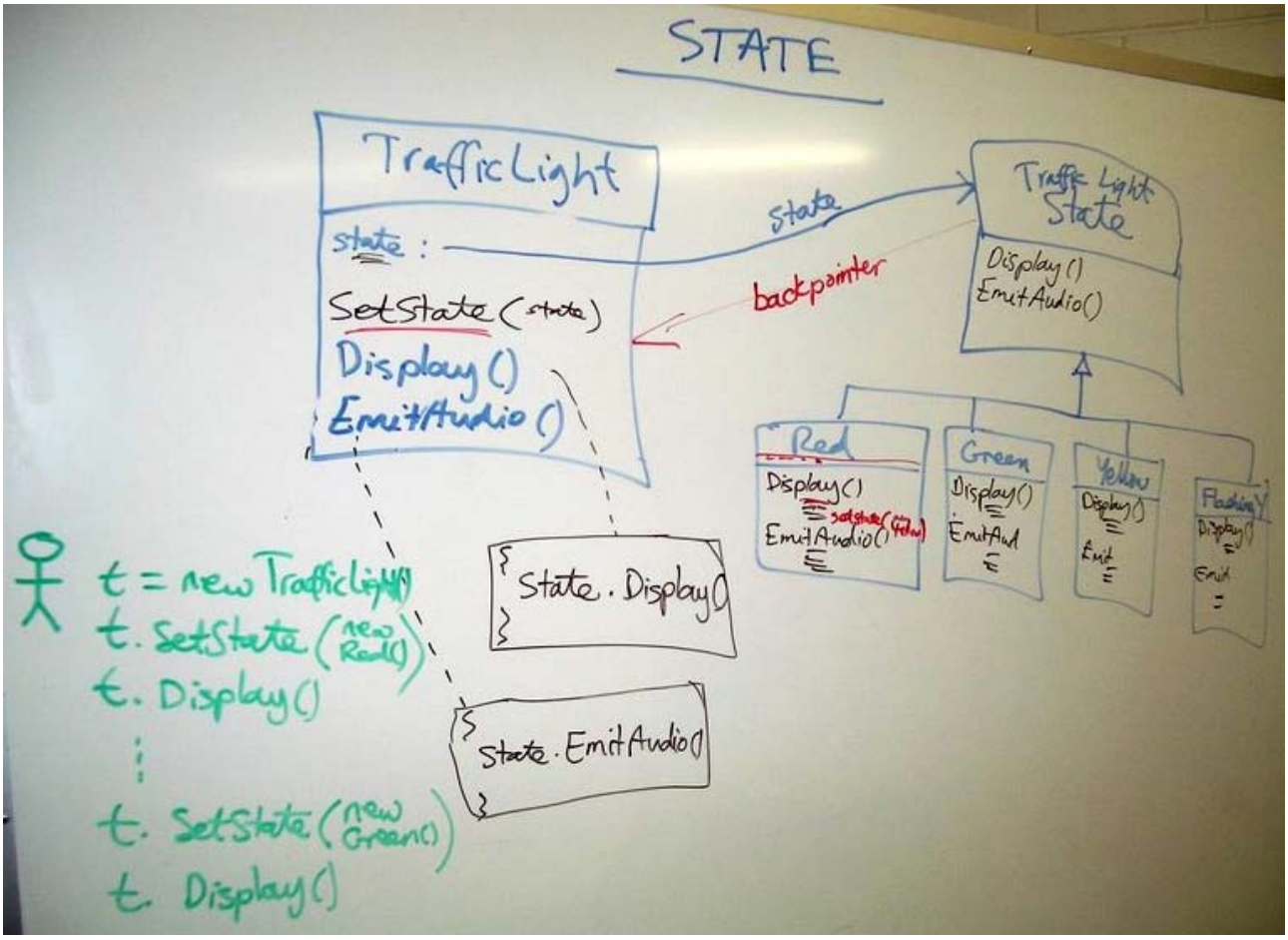


Figure 77 - Solution to traffic light modelling - using state pattern

Andy's Tips

One main issue in state pattern **is how do state instances access attributes in the context**. Simplest solution is to have a pointer back from each state object back to the context. Another solution is to completely decouple the state from the context and simply pass the state instance what it needs e.g. as parameters to state methods (as defined in the the state interface)

In both examples above the demo code (showing how to instantiate and use the classes) holds the state instances and controls the switching of states. Often the states themselves or the context (e.g. car) will take on this responsibility. If the states are given the power to set the next state – they will need need access to the context (via a backpointer) so that they can call a method called e.g. SetState().

You often want to have singleton states rather than recreating each state time and time again, and having to copy across any state attributes each time. Of course by keeping most of the attributes in the context, there is not much data in state to copy between state instance when the state switches.

Rules of thumb

State objects are often Singletons. [GOF, p313]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Interpreter can use State to define parsing contexts. [GOF, p349]

State is like Strategy except in its intent. [Coplien, C++ Report, Mar 96, p88]

Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, C++ Report, Mar 96, p88]

State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, Advanced C++, p58; see discussion under Bridge pattern]. They differ in **intent** - that is, they **solve different problems**.

The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently. [Coplien, C++ Report, May 95, p58]

Code Ideas

www.dofactory.com's structural code demonstrates the State pattern which allows an object to behave differently depending on its internal state. The difference in behavior is delegated to objects that represent this state.

www.dofactory.com's real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

Example 3: Modeling bank accounts using state pattern

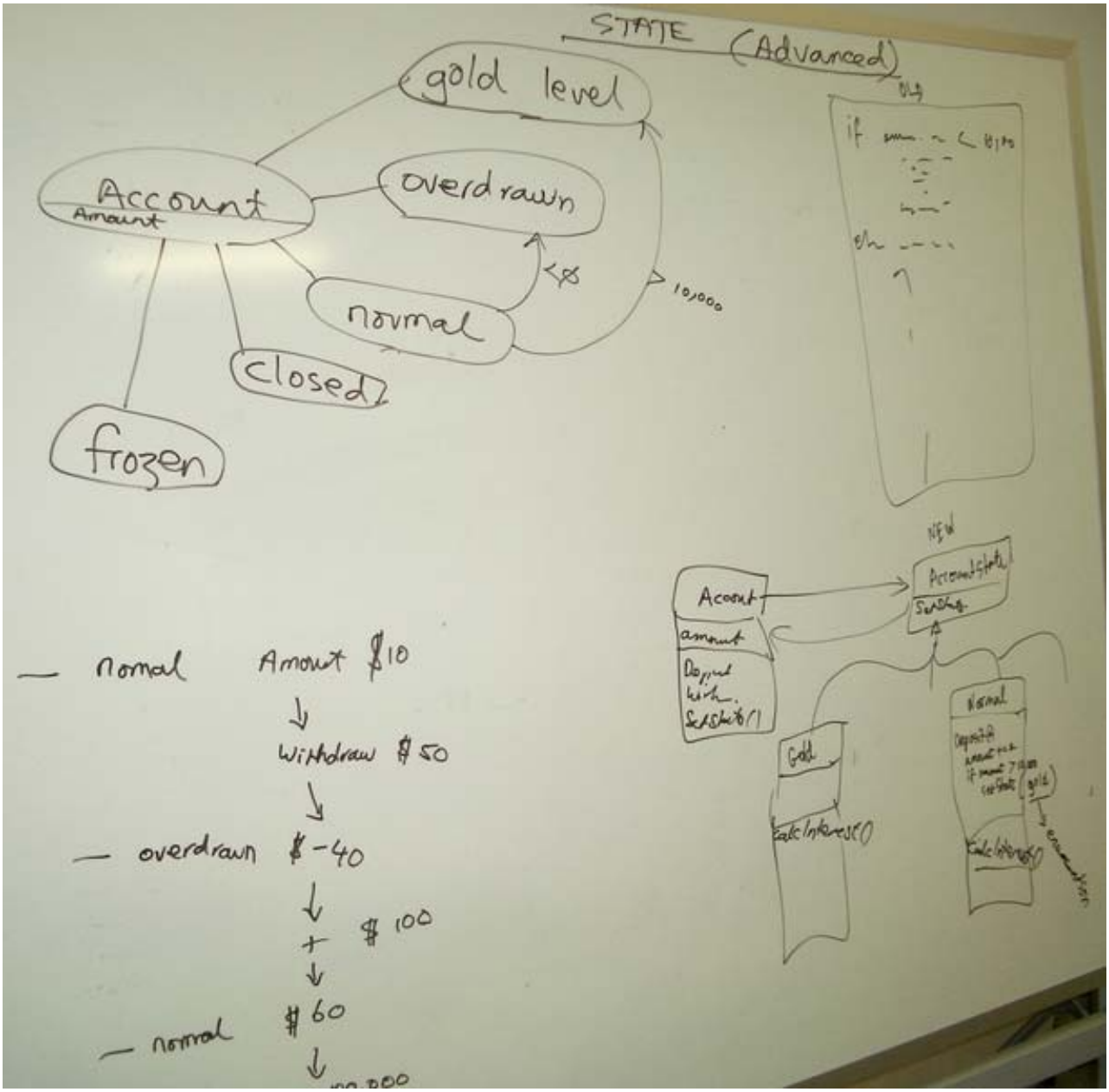


Figure 78 - Modeling bank accounts using state pattern

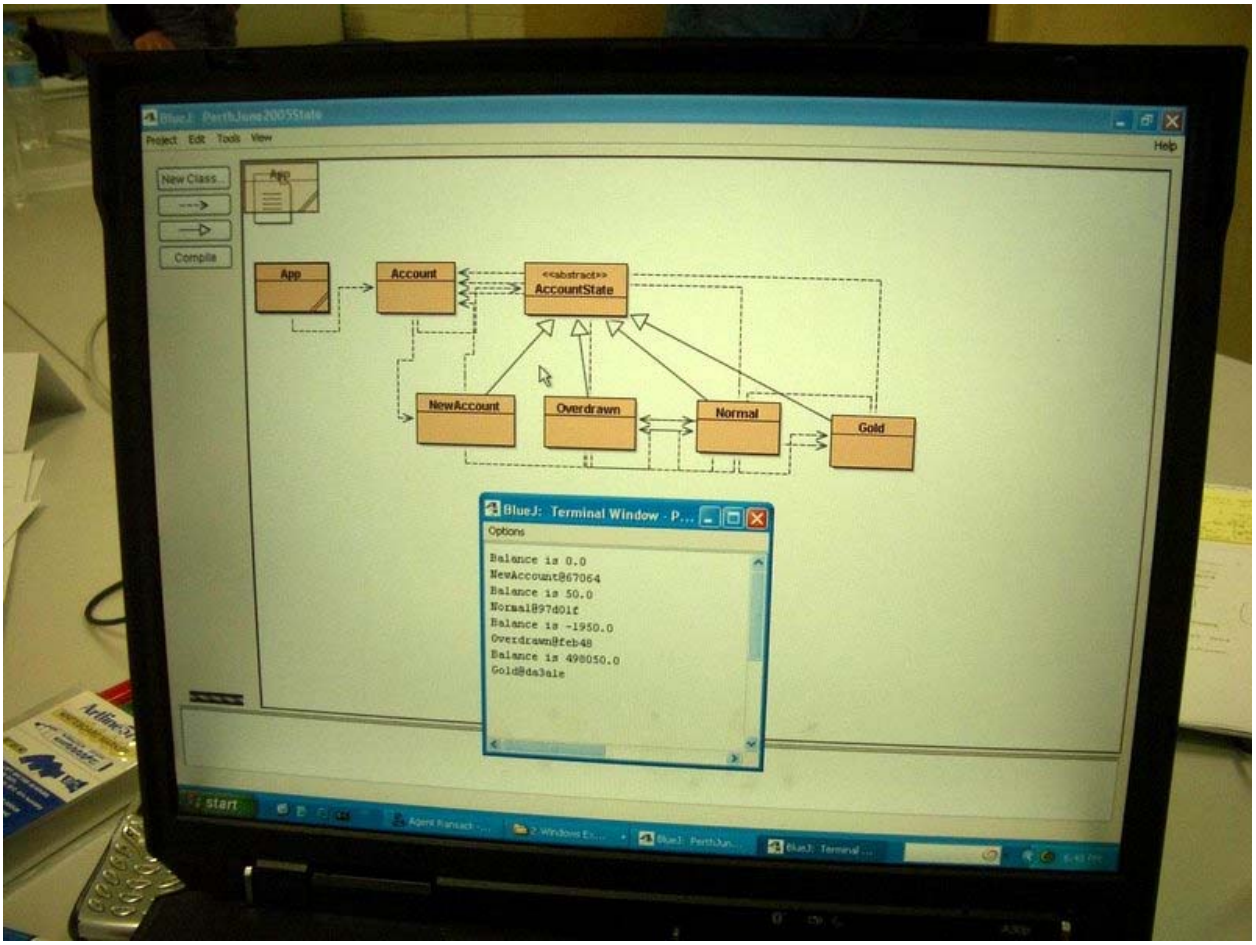


Figure 79 - Back Account State Pattern Example (UML done in BlueJ)

Account.java
Java Language Source file
1 KB

App.java
Java Language Source file
1 KB

DodgyAccountState.java
Java Language Source file
1 KB

GoldState.java
Java Language Source file
1 KB

OverdrawnState.java
Java Language Source file
1 KB

State.java
Java Language Source file
1 KB

```
public class App
{
    static Account account = new Account("Bill Gates");

    public static void main(String[] args) {
        Report();
        account.Deposit(50);
        Report();
        account.Withdraw(2000);
        Report();
        account.Deposit(500000);
        Report();
       }

   public static void Report() {
       System.out.println("Balance is " + account.balance);
       System.out.println(account.state);
   }
}


public class Account
{
    public AccountState state;
    public String ownername;
    public double balance;

    public Account(String name)
    {
        this.ownername = name;
        this.SetState(new NewAccount(this));
     }

    public void Deposit(double amount)
    {
        //this.balance += amount;
        this.state.Deposit(amount);
    }
    public void Withdraw(double amount)
    {
        //this.balance -= amount;
        this.state.Withdraw(amount);
    }

    public void SetState(AccountState newstate)
    {
        this.state = newstate;
    }

}
```

```
public abstract class AccountState
{
      public abstract void Deposit(double amount);
      public abstract void Withdraw(double amount);
}



public class NewAccount extends AccountState
{
      private Account account;

      public NewAccount(Account account)
      {
            this.account = account; // backpointer
      }

    public void Deposit(double amount)
    {
        this.account.balance += amount;
        if (amount > 0) {
            this.account.SetState(new Normal(this.account));
        }
    }
    public void Withdraw(double amount)
    {
        return;  // you have got to be kidding - new account has no money
    }
}


public class Overdrawn extends AccountState
{
      private Account account;

      public Overdrawn(Account account)
      {
            this.account = account; // backpointer
      }

    public void Deposit(double amount)
    {
        this.account.balance += amount;
        if (this.account.balance > 0) {
            this.account.SetState(new Normal(this.account));
        }
        if (this.account.balance > 250000) {
            this.account.SetState(new Gold(this.account));
        }
    }
    public void Withdraw(double amount)
    {
        return;  // can't withdraw from an already opverdrawn account!
    }
}
```

```
public class Normal extends AccountState
{
      private Account account;

      public Normal(Account account)
      {
            this.account = account; // backpointer
      }

    public void Deposit(double amount)
    {
        this.account.balance += amount;
        if (this.account.balance > 250000) {
            this.account.SetState(new Gold(this.account));
        }
    }
    public void Withdraw(double amount)
    {
        this.account.balance -= amount;
        if (this.account.balance < 0) {
            this.account.SetState(new Overdrawn(this.account));
        }
    }
}


public class Gold extends AccountState
{
      private Account account;

      public Gold(Account account)
      {
            this.account = account; // backpointer
      }

    public void Deposit(double amount)
    {
        this.account.balance += amount;
    }
    public void Withdraw(double amount)
    {
        this.account.balance -= amount;
        if (this.account.balance < 0) {
            this.account.SetState(new Overdrawn(this.account));
        }
    }
}
```

Coding Quiz:

Question: If the "balance" was not kept in the account class, but instead was kept in the State class – how would you preserve the balance when the states change?

Answer: states need to pass themselves to each state as part of the state constructor, in order to preserve the balance attribute.

For example - could have two constructors - one for startup and one for state change.

```
  // Constructors
  public GoldState( double balance, Account account )
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }
 public GoldState( State state )
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }
```

Notes:

Visitor

Defines a new operation to a class without changing the class.

Visitor implements "double dispatch".

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.

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Discussion

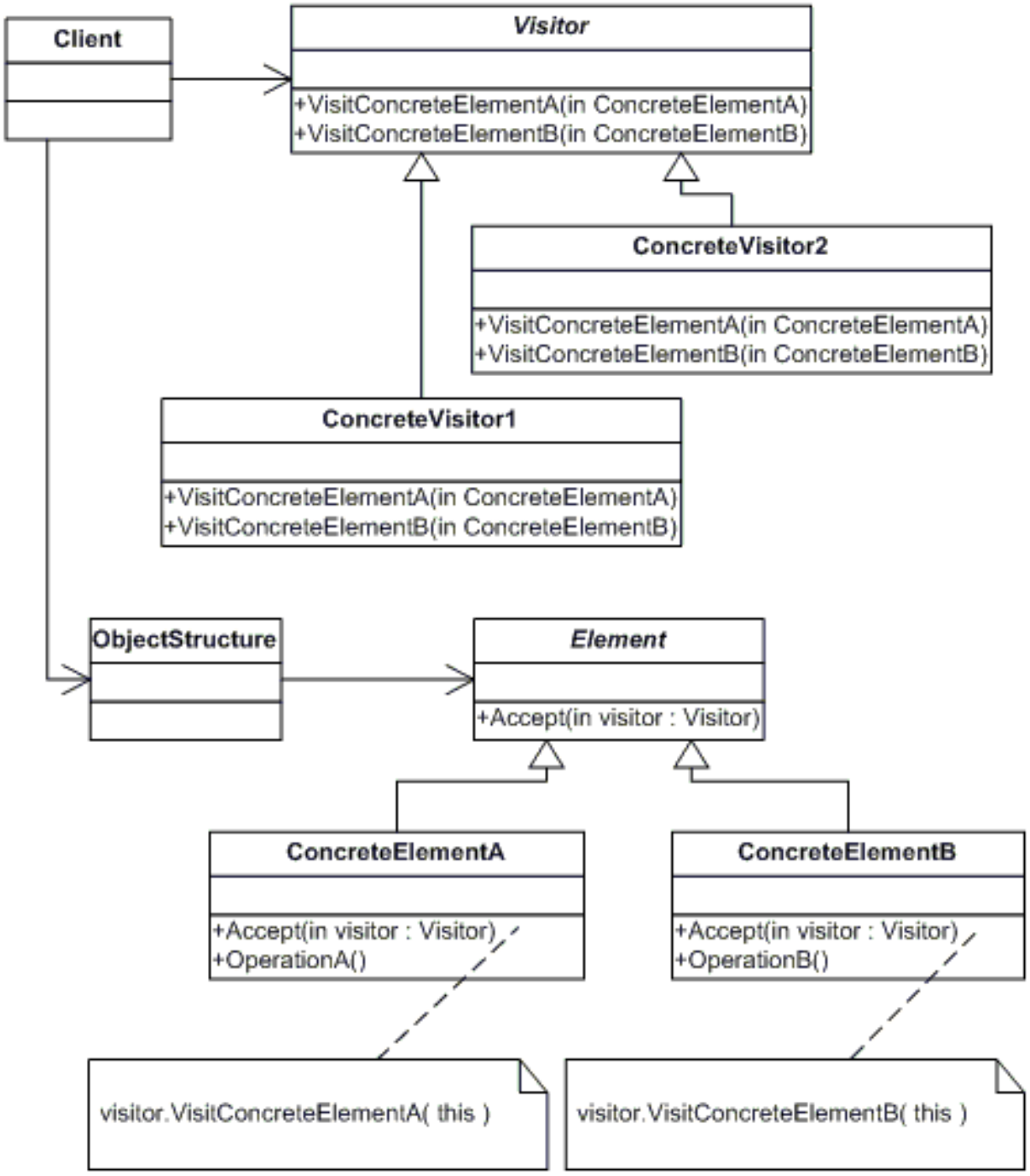
Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects. The approach encourages designing lightweight Element classes - because processing functionality is removed from their list of responsibilities. New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.

Visitor implements "double dispatch". OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver. In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).

The implementation proceeds as follows. Create a Visitor class hierarchy that defines a pure virtual visit() method in the abstract base class for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a pointer or reference to an original Element derived class.

Structure

This diagram is from javacoder.net. The NodeVisitor hierarchy is not modeled here. The first dispatch is accept(), and the second dispatch is visit(). Inside each visit() method, we now know the exact type of both the Node entity (the object being passed) and the NodeVisitor entity (the object being messaged).


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

More Discussion

Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

Add a single pure virtual accept() method to the base class of the Element hierarchy. accept() is defined to receive a single argument - a pointer or reference to the abstract base class of the Visitor hierarchy.

Each concrete derived class of the Element hierarchy implements the accept() method by simply calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as the sole argument.

Everything for "elements" and "visitors" is now set-up. When the client needs an operation to be performed, (s)he creates an instance of the Vistor object, calls the accept() method on each Element object, and passes the Visitor object. The accept() method causes flow of control to find the correct Element subclass. Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch.

The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class. But, if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort.

An acknowledged objection to the Visitor pattern is that is represents a regression to functional decomposition - separate the algorithms from the data structures. While this is a legitimate interpretation, perhaps a better perspective/rationale is the goal of promoting non-traditional behavior to full object status.

Andy's Tips

- Visitor is basically a way of getting different code called (using double dispatch), depending on the item you are "visiting". It haves you having big if-then-else statements.
- Visitor uses iterator (either internal or external) to drive the visitor across the items of the collection

Code Ideas

www.dofactory.com's structural code demonstrates the Visitor pattern in which an object traverses an object structure and performs the same operation on each node in this structure. Different visitor objects define different operations.

www.dofactory.com's real-world code demonstrates the Visitor pattern in which two objects traverse a list of Employees and performs the same operation on each Employee. The two visitor objects define different operations -- one adjusts vacation days and the other income.

Code Idea – How to drive the visiting of the collection from the visitor itself e.g.

```
  v.VisitAll(collectionIterator)

-----------
  Visitor
 -----------
  VisitAll(collectionIterator)
        o = collectionIterator.next()
        o.AcceptVisitor(this)
  VisitA()
  VisitB()
  VisitC()
 ------------
```

Participants

The classes and/or objects participating in this pattern are:

- Visitor (Visitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface
- ConcreteVisitor (IncomeVisitor, VacationVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- Element (Element)
  - defines an Accept operation that takes a visitor as an argument.
- ConcreteElement (Employee)
  - implements an Accept operation that takes a visitor as an argument
- ObjectStructure (Employees)
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set

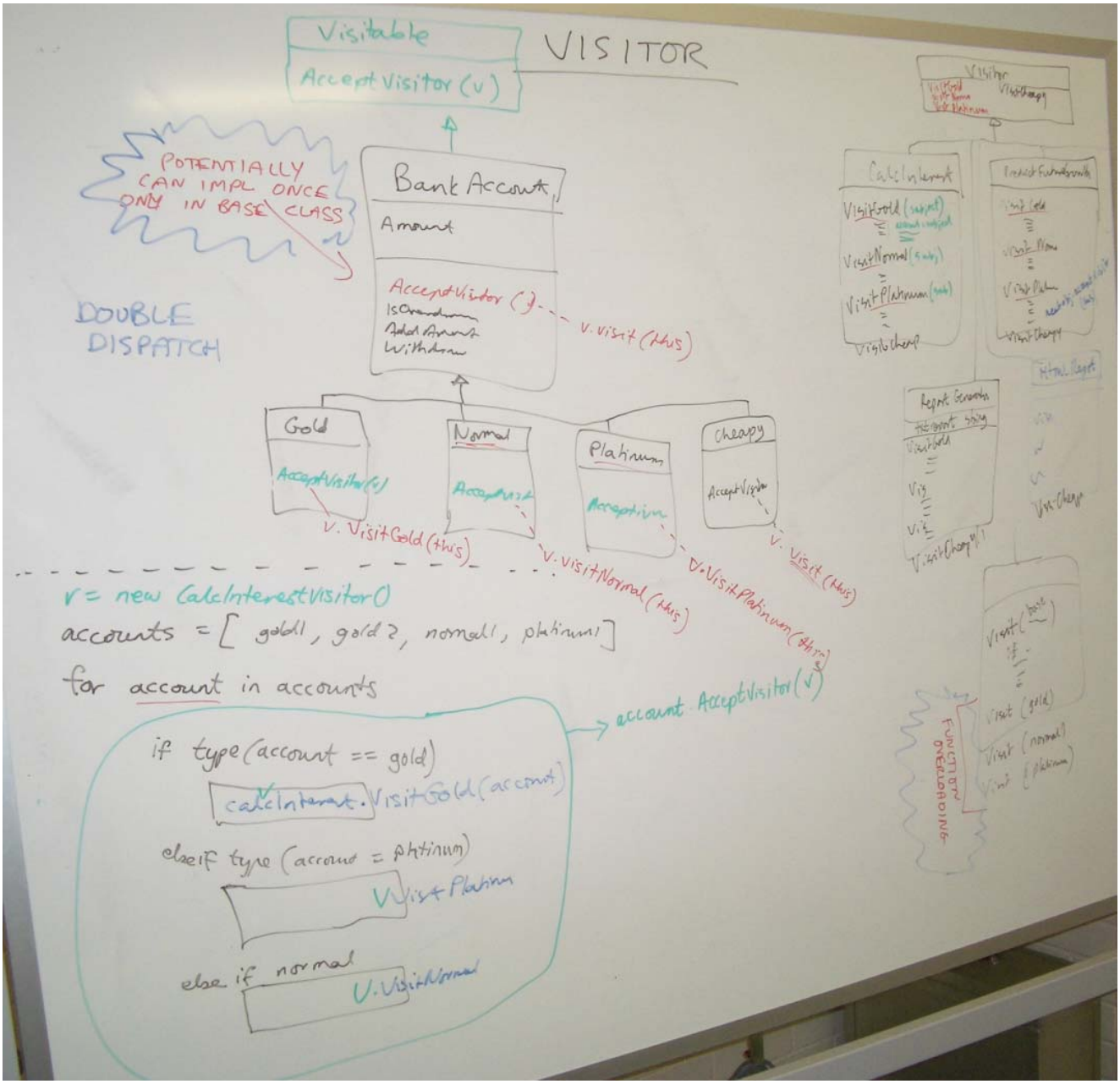


Figure 80 - Visitor as a way to add functionality to bank account class without changing the bank account class itself

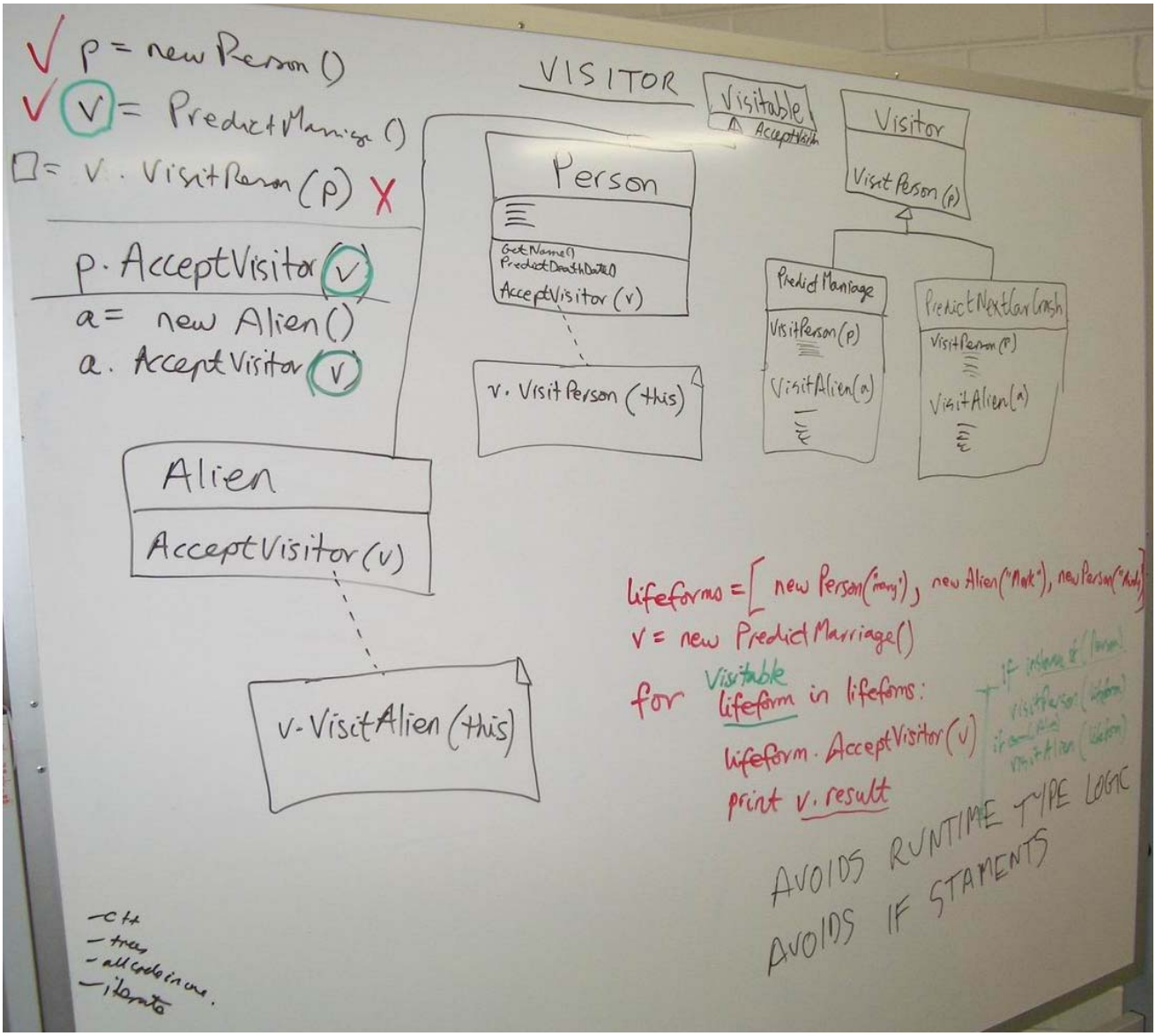


Figure 81 - Adding functionality to the Person class using Visitor Pattern

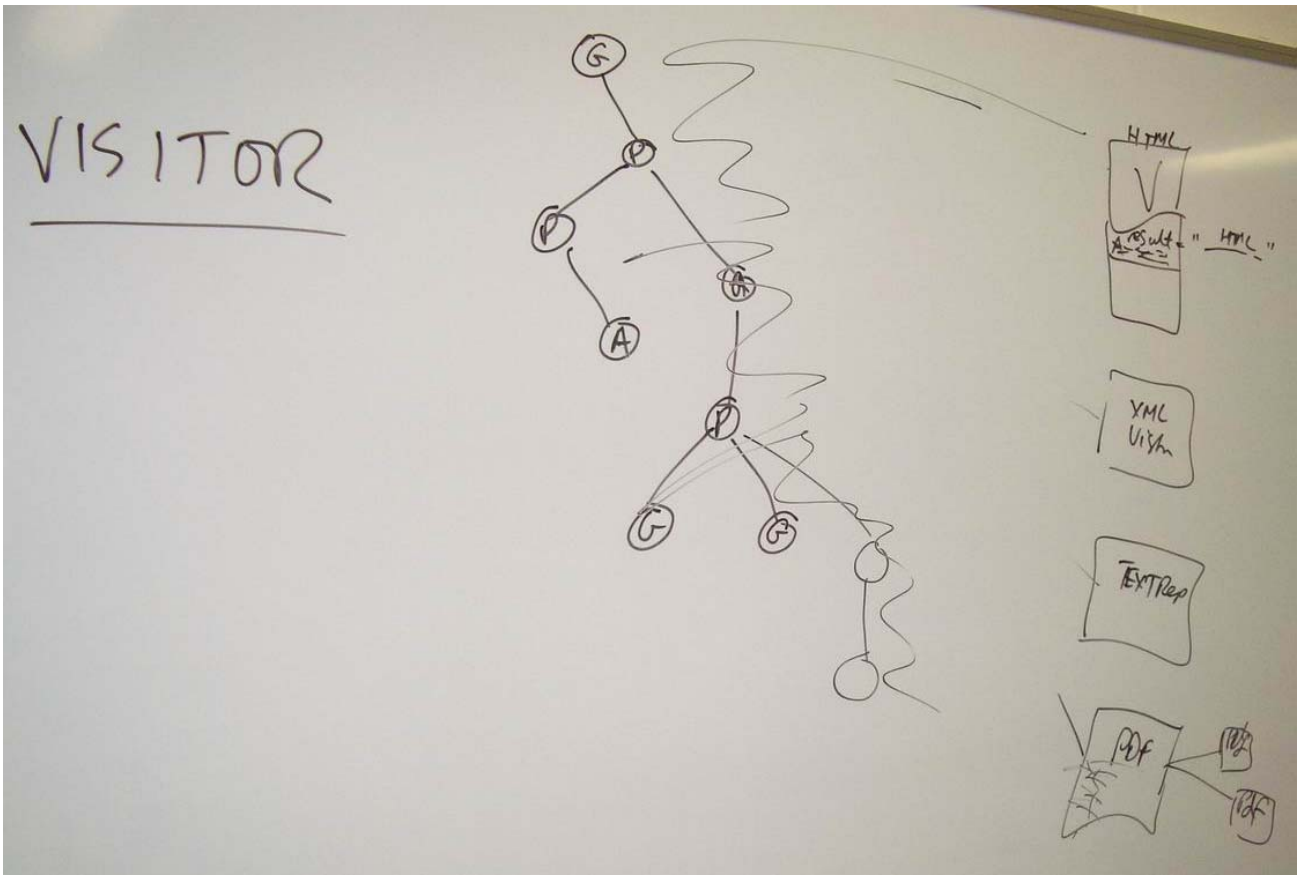


Figure 82 - Using Visitor in the context of iteration. Various visitors iterate over a data structure to produce different report types

Rules of thumb

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters. [Johnson, Huni, Engel, 1995, p8]

The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts. [Vlissides, "Type Laundering", C++ Report, Feb 97, p48]

Non Software Example

The Visitor example involves an outside consultant coming into a department to meet with employees. While the visitor is escorted to each cubicle, she does nothing. Once she arrives at a cubicle, she introduces herself, and the employee introduces himself (double dispatch). Following the introduction, the consultant springs into action interviewing the employee (sending messages and obtaining results). The basic idea is that the consultant is not required to have knowledge of the organization's structure to do her job. The escort supplies this knowledge.

Ideas

- The system shall allow the operations that can be performed on "whole-part" hierarchical assemblies of components to be defined and added without having to change (and potentially break) any existing code.
- The system shall decouple "data structures" from "algorithms" so that each can be developed, maintained, and used independent of the other.
- The design shall support the recovery of "lost type information" without resorting to the overhead associated with RTTI.
- "case" statements are a maintenance nightmare - they shall be avoided.
- The system shall be "open for extension, but closed for modification".

More Discussion

JavaPro has an article by James Cooper

The November 2000 issue of JavaPro has an article by James Cooper (author of a Java companion to the GoF) on the Visitor design pattern. He suggests it "turns the tables on our object-oriented model and creates an external class to act on data in other classes ... while this may seem unclean ... there are good reasons for doing it."

His primary example. Suppose you have a hierarchy of Employee-Engineer-Boss. They all enjoy a normal vacation day accrual policy, but, Bosses also participate in a "bonus" vacation day program. As a result, the interface of class Boss is different than that of class Engineer. We cannot polymorphically traverse a Composite-like organization and compute a total of the organization's remaining vacation days. "The Visitor becomes more useful when there are several classes with different interfaces and we want to encapsulate how we get data from these classes."

[It also is useful when you want to perform the right algorithm, based on the type of two (or more) objects (aka "double dispatch"). Example algorithms might be: process a "collision" between different kinds of SpaceGame objects, compute the distance between different kinds of shapes, or compute the intersection between different kinds of shapes.]

His Benefits for Visitor include:

- Add functions to class libraries for which you either do not have the source or cannot change the source
- Obtain data from a disparate collection of unrelated classes and use it to present the results of a global calculation to the user program
- Gather related operations into a single class rather than force you to change or derive classes to add these operations
- Collaborate with the Composite pattern

Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended.

Notes:

Flyweight

A fine-grained instance used for efficient sharing

Intent

Use sharing to support large numbers of fine-grained objects efficiently.

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

Discussion

The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

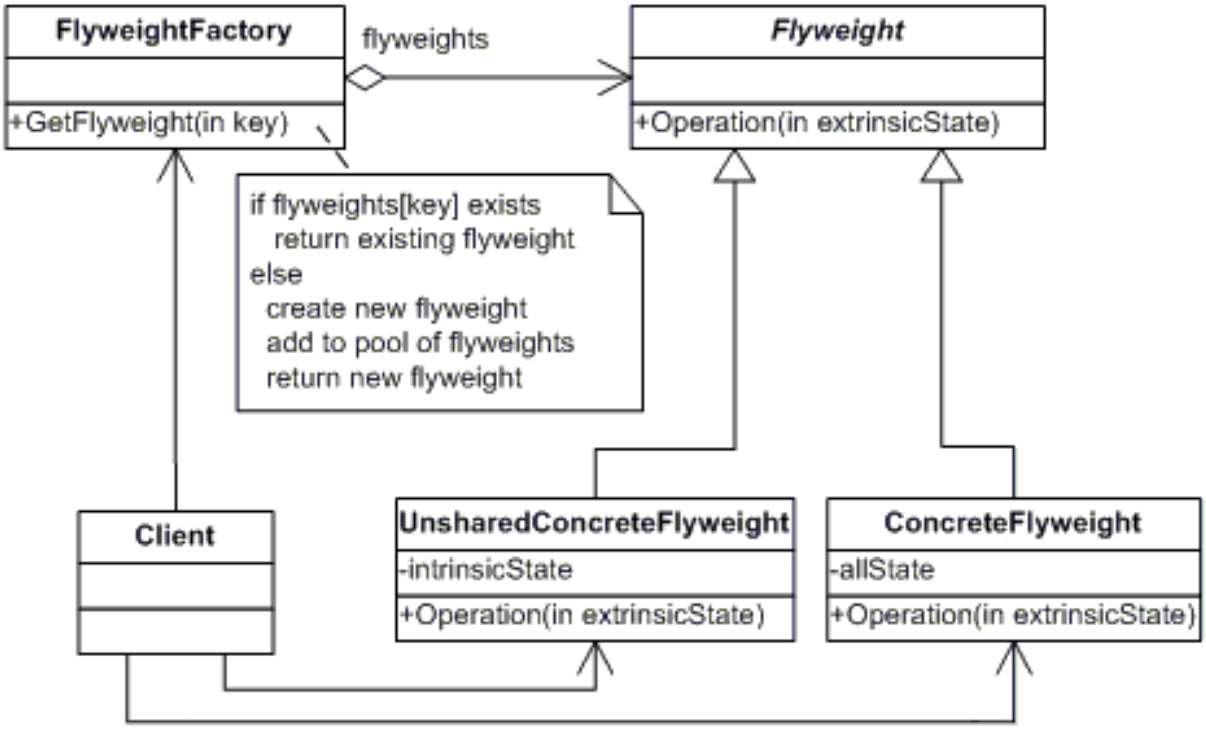
An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

Structure

This diagram is from javacoder.net. Note that Row and Column are playing the role of Composite in the Composite pattern. The Context parameters are providing the requisite extrinsic state to each method. No "factory" is modeled here for serving up flyweight objects.

The system shall employ hundreds of components at very fine levels of granularity without prohibitive cost.

The system shall support pooling and reuse of a finite supply of "x" across an inexhaustible demand.


FlyweightFactory
flyweights
Flyweight
+GetFlyweight(in key)
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweight
Client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

Participants

The classes and/or objects participating in the Flyweight pattern are:

- Flyweight (Character)
  - declares an interface through which flyweights can receive and act on extrinsic state.
- ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.
- UnsharedConcreteFlyweight (not used)
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- FlyweightFactory (CharacterFactory)
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects supplies an existing instance or creates one, if none exists.
- Client (FlyweightApp)
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

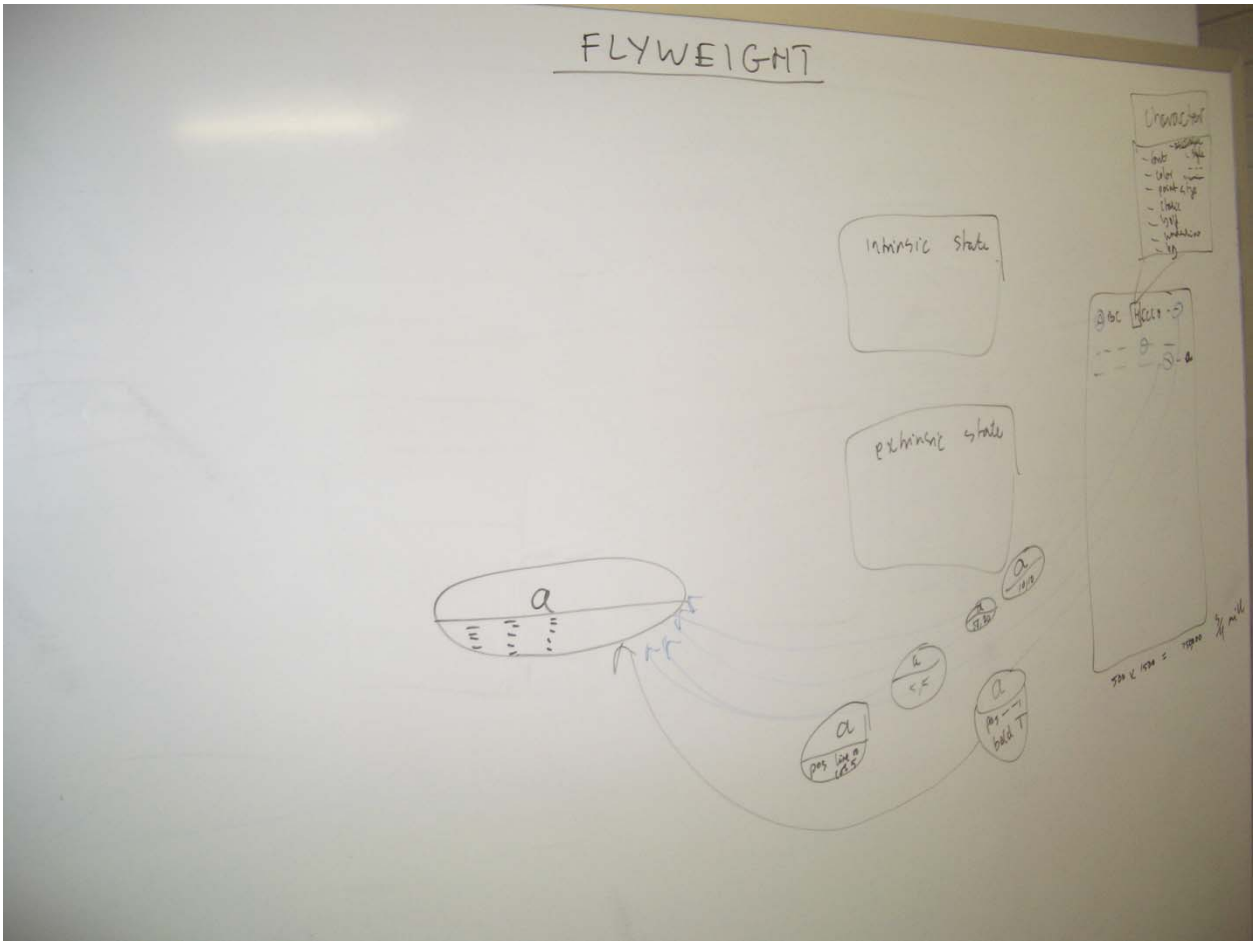


Figure 83 - Basic Idea of Flyweight

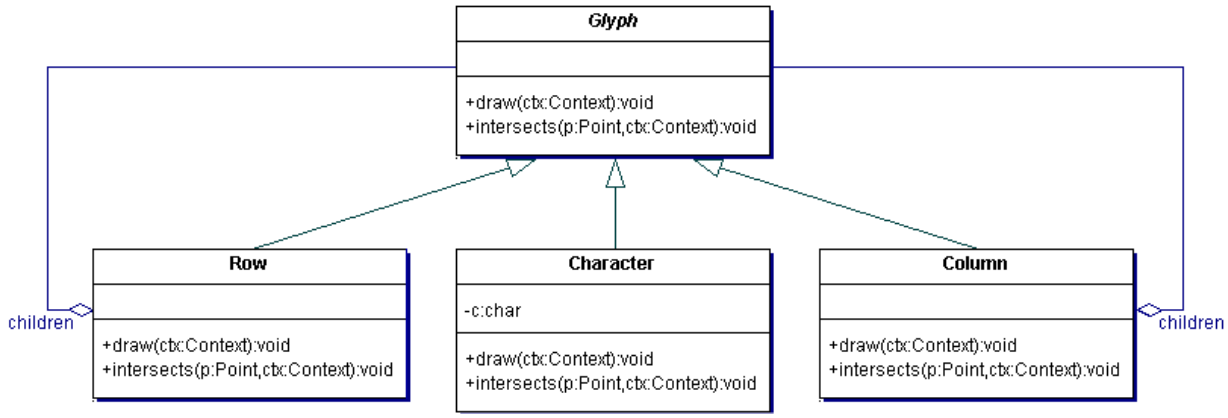
Code Ideas

www.dofactory.com's structural code demonstrates the Flyweight pattern in which a relatively small number of objects is shared many times by different clients.

www.dofactory.com's real-world code demonstrates the Flyweight pattern in which a relatively small number of Character objects is shared many times by a document that has potentially many characters.

Andy's Tips

- Basically you have a problem of say, millions of objects - which is expesive to store, so you refactor the common stuff out into a single object (intrinsic state) and the varying stuff into extrinsic state.
- Extrinsic state might be implemented in a very efficient manner e.g. not separate objects but an array or database or binary bits of a number etc.
- Flyweight is not a pattern you would typically use very often.



Example

The Flyweight uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF. p255]

Flyweight explains when and how State objects can be shared. [GOF, p313]

Notes:

Interpreter

A way to include language elements in a program

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

Discussion

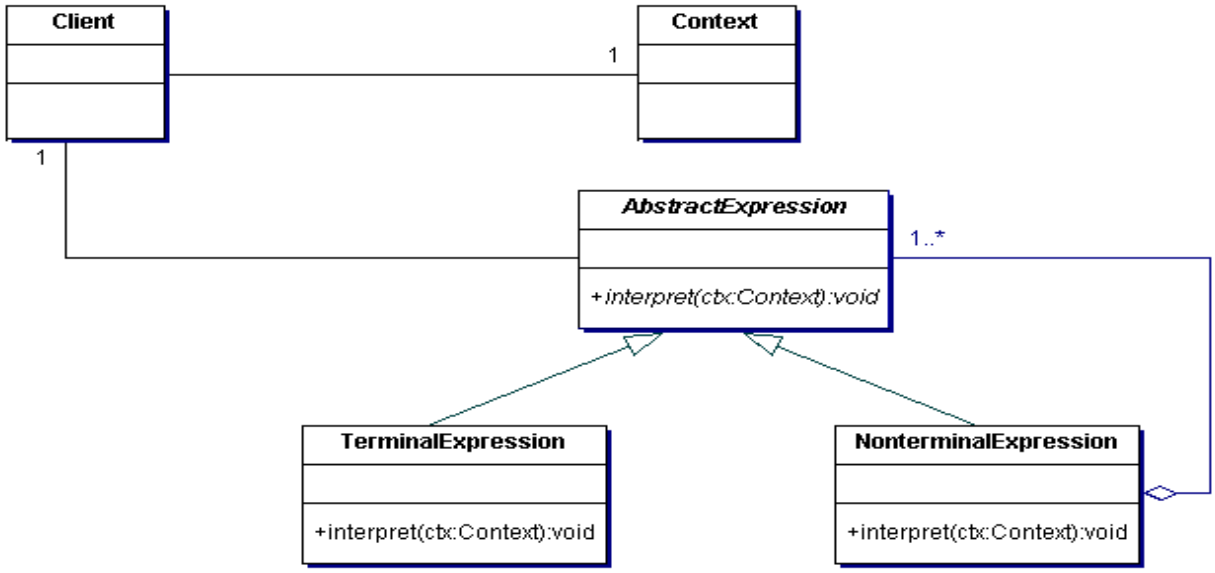
The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

An abstract base class specifies the method interpret(). Each concrete subclass implements interpret() by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

Non Software Example

The Intepreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

The system shall characterize the domain as a set of rules, then define a language capable of specifying the rules, and a grammar for defining the language, and an "engine" for interpreting the grammar.


Client
Context
1
1
AbstractExpression
+interpret(ctx:Context):void
1..*
TerminalExpression
+interpret(ctx:Context):void
NonterminalExpression
+interpret(ctx:Context):void

Participants

The classes and/or objects participating in this pattern are:

- AbstractExpression (Expression)
  - declares an interface for executing an operation
- TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.
- NonterminalExpression (not used)
  - one such class is required for every rule R ::= R1R2...Rn in the grammar
  - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
- Context (Context)
  - contains information that is global to the interpreter
- Client (InterpreterApp)
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - invokes the Interpret operation

Basic Ideas

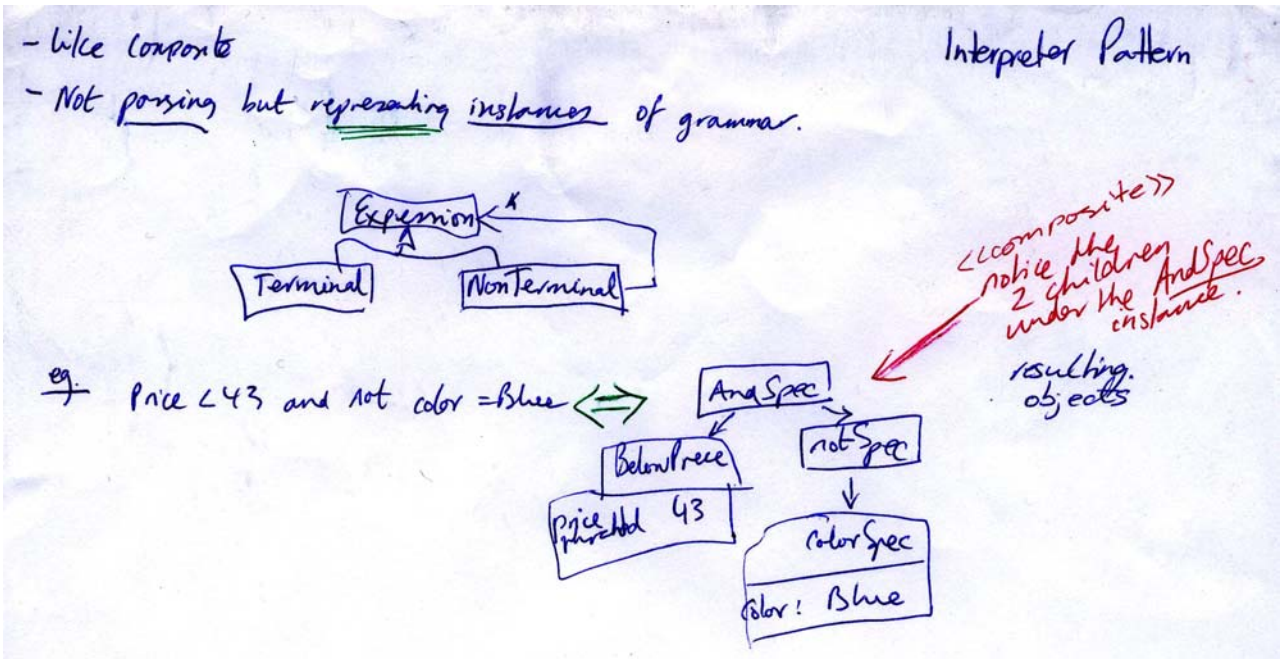


Figure 84 - Interpreter is typically about representing e.g. instances of a grammer. Its not about parsing. UML is similar to composite since a grammer is often a tree.

Code Ideas

www.dofactory.com's code demonstrates the Interpreter patterns, which using a defined grammer, provides the interpreter that processes parsed statements.

www.dofactory.com's real-world code demonstrates the Interpreter pattern which is used to convert a Roman numeral to a decimal.

Rules of thumb

Interpreter can use State to define parsing contexts. [GOF, p349]

The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF, p255]

Notes:

Facade

A single class that represents an entire subsystem

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.

Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Discussion

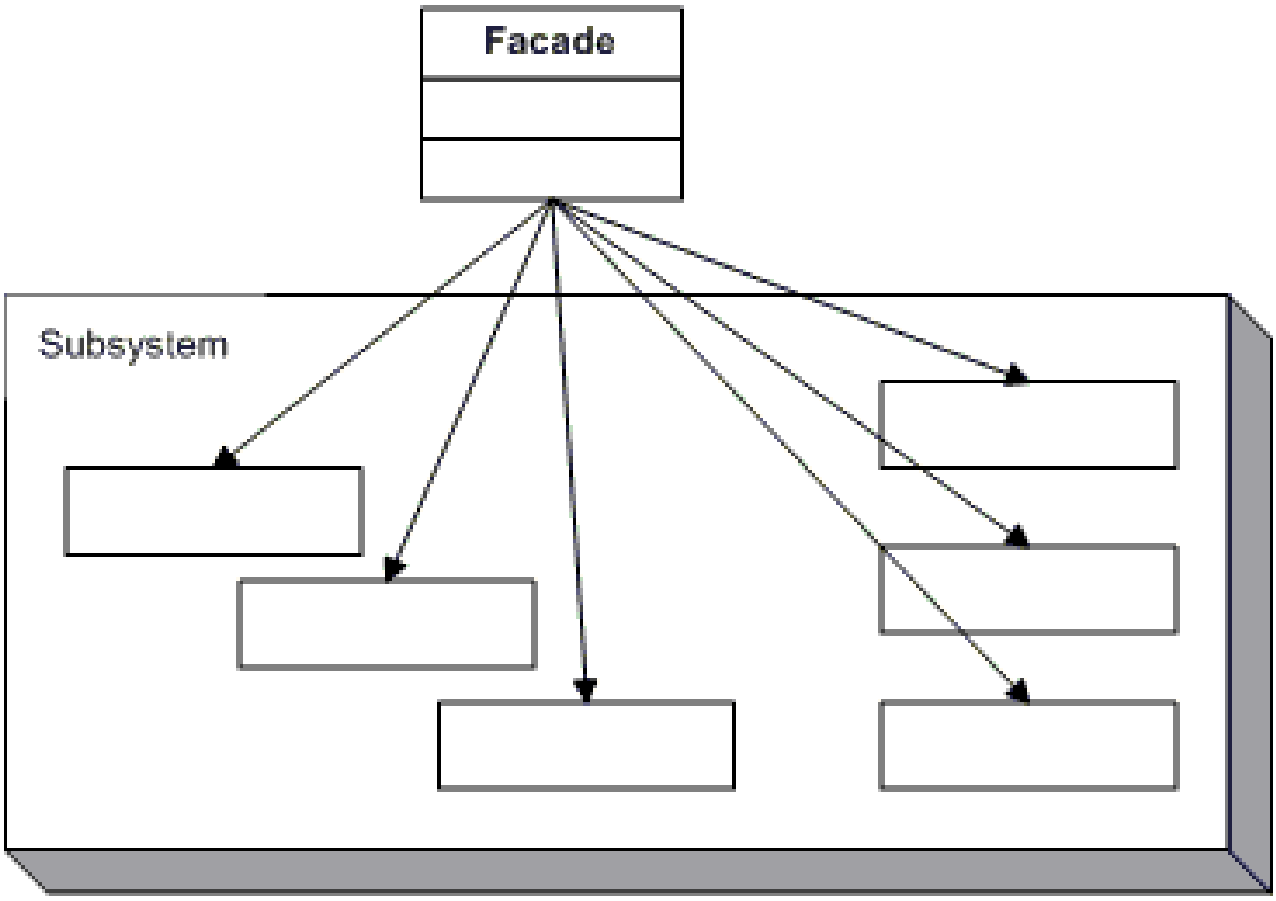
Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Participants

The classes and/or objects participating in the Façade pattern are:

- Facade (MortgageApplication)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- Subsystem classes (Bank, Credit, Loan)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.


Facade
Subsystem

Ideas

- The system shall provide a simple interface to a complex subsystem.
- The system shall provide alternative novice, intermediate, and "power-user" interfaces.
- The system shall decouple subsystems from each other.
- The system shall support "layering" of subsystems.

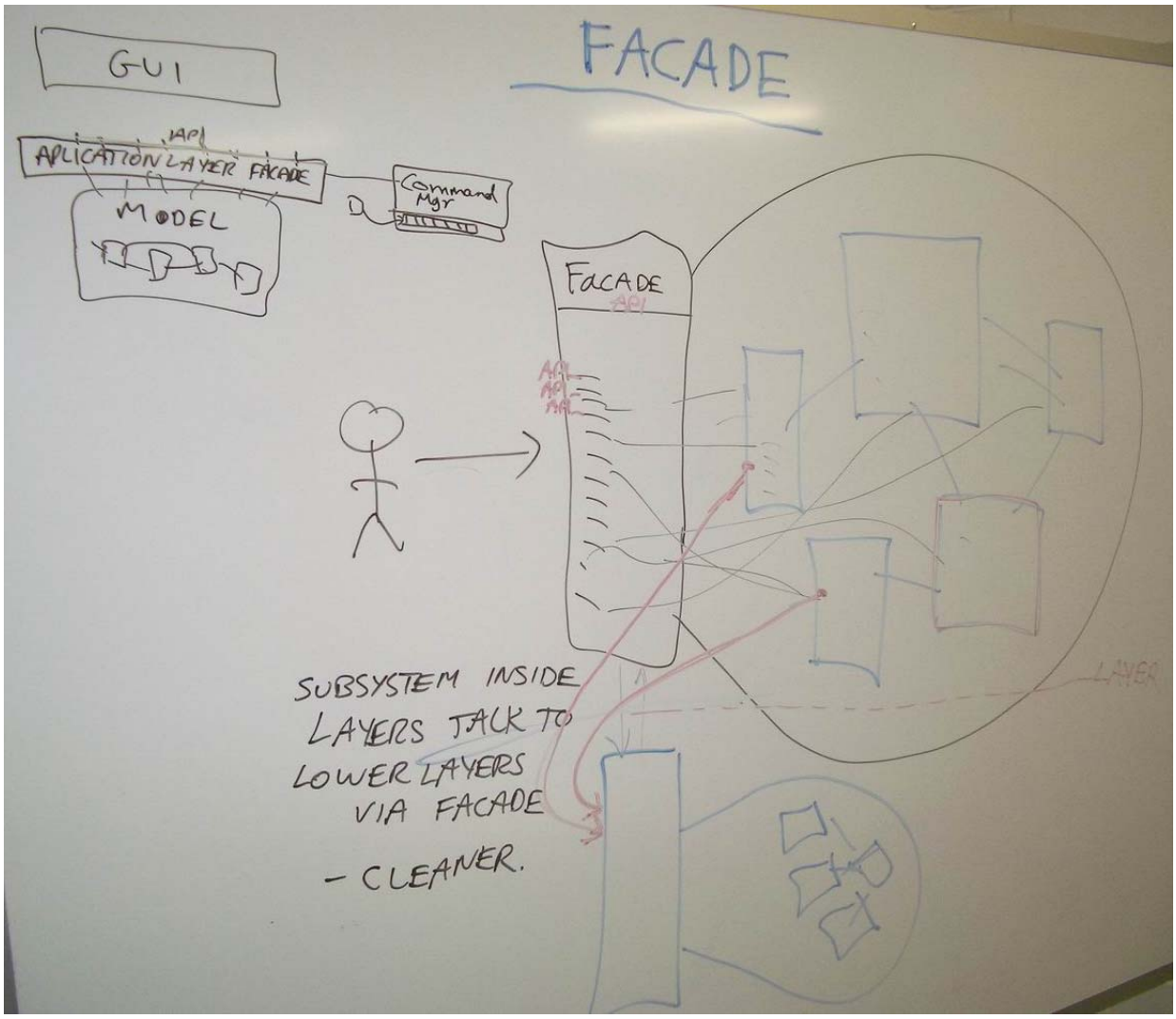


Figure 85 - Basic idea of façade

Andy's Tips

- Simplify the interface to a complex system. Shield the client code.
- Façade to façade to façade – layers are possible. See GOF p186
- Vary subsystem without affecting clients. Weak coupling.
- Client code can still bypass the façade and use the subsystem directly if it wants to.
- Façade often a Singleton.
- Similar to Mediator, different intent. GOF 193.

- Negative: Facades can end up being very large API's
- The subsystems do not know about the façade!

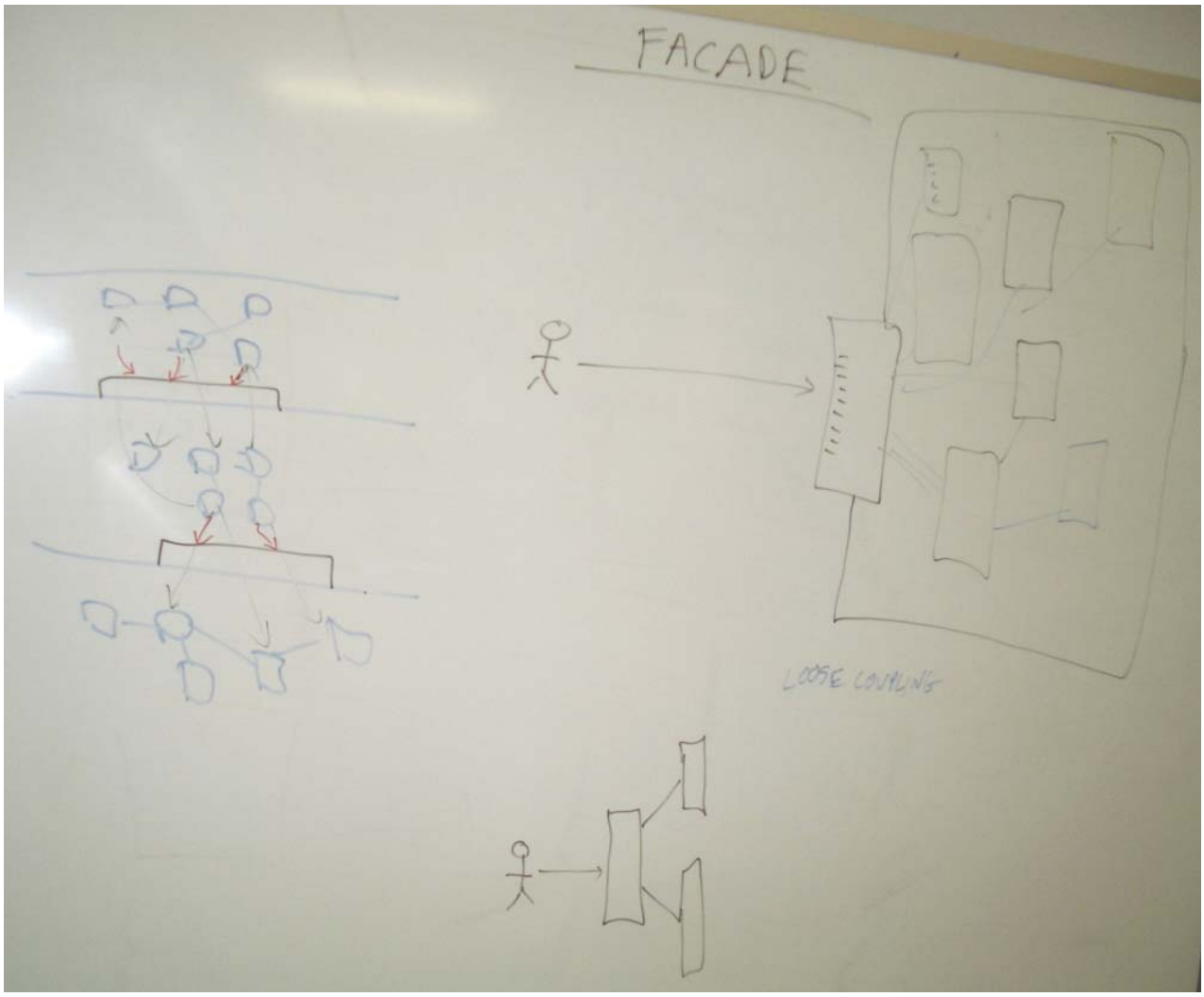


Figure 86 - Facade and the layering of subsystems

Non-software example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Code Ideas

www.dofactory.com's structural code demonstrates the Facade pattern which provides a simplified and uniform interface to a large subsystem of classes.

www.dofactory.com's real-world code demonstrates the Facade pattern as a MortgageApplication object which provides a simplified interface to a large subsystem of classes measuring the creditworthyness of an applicant.

Rules of thumb

Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF. p193]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

Notes:

Null Object

Taken from http://occs.cs.oberlin.edu/~jwalker/nullObjPattern/

Intent

Provide an object as a surrogate for the lack of an object of a given type. The Null Object provides intelligent do nothing behavior, hiding the details from its collaborators.

Also Known as

Stub, Active Nothing

Motivation

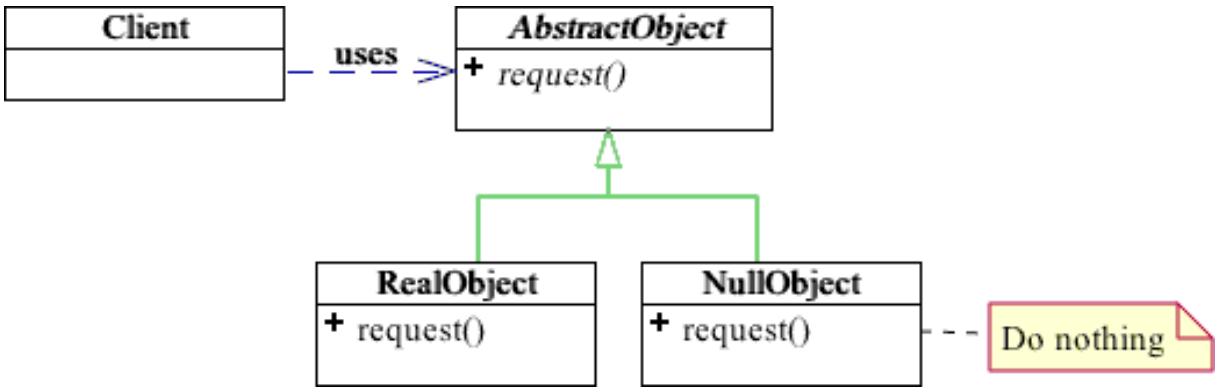
Sometimes a class that requires a collaborator does not need the collaborator to do anything. However, the class wishes to treat a collaborator that does nothing the same way it treats one that actually provides behavior.

Consider for example a simple screen saver which displays balls that move about the screen and have special color effects. This is easily achieved by creating a Ball class to represent the balls and using a Strategy pattern [GHJV95, page 315] to control the ball's motion and another Strategy pattern to control the ball's color. It would then be trivial to write strategies for many different types of motion and color effects and create balls with any combination of those. However, to start with you want to create the simplest strategies possible to make sure everything is working. And these strategies could also be useful later since you want as strategies as possible strategies.

Now, the simplest strategy would be no strategy. That is do nothing, don't move and don't change color. However, the Strategy pattern requires the ball to have objects which implement the strategy interfaces. This is where the Null Object pattern becomes useful. Simply implement a NullMovementStrategy which doesn't move the ball and a NullColorStrategy which doesn't change the ball's color. Both of these can probably be implemented with essentially no code. All the methods in these classes do "nothing". They are perfect examples of the Null Object Pattern.

The key to the Null Object pattern is an abstract class that defines the interface for all objects of this type. The Null Object is implemented as a subclass of this abstract class. Because it conforms to the abstract class' interface, it can be used any place this type of object is needed. As compared to using a special "null" value which doesn't actually implement the abstract interface and which must constantly be checked for with special code in any object which uses the abstract interface.

It is sometimes thought that Null Objects are over simple and "stupid" but in truth a Null Object always knows exactly what needs to be done without interacting with any other objects. So in truth it is very "smart."


Client
uses
AbstractObject
+ request()
RealObject
+ request()
NullObject
+ request()
Do nothing

Applicability

Use the Null Object pattern when:

- an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists.
- some collaborator instances should do nothing.
- you want to abstract the handling of null away from the client.

Participants

- Client
  - requires a collaborator.
- AbstractObject
  - declares the interface for Client's collaborator.
  - implements default behavior for the interface common to all classes, as appropriate.
- RealObject
  - defines a concrete subclass of AbstractObject whose instances
  - provide useful behavior that Client expects.
- NullObject
  - provides an interface identical to AbstractObject's so that a null object can be substituted for a real object.
  - implements its interface to do nothing. What exactly it means to do nothing depends on what sort of behavior Client is expecting.
  - when there is more than one way to do nothing, more than one NullObject class may be required.

Collaborations

Clients use the AbstractObject class interface to interact with their collaborators. If the receiver is a RealObject, then the request is handled to provide real behavior. If the receiver is a NullObject, the request is handled by doing nothing or at least providing a null result.

Consequences

The Null Object pattern:

- defines class hierarchies consisting of real objects and null objects. Null objects can be used in place of real objects when the object is expected to do nothing. Whenever client code expects a real object, it can also take a null object.
- makes client code simple. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write testing code which handles the null collaborator specially.
- encapsulates the do nothing code into the null object. The do nothing code is easy to find. Its variation with the AbstractObject and RealObject classes is readily apparent. It can be efficiently

coded to do nothing. It does not require variables that contain null values because those values can be hard-coded as constants or the do nothing code can avoid using those values altogether.

- makes the do nothing code in the null object easy to reuse. Multiple clients which all need their collaborators to do nothing will all do nothing the same way. If the do nothing behavior needs to be modified, the code can be changed in one place. Thereafter, all clients will continue to use the same do nothing behavior, which is now the modified do nothing behavior.
- makes the do nothing behavior difficult to distribute or mix into the real behavior of several collaborating objects. The same do nothing behavior cannot easily be added to several classes unless those classes all delegate the behavior to a class which can be a null object class.
- can necessitate creating a new NullObject class for every new AbstractObject class.
- can be difficult to implement if various clients do not agree on how the null object should do nothing as when your AbstractObject interface is not well defined.
- always acts as a do nothing object. The Null Object does not transform into a Real Object.

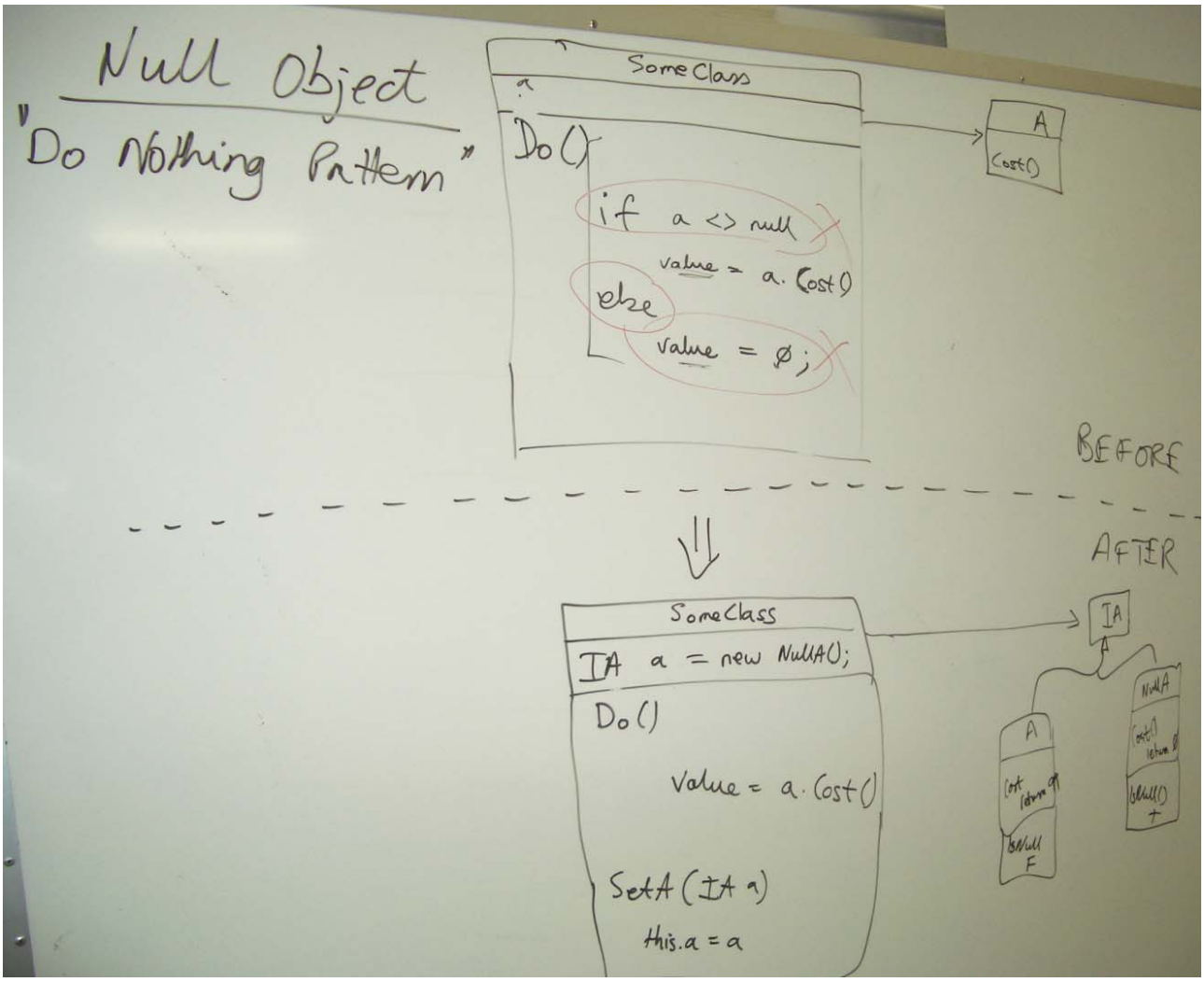


Figure 87 - Before and After - Null Object pattern gets rif of "if" statements

Implementation

There are several issues to consider when implementing the Null Object pattern:

1. Null Object as Singleton. The Null Object class is often implemented as a Singleton [GHJV95, page 127]. Since a null object usually does not have any state, its state can't change, so multiple instances are identical. Rather than use multiple identical instances, the system can just use a single instance repeatedly.

2. Clients don't agree on null behavior. If some clients expect the null object to do nothing one way and some another, multiple NullObject classes will be required. If the do nothing behavior must be customized at run time, the NullObject class will require pluggable variables so that the client can specify how the null object should do nothing (see the discussion of pluggable adaptors in the Adapter pattern [GHJV95, page 142]). This may generally be a symptom of the AbstractObject not having a well defined (semantic) interface.

3. Transformation to Real Object. A Null Object does not transform to become a Real Object. If the object may decide to stop providing do nothing behavior and start providing real behavior, it is not a null object. It may be a real object with a do nothing mode, such as a controller which can switch in and out of read-only mode. If it is a single object which must mutate from a do nothing object to a real one, it should be implemented with the State pattern [GHJV95, page 305] or perhaps the Proxy pattern [GHJV95, page 207]. In this case a Null State may be used or the proxy may hold a Null Object.

4. Null Object is not Proxy. The use of a null object can be similar to that of a Proxy [GHJV95, page 207], but the two patterns have different purposes. A proxy provides a level of indirection when accessing a real subject, thus controlling access to the subject. A null collaborator does not hide a real object and control access to it, it replaces the real object. A proxy may eventually mutate to start acting like a real subject. A null object will not mutate to start providing real behavior, it will always provide do nothing behavior.

5. Null Object as special Strategy. A Null Object can be a special case of the Strategy pattern [GHJV95, page 315]. Strategy specifies several ConcreteStrategy classes as different approaches for accomplishing a task. If one of those approaches is to consistently do nothing, that ConcreteStrategy is a NullObject. For example, a Controller is a View's Strategy for handling input, and NoController is the Strategy that ignores all input.

6. Null Object as special State. A Null Object can be a special case of the State pattern [GHJV95, page 305]. Normally, each ConcreteState has some do nothing methods because they're not appropriate for that state. In fact, a given method is often implemented to do something useful in most states but to do nothing in at least one state. If a particular ConcreteState implements most of its methods to do nothing or at least give null results, it becomes a do nothing state and as such is a null state. [Woolf96]

7. Null Object as Visitor host. A Null Object can be used to allow a Visitor [GHJV95, page 331] to safely visit a hierarchy and handle the null situation.

8. The Null Object class is not a mixin. Null Object is a concrete collaborator class that acts as the collaborator for a client which needs one. The null behavior is not designed to be mixed into an object that needs some do nothing behavior. It is designed for a class which delegates to a collaborator all of the behavior that may or may not be do nothing behavior. [Woolf96]

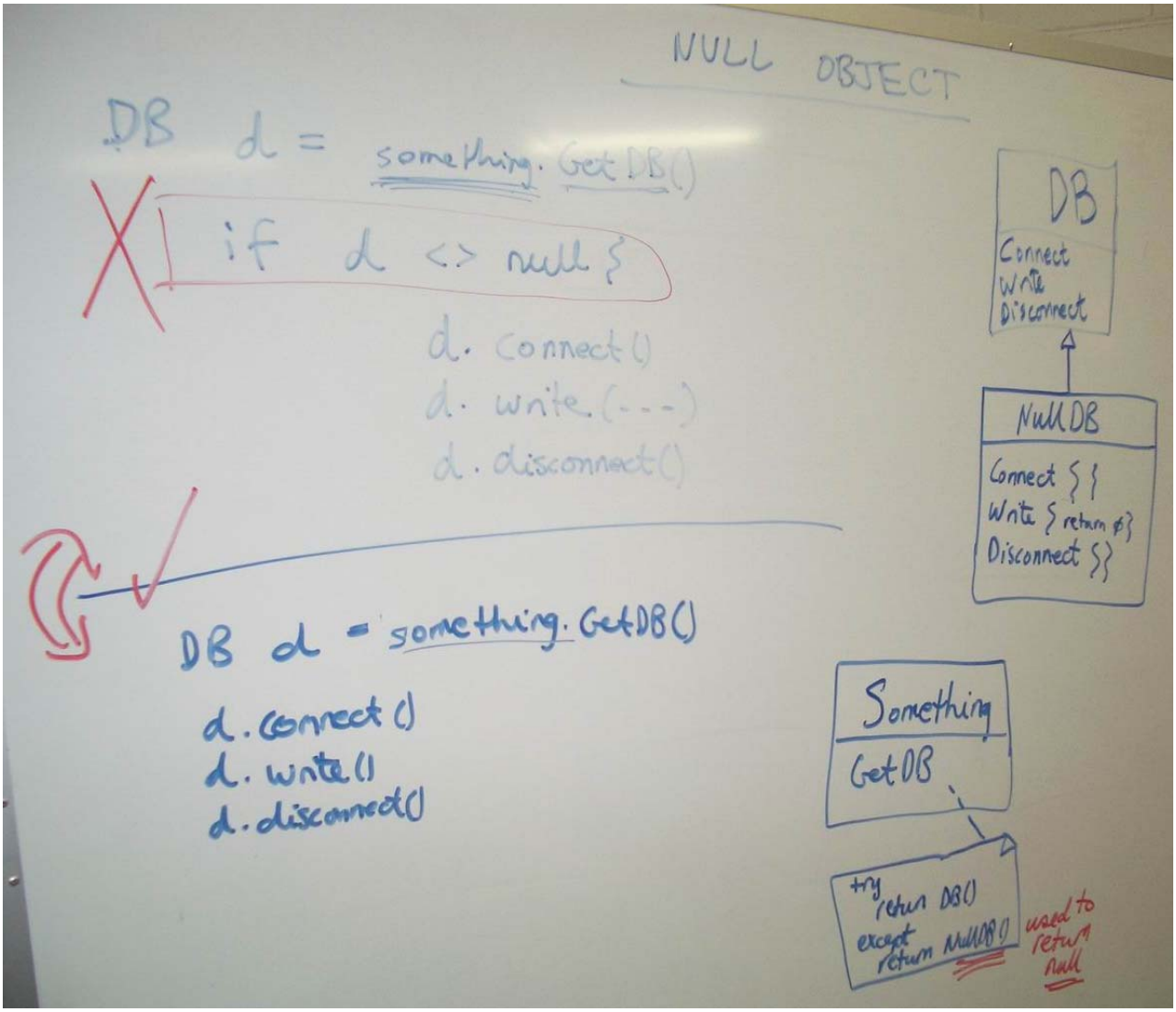
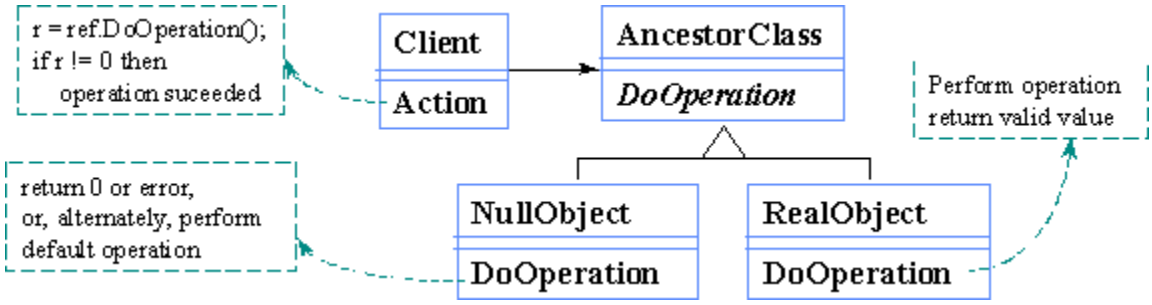


Figure 88 - Using Null Object pattern for Mocking Databases

Andy Tips:

- A Null Object provides a surrogate for another object that shares the same interface, but does nothing.
- Thus the Null Object encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators.



Code:

Without Null Object

```
from time import asctime, localtime

class RealLogging:
    def Log(self, msg):
        print 'Logged at', asctime(localtime()), msg

# Proxy / wrapper around either null or real logger.

class Logger:
    def __init__(self):
        self.logger = RealLogging()
    def Log(self, msg):
        if self.logger:
            self.logger.Log(msg)
    def On(self):
        self.logger = RealLogging()
    def Off(self):
        self.logger = None
Logger = Logger()

# Usage:

class API:
    def doA(self):
        if Logger.logger:
            Logger.Log('Am calling A')
        print 'A done.'
    def doB(self):
        if Logger.logger:
            Logger.Log('Am calling B')
        print 'B done.'

o = API()
o.doA()
o.doB()
```

```
Logger.Off()
o.doA()
o.doB()
```

With Null Object

```
# Null Object Pattern

class AbstractLogging:
    def Log(self, msg): pass

from time import asctime, localtime

class RealLogging(AbstractObject):
    def Log(self, msg):
        print 'Logged at', asctime(localtime()), msg

class NullLogging(AbstractObject):
    def Log(self, msg):
        return

# Proxy / wrapper around either null or real logger.

class Logger:
    def __init__(self):
        self.On()
    def Log(self, msg):
        self.logger.Log(msg)
    def On(self):
        self.logger = RealLogging()
    def Off(self):
        self.logger = NullLogging()
Logger = Logger()

# Usage:

class API:
    def doA(self):
        Logger.Log('Am calling A')
        print 'A done.'
    def doB(self):
        Logger.Log('Am calling B')
        print 'B done.'

o = API()
o.doA()
```

```
o.doB()

Logger.Off()
o.doA()
o.doB()
```

Notice – no more “if” statements in the client code (API class).

Notes:

Blending Patterns

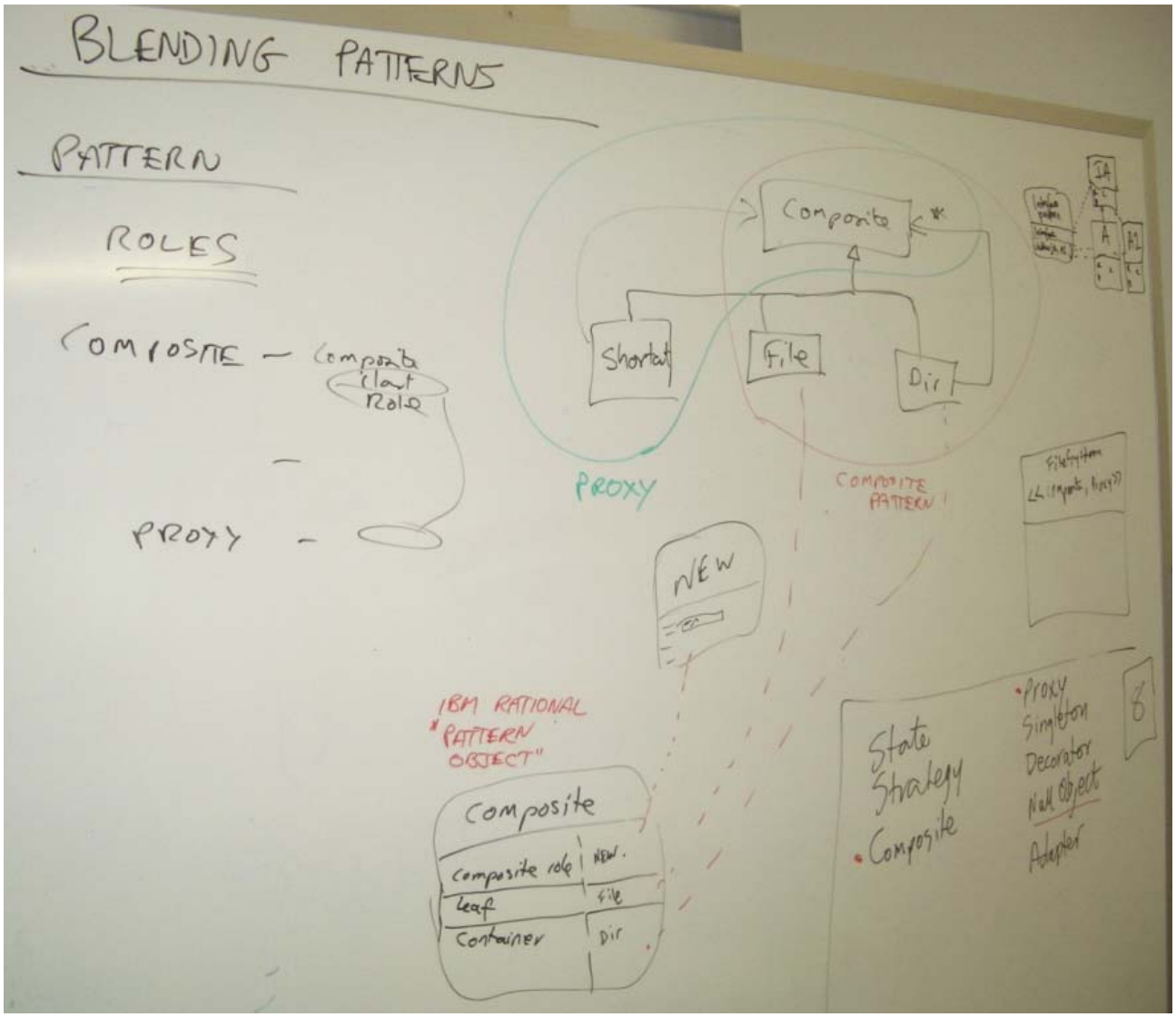


Figure 89 - Combining Composite with Proxy - creating the "shorcut" in our filesystem example

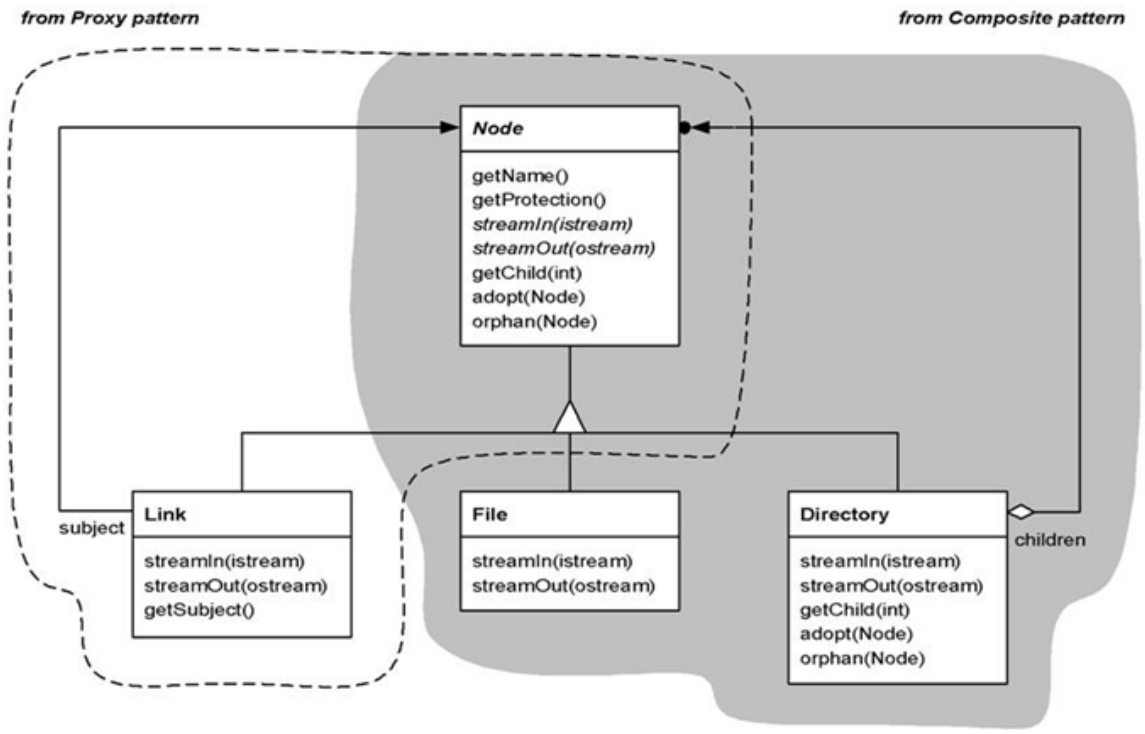


Figure 2: Venn diagram-style pattern annotations

Figure 90 - From "Pattern Hatching" by Vlissides

Blending 7 patterns

Design Patterns Implementation in a Storage Explorer Application

By Breman Sinaga.

From
http://www.codeproject.com/cs/design/sinagastorageexplorer.asp?df=100&forumid=39635&exp=0&select=1283126#xx1283126xx

Introduction

A lot of information about GoF design patterns has been published, however most of the examples use separate illustration for each pattern, and the sample code is not an application with a real function. The Storage Explorer is a small application that is designed to use several design patterns together. While the main purpose of this submission is to share the code than to teach the reader about design patterns, the discussion about the application design itself is explained using design patterns.

This article discusses the structure of the application, the design patterns that I use, how it works and the advantages of using them. In my opinion design pattern is worth learning. Developers who have design pattern skill can recognize the intent of a program and the purpose by identifying the patterns. The classes

with complex relations then can be understood better and faster than if the developers only understand basic OO design. The design patterns can be used as a thinking model for the designer and the code reader.

The application is used to explore file composition inside a computer storage. The design patterns that are used are: Strategy, Observer, Adapter, Template Method, Singleton and Wrapper Façade. The first five are known as GoF design patterns and the last one is a POSA pattern (POSA book volume-2).

Background

The idea of this application came when I desperately needed an application similar to Windows Explorer that can show me the file size composition inside my storage. I want to know, for example, under the Program Files folder, which folder uses what size of space, including the numbers in percentage. I want also to know how big my mp3 total size in drive D compare to my jpg files. The information I want to see should look like this:

```
C:\Program Files:
        Microsoft           445,123 KB        43%
        Adobe               234,744 KB        25%
        Symantec             98,906 KB        10%
        ...
```

And this

```
D:\
        *.mp3               602,456 KB        47%
        *.jpg               305,830 KB        30%
        *.doc               245,355 KB        20%
        ....
```

I need the information is presented in a chart to help me visualize the numbers as well.

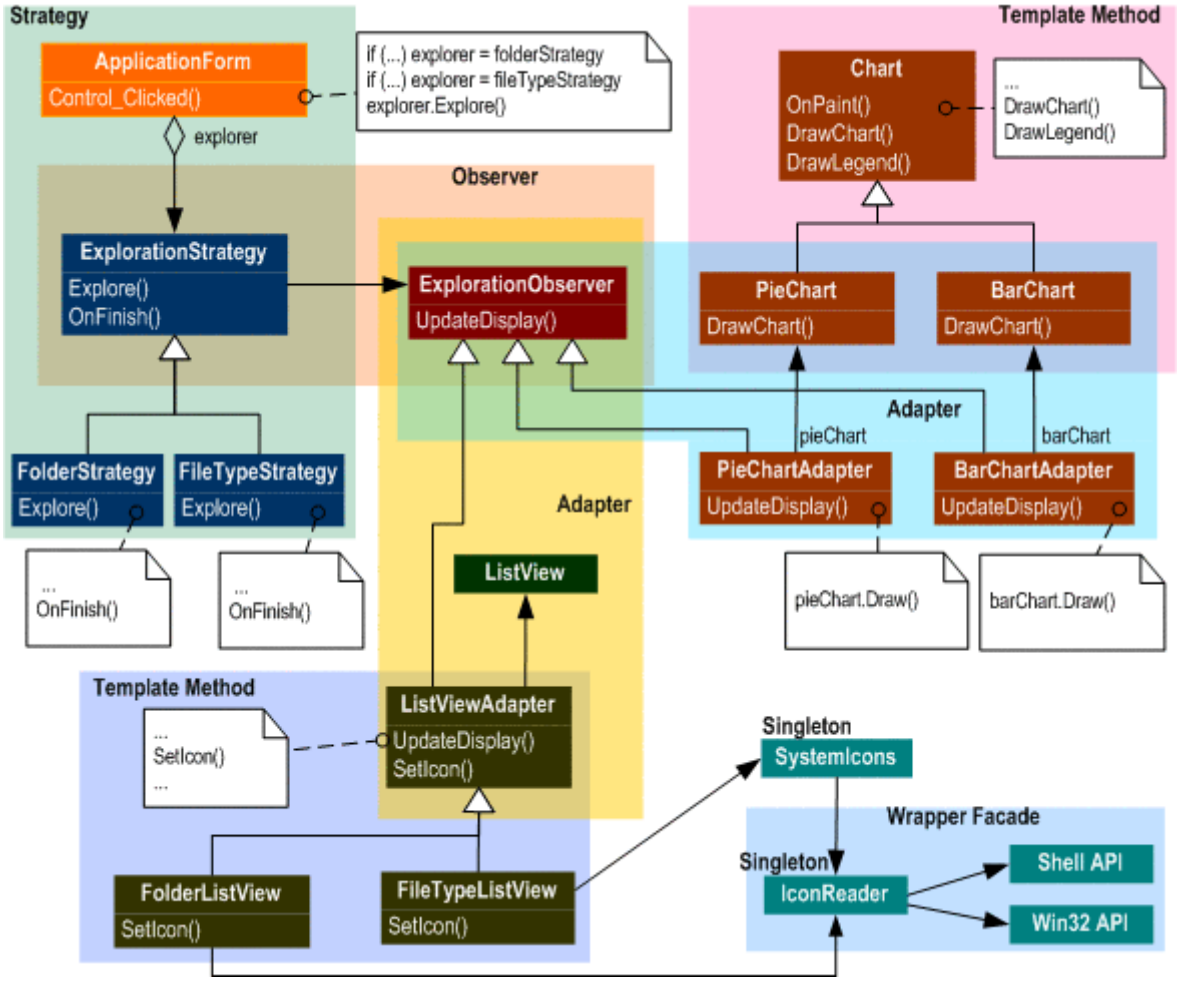
The Features

The application features are:

- A TreeView containing nodes that represents file system folders. The folders can be selected to see the information about file size structure within that path.
- It shows information about files size grouped by folders
- It shows information about files size grouped by file type
- The information is presented in listview, pie chart and bar chart on the right panel.

The Design Patterns inside the Application

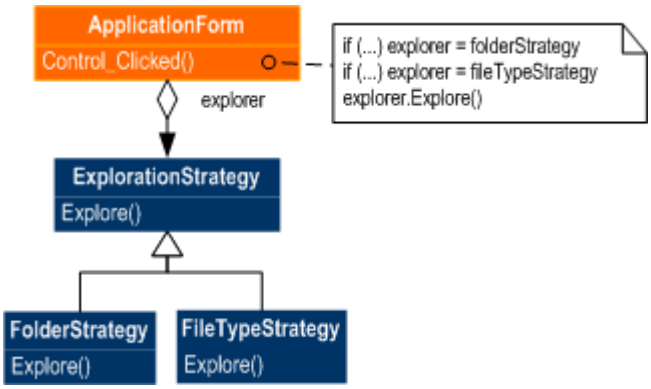
The design patterns used in this application is shown in the following figure.



The discussion for each design pattern is given below.

Strategy

Intent: “Define a family of algorithms, encapsulate each one, and make them interchangeable” (GoF).



The application explores the file system using two different algorithms, which is implemented in two strategy classes: `FolderStrategy` and `FileTypeStrategy` (see Figure above). `FolderStrategy` class implement an algorithm that explores files and sums the file size under different folders in a path. `FileTypeStrategy` class implements an algorithm that explores files and sums the file size that has the same type.

Now suppose we want to add a new algorithm. It can be done by creating a new subclass under `ExporationStrategy` class and putting the new algorithm code under the `Explore()` method. During the runtime create the instance of the new class and assign it dynamically to the explorer object. When we call `explore.Explore()` method the algorithm is run. That is the idea of the strategy pattern: encapsulate and make the algorithm interchangeable. The implementation looks like this:

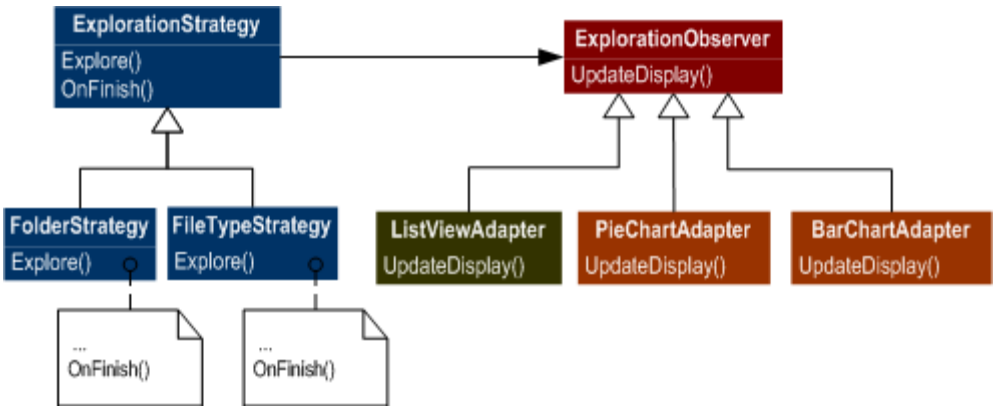
Collapse

```
public abstract class ExplorationStrategy
{
  public virtual void Explore (...){}
}

public class NewAlgorithm : ExplorationStrategy
{
  public override void Explore (...)
  {
    // the new algorithm
  }
}

public class StorageExplorerForm : System.Windows.Forms.Form
{
  // Somewhere in the initialization section
  ExplorationStrategy explorer;
  ExplorationStrategy folderStrategy = new FolderStrategy ();
  ExplorationStrategy fileTypeStrategy = new FileTypeStrategy ();
  ExplorationStrategy newStrategy = new NewAlgorithm ();

  // Somewhere else, an algorithm is selected
  switch (...)
  {
```

```
    case 0: explorer = folderStrategy; break;
    case 1: explorer = fileTypeStrategy; break;
    case 2: explorer = newStrategy; break;
  }

  // Execute the algorithm
  explorer.Explore (...);
}
```

Observer

Intent: “Define a one-to-many dependency between object so that when one object changes state, all its dependent are notified and updated automatically” (GoF).



In the application, this pattern creates a relationship between the subjects (concrete `ExplorationStrategy` object) and the observers (concrete `ExplorationObserver` object), so that when an exploration process finishes, the subject notifies the observer, and then the observer display the result (see Figure above). The concrete subject (`FolderStrategy` or `FileTypeStrategy`) notify the observers (`ListViewAdapter`, `PieChartAdapter` and `BarChartAdapter`) by calling `OnFinish()` method, which in turn generates an event. The event is sent to the observers and then handled by them. The relationship is defined at the abstract class and the concrete classes then inherits it.

The object and observer relationship is established by registering the observer `UpdateDisplay()` method to `ExplorationStrategy.Finish` event, as shown below:

```
public abstract class ExplorationObserver
{
  public void SubscribeToExplorationEvent (ExplorationStrategy obj)
  {
    obj.Finish += new ExplorationFinishEventHandler (UpdateDisplay);
  }
}
```

And during application initialization in the client (the application form), the concrete observer object call `SubscribeToExplorationEvent()` method and pass the instance of the concrete strategy as the parameter. Since then the subject-observer relationship has been established. It shows as folows:

```
// Initialization in the client:
// Create the concrete subject
```

```
ExplorationStrategy folderStrategy = new FolderStrategy ();
ExplorationStrategy fileTypeStrategy = new FileTypeStrategy ();

// Create the concrete observer
ExplorationObserver pieChart = new PieChartAdapter ();

// Subscribe the concrete observer object to the concrete strategy object
pieChart.SubscribeToExplorationEvent (folderStrategy);
pieChart.SubscribeToExplorationEvent (fileTypeStrategy);
```

Now let see the beauty of this pattern. Suppose we want to change the application to accommodate a new requirement. We want to save the exploration result into a file in addition to displaying it on the screen. For that purpose, inherits a new class from `ExplorationObserver` class and put the code that saves the result to a file inside `UpdateDisplay()` method. Create the instance of new class then call `SubscribeToExplorationEvent()` with the concrete `ExplorationStrategy` object as the parameter. Finish. When the application is run, information will be sent to the display and a file as well. The code is shown below.

```
public class NewConcreteObserver : ExplorationObserver
{
  public override void UpdateDisplay (object o, ExplorationFinishEventArgs e)
  {
    Hashtable result = e.ExplorationResult;

    // Write the result to a file
  }
}

// Somewhere, during the initialization in the client
...
ExplorationObserver fileSaver = new NewConcreteObserver ();
fileSaver.SubscribeToExplorationEvent (folderStrategy);
fileSaver.SubscribeToExplorationEvent (fileTypeStrategy);
```

Observer design pattern is actually always used in the event driven programming. The idea is often used without being realized. However knowing the conceptes is still important if we want to collaborate it with another pattern like Adapter, as explained in the next section.

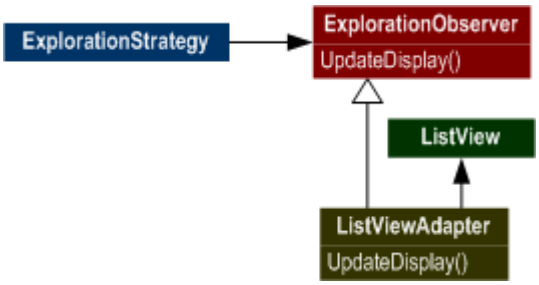
Adapter

Intent: “Convert the interface of a class into another interface client expects. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces” (Gof).

Now here is the case: I want the exploration result to be displayed in a .NET `ListView` class. However I need to modify the `ListView` capability so it can receive auto notification when an exploration finishes. It means I need to change `ListView` class to have `ExplorationObserver` class signatures and therefore I can override the `UpdateDisplay()` method, which is called when the `Finish` event comes. Unfortunately I cannot modify .NET ListView class. The solution is to create a new class, namely `ListViewAdapter`, that uses `ListView` capability while still has `ExplorationObserver` class signatures. Here adapter pattern plays the role. The `ListViewAdapter` class is an Adapter. In GoF terms: the `ListViewAdapter` class modify the interface of `ListView` class to have `ExplorationObserver` interface that `ExplorationStrategy` expects.

The adapter pattern has two forms: class adapter and object adapter. The class adapter is implemented by creating `ListViewAdapter` that inherits from `ListView` and `ExplorationObserver` class. Because C# does not support multiple inheritances then this is not possible. Actually this can be possible if `ExplorationObserver` is an interface instead of an abstract class. `ListViewAdapter` then could implement the interface and inherit from the`ListView` class. But in this case `ExplorationObserver` has some implementation inside the class, which makes being an interface is not an option.

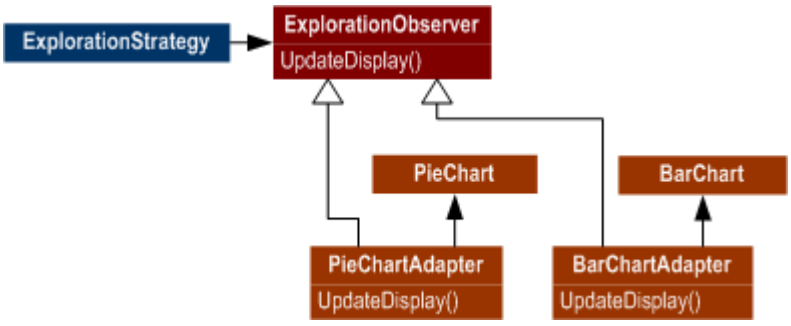
The second form, the object adapter, which is used in this application, uses object composition (see Figure below).



The `ListViewAdapter` class inherits the `ExplorationObserver` class. To have a `ListView` capability, the adapter creates a `ListView` object and use the object when necessary. The advantage of this approach is the `ListView` object and its members is hidden from the `ListViewAdapter` user. The code is shown below:

```
public class ListViewAdapter : ExplorationObserver
{
  protected ListView listView;

  public ListViewAdapter()
  {
    // Create a ListView object and initialze it
    listView = new ListView();
    ...
  }
}
```

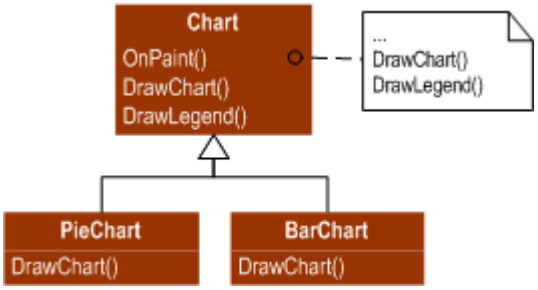
The same approach is applied to the `PieChartAdapater` and `BarChartAdapter` class. Again I assume that I have already had those chart classes and I do not want to modify them. The solution is again to create an adapter class that inherits from `ExplorationObserver` class and create the chart object inside the adapter (see Figure below).



Template Method

Intent: “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain step of an algorithm without changing the algorithm structure” (GoF).

The template method itself is a method in a class that calls other primitive method(s), so that when the subclass overrides the primitive method(s), the template method in the descendant class produces different results. Now let see the template method implementation in this application (see Figure below).

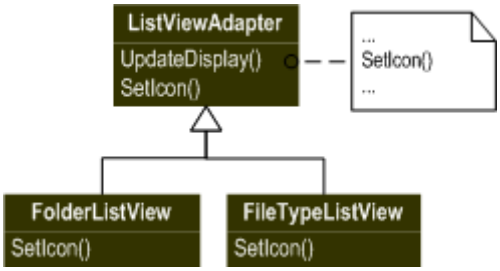


The template method inside the Chart is the `OnPaint()` method. Inside the `OnPaint()` there has been defined some steps to draw a chart. The steps include the process to initialize the Graphics object, draw the legend and draw the chart image. If we want to inherits this class to be some different types of chart like `PieChart` class and `BarChart` class , then inside the `OnPaint()` method convert the drawing chart operation to become a `DrawChart()` method call. In the `Chart` class itself the `DrawChart()` is a blank method. In the `PieChart` class, which inherits from the `Chart` class, the `DrawChart()` contains implementation to draw a pie images, and in the `BarChart`, the `DrawChart()` draws a bar images. We can do the same thing to the drawing the legend, the title, and every steps involved in the drawing chart process. The code is shown below:

```
public abstract class Chart : Control
{
  // OnPaint is a template method. It contains steps
  // necessary to draw a complete chart
  public OnPaint ()
  {
    ...
    DrawLegend (...);
    DrawChart (...);
  }

  protected abstract void DrawChart (...)
  protected virtual void DrawLegend (...) { // Draw legend }
}

public abstract class PieChart : Chart
{
  protected virtual void DrawChart (...)
  {
    // Draw the pie chart image
  }
}

public abstract class BarChart : Chart
{
```

```
  protected virtual void DrawChart (...)
  {
    // Draw the bar chart image
  }
}
```

I use the same approach with the `ListViewAdapter` class. It has two descendant class, `FolderListView` and `FileTypeListView`. Both classes have very similar steps when displaying information, except a step to set the icon. The `FolderListView` set a single icon representing folder image while the `FileTypeListView` select the icon from the series of image in the `ImageList` . Because the difference, that step is deferred to the descendant class (see Figure below)



The key to use the Template Method is to define a skeleton of an algorithm as general as possible in the base class and defer the steps that could vary to the descendant class. Also in the base class, we need to create some default implementation of the deferred step in case the descendant classes do not want to override the steps. Once we have a good template and default implementation, then we can enjoy creating variety of descendant classes with less effort.

Singleton

Intent: “Ensure a class only has one instance, and provide a global point of access to it” [GoF].

In this application there are two classes in which each of them needs only one instance to run: `FileIcons` class and `IconReader` class. The`FileIcon` class is used to retrieve and buffer the shell icons from the Windows Shell. This class represents the collection of the icon in the system. From the definition we know that it needs only one instance of the class running in the application. The `IconReader` class is used to retrieve the icon from the system without buffering. It calls the Shell API function. We need only one instance of this class as well and therefore it is implemented as Singleton.

MSDN provides an example on how to implement the singleton in .NET using C#. In this application the .NET singleton is implemented as follows:

```
sealed class FileIcons
{
  ...

  // By making it private, the class cannot be explicitly created
  private FileIcons () {}

  // This is the statement that ensures only one instance exists
  public static readonly FileIcons Instance = new FileIcons();
```

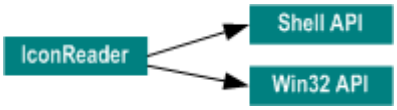
```
  public int GetIconIndex (...)
  {
    ...
  }
}
```

To use the `FileIcons` class we just use `FileIcons.Instance` property without needing to create the class, like this:

```
int iconIndex = FileIcons.Instance.GetIconIndex (...);
```

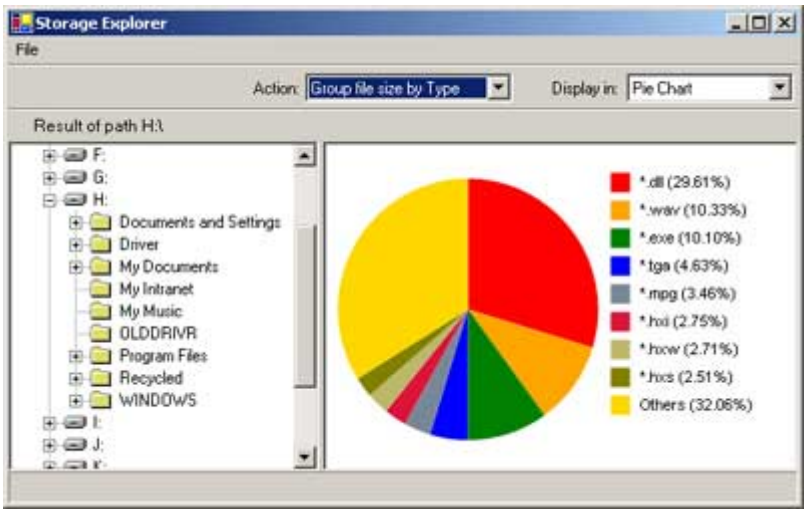
Wrapper Facade

Intent: " Encapsulate the functions and data provided by existing non-object oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces" (POSA volume 2).



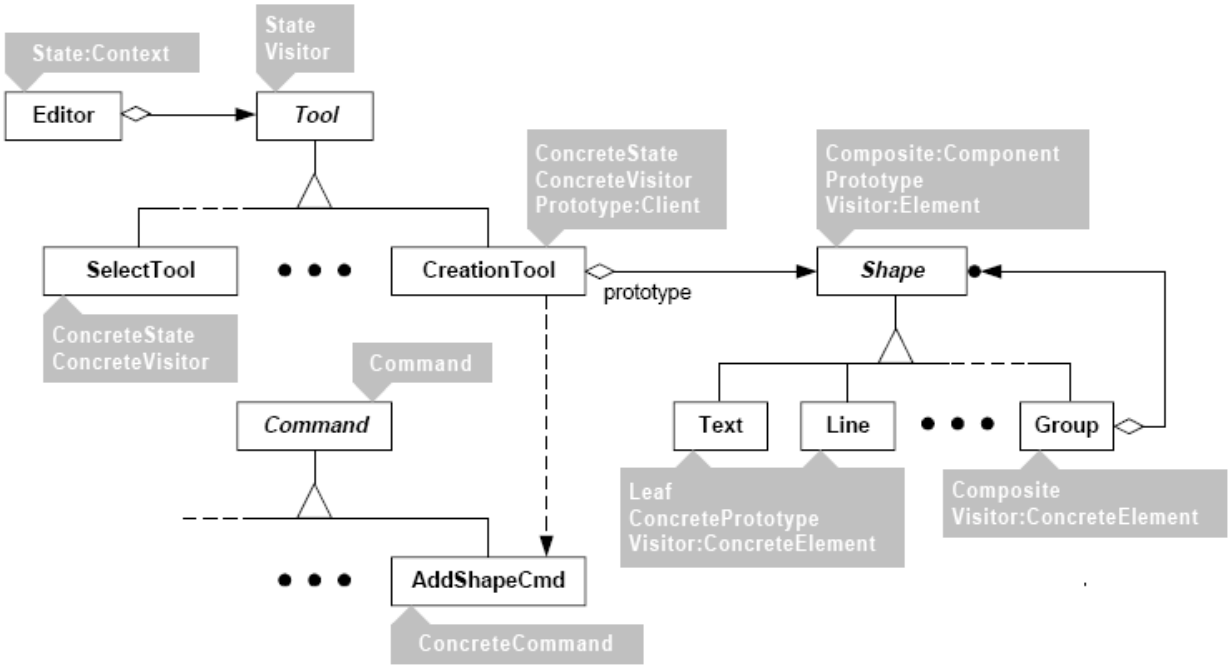
To access the shell icons from the windows operating system we must call the Shell API and Win API functions. The function call involves strict usage of data structure and complex parameter. To encapsulate that complexity I created an `IconReader` class. POSA says this is different from Facade pattern since the Facade encapsulates relationship complexity among classes, while the Wrapper Facade encapsulates non OO functions. Also this pattern is more specific in its purpose. For example, the Shell API function that is used to retrieve icon from the Windows Shell is the `SHGetFileInfo()` function. This function can be used to do a lot of job, not only for icon retrieval. Because the `IconReader` class is specific for Icon related jobs then I create only methods specific for it, they are `GetSmallFileTypeIcon()` method (to retrieve icon representing file extension) and `GetClosedFolderIcon()` (to retrieve icon representing a closed folder). The icon retrieval also needs to call Win32 API `DestroyIcon()` function. The idea behind this pattern is: don't look at to the complexity behind the process, as long as it is for delivering robust and cohesive functionalities, then use all the resources needed, and present only the cohesive interface to the class user.

Result:


Storage Explorer
File
Action: Group file size by Type
Display in: Pie Chart
Result of path H:\
F:
G:
H:
Documents and Settings
Driver
My Documents
My Intranet
My Music
OLDDRIVR
Program Files
Recycled
WINDOWS
I:
J:
*.dll (29.61%)
*.wav (10.33%)
*.exe (10.10%)
*.tga (4.63%)
*.mpg (3.46%)
*.hoi (2.75%)
*.hcw (2.71%)
*.hcs (2.51%)
Others (32.06%)

Blending 5 patterns - Tooled Composite – Architectural Pattern

This is a pattern my company Austhink Software implemented when building a highly interactive diagramming tool. I presented this pattern to the Melbourne Patterns Group in 2006. –Andy



The Problem

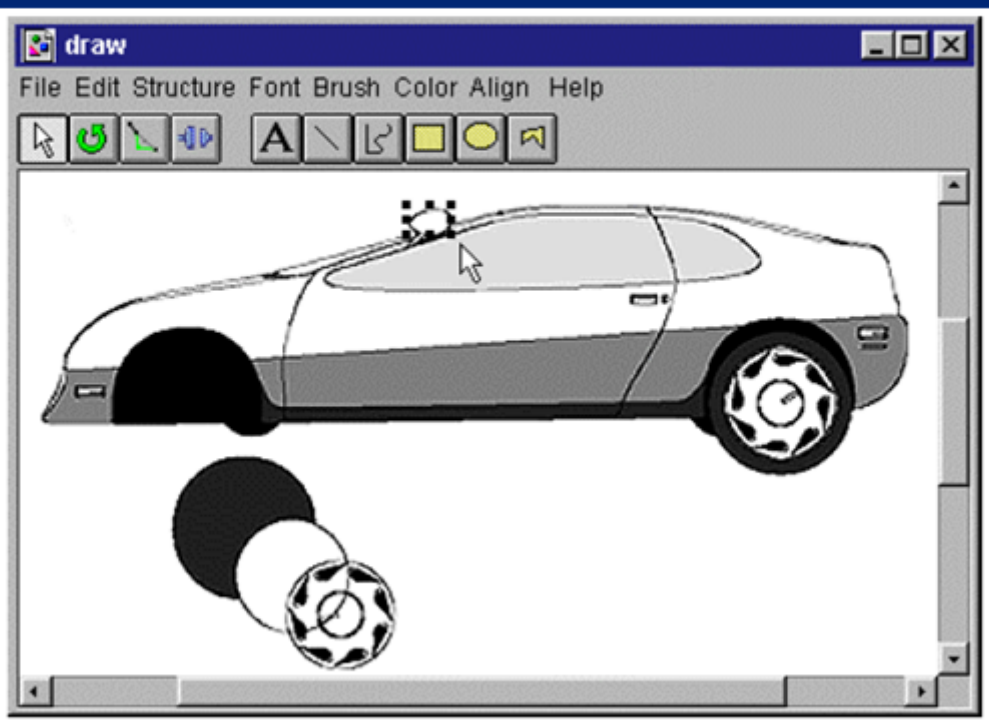
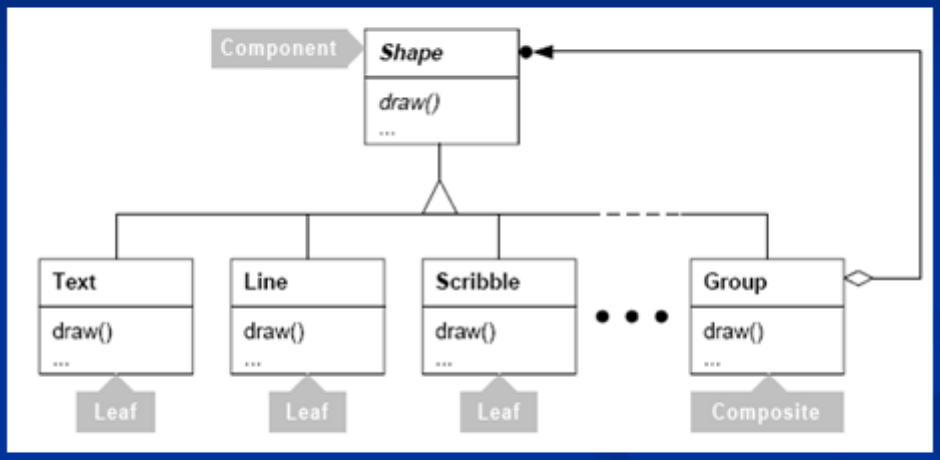


Figure 1: A simple drawing editor

"direct manipulation" metaphor.

- *How do you represent shapes?*
- *How do you represent tools?*
- *How do tools and shapes interact?*
- *How do you enhance the editor with new shapes and tools?*

Representing shapes



COMPOSITE – design pattern

Representing "Tools"

- *usually a palette of tools*
- *editor's behavior changes with the current tool*
- *E.g. when **drawing** tool is active we **create** shapes;*
- *E.g. when the **selection** tool is active we **select** shapes*

Representing "Tools"

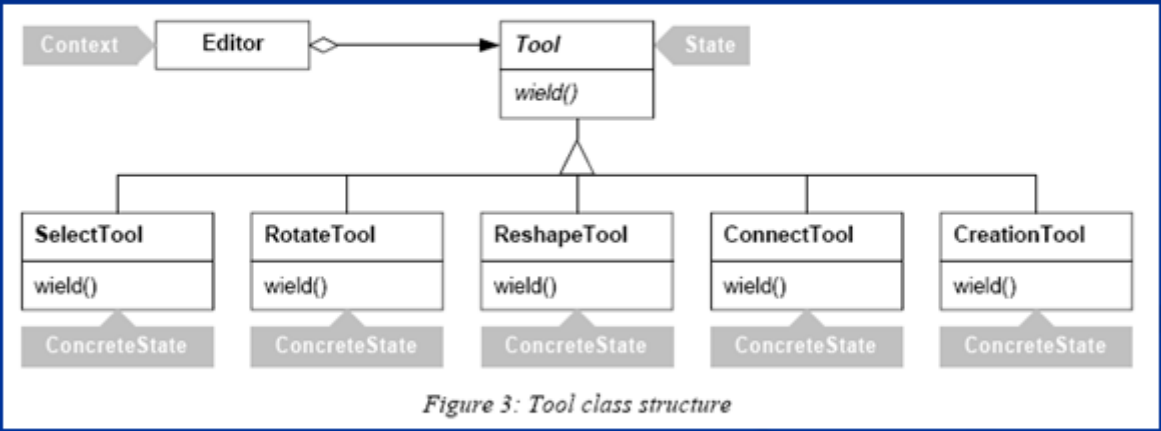


Figure 3: Tool class structure

STATE – design pattern

How tools and shapes interact

- How a tool interacts with each shape is usually different
- *m* x *n* possible interactions between *m* tools and *n* shapes.
- How do we manage and organise this complexity?

Visitor


Canvas
Tool
Visitor
Delete
Create
Select/ Hover
Resize
Edit
Move
onclick
onDblClick
Rescale
HorzResize

Visitor

- Shapes don't have tool knowledge – tools do all the work. Shapes just implement **AcceptVisitor(v)** and then do a v.VisitShape() or v.VisitEdge() or whatever they themselves are.
- Each tool has a
  VisitShape
  VisitEdge
  VisitShapeRhEdgeZone
  method that gets triggered in this way.

Example Sequence continued

- User Left Clicks
- HoverTool.OnLeftClick sees that you are over a resize zone shape, so switches to the Resize tool
  - Zones (e.g. resize zone) within shapes are also 'shapes'
- Resize tool.OnMouseMove resizes the shape you are on. Repeatedly (as MouseMove events arrive).
- Resize tool.OnMouseUp switches back to the hover tool.

Example Sequence

- Start in Hover Tool
- All mouseMove events go to Hover tool
- As hover over shapes/edges you ask what is under me and change cursor. You "visit" the shape and change cursor accordingly

ToolHover

VisitShape()
  cursor = HAND

VisitEdge()
  cursor = ARROW

Tooled Composite pattern

Rewire events as you swap tools

Events

- Funnel all events through to the **current tool**.
- Each tool has custom handling for all the gui events e.g. mouseDown, mouseClick, mouseMove etc.

```
public void OnMouseDown(MouseEventArgs e)
{
    if (MouseDown != null)
    {
        MouseDown();
    }

    OnMouseAction(MouseEvent.OnMouseDown, e);
}

public void OnMouseMove(MouseEventArgs e)
{
    OnMouseAction(MouseEvent.OnMouseMove, e);
}

public void OnMouseUp(MouseEventArgs e)
{
    OnMouseAction(MouseEvent.OnMouseUp, e);
}

public void OnMouseWheel(MouseEventArgs e)
{
    OnMouseAction(MouseEvent.OnMouseWheel, e);
}

/// <summary>
/// Corrects for graphics transformations for scrolling and
/// </summary>
private void OnMouseAction(MouseEvent me, MouseEventArgs e)
{
    Mouse.Store(me, e);

    CurrentTool.OnMouseAction();
    Refresh();
}
```

Classic STATE pattern, passing through method invocations to the current state object

Blending Strategy and Template Method

java.net > All Articles > http://today.java.net/pub/a/today/2004/10/29/patterns.html

Principles, Patterns, and Practices: The Strategy, Template Method, and Bridge Patterns

by Robert C. Martin
10/29/2004

The Strategy, Template Method, and Bridge Patterns

One of the great benefits of object-oriented programming is polymorphism; i.e., the ability to send a message to an object without knowing the true type of the object. Perhaps no pattern illustrates this better than the Strategy pattern.

To illustrate the Strategy pattern let's assume that we are working on a debug logger. Debug loggers are often very useful devices. Programmers can send messages to these loggers at strategic places within the code. If the system misbehaves in some way, the debug log can provide clues about what the system was doing internally at the time of the failure.

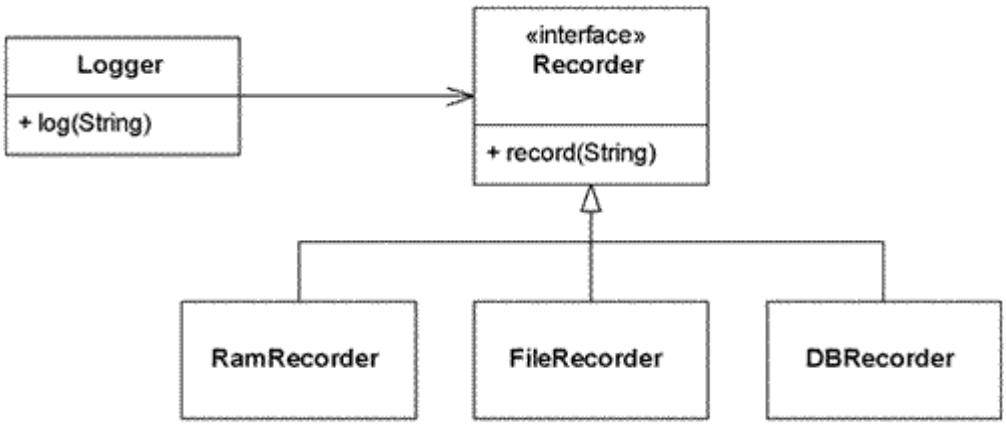
In order to be effective, loggers need to be simple for programmers to use. Programmers aren't going to frequently use something that is inconvenient. You should be able to emit a log message with something no more complicated than:

```
logger.log("My Message");
```

On the other hand, what we want to see in the log is quite a bit more complex. At very least we are going to want to see the time and date of the message. We'll also probably want to see the thread ID. Indeed, there may be a whole laundry list of system states that we want to log along with the message. So the logger needs to gather all of this peripheral information together, format it into a log message, and then add it to the growing list of logged messages.

Where should the logged messages be stored? Sometimes we might like them stored in a text file. Sometimes we might want to see them added to a database table. Sometimes we might like them accumulated in RAM. The choices seem endless. However, the final destination of the logged messages has nothing to do with the format of the messages themselves.

We have two algorithms: one formats the logged message, and the other records the logged message. These two algorithms are both in the flow of logging a message, but both can vary independently of each other. The formatter does not care where the message is recorded, and the recorder does not care about the format of the message. Whenever we have two connected but independent algorithms, we can use the **Strategy** pattern to connect them. Consider the following structure:



Here the user calls the `log` method of the `Logger` class. The `log` method **formats** the message and then **calls** the `record` method of the `Recorder` interface. There are many possible implementations of the `Recorder` interface. Each does the recording in a different way.

The structure of the `Logger` and `Recorder` is exemplified by the following unit test, which uses the Adapter pattern:

```
public class LoggerTest extends TestCase {
  private String recordedMessage;
  protected String message;

  public void testLogger() throws Exception {
    Logger logger = new Logger(new Recorder() {
      public void record(String message) {
        recordedMessage = message;
      }
    });

    message = "myMessage";
    logger.log(message);
    checkFormat();
  }

  private void checkFormat() {
    String datePattern = "\\d{2}/\\d{2}/\\d{4} \\d{2}:\\d{2}:\\d{2}.\\d{3}";
    String messagePattern = datePattern + " " + message;
    if(!Pattern.matches(messagePattern, recordedMessage)) {
      fail(recordedMessage + " does not match pattern");
    }
  }
}
```

As you can see, the `Logger` is constructed with an instance of an object that implements the `Recorder` interface. [East Facing coding! - Andy] `Logger` does not care what that implementation does. It simply builds the string to be logged and then calls the record method. This is very powerful decoupling. It allows the formatting and recording algorithms to change independently of each other.

`Logger` is a simple class that simply formats the message and forwards it to the `Recorder`.

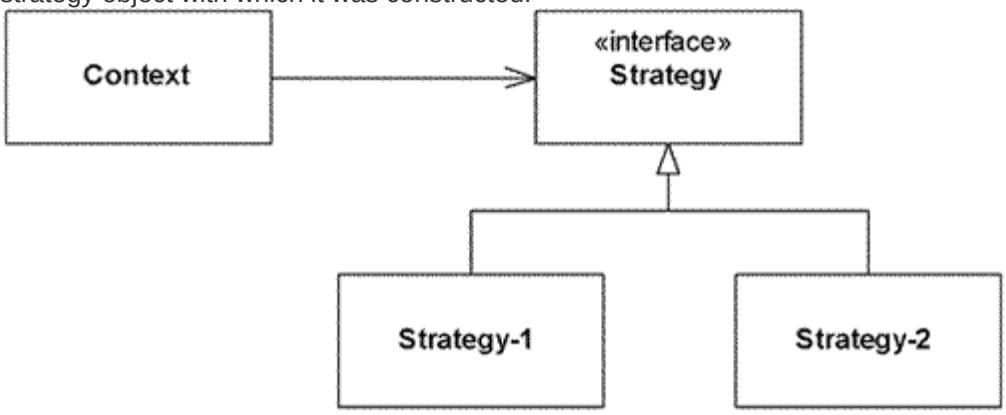
```
public class Logger {
  private Recorder recorder;

  public Logger(Recorder recorder) {
    this.recorder = recorder;
  }

  public void log(String message) {
    DateFormat format = new SimpleDateFormat("MM/dd/yyyy kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
```

```
    recorder.record(prefix + " " + message);
  }
}
```

And `Recorder` is an even simpler interface.

```
public interface Recorder {
  void record(String message);
}
```

The canonical form of the Strategy pattern is shown below. One algorithm (the *context*) is shielded from the other (the *strategy*) by an interface. The context is unaware of how the strategy is implemented, or of how many different implementations there are. The context typically holds a pointer or reference to the strategy object with which it was constructed.

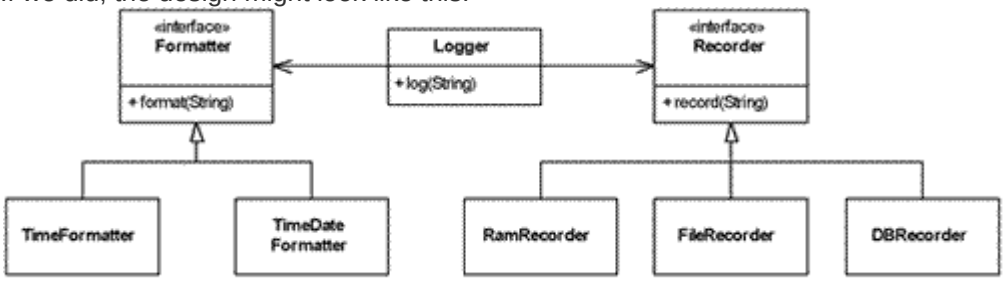


In our `Logger` example, the `Logger` is the **context**, the `Recorder` is the **strategy interface**, and the anonymous inner class within the unit test acts as one of the implemented **strategies**.

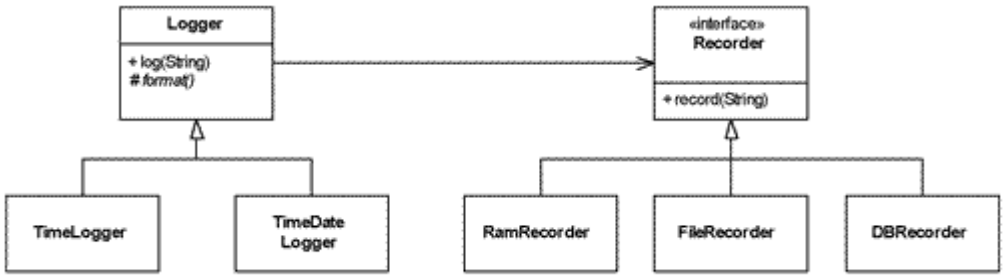
If you have been an object-oriented programmer for any length of time, you have seen this pattern many times. Indeed, it is so common that some folks shake their heads and wonder why it even has a name. It's rather like giving the name "DO NEXT STATEMENT" to the fact that execution proceeds statement by statement. However, there is a good reason to give this pattern a name. It turns out that there is another pattern that solves the same problem in a slightly different way; and the two names help us differentiate between them.

This second pattern is called *Template Method*, and we can see it by adding the next obvious layer of polymorphism to the `Logger` example. We already have one layer that allows us to change the way log messages are recorded. We could add another layer to allow us to change how log messages are formatted. Let's suppose, for instance, that we want to support two different formats. One prepends the time and date to the message as above; the other prepends only the time.

Clearly, this is just another problem in polymorphism, and we could use the Strategy pattern once again. If we did, the design might look like this:



Here we see two uses of the Strategy pattern. One provides polymorphic recording and the other provides polymorphic formatting. This is a common enough solution, but it is not the only solution. Indeed, we might have opted for a solution that looked more like this:



Notice the `format` method of `Logger`. It is protected (that's what the `#` means) and it is abstract (that's what the italics mean). The `log` method of `Logger` calls its own abstract `format` method, which deploys to one of the derived classes. The formatted string is then passed to the `record` method of the `Recorder`. Consider the unit test below. It shows tests for both the `TimeLogger` and the `TimeDateLogger`. Notice that each test method creates the appropriate `Logger` derivative and passes a `Recorder` instance into it.

```
import junit.framework.TestCase;

import java.util.regex.Pattern;

public class LoggerTest extends TestCase {
  private String recordedMessage;
  protected String message;
  private static final String timeDateFormat =
    "\\d{2}/\\d{2}/\\d{4} \\d{2}:\\d{2}:\\d{2}.\\d{3}";
  private static final String timeFormat = "\\d{2}:\\d{2}:\\d{2}.\\d{3}";
  private Recorder recorder = new Recorder() {
    public void record(String message) {
      recordedMessage = message;
    }
  };

  public void testTimeDateLogger() throws Exception {
    Logger logger = new TimeDateLogger(recorder);
    message = "myMessage";
    logger.log(message);
    checkFormat(timeDateFormat);
  }

  public void testTimeLogger() throws Exception {
    Logger logger = new TimeLogger(recorder);
    message = "myMessage";
    logger.log(message);
    checkFormat(timeFormat);
  }

  private void checkFormat(String prefix) {
    String messagePattern = prefix + " " + message;
    if (!Pattern.matches(messagePattern, recordedMessage)) {
      fail(recordedMessage + " does not match pattern");
    }
  }
}
```

The `Logger` has changed as follows. Notice the protected abstract `format` method.

```
public abstract class Logger {
```

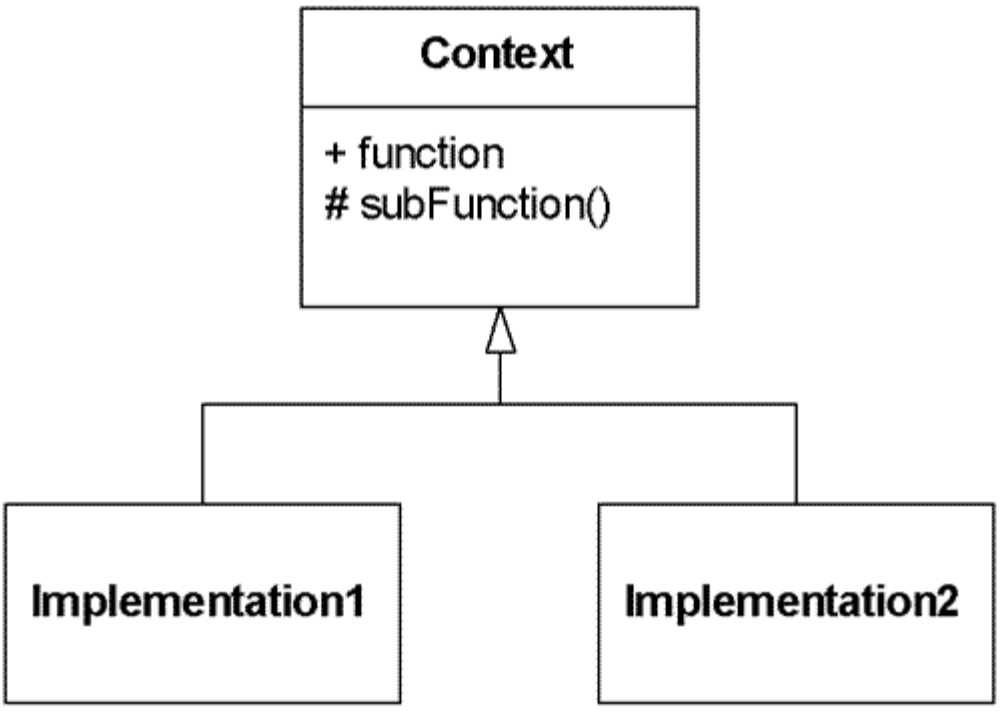
```
  private Recorder recorder;

  public Logger(Recorder recorder) {
    this.recorder = recorder;
  }

  public void log(String message) {
    recorder.record(format(message));
  }

  protected abstract String format(String message);
}
```

`TimeLogger` and `TimeDateLogger` simply implement the format method appropriate to their type, as shown below:

```
import java.text.*;
import java.util.Date;

public class TimeLogger extends Logger {
  public TimeLogger(Recorder recorder) {
    super(recorder);
  }

  protected String format(String message) {
    DateFormat format = new SimpleDateFormat("kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
    return prefix + " " + message;
  }
}
```

```
import java.text.*;
import java.util.Date;

public class TimeDateLogger extends Logger {
  public TimeDateLogger(Recorder recorder) {
    super(recorder);
  }

  protected String format(String message) {
    DateFormat format = new SimpleDateFormat("MM/dd/yyyy kk:mm:ss.SSS");
    Date now = new Date();
    String prefix = format.format(now);
    return prefix + " " + message;
  }
}
```

The canonical form of Template Method looks like this:

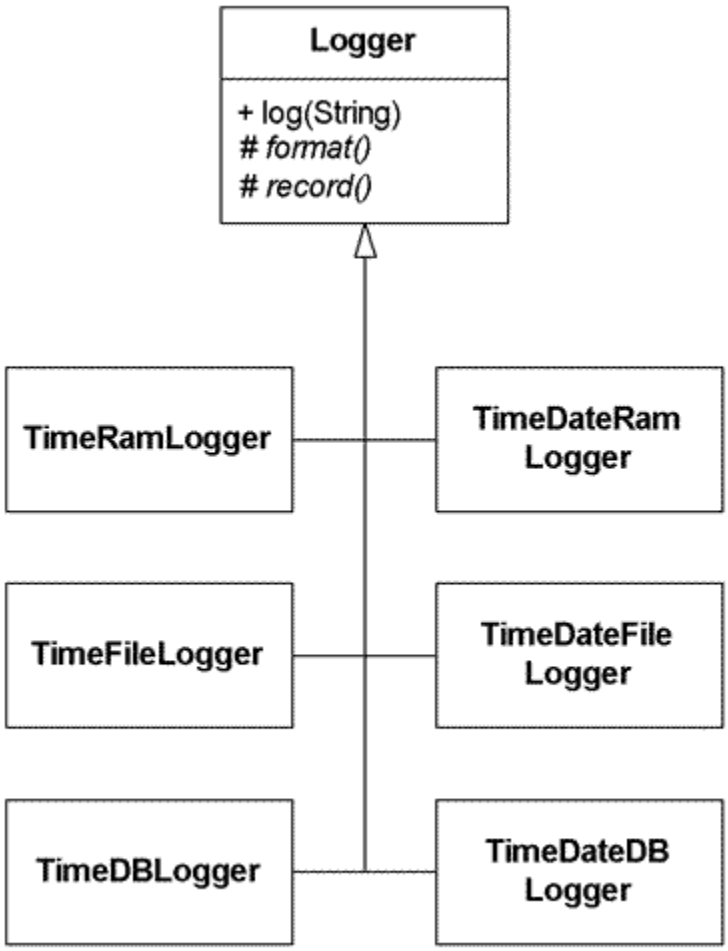


The `Context` class has at least two functions. One (here called `function`) is generally public, and represents some high-level algorithm. The other function (here called `subFunction`) represents some lower-level algorithm called by the higher-level algorithm. The derivatives of `Context` implement `subFunction` in different ways.

It should be clear how Strategy and Template Method solve the same problem. The problem is simply to separate a high-level algorithm from a lower-level algorithm in such a way that the two can vary independently. In the case of **Strategy**, this is solved by creating an interface for the lower-level algorithm. In the **Template Method** case, it is solved by creating an abstract method.

Strategy is preferable to Template Method when the lower-level algorithm needs to change at run time. This can be accomplished with Strategy simply by swapping in an instance of a different derivative. Template Method is not so fortunate; once created, its lower-level algorithm is locked in. On the other hand, Strategy has a slight time and space penalty compared to Template Method, and is more complex to set up. So Strategy should be used when flexibility is important, and Template Method should be used when time and space efficiency and simplicity are more important.

Could we have used Template Method to solve the whole `Logger` problem? Yes, but the result is not pleasant. Consider the following diagram.



Notice that there is one derivative for each possible combination. This is the dreaded `m x n` problem. Given two polymorphic degrees of freedom (e.g., recording and format) the number of derivatives is the product of those degrees.
This problem is common enough that the combined use of Strategy and Template Method to solve it (as we did in the previous example) is a pattern in and of itself, called Bridge.
Robert C. Martin (Uncle Bob) has been a software professional since 1970 and an international software consultant since 1990.

J2EE Patterns Catalog

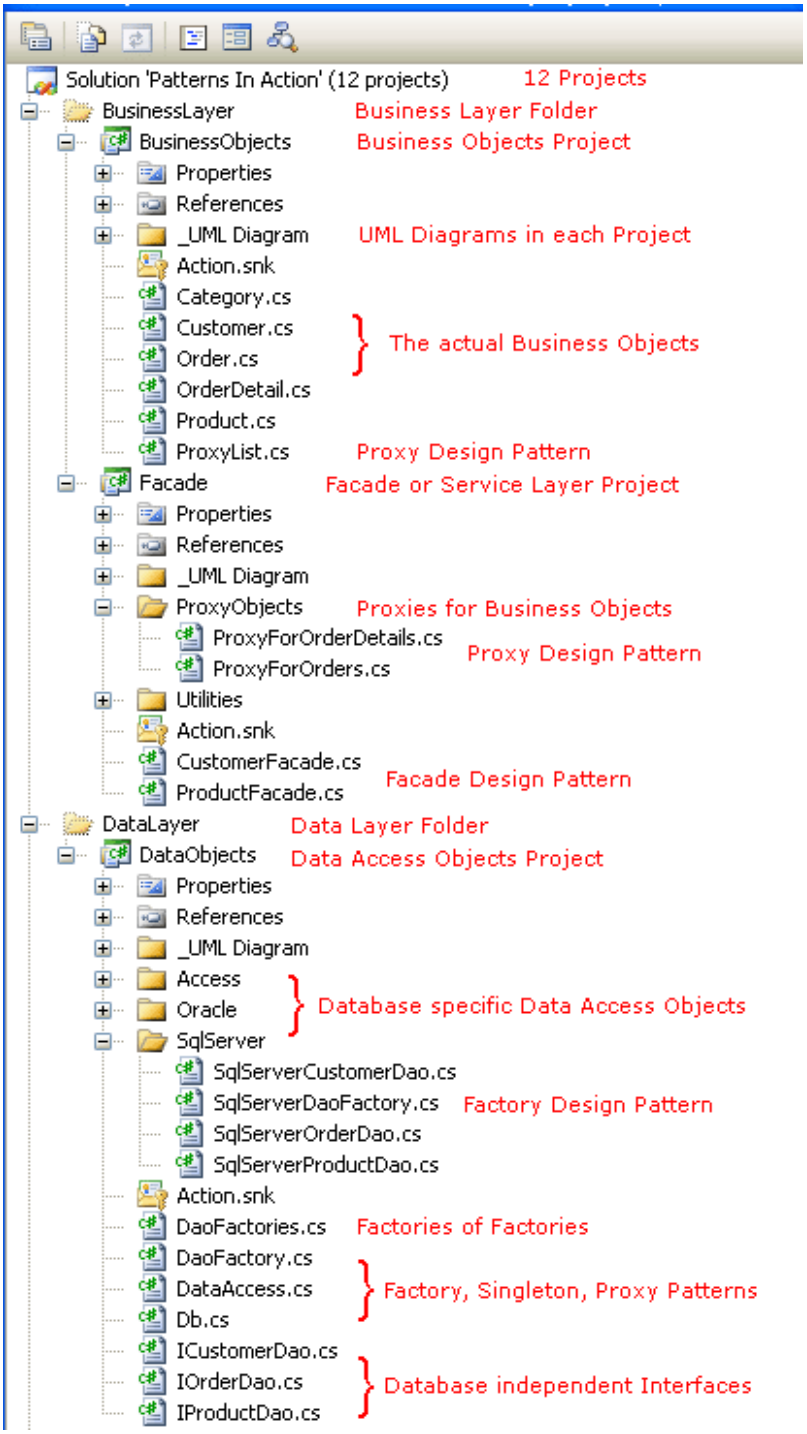
Each pattern in this catalog includes sample code from JavaTM BluePrints reference applications such as the Java Pet Store sample application.

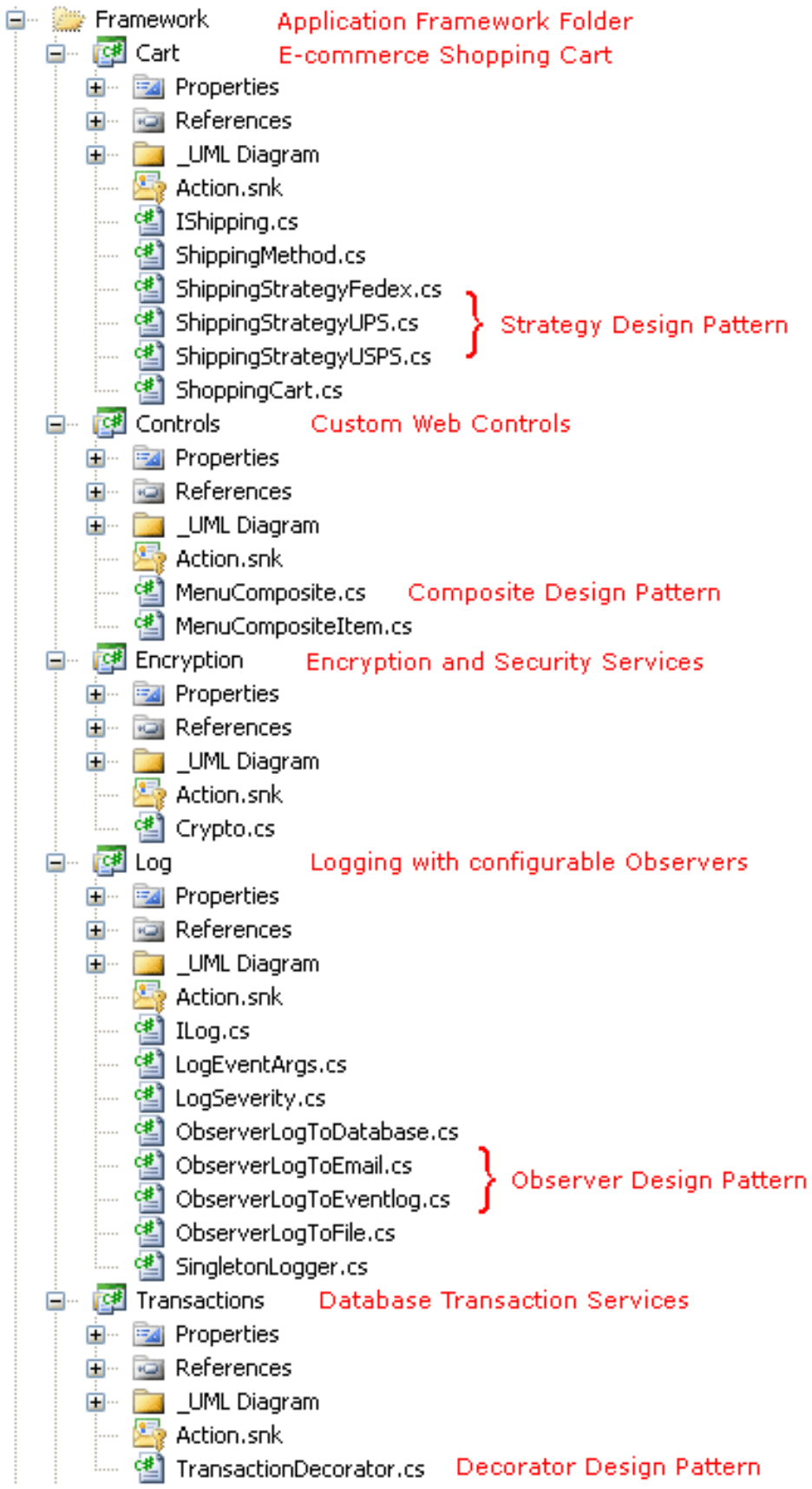
Pattern Name	Description
Business Delegate [ACM01]	Reduce coupling between Web and Enterprise JavaBeans™ tiers
Composite Entity [ACM01]	Model a network of related business entities
Composite View [ACM01]	Separately manage layout and content of multiple composed views
Data Access Object (DAO) [ACM01]	Abstract and encapsulate data access mechanisms
Fast Lane Reader	Improve read performance of tabular data
Front Controller [ACM01]	Centralize application request processing
Intercepting Filter [ACM01]	Pre- and post-process application requests
Model-View-Controller	Decouple data representation, application behavior, and presentation
Service Locator [ACM01]	Simplify client access to enterprise business services
Session Facade [ACM01]	Coordinate operations between multiple business objects in a workflow
Transfer Object [ACM01]	Transfer business data between tiers
Value List Handler [ACM01]	Efficiently iterate a virtual list
View Helper [ACM01]	Simplify access to model state and data access logic

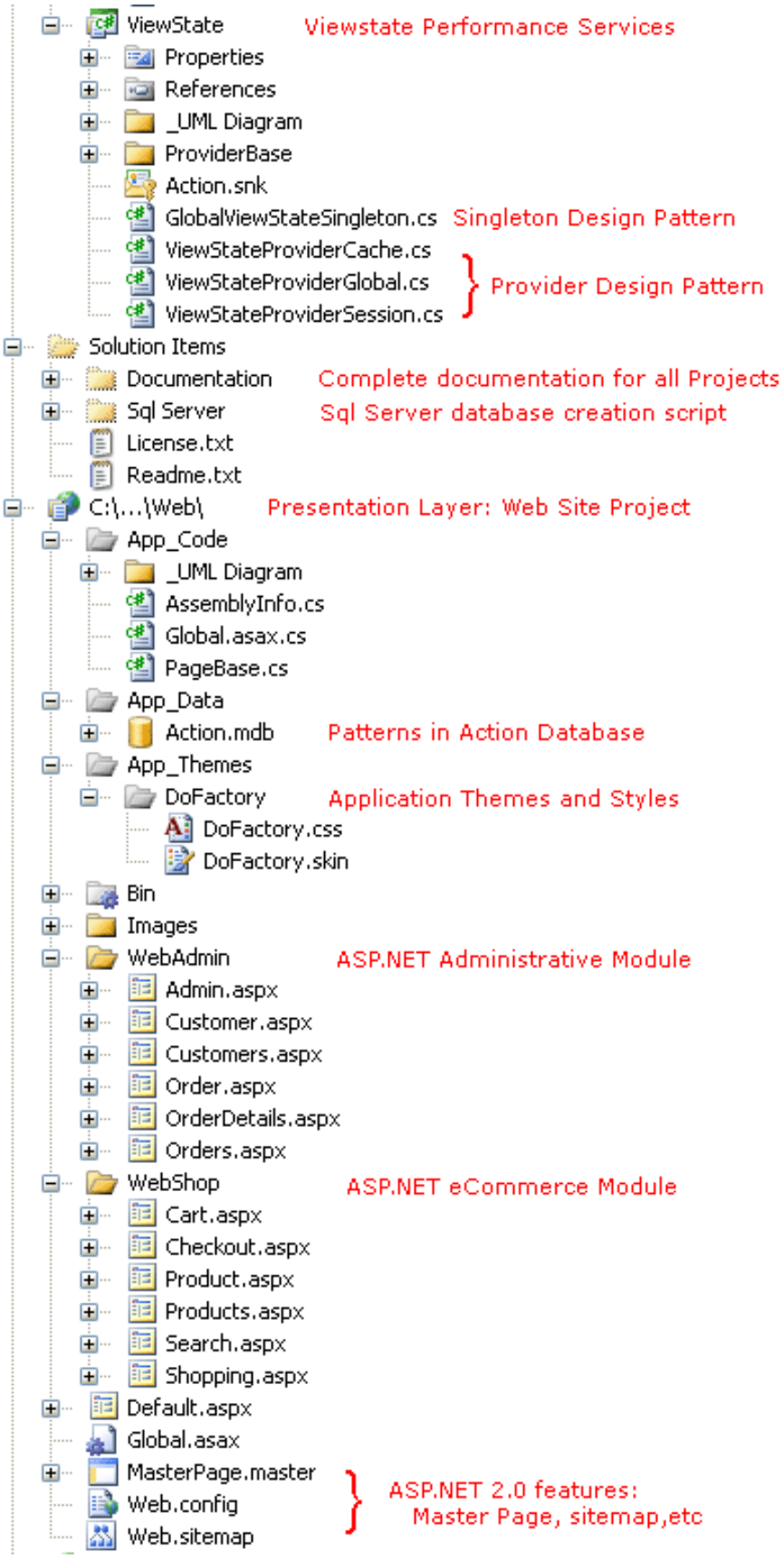
Code Examples Index

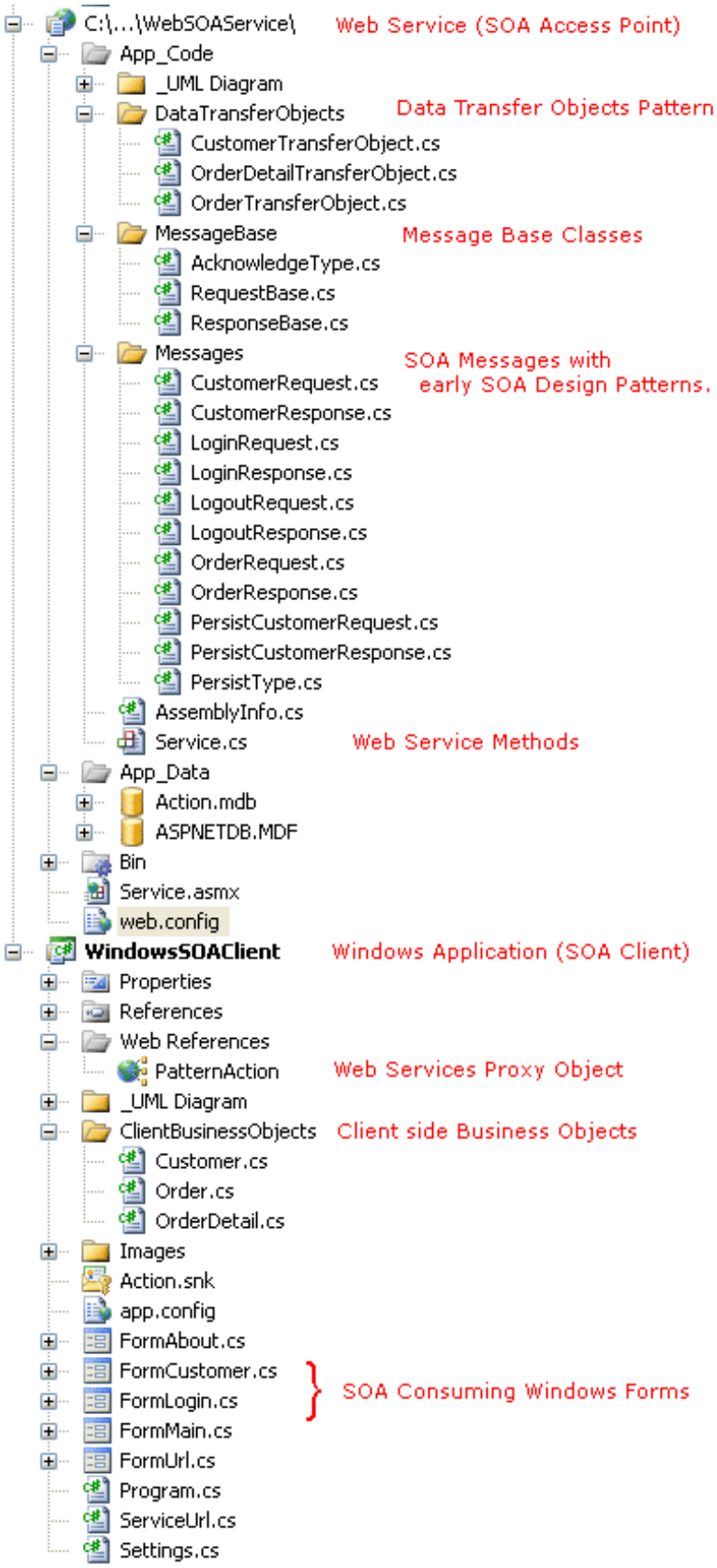
Courtesy dofactory design patterns examples.You can purchase these code examples online from http://www.dofactory.com/Framework/Framework.aspx

These series of screens demonstrates how code can be organized in terms of design pattern roles.




Framework
Application Framework Folder
Cart
E-commerce Shopping Cart
Properties
References
_UML Diagram
Action.snk
IShipping.cs
ShippingMethod.cs
ShippingStrategyFedex.cs
ShippingStrategyUPS.cs
ShippingStrategyUSPS.cs
Strategy Design Pattern
ShoppingCart.cs
Controls
Custom Web Controls
Properties
References
_UML Diagram
Action.snk
MenuComposite.cs
Composite Design Pattern
MenuCompositeItem.cs
Encryption
Encryption and Security Services
Properties
References
_UML Diagram
Action.snk
Crypto.cs
Log
Logging with configurable Observers
Properties
References
_UML Diagram
Action.snk
ILog.cs
LogEventArgs.cs
LogSeverity.cs
ObserverLogToDatabase.cs
ObserverLogToEmail.cs
ObserverLogToEventlog.cs
Observer Design Pattern
ObserverLogToFile.cs
SingletonLogger.cs
Transactions
Database Transaction Services
Properties
References
_UML Diagram
Action.snk
TransactionDecorator.cs
Decorator Design Pattern


ViewState
Viewstate Performance Services
Properties
References
_UML Diagram
ProviderBase
Action.snk
GlobalViewStateSingleton.cs
Singleton Design Pattern
ViewStateProviderCache.cs
ViewStateProviderGlobal.cs
ViewStateProviderSession.cs
Provider Design Pattern
Solution Items
Documentation
Complete documentation for all Projects
Sql Server
Sql Server database creation script
License.txt
Readme.txt
C:\...\Web\
Presentation Layer: Web Site Project
App_Code
_UML Diagram
AssemblyInfo.cs
Global.asax.cs
PageBase.cs
App_Data
Action.mdb
Patterns in Action Database
App_Themes
DoFactory
Application Themes and Styles
DoFactory.css
DoFactory.skin
Bin
Images
WebAdmin
ASP.NET Administrative Module
Admin.aspx
Customer.aspx
Customers.aspx
Order.aspx
OrderDetails.aspx
Orders.aspx
WebShop
ASP.NET eCommerce Module
Cart.aspx
Checkout.aspx
Product.aspx
Products.aspx
Search.aspx
Shopping.aspx
Default.aspx
Global.asax
MasterPage.master
Web.config
Web.sitemap
ASP.NET 2.0 features:
Master Page, sitemap,etc


C:\...\WebSOAService\
Web Service (SOA Access Point)
App_Code
_UML Diagram
DataTransferObjects
Data Transfer Objects Pattern
CustomerTransferObject.cs
OrderDetailTransferObject.cs
OrderTransferObject.cs
MessageBase
Message Base Classes
AcknowledgeType.cs
RequestBase.cs
ResponseBase.cs
Messages
SOA Messages with early SOA Design Patterns.
CustomerRequest.cs
CustomerResponse.cs
LoginRequest.cs
LoginResponse.cs
LogoutRequest.cs
LogoutResponse.cs
OrderRequest.cs
OrderResponse.cs
PersistCustomerRequest.cs
PersistCustomerResponse.cs
PersistType.cs
AssemblyInfo.cs
Service.cs
Web Service Methods
App_Data
Action.mdb
ASPNETDB.MDF
Bin
Service.asmx
web.config
WindowsSOAClient
Windows Application (SOA Client)
Properties
References
Web References
PatternAction
Web Services Proxy Object
_UML Diagram
ClientBusinessObjects
Client side Business Objects
Customer.cs
Order.cs
OrderDetail.cs
Images
Action.snk
app.config
FormAbout.cs
FormCustomer.cs
FormLogin.cs
FormMain.cs
FormUrl.cs
SOA Consuming Windows Forms
Program.cs
ServiceUrl.cs
Settings.cs

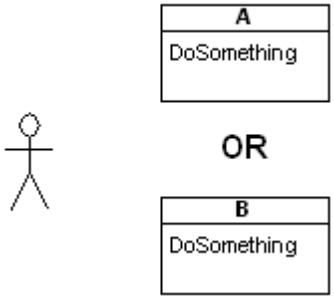
The "Road to the Bridge"

This is a story of how we get from patterns like the Interface pattern to Factory, Strategy, Proxy, Adapter, and finally to the Bridge Design Pattern.
This is an exploration of how to swap implementations of objects within our software architectures.

The problem

We want to be flexible in our architecture. We want to be able to swap implementations of objects/classes easily.

- Build to anticipate and celebrate change.
- Program to interfaces, not implementations..



The road to journey on...

Let's examine the range of solutions - showing the story of how to bind to two different implementations of the same interface - simple ways and more complex ways. Specifically how we move from:

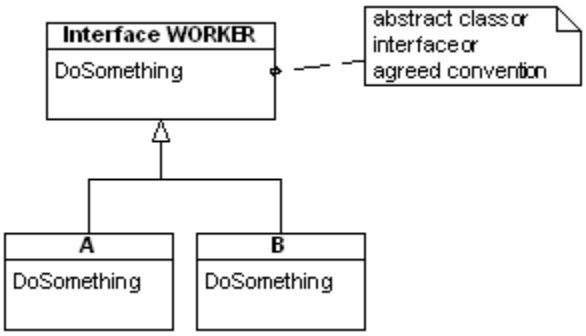
1. compile-time binding (one or the other is chosen by compiled code) to
2. factory based binding (one or the other is returned by a factory) to
3. dynamic binding using an intermediary object,

where #3 is achieved using Strategy, Adapter and Proxy, then this ultimately leads us to Bridge.
This journey strikes me as a powerful way of looking at a deep and common problem (building to embrace change), and that also unites multiple patterns under the one theme. Given programmers love the ideal of 'programming to interfaces' and being able to swap in different implementations, this story will show how to do it at many different levels and in fact how many of the design patterns are all about helping us achieve it.

Interface pattern

Interface, compile time choice

Alternative implementations of an interface. Instantiate one or the other implementation of that interface. The code that uses the object is unaware of which object it is using. "Program to an interface"




```
Worker o = new A()
// Worker o = new B()
o.DoSomething()
```

Here the choice is at compile time, by commenting out one or the other instantiation.

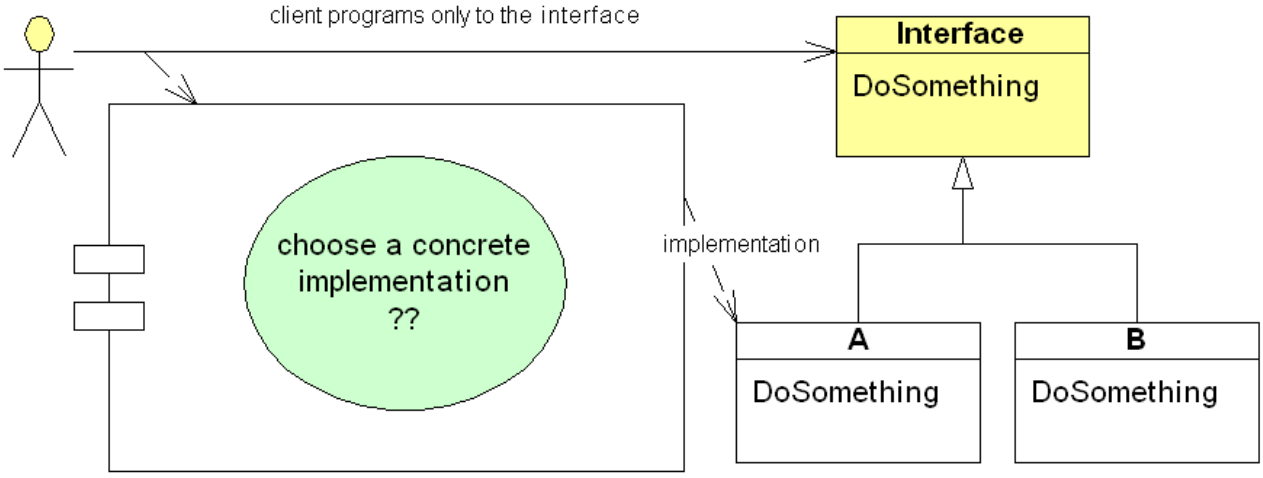
Interface, dynamic run time choice

Same solution except choose particular implementation dynamically at runtime using a flag.

```
if flag
  Worker o = new A()
else
  Worker o = new B()
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

Factory

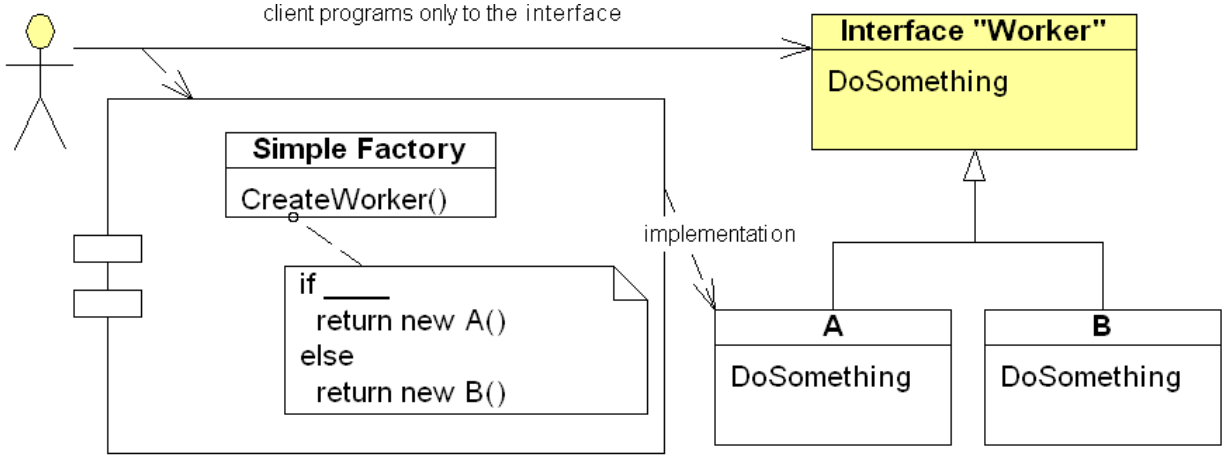
Create A or B at runtime by asking another class to create the concrete object for us. Pass in the flag to the factory or let the factory decide for itself which implementation we want.
Factory class is the only class to refer to concrete products. The client refers to the interface/abstract class only.
We are still talking directly to the concrete object (either an A or a B).



There are a number of factory method variants:

Simple Super Dumb Factory

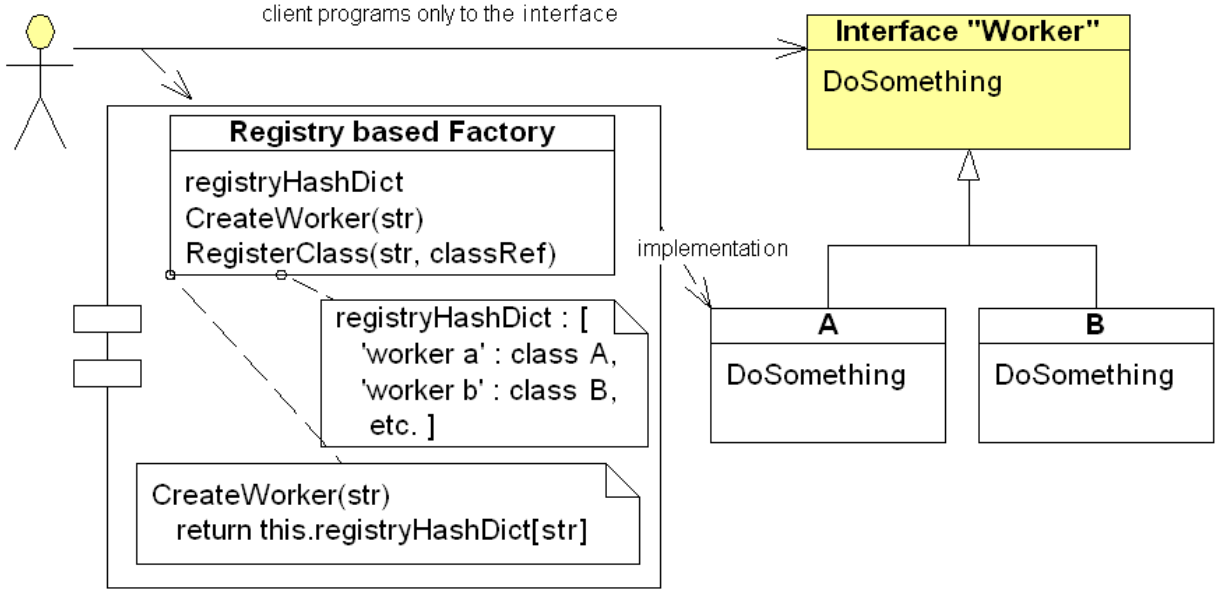
Encapsulates the "dynamic run time choice" solution discussed in the beginning of this talk. Benefit is that the conditional logic containing the if statement is hidden and possibly centralized in a factory class.
Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.
The choice is made via conditional code.




```
Factory f = new SimpleFactory()
Worker o = f.CreateWorker()
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

Registry Based Factory

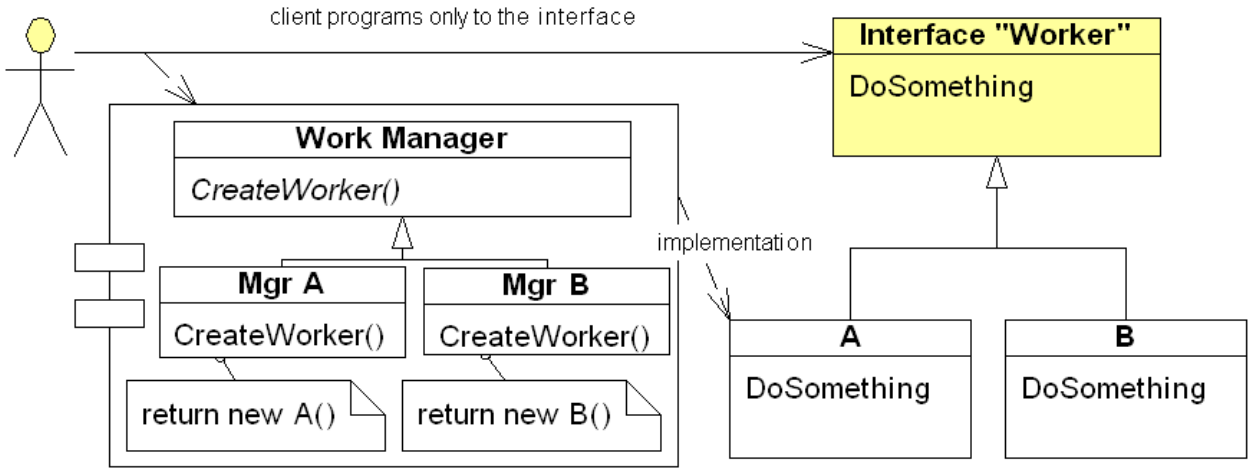
Maintains a registry of mappings between strings (or any type of key e.g. objects, class references, numbers etc.) and class references. Benefit: more generalized, no if statements.
Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.
The choice is made via registry key.




```
key = 'worker a'   // in setup code somewhere
Factory f = new RegistryFactory()
Worker o = f.CreateWorker(key)
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

GOF Factory Method

Assumes the client *already has* an instance of some class which needs either a A or B version of a worker class.
Each alternative instance of the existing class overrides a create method differently, each instantiating a different concrete product - typically one matching their own functionality. Benefit: no class reference language facilities required.
Factory class is the only class that refers directly to concrete products. Client refers only to interface/abstract class.
The choice is made via polymorphic override.



Note that the choice as to which Work Manager (MgrA or MgrB) to instantiate in the first place is going to be an issue, but is not the point of this example. The point is that once you have a particular brand of work manager, then you will get a related brand of of worker via the suitably overriden CreateWorker factory method.

```
WorkManager f = new MgrA()      // done somewhere in setup
Worker o = f.CreateWorker(key)
o.DoSomething()    // we don't know if its an A or a B.  Everything works ok.
```

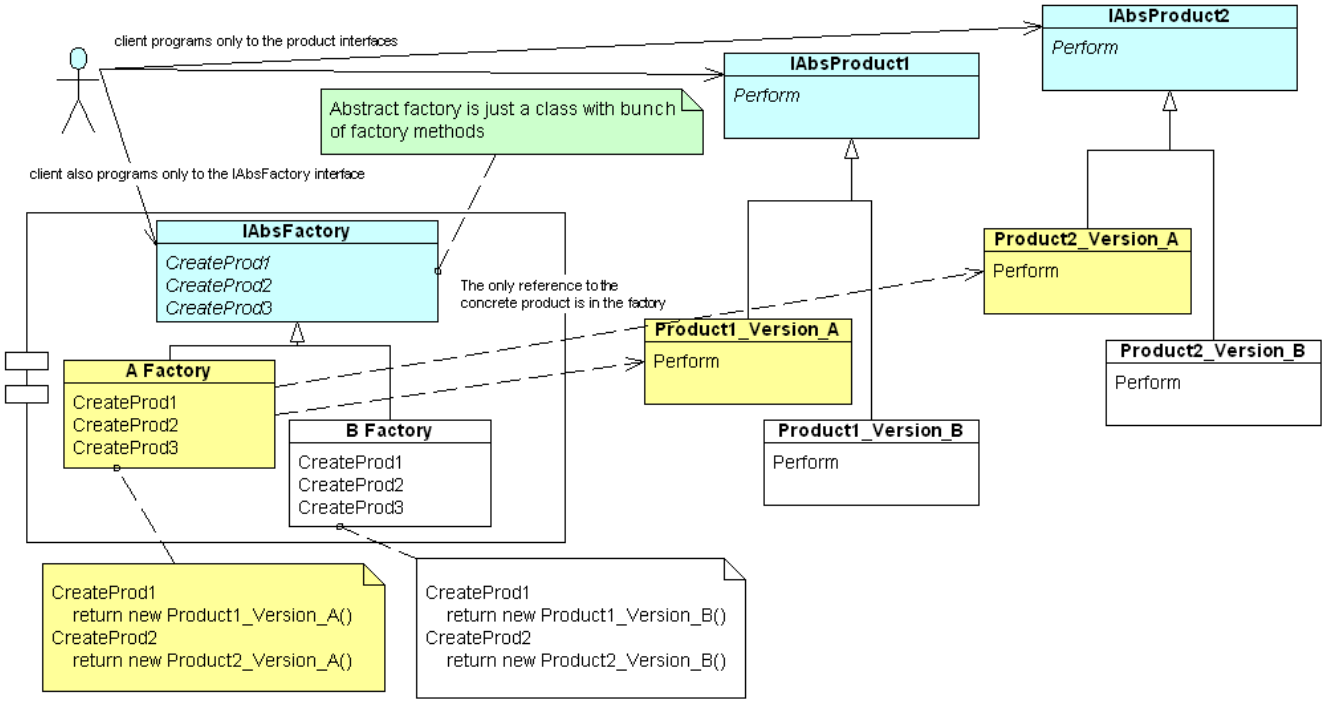
There will be parallel hierarchies, e.g. the WorkManager and the Worker hierarchies closely match, with A and B versions of their subclasses. Start to think of a *family* of classes.
My further thoughts, including a more detailed example of Factory Method here.

Abstract Factory

Abstract factory similar to factory method, in that there is something being overridden.
Abstract factory is the same as factory method, except there is more than one Creation method. E.g. CreateWorker, CreateAdministrator, CreatePoliceman - such that the class containing the factory methods might as well become a sole purpose class for dispensing these related classes.
The abstract factory is a mere mechanism for delivering A versions of B versions. E.g. Client wants A version of products
Client programs against interfaces thus can switch between A or B. Specifically, the client only talks to

- IAbstractProductFactory
- IProduct1
- IProduct2
- IProduct3

Abstract Factory

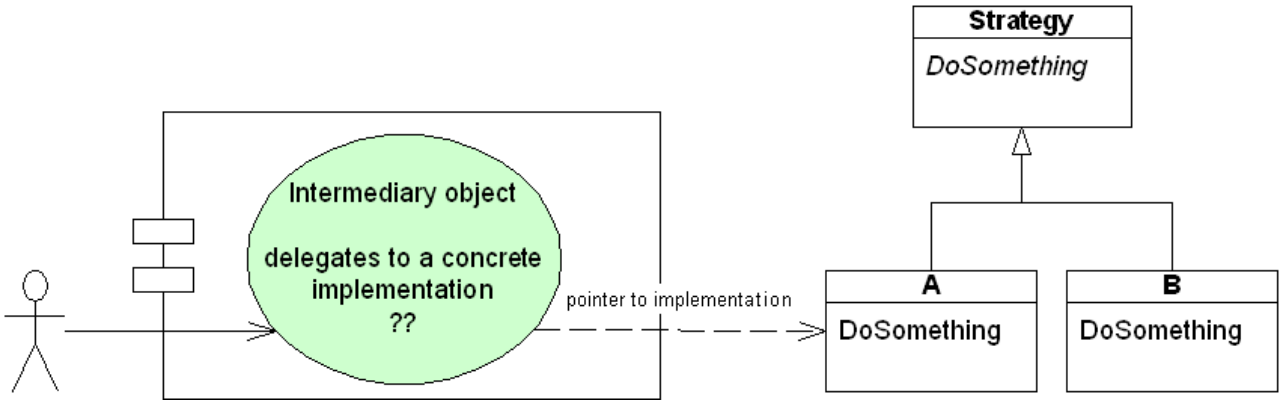


IAbstractProductFactory f = new ProductFactoryVersionA()
// choice is made at compile time, via factory method (run time) via strategy (runtime)
IProduct1 p1 = f.CreateProduct1()
IProduct2 p2 = f.CreateProduct2()
IProduct3 p3 = f.CreateProduct3()
All products p1, p2, p3 are in the above example A versions, and compatible with each other.

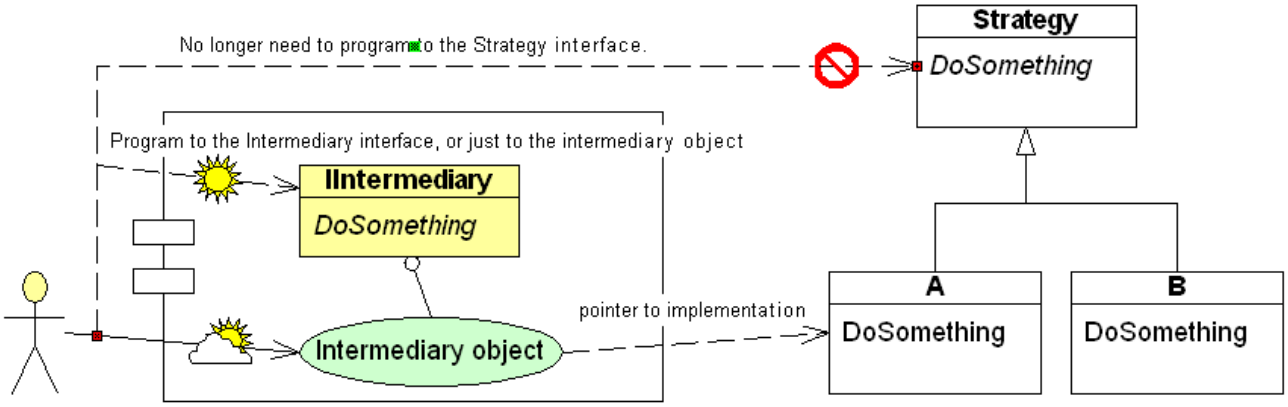
My further thoughts on Abstract Factory www.atug.com/andypatterns/abstract_factory_thoughts.htm
.

Indirection - get to implementation A or B via intermediary

Rather than instantiate A or B and refer to them directly (albiet via a flexible interface variable), another approach is to refer to the same object all the time and hide the switching *behind* that object.

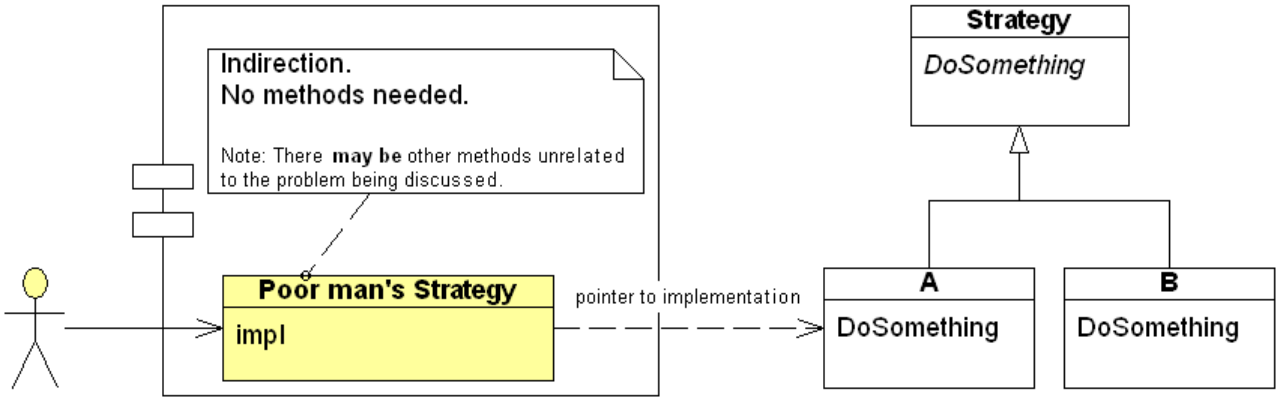


Now, because what is behind the intermediate object is hidden (and rightly so), you no longer need to program to the Strategy interface.



If you want to still program to an interface (good idea) then program to the Intermediary interface. If you want to run free and wild, program to the intermediary object api.
Variants are as follows:

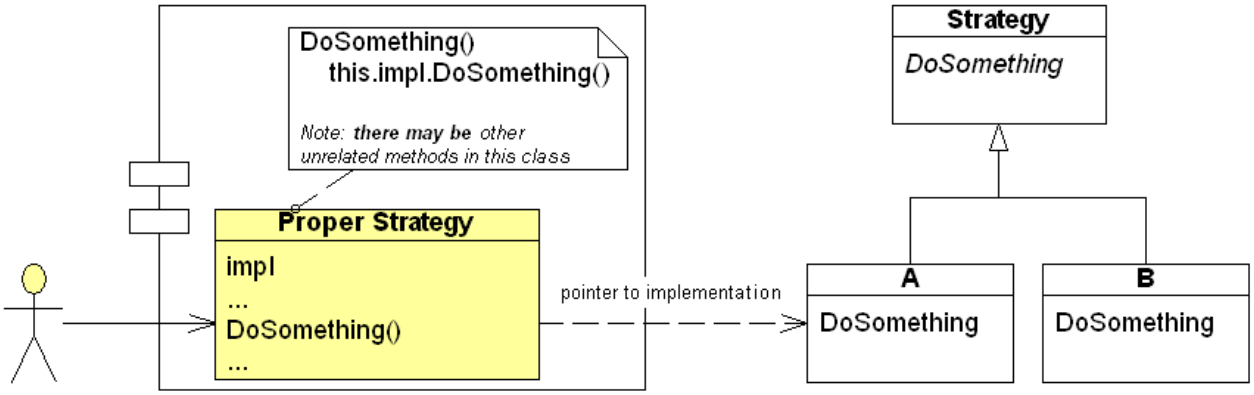
Proxy - methodless indirection using "demeter" referencing



Responsibility of the client to know the API of the strategy. So still programming to the strategy interface. You have to since the intermediary has no methods, or rather, has no methods specifically related to accessing the A & B classes.

o = new Intermediary()
o.SetStrategy(new A()) // done in setup somewhere, or via a factory or via dependency injection framework
o.impl.DoSomething()

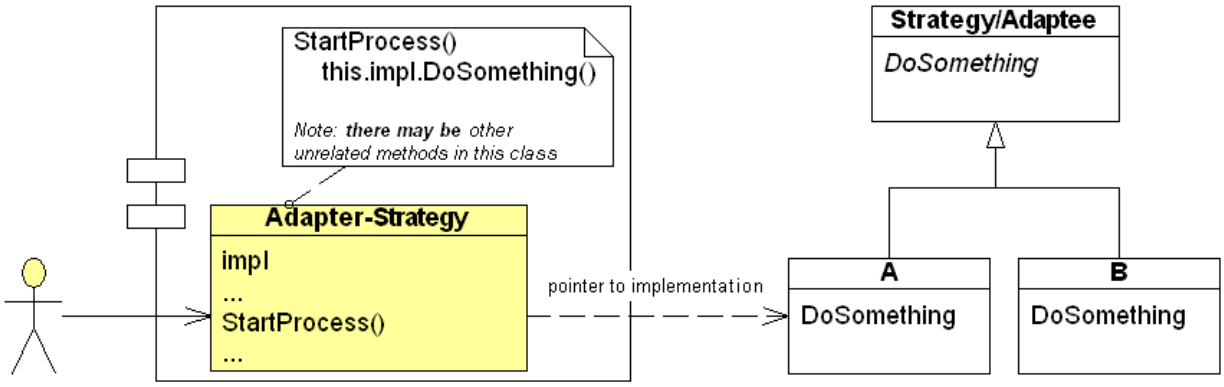
Proper Strategy



o = new Intermediary()
o.SetStrategy(new A()) // done in setup somewhere, or via a factory or via dependency injection framework
o.DoSomething()
later you can switch the strategy without the client code caring.
o.SetStrategy(new B())
o.DoSomething() // different behaviour or different implementation occurs

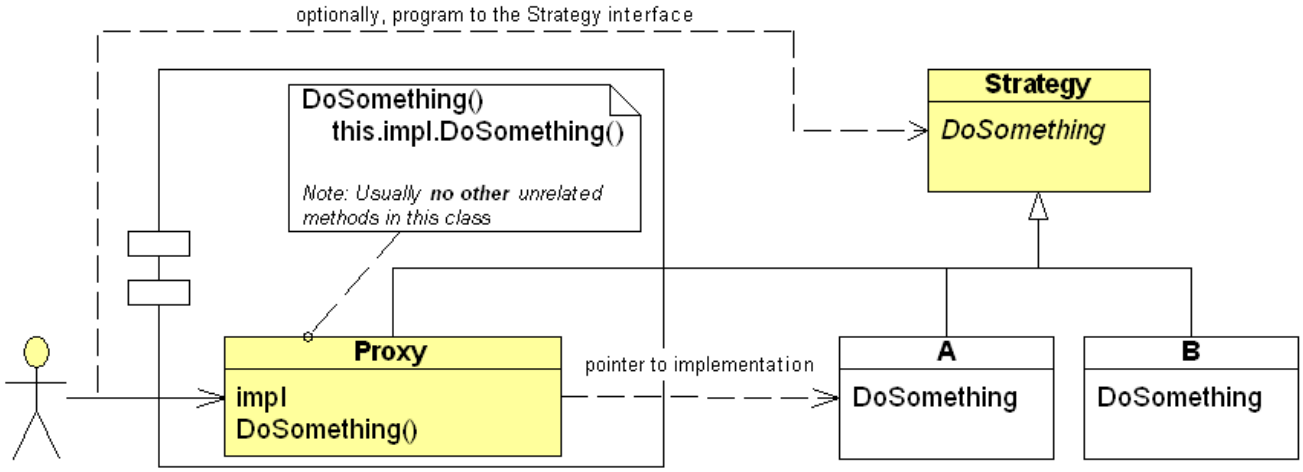
Strategy with a touch of the Adapter pattern

If your implementation has a slightly different API than the one your client code wants to use, then you can adapt it at the same time as you are strategizing...



Proxy - going all the way

If your only have the same methods in your intermediary object as you have in your implementation, then you can have the intermediary inherit from the abstract implementation interface. This turns the pattern into proxy, and lets you optionally, program to the Strategy interface again.



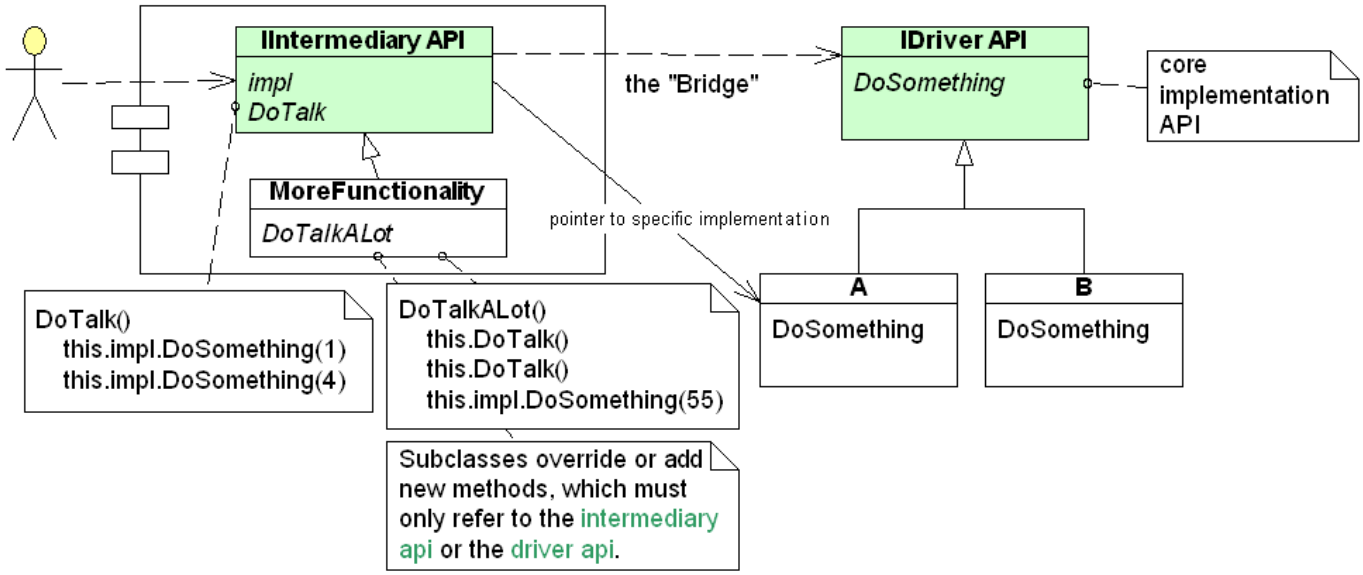
The proxy, whilst *inheriting* from Strategy, can also implement extra methods, though this is diverging a little from the intent of Proxy.

An alternative to inheritance, the proxy can *implement the interface* of the Strategy class, and get some similar polymorphic substitutability benefits.

Bridge

This is still a variant on accessing different behaviour via an intermediary.
Bridge is just strategy with a oversized lhs context.
Same as strategy except there is

- Massive subclassing going on on the lhs (the 'context' side).
- The nature of the lhs methods are more compositional, adaptive and far reaching (not just a simply strategy delegation)



Massive subclassing going on in the lhs. context

The reason is that you are wanting lots of methods and lots of functionality, lots of classes. E.g. you want to have a GUI or DB subsystem, not just a single strategy.

The nature of the lhs and rhs methods

Typically rhs (implementation/driver) calls are more primitive, and one lhs method will call the rhs. many times. e.g. see the DoTalk() method, above.
The lhs methods can be diverse, comprising

- lhs method simply calls rhs method. Method names can change or be the same. Simple delegation with no extra work.
- lhs methods more complex and adapt and do extra lines of code as needed
- Lots of logic in the lhs methods and may have associated helper classes. But in the end they call stuff on the intermediary api.

Insulated from change. Allow lhs and rhs to vary independently.

Client is insulated from changes. Should not talk talk to implementation, even if it is the abstract implementation interface because the abs impl. may change. If the abstract implementation interface does change then this affects only the Intermediary but not the client code. Client code should thus only talk to intermediary.
Similarly, if you change the Intermediary API, then only the client is affected - the r.h.s. (the abstract implementation interface and concrete implementations) are not affected.
In this sense the lhs and rhs can vary independently. Ok - so there are repercussions when things vary - but they are limited, as discussed above.

Final thought on Bridge

You could simplify Bridge and have the client code talk directly to the rhs. abstract implementation interface. You would be reverting to where we started on this "road to Bridge". Nothing wrong with that - but you would lose the 'insulation against change' that Bridge gets you. And with Bridge the lhs can have lots of complex logic and the rhs implementations need only implement the more primitive operations. That is a big win.

Solutions overview

Summary of the ways of coupling your components

Technique	Meta-Pattern	Pattern	Description	
To implementation A or B directly	Program to Interface	Interface	Interface, compile time choice	Alternative implementations of an interface. Instantiate one or the other implementation of that interface. The code that uses the object is unaware of which object it is using. "Program to an interface"
			Interface, conditional code	Same solution as above, except choose particular implementation dynamically at runtime using a flag.
	Factory	Factory Method	conditional code	
			registry	
			polymorphic factory method	
		Abstract Factory	abstract factory - polymorphic	
			conditional code	
			class registry	
To implementation A or B via intermediary	Indirection	Dot notation drilling	methodless proxy using demeter referencing	
		Strategy	strategy - may be extra methods not related to the strategising	
		Proxy	proxy, all methods mapped (demeter is happy). inherit	

		Adapter	adapted proxy-like strategy. different method names sometimes	
		Bridge	rhs - methods usually more primitive. Only talk to abs. impl. lhs - all adapted & thus changeable. can build hierarchies	

Final thoughts

The presentation of the patterns form a story of simple to complex.
And its a story of two broadly different techniques,

- Getting to the implementation A or B directly
- Getting to the implementation A or B via an intermediary object

Adapter vs. Bridge

Adapter is closer to Bridge in that the adaptation on the lhs. (the context) can be not just a renaming and mapping of methods, but extra logic and whatever it takes to make the adapation work. So the lhs. is closer to the free wheeling compositional lhs of Bridge Pattern. By compositional I mean that a single lhs. method can comprise of complex code and multiple calls to the rhs. methods. In Bridge the lhs methods can even call on other methods in the same lhs, whereas in Adapter this is not really the intent.

IOC (inverison of control) also fits in here somewhere.

Dependency injection. Inject a context object or wire up dependent objects. Allows you to program normally. Allows different implementations to be injected in.

Microkernels also fit in here.

Amongst other things, a Microkernel style architecture allows alternative plugins (services) to fulfil the implementation.

- Maybe think of it as *service* A or B.
- Or *plugin* A or B.

There seem to be three types of MicroKernel:

- Service location, like COM where you either ask for a service and get an interface which you use, or you call a service and the late binding binds to an appropriate service at the last minute. **Style of programming:** slightly different - must ask for an interface before using it.
- Message broadcasting kernel, where messages are broadcast to all plugins and the chain of responsibility pattern is used, and a plugin/service which can make sense of the message acts on it (either consuming it or passing it on for someone else to have a go at). **Style of programming:** different - you must create messages send them into the kernel, either synchronously or asynchronously.
- Dependency Injection Microkernel, where all object attributes refering to other objects (dependencies) are injected for you by a framework. Rather than setting up these references

yourself manually, as normal programming style dictates, you leave it to magic - which allows other implementations to be swapped in. You must of course program to interfaces not to concrete classes, in order for this trick to work. **Style of programming:** normal, you just call methods on objects that you have references to. The fact that the references have been wired up by a framework (which consults a plugin directory & setup file telling us which plugins are active) is hidden from us.

Maybe one or more of the above three descriptions of a Microkernel is not actually a microkernel - I am just learning about this stuff. But I have seen references that suggest my analysis is correct. E.g. The Castle IOC framework for .NET calls itself a microkernel.

A variable of type interface is really a another 'secret' form of indirection

I have made a broad distinction between accessing implementations A or B either directly or via an intermediary. Thinking about it some more, when you do access A or B directly, you do so via an an intermediary variable declared of type abstract/interface. This is when you are being good and 'programming to interfaces'.

Thus you could argue that even even when you are accessing an object (implementation A or B) directly, you are in fact still acting through an intermediary - the interface variable. !

A Pattern Language

From http://xianlandia.com/te-amo/2007/10/12/when_is_a_pattern_not_a_pattern.html

> *Going back to Christopher Alexander, the "pattern language" concept started off with a fairly strict, well defined structure. Alexander's patterns all have a sensitizing example, a context (when to use or apply the pattern), a problem (expressed in terms of conflicting forces), and a solution (a way to balance or reconcile those forces). Application of the pattern produces a new context (and hence a way to "chain" patterns together).*
>
> *The pattern language, then, is a vocabularly of patterns that relate to each other. In Alexander's work, there was a hierarchical, or scale, dimension. His A Pattern Language book starts at the level of nation-states across the world and works its way down to things like doorknobs.*
>
> *When the extreme programming folks involved with Ward's wiki and the "Gang of Four" adapted the pattern metaphor to software engineering, they did not really preserve the pattern language concept. They also debated among themselves between what they called descriptive and prescriptive patterns (actually, I'd better check if those were the real terms they used). They were aware of the Alexander precedent and conscious at least about which parts they were applying to the computer software context. (Alexander himself foresaw and promoted this application, btw, in the 1980s.)*

Pattern Relationships

From Huston Design Patterns
http://home.earthlink.net/~huston2/dp/patterns.html

Creational Patterns

1. Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementory: Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126], Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, pp81,134]

2. Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object. [GOF, p135]

3. Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]

4. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

5. Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately. [GOF, p105]
6. Builder is to creation as Strategy is to algorithm. [Icon, p8-13]

7. Builder often builds a Composite. [GOF, p106]

8. Factory Methods are usually called within Template Methods. [GOF, p116]
9. Factory Method: creation through inheritance. Prototype: creation through delegation.

10. Often, designs start out using Factory Method (less complicated, more customizable, subclasses

proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

11. Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]
12. Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. [GOF, p126]

Structural Patterns

1. Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]

2. Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, p161]

3. Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

4. Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters. [GOF, p149]

5. Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects. [GOF, p219]

6. Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes. [GOF, pp173,349]

7. Composite can let you compose a Mediator out of smaller pieces through recursive composition. [Vlissides, Apr96, p18]

8. Decorator lets you change the skin of an object. Strategy lets you change the guts. [GOF, p184]

9. Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert. [GOF, p220]

10. Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

11. Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

12. Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

13. Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF, p193]

14. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

15. Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

16. Flyweight is often combined with Composite to implement shared leaf nodes. [GOF, p206]

17. Flyweight explains when and how State objects can be shared. [GOF, p313]

Behavioral Patterns

1. Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it. [GOF, p222]

2. Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time. [GOF, p347]

3. Chain of Responsibility can use Command to represent requests as objects. [GOF, p349]

4. Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor. [GOF, p232]

5. Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value. [GOF, p346]

6. Command can use Memento to maintain the state required for an undo operation. [GOF, p242]

7. MacroCommands can be implemented with Composite. [GOF, p242]

8. A Command that must be copied before being placed on a history list acts as a Prototype. [GOF, p242]

9. Interpreter can use State to define parsing contexts. [GOF, p349]

10. The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable). [GOF, p255]

11. Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight. [GOF. p255]

12. Iterator can traverse a Composite. Visitor can apply an operation over a Composite. [GOF, p173]

13. Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass. [GOF, p271]

14. Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. [GOF, p346]

15. On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them. [GOF, p282]

16. Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional

protocol where it makes requests of the subsystem classes but not vice versa). [GOF, p193]

17. Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally. [GOF, p271]

18. State is like Strategy except in its intent. [Coplien, Mar96, p88]

19. Flyweight explains when and how State objects can be shared. [GOF, p313]

20. State objects are often Singletons. [GOF, p313]

21. Strategy lets you change the guts of an object. Decorator lets you change the skin. [GOF, p184]

22. Strategy is to algorithm. as Builder is to creation. [Icon, p8-13]

23. Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic). [Coplien, Mar96, p88]

24. Strategy objects often make good Flyweights. [GOF, p323]

25. Strategy is like Template Method except in its granularity. [Coplien, Mar96, p88]

26. Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]
27. The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters. [Johnson, Huni, Engel, 1995, p8]

GoF Structure Similarities

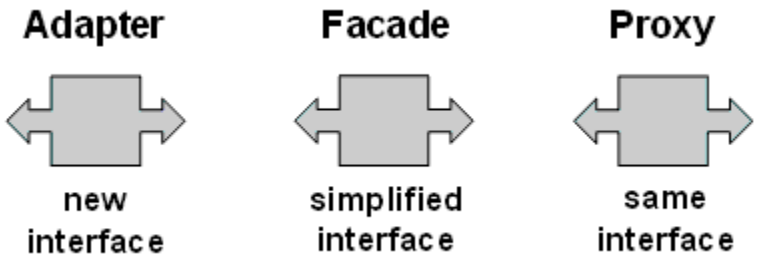
Taken from Huston Patterns Page http://home.earthlink.net/~huston2/dp/patterns.html

Left-Right symbol = wrapper/wrappee or delegation or "has a" relationship

Adapter ... wrap a legacy object that provides an incompatible interface with an object that supports the desired interface

Facade ... wrap a complicated subsystem with an object that provides a simple interface

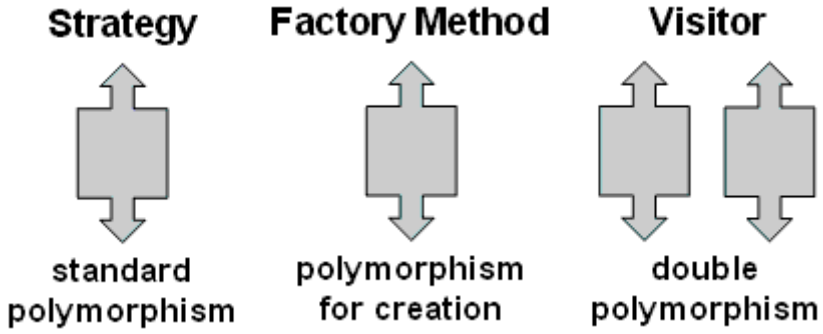
Proxy ... wrap an object with a surrogate object that provides additional functionality



Up-Down symbol = inheritance hierarchy (promote interface to a base class and bury implementation alternatives in derived classes)

Strategy ... define algorithm interface in a base class and implementations in derived classes

Factory Method ... define "createInstance" placeholder in the base class, each derived class calls the "new" operator and returns an instance of itself



Visitor ... define "accept" method in first inheritance hierarchy, define "visit" methods in second hierarchy ... a.k.a. "double dispatch"

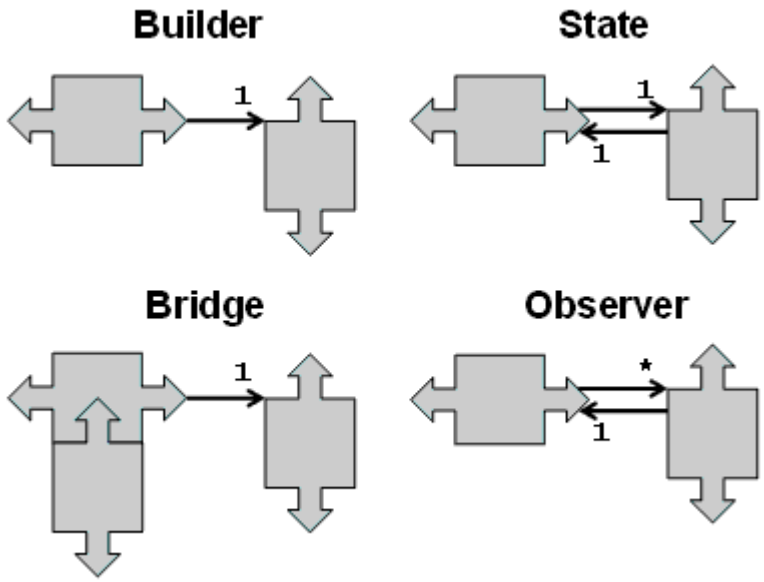
Category: a wrapper wraps an inheritance hierarchy

Builder ... the "reader" delegates to its configured "builder" ... each builder corresponds to a different representation or target

State ... the FiniteStateMachine delegates to the "current" state object, and that state object can set the "next" state object

Bridge ... the wrapper models "abstraction" and the wrappee models many possible "implementations" ... the wrapper can use inheritance to support abstraction specialization

Observer ... the "model" broadcasts to many possible "views", and each "view" can dialog with the "model"



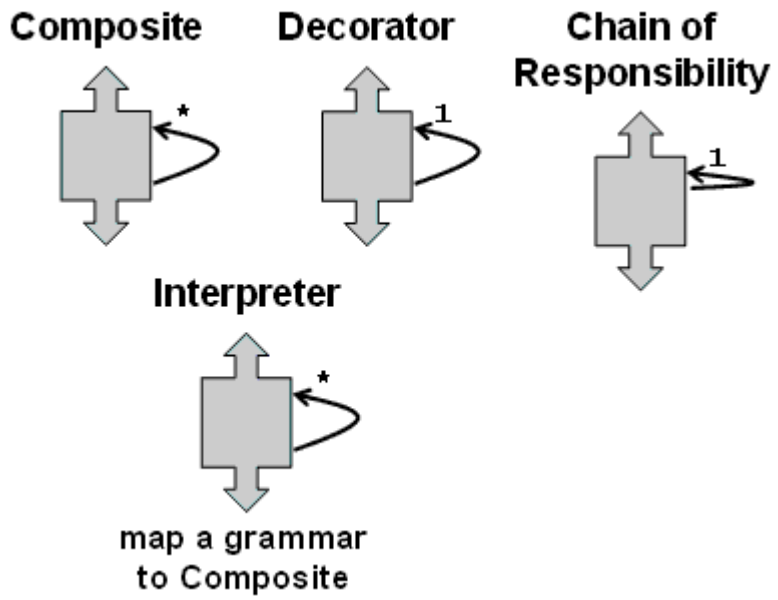
Category: recursive composition

Composite ... derived Composites contain one or more base Components, each of which could be a derived Composite

Decorator ... a Decorator contains a single base Component, which could be a derived ConcreteComponent or another derived Decorator

Chain of Responsibility ... define "linked list" functionality in the base class and implement "domain" functionality in derived classes

Interpreter ... map a domain to a language, the language to a recursive grammar, and the grammar to the Composite pattern

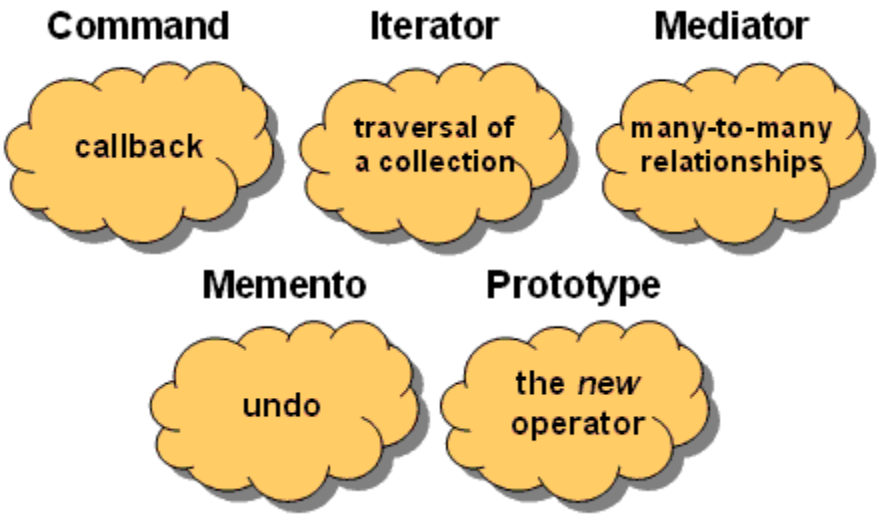


Cloud symbol = promote X to "full object status"

Command ... encapsulate an object, the method to be invoked, and the parameters to be passed behind the method signature "execute"

Iterator ... encapsulate the traversal of collection classes behind the interface "first..next..isDone"

Mediator ... decouple peer objects by encapsulating their "many to many"



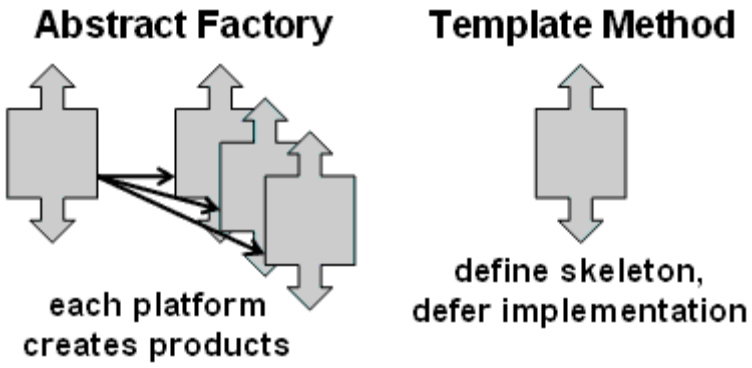
linkages in an intermediary object

Memento ... encapsulate the state of an existing object in a new object to implement a "restore" capability

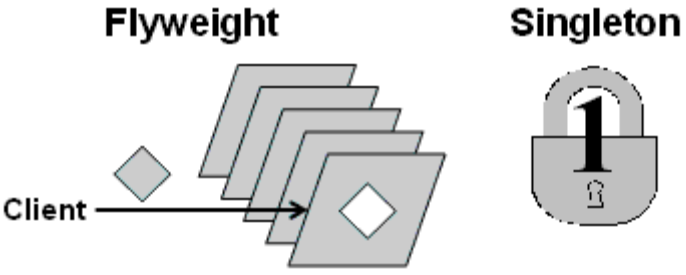
Prototype ... encapsulate use of the "new" operator behind the method signature "clone" ... clients will delegate to a Prototype object when new instances are required

Category: miscellaneous

Abstract Factory ... model "platform" (e.g. windowing system, operating system, database) with an inheritance hierarchy, and model each "product" (e.g. widgets, services, data structures) with its own hierarchy ... platform derived classes create and return instances of product derived classes



Template Method ... define the "outline" of an algorithm in a base class ... common implementation is staged in the base class, peculiar implementation is represented by "place holders" in the base class and then implemented in derived classes



Flyweight ... when dozens of instances of a class are

desired and performance boggs down, externalize object state that is peculiar for each instance, and require the client to pass that state when methods are invoked

Singleton ... engineer a class to encapsulate a single instance of itself, and "lock out" clients from creating their own instances

Are Design Patterns simply Missing Language Features?

From http://c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures

On various places, it has been claimed that use of DesignPatterns, especially complex ones like VisitorPattern, are actually indicators that the language being used isn't powerful enough. Many DesignPatterns are by convention rather than encapsulable in a library or component, and as such contain repetition and thus violate OnceAndOnlyOnce. If it didn't contain at least ***some*** repetition, or something that could be Refactored out, then it wouldn't be a pattern.

Discussion on this topic culled from elsewhere on WardsWiki:

Here is an interesting quote from PaulGraham, which leads to the question "Are Patterns a DesignSmell?"

This practice is not only common, but institutionalized. For example, in the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of case (c), the human compiler, at work. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough - often that I'm generating by hand the expansions of some macro that I need to write.

from http://www.paulgraham.com/icad.html

PeterNorvig argues in the same vein in "DesignPatternsInDynamicProgramming".

PaulGraham said "Peter Norvig found that 16 of the 23 patterns in Design Patterns were 'invisible or simpler' in Lisp." http://www.norvig.com/design-patterns/

"Pattern" generally implies there is some kind of duplication. Duplication should ideally be factored out (OnceAndOnlyOnce) so that only the differences remain. It appears to me that stronger languages can more easily remove such duplication because sometimes one has to make a kind of sub-language to do it.

A list of DesignPatterns and a language feature that (largely) replaces them:

```
VisitorPattern  ..............  GenericFunctions (MultipleDispatch)
```

```
 FactoryPattern  ..............  MetaClasses, closures
 SingletonPattern ..............  MetaClasses
 IteratorPattern................  AnonymousFunctions
              (used with HigherOrderFunctions,
               MapFunction, FilterFunction, etc.)
 InterpreterPattern.............  Macros (extending the language)
               EvalFunction?, MetaCircularInterpreter
               Support for parser generation (for differing syntax)
 CommandPattern ................  Closures, LexicalScope,
              AnonymousFunctions, FirstClassFunctions
 HandleBodyPattern..............  Delegation, Macros, MetaClasses
 RunAndReturnSuccessor..........  TailCallOptimization
 Abstract-Factory,
 Flyweight,
 Factory-Method,
 State, Proxy,
 Chain-of-Responsibility.......  FirstClass types (Norvig)
 Mediator, Observer.............  Method combination (Norvig)
 BuilderPattern.................  Multi Methods (Norvig)
 FacadePattern..................  Modules (Norvig)
 StrategyPattern................  higher order functions (Gene Michael
Stover?), ControlTable
 AssociationList................Dictionaries, maps, HashTables
                   (these go by numerous names in different languages)
```

Makes me wonder why we continue to put up with languages that don't have these features!

For the same reason we use the shoddy bug-ridden self-contradicting obsolete language called "English"!

- Because we were taught inadequate programming languages as children?

Because we didn't have a choice, at best we can become multi-lingual - but we still have to talk to people who think English is "the best language".

- I'm curious: What (natural) language ***is*** the best language, and what metric would determine such a thing? I'm a native speaker of English, and I think it a ***reasonable*** language. I learned it because that's what my family speaks, and that's what they speak where I live. (That said, my wife is Chinese, and our son is comfortably bilingual). I certainly won't claim it to be the ***best***, it certainly has its flaws (rather irregular spelling and grammar top the list, along with a wide variety of dialects and imprecision in technical matters - compared to, at least, German). It also certainly has its strengths - chief among them is that it's easy to speak BadEnglish? and be understood. Also, English speakers and authorities are generally not concerned with the PurityOfEnglish; hence the language is quite malleable and adaptable.

I don't see how language features replace design patterns. Language features can make them easier to use, but design patterns (the good ones) aren't language specific. -- EricHodges

[Are you sure about that, many of those patterns listed above come strait from the GOF, seems like patterns are workarounds for things that can't be further simplified, and that's a language smell. Why

would you ever use InterpreterPattern if you have a language that allows you to extend it... you wouldn't, because it's a hack for languages that don't allow that. Personally, after learning a bit about functional programming idioms, it seems many of the GOF patterns are OO hacks to emulate functional idioms.]

You'd still be using the InterpreterPattern, the language would just make it easier. A pattern isn't a set of source code lines. It's a way of doing things. -- EH

If you have double dispatch, why use Visitor?

Why not? Double dispatch doesn't do away with the need to abstract behavior. -- EH

Eh? The Visitor pattern implemented with MultiMethods is indistinguishable from dozens of other mundane multi-dispatched method calls. Yeah, I suppose you could say that the pattern's still there, but it's barely visible against the background - so it's not much use to identify it in that context.

I wouldn't say the GOF patterns are hacks to emulate ***functional*** idioms - at least some the patterns are structural. Rather, I think that some are language-specific idioms that have been generalized enough to apply to a class of languages. Visitor is a good example. -- DanMuller

[Yes, that class being OO languages, and Visitor allowing DoubleDispatch, a limited form of MultipleDispatch, I'd call it a hack. If you want to think of having a generic function mapped over a collection as still being the visitor pattern, then I can understand that point of view, but I'd demote it to the visitor idiom, because it isn't a pattern anymore.]

Yes, I agree that it isn't really a pattern at that point. Note that MultiMethods are not a feature specific to the functional paradigm - their defining characteristic is their polymorphic nature, and they could certainly have side-effects. But that's a digression from the topic at hand. -- DanM

Perhaps I'm not making myself clear. If you're applying Alexander's ThickWalls? pattern when building a house and you use a material that makes thick walls easy it doesn't mean you aren't applying that pattern. -- EH

The summary of VisitorPattern (from that page) is: ***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.*** Well, that's how MultiMethods work ***inherently***. VisitorPattern is only significant in languages that restrict polymorphism to methods that are part of the definition of "class".

A better analogy would be trying to apply Alexander's ThickWalls? pattern to the construction of bunkers, which ***of course*** have thick walls. -- DanM

Now I think you're not making yourself clear. If you build a bunker and it has to have thick walls then you're not applying the ThickWalls? pattern? I don't follow. -- EH

[If you build the thick wall, then you're applying the pattern... but if the wall was already there, as in multiple dispatch in some languages, using it does not constitute applying that pattern, because you didn't have to build it. I guess I'm saying design patterns are what you call things that haven't become part of the base language. Which is Paul Grahm's point, a sufficiently flexible language has no use for "patterns", anything that looks like a pattern quickly becomes part of the language, thus killing its status as a pattern. Things that were called patterns in assembly, are now just features of more modern languages, design patterns will eventually become the same.]

- It doesn't matter if I code the application of a pattern or the language does that for me. The pattern is still being used. I think you're focusing on specific instances of patterns. I'm talking about the pattern itself. Building a language that makes applying patterns easy doesn't make patterns go away. It just makes them easier to use.
    - Of course, nobody ever speaks of the "function" pattern, or the "class" pattern, or numerous other things that we take for granted because most languages provide them as built-in features. OTOH, programmers in a purely PrototypeOrientedLanguage? might well find it convenient to simulate classes with prototypes...and programmers in a hypothetical language that provided continuations (or AssemblyLanguage programmers using an architecture with only branch instructions) as the only flow control mechanism (and no macro capability) might well construct code sequences that emulate things like function/procedure call, loops, if/then/else, etc. If one were to study the assembly-language output of a CeeLanguage compiler, without knowing the source of the code, lots of patterns would emerge.
    - And what about the AssociationList? What are alists, if not a common Lisp DesignPattern? There isn't any official "alist" library anywhere which performs manipulations on alists; generally most manipulations on alists are sufficiently easy that they are hand-coded ***in situ***, rather than being abstracted in a common place. Of course, modern Lisp dialects now have native associative containers based on HashTables, RedBlackTrees, or other data structures capable of supporting fast key retrieval, so the alist is no longer strictly necessary (though still frequently used). In early Lisp dialects, OTOH, the alist was how associative data structures were implemented. In my mind, the AssociationList does qualify as a design pattern. However, it would be absurd to speak of a dictionary design pattern in Python, or a "map" design pattern in C++, etc.
    - ***I question whether an AssociationList deserves the label 'pattern'. If there's a continuum of concepts from patterns down to trivial idioms, AssociationList's are certainly closer to an idiom. An associative data structure is a higher-level concept that might deserve to be called a (still trivial) pattern, with the AssociationList as a means of implementing it in a particular family of languages.*** -- DanM

(A more verbose explanation...) Bunkers by definition have thick walls. (A thin-walled bunker would be no use to anyone.) If you restrict the realm of discourse to bunkers rather than buildings in general, then the ThickWalls? pattern disappears. Similarly, if you restrict the realm of discourse from OO languages in general down to OO languages that support MultiMethods, then the VisitorPattern disappears.

Delving more particularly into MultiMethods vs the VisitorPattern: VisitorPattern invents a class to embody the behavior of an operation separately from the multiplicity of classes that the operation applies to. With MultiMethods there are two ways that you can approach this. A GenericFunction can replace the second class, and the implementing methods replace the methods of that class. In the VistiorPattern?, Accept() together with the specific type of the Visitor argument invoke an operation polymorphically on several types of operand objects. With this first MultiMethod?-based approach, the artificiality of Accept() is not needed. MultiMethods trivially embody the VisitorPattern every time you define a MultiMethod? that is polymorphic on a single parameter.

The alternative arrangement, if you need the level of indirection that Accept() can buy you (you don't need to know the type of the operation to invoke Accept(), which is important if you choose to use internal recursion through a hierarchy of objects), you can define an Accept GenericFunction that takes two polymorphic arguments, and specialize the methods on both. Again, this is a trivial case of a MultiMethod? that's polymorphic on ***two*** arguments. Although this latter arrangement is more readily recognizable as a VisitorPattern, its most interesting feature is actually the internal recursion, and not the VisitorPattern-like separation of operator and operand (the latter being ***inherent*** to MultiMethods). -- DanM

Andy's Thoughts on Patterns in Languages etc.:

patterns help when language features are missing e.g. run time type checking was missing in C++, and a double dispatch mechanism (that visitor uses) is missing from most languages.

Over time, some patterns become language features if they are useful enough e.g. the iterator pattern is now reified in both java C# and python and e.g. delegates in C# and listeners in Java usually make the need for writing your own "observer pattern" unnecessary.

Also many language "idioms" are can be seen as low level "patterns".

I think it is important to note that although there is this evolving continuum of what is officially a pattern vs. language feature, the "idea" of what each pattern is for is still important to understand at a broader more abstract level. Appreciating the abstract idea allows us to use the concept of the pattern at both the mico and macro levels. E.g. A little bit of pointer indirection at the language level becomes wrapper at the class level and then proxy at the architectural level. Same idea. Reifying the abstraction allows design pattern papers to be written at the abstract level, which allows the detailed (and admittedly sometimes dull) discussion of pros and cons of different implementations, and allows the discussion of relationships between patterns.

Personally I would like all patterns to be available at a language level – perhaps some sort of customisable language like boo or groovy(?) would allow this. But to reiterate, I believe you would still need to be using patterns at the design/architectural level too, since they are an idea without specific implementation, which can be applied at many levels of granularity/abstraction.

Diagramming - UML and the future?

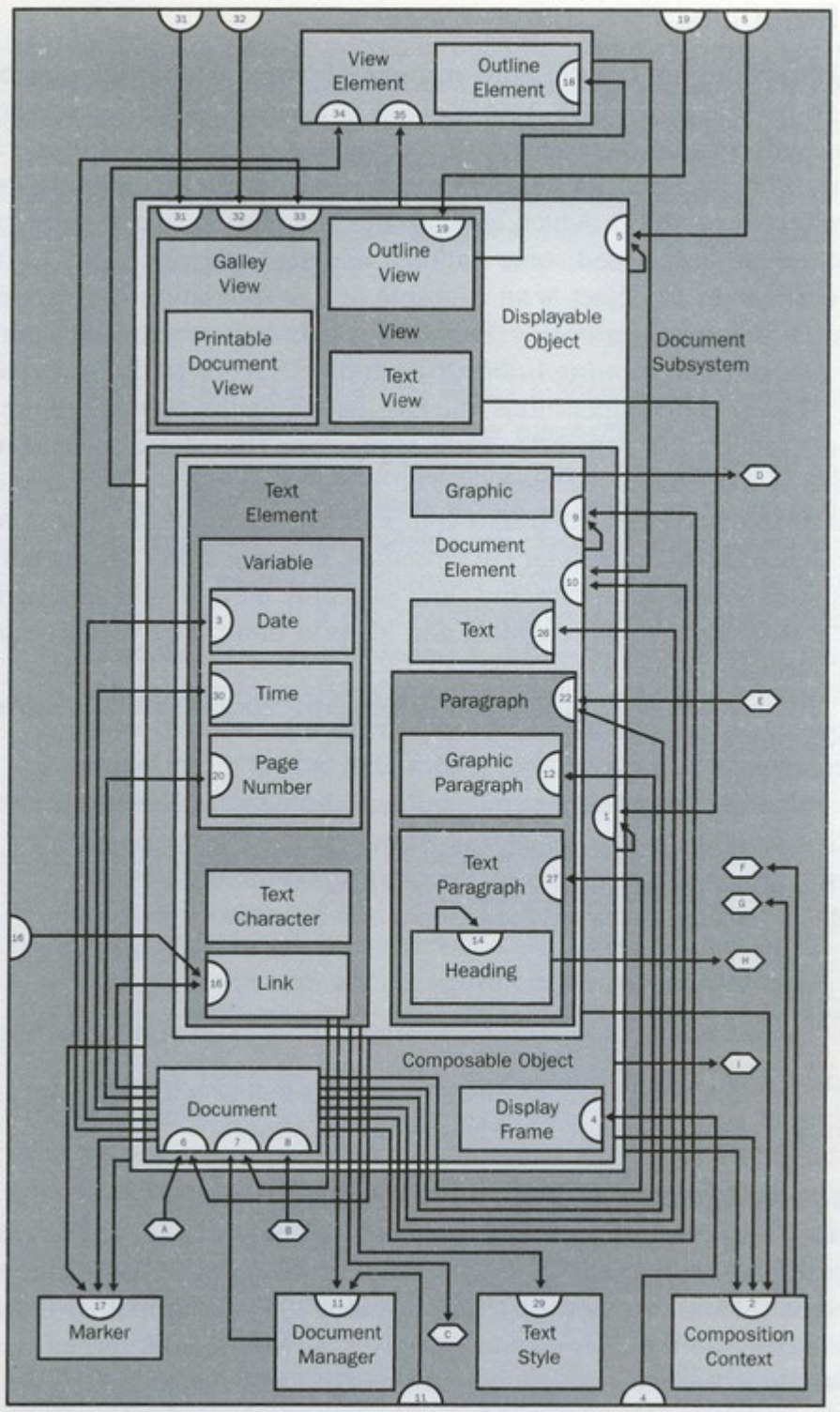


Figure 9-2 A visual model of collaborations, based on a graph in *Designing Object Oriented Software,* by Wirfs-Brock et al., 1991.

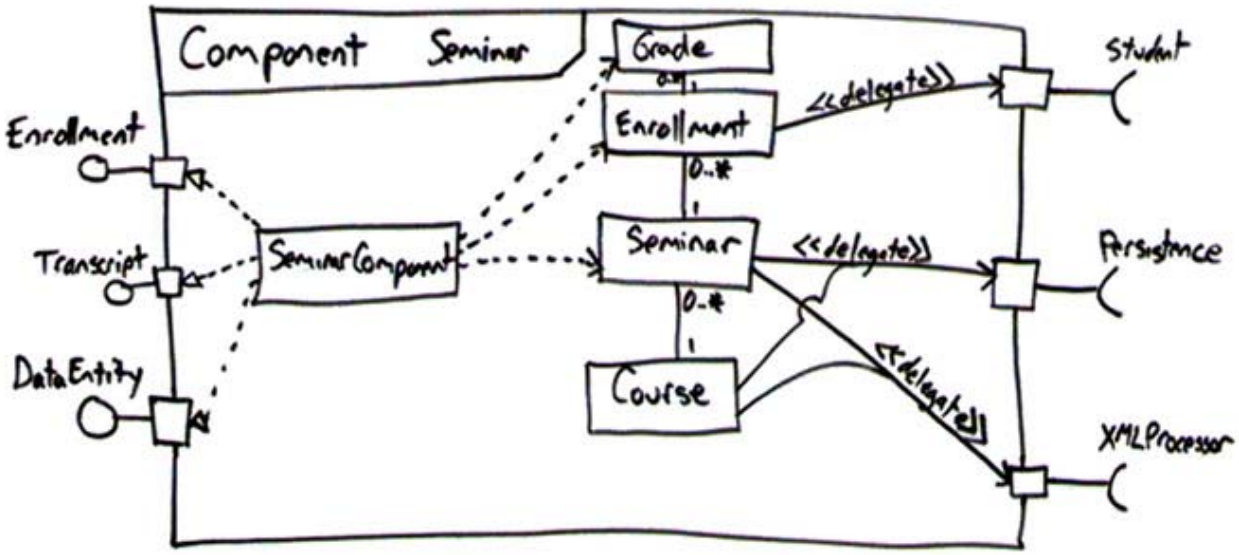


Figure 91 - from http://www.agilemodeling.com/style/interface.htm

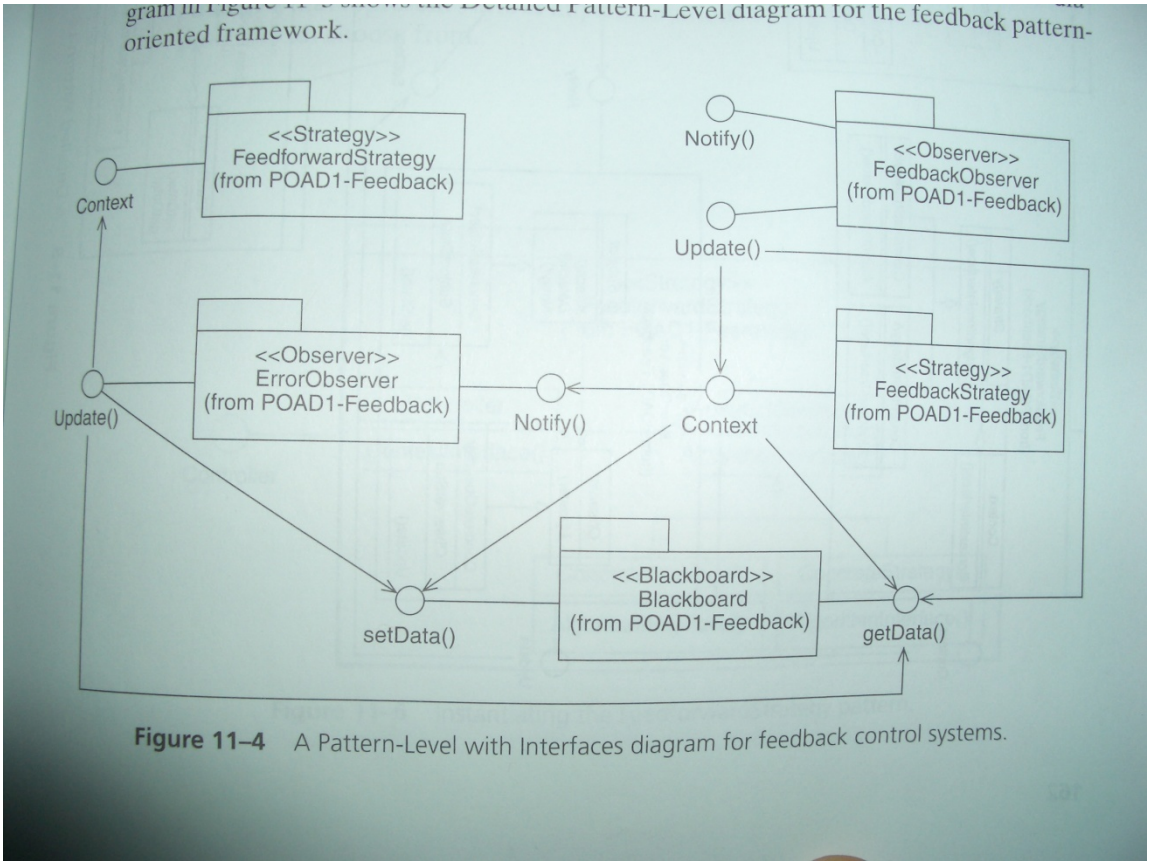


Figure 92 - Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems

Role Diagrams

Dirk Riehle
Ubilab, Union Bank of Switzerland
Bahnhofstrasse 45, CH-8021 Zurich
e-mail: Dirk.Riehle@ubs.com

Design patterns are patterns of classes and objects that represent solutions to recurring design problems. They are usually described using class diagrams. Class diagrams, however, often intertwine the actual solution with efficient ways of implementing it. This paper uses role diagrams to describe and compose patterns. Role diagrams help designers focus on the collaborations and distribution of responsibilities between objects. Role diagrams also are a better starting point for composing patterns. This paper presents several examples and reports on first experiences with using role diagrams for composing patterns which have been promising.

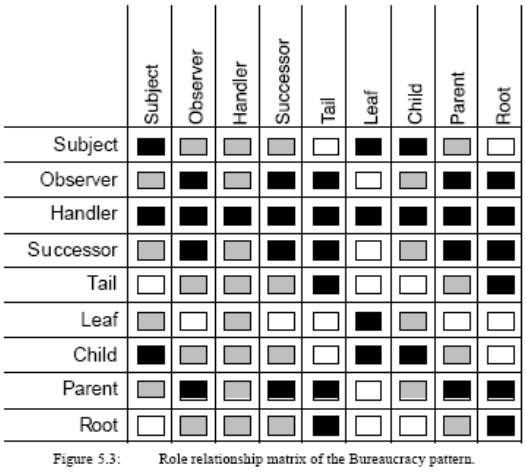


Figure 5.3: Role relationship matrix of the Bureaucracy pattern.

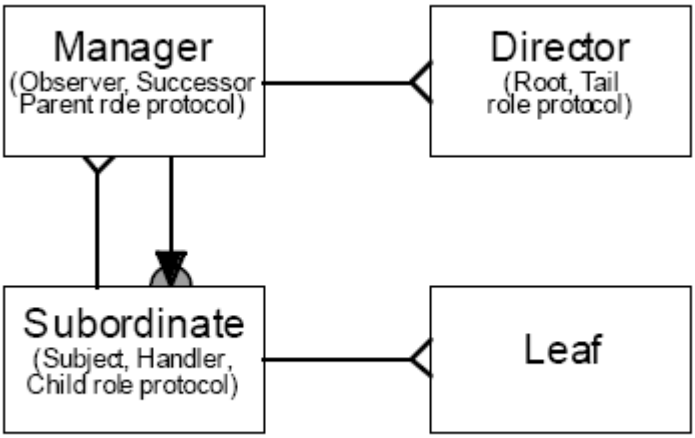
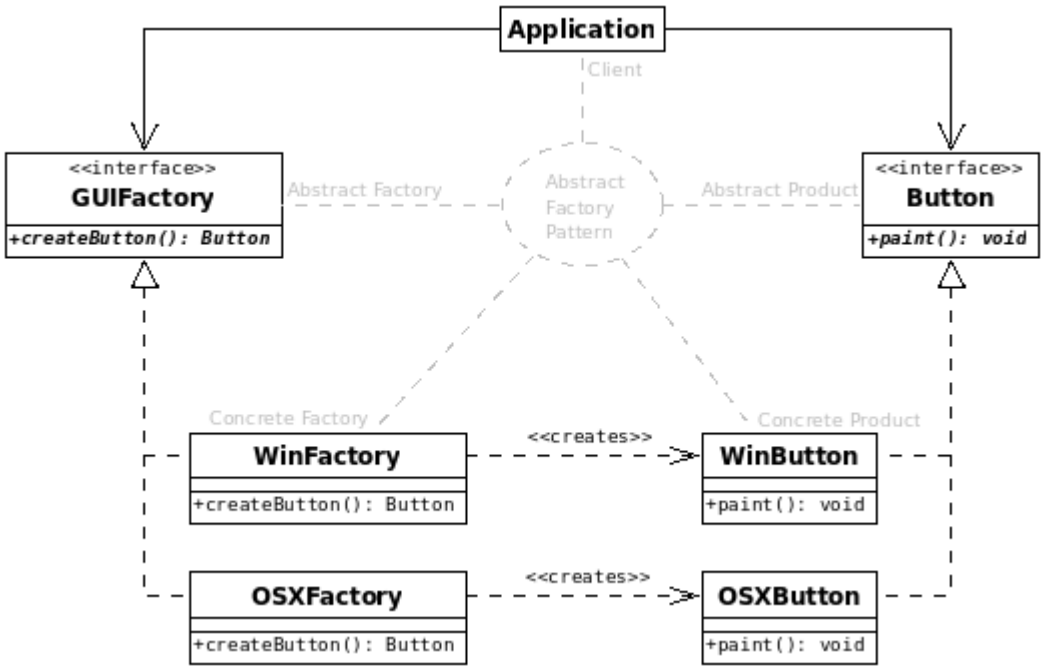


Figure 5.4: Class diagram for the Bureaucracy pattern.

A way of representing patterns in UML

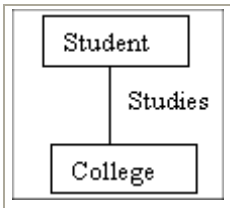
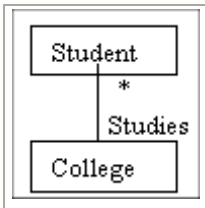
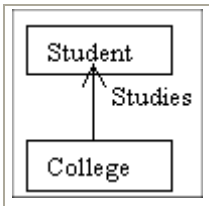
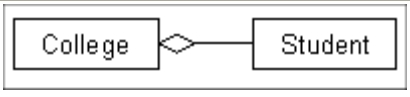
Using collaboration diagrams.

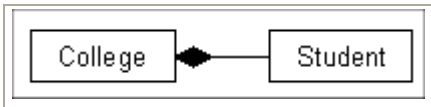
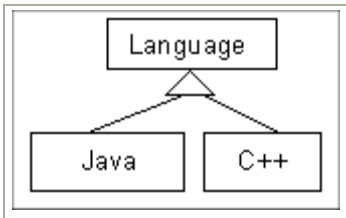
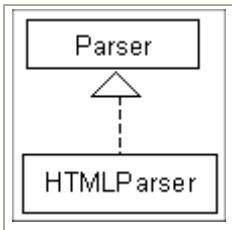


There are still problems with this approach since you don't get to see which specific method or fragment of code is involved in the pattern.

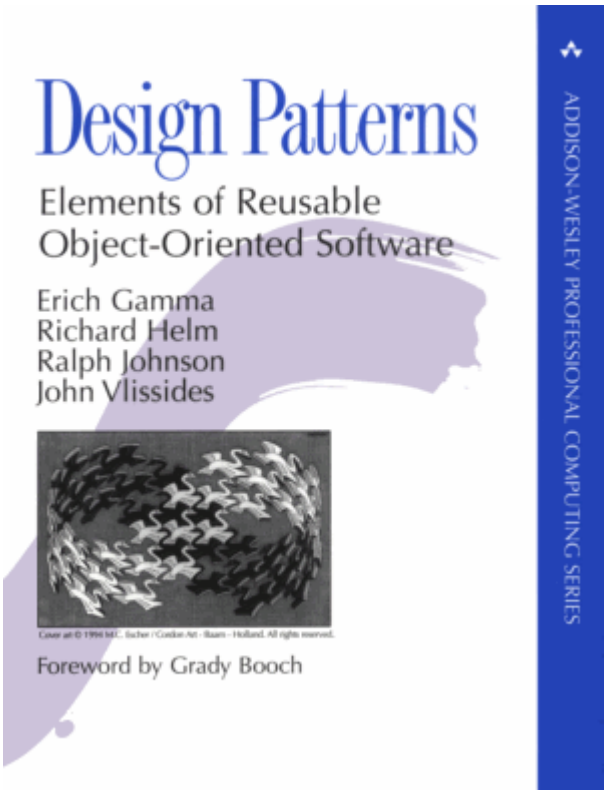
Basic UML

Taken from http://www.developer.com/design/article.php/2206791

Sr. No.	Relation	Symbol	Description
1	Association		When two classes are connected to each other in any way, an association relation is established. For example: A "student studies in a college" association can be shown as:
1 a.	Multiplicity		An example of this kind of association is many students belonging to the same college. Hence, the relation shows a star sign near the student class (one to many, many to many, and so forth kind of relations).
1 b.	Directed Association		Association between classes is bi-directional by default. You can define the flow of the association by using a directed association. The arrowhead identifies the container-contained relationship.
1 c.	Reflexive Association	No separate symbol. However, the relation will point back at the same class.	An example of this kind of relation is when a class has a variety of responsibilities. For example, an employee of a college can be a professor, a housekeeper, or an administrative assistant.
2	Aggregation		When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It

			is also called a "**has a**" relationship.
2 a.	Composition		Composition is a variation of the aggregation relationship. Composition connotes that a strong life cycle is associated between the classes.
3	Inheritance/Generalization		Also called an "**is a**" relationship, because the child class **is a** type of the parent class. Generalization is the basic type of relationship used to define reusable elements in the class diagram. Literally, the child classes "inherit" the common functionality defined in the parent class.
4	Realization		In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract.

Design Pattern books



Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

See Also

Martin Fowler (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley. ISBN 0-201-48567-2.

Joshua Kerievsky (2004). Refactoring To Patterns. Addison-Wesley. ISBN 0-321-21335-1.

Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates (2004). Head First Design Patterns. O'Reilly. ISBN 0-596-00712-4.

Books

- **Head First Design Patterns** | summary | amazon | sample chapter | downloads | your brain poster | head first labs
- **Pattern-Oriented Software Architecture** | amazon | Table of Intents | Layers | Vol II Wiley | Vol II amazon
- **Java Design Patterns: A Tutorial** | amazon | download the PDF file
- **Patterns in Java, Volume 1** | amazon | author's Web site
- **Design Patterns Explained** | amazon
- **Design Patterns Java Workbook** | amazon
- **Applied Java Patterns** | amazon
- **C# Design Patterns: A Tutorial** | amazon
- **The Design Patterns Smalltalk Companion** | amazon
- **Pattern Hatching: Design Patterns Applied** | amazon
- **The Patterns Handbook** | amazon
- **Core J2EE Patterns** | amazon | summary
- **Pattern Languages of Program Design 1** | amazon | Table of Contents
- **Pattern Languages of Program Design 2** | amazon | Table of Contents
- **Pattern Languages of Program Design 3** | amazon | Table of Contents | Extension Object | C++ demo
- **Pattern Languages of Program Design 4** | amazon | Table of Contents
- **Pattern Languages of Program Design 5**
- **Analysis Patterns** | amazon
- **AntiPatterns** | amazon | subset summary
- **San Francisco Design Patterns** | amazon | Table of Intents
- **Concurrent, Parallel, and Distributed Systems** | Doug Schmidt

Design Patterns Periodic Table

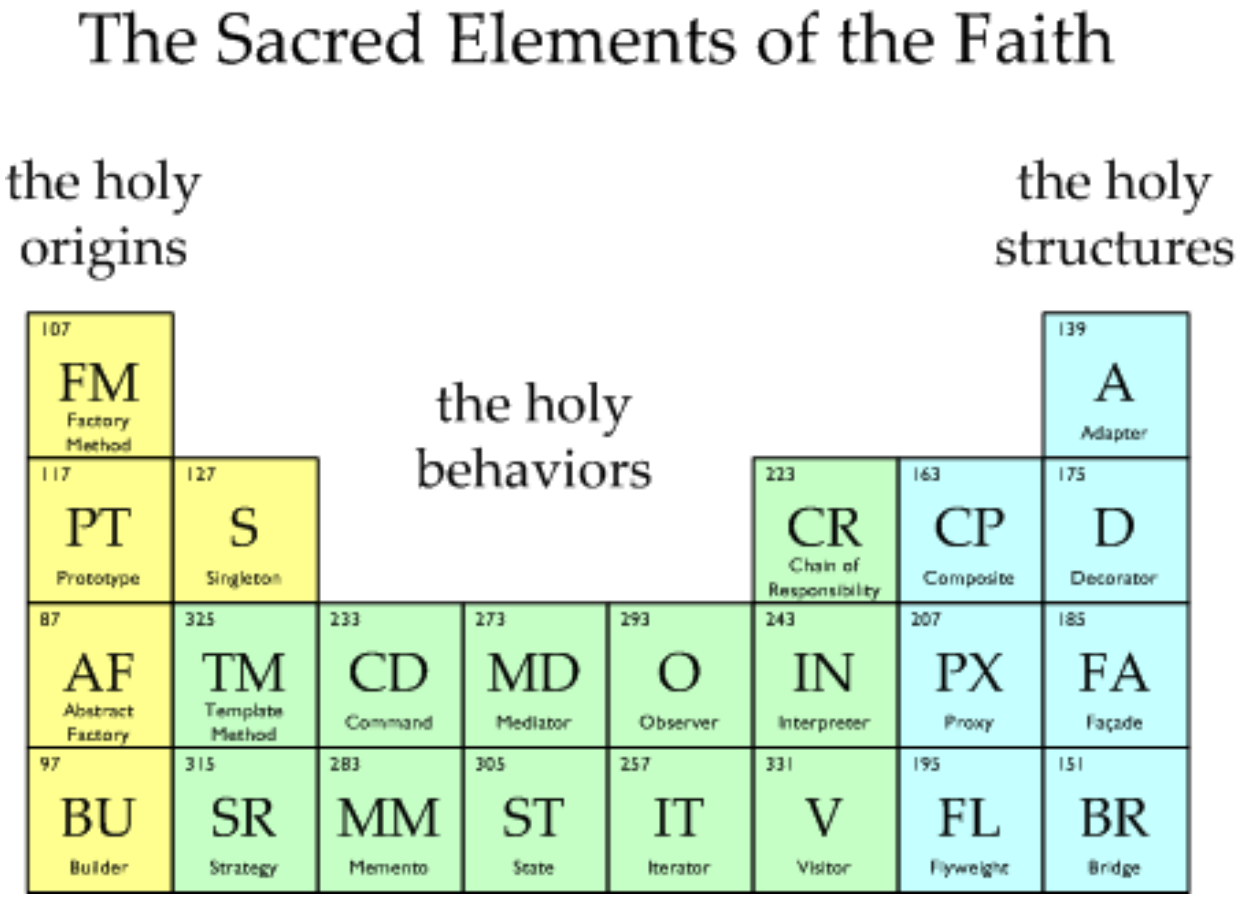


Diagram courtesy Huston Patterns

http://home.earthlink.net/~huston2/dp/patterns.html

Table of Figures