

Thinking in Patterns
Using Design Patterns to Maximize
Java's Object-Oriented Capabilities
by

Alan Shalloway



www.netobjectives.com

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

1

What We Are Going to Do

- This material is an excerpt from our Pattern Oriented Design courses
- Give a quick overview of where we are going tonight
- Talk about what design patterns are
- Learn a few design patterns
- Present a problem and see how we'd solve it using patterns to do the design
- Talk about where you can learn more about this approach

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

2

In the Old Days ...

- “Two things in life are certain -- death and taxes.”
-- Ben Franklin

- In the information age, there is a third ...

In the Information Age ...

- Three things in life are certain --
 - death
 - taxes
 - requirements will change

Pattern Oriented Design and Changing Requirements

- Pattern Oriented Design is about anticipating and handling changing requirements.
- It focuses on relationships more than specific entities and behaviors.
- Relationships vary more slowly than do our entities and the behaviors of these entities.
- Focusing on relationships allows us to disentangle the dependencies between things.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

5

Pattern Oriented Design Vs Object-Oriented Design

- *Is* object-oriented design.
- Integrates advanced techniques into its approach.
- Goes beyond merely finding the nouns in your functional specifications and hoping insight and experience will take over.
- Uses relationships between the entities in your problem domain to identify the objects you need to define.
- Is iterative, so as you define more relationships, object definition becomes easier.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

6

Pattern Oriented Design Vs OOD - cont'd

- Makes use of advanced design techniques espoused in Gang of Four Design Patterns book:
 - find what varies and encapsulate it
 - favor composition over inheritance
- However, use of design patterns makes this straightforward, not complicated.

Learning Pattern Oriented Design

- POD, is simple since it follows an intuitive approach.
- We must look at our design problem from a different perspective, however.
- It requires the same paradigmatic shift of looking at objects that object-oriented design requires.
- It also requires looking at relationships between these objects early in the design.

Three Levels of Perspective

- Conceptual Perspective
- Specification Perspective
- Implementation Perspective

- From Martin Fowler's UML Distilled

Conceptual Perspective

- Describes *what* you want
- *Not* how you'll get it
- Can be very detailed
- Example: want house with lots of bedrooms, should be light and airy, big kitchen

Specification Perspective

- We talk about *how* things behave
- What are the relationships between entities
- We have identified behaviors, but not implementations. That is, we have classes and interfaces, not code.
- For example: all bedrooms on top floor, kitchen overlooks garden, connected to dining room, ...

Implementation Perspective

- We have classes and how we are going to implement it.
- Methods (internal and external) are designed and laid out
- Language specific, actual modules, code
- For example: detailed blueprints from which you could build a house

Using Patterns to Learn Principles

- Throughout this process we will:
 - Review core object-oriented terms
 - Review some object-oriented design principles/strategies and see how patterns utilize them
- Practice has shown this to be the best way to learn OO concepts

What Are Patterns?

- “A pattern is a solution to a problem in a context”¹
- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”¹
- ¹ Christopher Alexander - A Pattern Language

The Courtyard Pattern - Christopher Alexander

In the same way, a courtyard which is properly formed, helps people come to life in it.

Consider the forces at work in a courtyard. Most fundamental of all, people seek some kind of private outdoor space, where they can sit under the sky, see the stars, enjoy the sun, perhaps plant flowers. This is obvious. But there are more subtle forces too. For instance, when a courtyard is too tightly enclosed, has no view out, people feel uncomfortable, and tend to stay away ... they need to see out into some larger and more distant space. Or again, people are creatures of habit. If they pass in and out of the courtyard, every day, in the course of their normal lives, the courtyard becomes familiar, a natural place to go ... and it is used. But a courtyard with only one way in, a place you only go when you "want" to go there, is an unfamiliar place, tends to stay unused ... people go more often to places which are familiar. Or again, there is a certain abruptness about suddenly stepping out, from the inside, directly to the outside ... it is subtle, but enough to inhibit you. If there is a

Courtyard cont'd

transitional space, a porch or a veranda, under cover, but open to the air, this is psychologically half way between indoors and outdoors, and makes it much easier, more simple, to take each of the smaller steps that brings you out into the courtyard ...

When a courtyard has a view out to a larger space, has crossing paths from different rooms, and has a veranda or a porch, these forces can resolve themselves. The view out makes it comfortable, the crossing paths help generate a sense of habit there, the porch makes it easier to go out more often ... and gradually the courtyard becomes a pleasant customary place to be.

What we've done:

Identified the pattern.

Discussed what we were trying to accomplish.

State how we could accomplish this.

Talked about the forces that would hurt what we were trying to accomplish.

Elements of Patterns

- **Pattern name.** Gives us a way to refer to the pattern. ²
- **The problem.** A particular pattern is applicable to certain types of problems. Part and parcel of a pattern is a description of what types of problems for which it is useful. ²
- **The solution.** Patterns define a particular conceptual solution to the problem. ²
- **Consequences.** Implementation decisions have certain tradeoffs. The consequences of these decisions and the forces underlying the pattern are essential aspects of the pattern. ²
- ² Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides.

Common Reasons to Study Patterns

- Re-uses existing, quality solutions
- Use common terminology
- *Shift to new level of thinking*

A Pattern in Carpentry

- Let's say two carpenter's are trying to decide on how to build a dresser.
- One says to the other: "should we build the joint by first cutting down, then cutting up at a 45 degree angle..."



- "then back down and back up the other way, then back down"



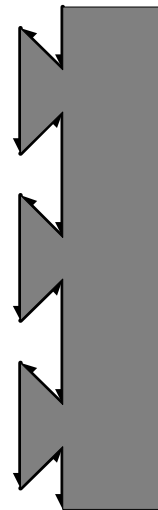
9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

19

We Are Describing a Dove-Tail Joint

- And keep this up until done.



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

20

Focusing on Detail Loses the Big Picture

- By having to describe how we implement the dove-tail joint, we can lose sight of why we might want to use it in the first place.
- Dove-tail joint Vs miter joint emphasizes:
 - do we want a strong, relatively expensive joint that is of high quality and will last forever
 - or do we want a weak, relatively inexpensive joint that is easy to make but not of high quality

Other Reasons to Study Patterns

- Common terminology assists learning across all levels
- Designs improved for modifiability (expand on good solutions)
- We will find that design is not a process of synthesis, but one of differentiation.
- Design patterns can be used to help us build our designs better than normal methods.

Core Object-Oriented Concepts

- Class
- Object
- Encapsulation *
- Inheritance *
- Polymorphism *
- Abstract classes
- Composition
- * Considered minimal concepts for object-oriented design

Classes and Objects

- Classes are our blueprints
- Objects are specific instances of our classes

Open - Closed Principle

- Ivar Jacobson said: “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.
- Bertrand Meyer summarized this as: *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
- In English this means: *design modules so that they never change. When requirements change, add new modules to handle things.*
- (good article on Open-Closed Principle at www.oma.com under publications)

Characteristics of Classes that Follow Open/Closed Principle

- **“Open for extension”** Behavior of modules can be extended as requirement change.
- **“Closed for modification”** The source code does not change.

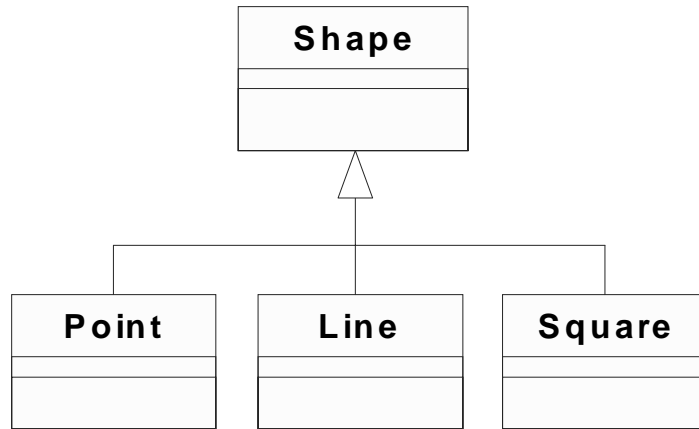
The Adapter Pattern

- 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. ³
- ³Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

Inheritance

- When have a specialized class, can inherit from another
- Only requires that you specify the differences
- The derived class is a kind of the base (super) class

Our Problem



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

29

This Is an Example of Polymorphism

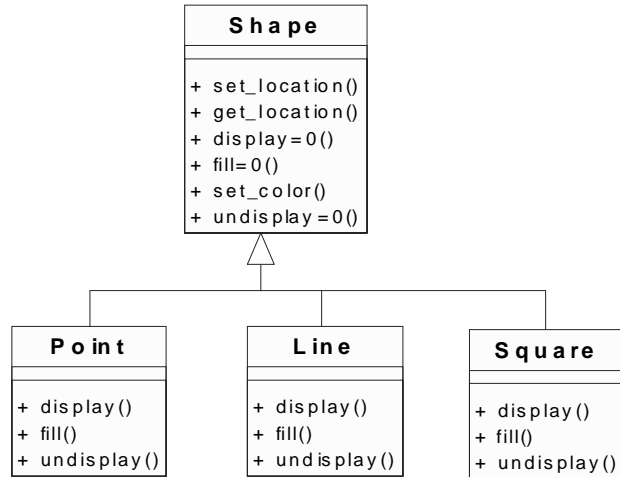
- Allows for different behavior
- Calling object does not need to know the exact type of object involved
- Calling object only needs to know the conceptual type of object
- In this case, anybody having *points*, *lines*, and *squares* only knows it has *shapes*.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

30

More Detail

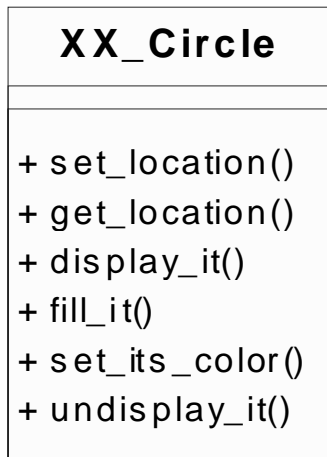


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

31

What We Have

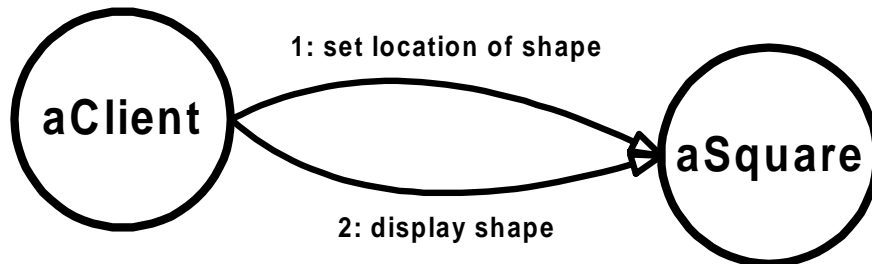


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

32

How Client Behaves With a Square



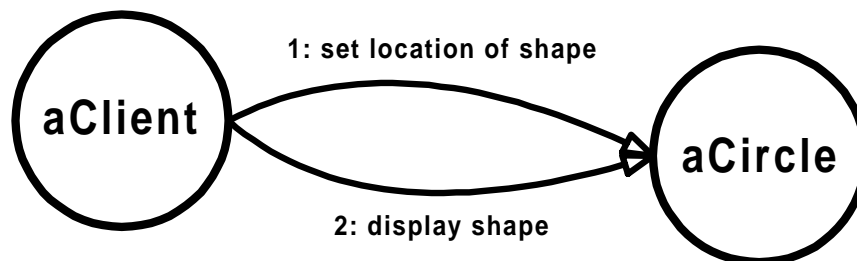
***aClient* works with *aSquare* but
only knows it's a *Shape***

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

33

What We Want



**want *aClient* to work with
aCircle in the same way**

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

34

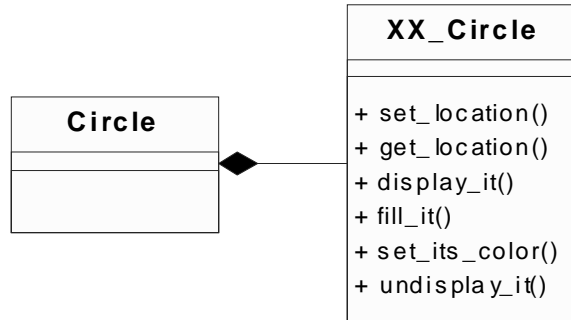
Our Problem

- Our client can't behave with our XX_Circle in the same way it does with the Square because XX_Circle is not a Shape
- Author of XX_Circle may not be willing or able to change XX_Circle's interface

Composition

- When one object 'contains' another object.
- Like an engine in a car.

How to Implement the Pattern



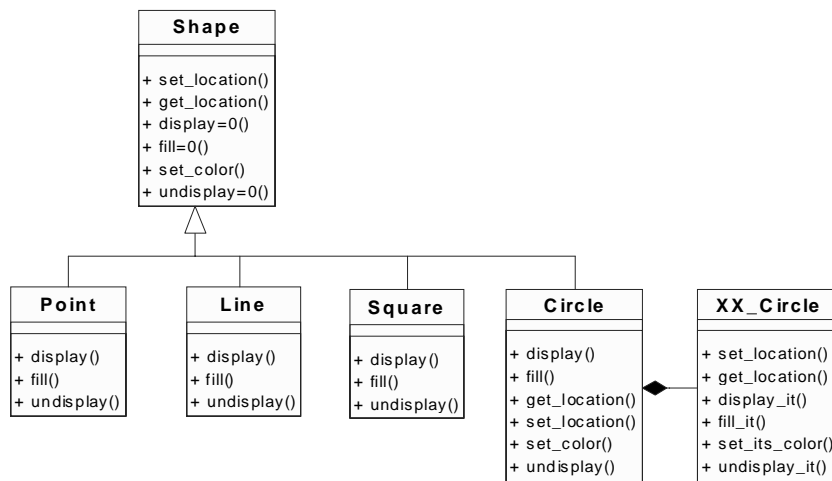
If circle contained an `XX_circle`, it could handle the 'communication' with the client and let `XX_circle` handle the work

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

37

The Pattern in its Context



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

38

A C++ Example

<p>IN HEADER class circle : public shape { ... private: XX_circle *pxc; ... }</p> <p>IN CONSTRUCTOR instantiate and initialize XX_circle point to it with pxc</p>	<p>IN CODE void circle::display () { pxc->display_it(); }</p>
---	---

A Java Example

```
class circle extends shape
{
  ...
  private XX_circle pxc;
  ...
  void display()
  {
    pxc.display_it();
  }
}
```

A Smalltalk Example

```
Shape subclass: #Circle
instance Variables: 'xCircle...'
class Variables: ' '
pools: ''

display
^xCircle display_it
```

How'd We Do Re Principles?

- Did we follow the open-closed principle?
- Specifically:
 - were we open for extension? (i.e., were we able to make the changes we wanted to?)
 - were we closed for modification? (i.e., did we avoid changing anything we had?)

Abstract Class

- A class that is never instantiated
- Used to define an interface for the real-world classes
- Represents a concept
- For example, in our case a 'shape' was an abstract class for 'points', 'lines', 'squares', and 'circles'.

9/8/99 v.2

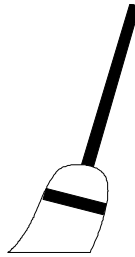
(1) Copyright © 1998-1999 Net Objectives

43

Are these the same??



Mop



Broom



Sponge

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

44

As Any Good Consultant Will Tell You: “IT DEPENDS!”

- As specific objects: they are different
- As a concept: they are all Cleaning Utensils

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

45

Where does “Cleaning Utensil” exist?



Not in the real world, but in our thoughts as an
abstraction classification!

A “Cleaning Utensil” does not exist, but specific kinds do!

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

46

Abstract Class

- A class that is never instantiated
- Used to define an interface for the real-world classes
- Represents a concept
- For example, a 'cleaning utensil' was an abstract class for 'mops, 'brooms', and 'sponges'.

This Is an Example of Polymorphism

- Allows for different behavior
- Calling object does not need to know the exact type of object involved
- Calling object only needs to know the conceptual type of object
- In our 'shape' example, anybody having *points*, *lines*, and *squares* only knows it has *shapes*.
- Both hides implementation details and allows for new types of implementations

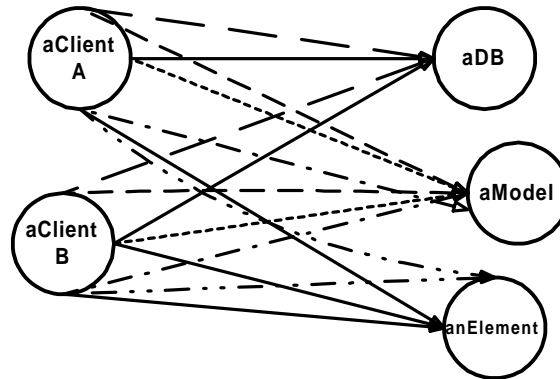
Encapsulation

- Often called data-hiding
- Can also hide behavior
- Means can't see what is being encapsulated

The Facade Pattern

- 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. ⁴
- ⁴ Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

Our Situation



**aDB, aModel, anElement are all needed to get the job done
Many different methods are used, often in the same way.**

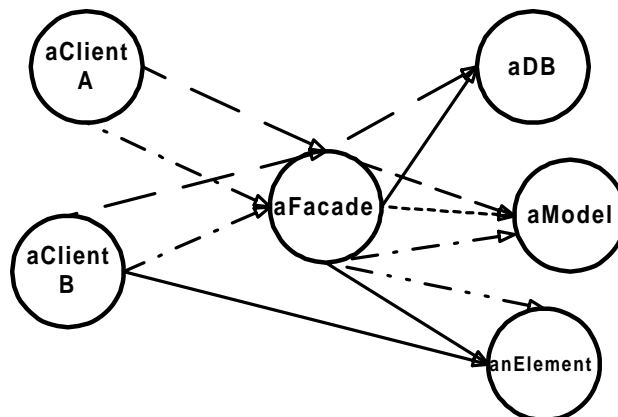
Our Problem

- Whoever writes the clients must learn pretty much the same things.
- Also, there is considerable duplication in the clients.
- We want to lower the learning curve and eliminate the duplication

Our Solution

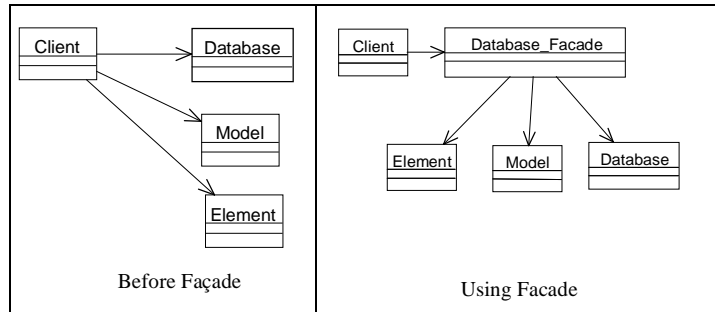
- Create a new class that both clients can use
- It handles the most common functions
- It allows clients to go directly to the underlying methods when needed

Our Solution



The Facade class creates a new interface for the commonly used classes that is easier to use.

Class Diagrams



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

55

Comparing Façade With Object Adapter

	Façade	Object Adapter
Have pre-existing classes	Yes	Yes
Have an interface we must design to	No	Yes
Have to make an object behave polymorphically	No	Yes
Want to make a new interface to simplify things	Yes	No

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

56

Commonality/Variability Analysis

- First find what is common across domain
- Identify where things vary
- Identify how things vary
- This is preparation for identifying potential classes

Find What Varies and Encapsulate It

- Identify varying behavior
- Define abstract class that allows for communicating with objects that have one case of this varying behavior

Favor Composition Over Inheritance

- If can define a class that encapsulates variation, contain (via composition) an instance of a concrete class of the abstract class defined earlier
- Allows for decoupling of concepts
- Allows for deferring decisions until runtime
- Small performance hit

Commonality and Variability Analysis

- “Commonality analysis is the search for common elements that helps us understand how family members are the same.”²
- Variability analysis determines patterns in how family members vary

- Jim Coplien - Multi-Paradigm DESIGN for C++

Commonality Provides the Base

- Variability only makes sense within a given commonality
- “From an architectural perspective, commonality analysis gives the architecture its longevity; variability analysis drives its fitness for use”

- Jim Coplien - Multi-Paradigm DESIGN for C++

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

61

How to Use Commonality and Variability Analysis

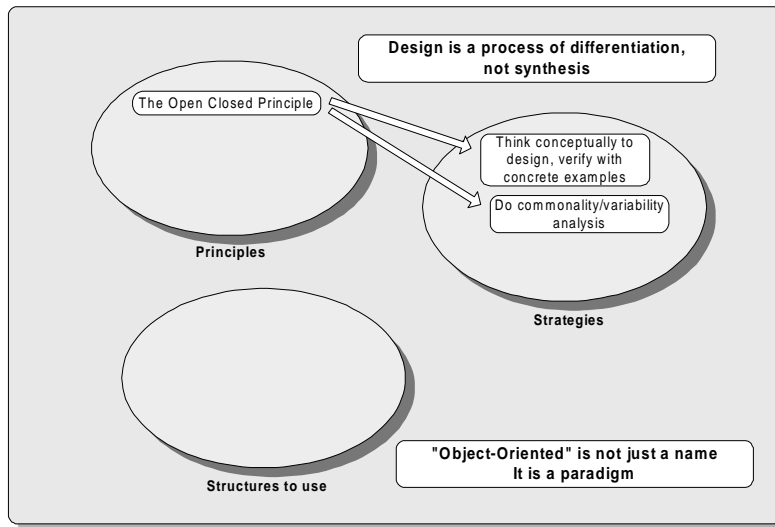
- Those things that are conceptually the same (found by commonality analysis) become the abstract classes
- The concrete implementations are defined by variability analysis
- Since inheritance doesn't work all the time - what does?
- Design patterns give us different structures to use

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

62

Relationship Between OCP and Commonality/Variability Analysis



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

63

The Strategy Pattern

- We are going to look at a new problem
- See some challenges we will have making 'fat' objects
- Learn the strategy pattern and how it can add flexibility

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

64

The Strategy Pattern - Cont'd

- Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. ³

- ³Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

The Differentiator of the Strategy Pattern

- We need different behavior at different times; either for different things we are working on or for different clients asking us to do work.

The Problem

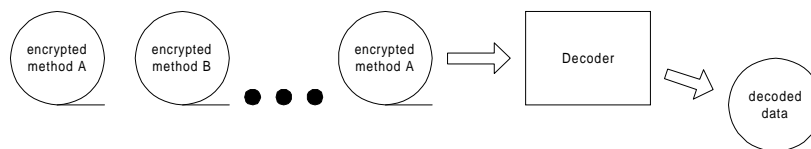
- We are writing a decoder.
- Streams of messages come to us with a header saying which code to use.
- The message following the header is encoded in that method.
- We have a decoder object that takes in the encoded stream and outputs a decoded stream.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

67

The Problem Illustrated



The encryption methods for each data packet are independent from each other.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

68

One Solution

```
DECODER OBJECT

method DO_DECODE

    // FOR EACH MESSAGE:
    //   read message header (GO TO END IF NO
    //   see type of encoding used
    //   get rest of message
    //   use switch on type of encoding used
    //     TYPE A: decode with decoder A; break
    //     TYPE B: decode with decoder B; break
    //     ...
    //     TYPE Z: decode with decoder Z; break
    //   output decoded message
```

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

69

The Problem

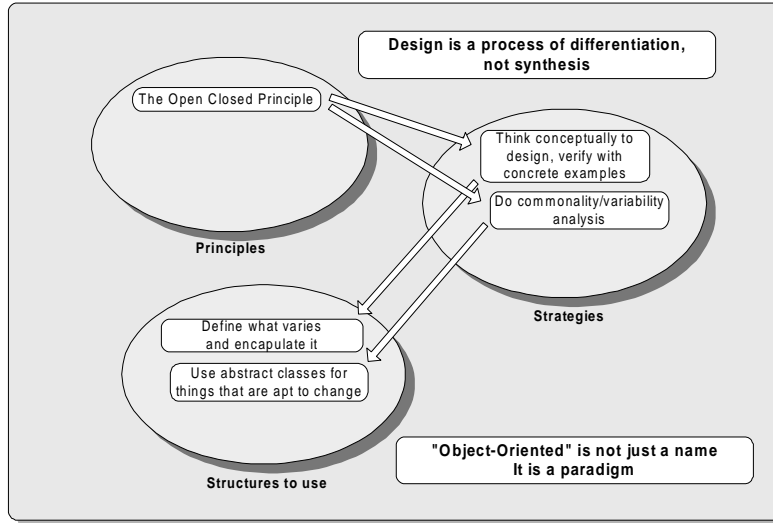
- This works fine, but has the problem of not being extensible and following the Open-Closed Principle.
- As soon as we get a new encoding/decoding type we must change our switch statement.
- It also has the problem of possibly creating a large object if it is responsible for more than one thing
- For example, maybe the decoding object is also responsible for routing of the message.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

70

Following Open Closed Principle Can Suggest Strategies



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

71

A Note About The Roadmap

- It is not complete
- New principles, strategies, structures can easily be added
- For example, you might also use abstract classes to encapsulate a rule (since the implementation of the rule may change)

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

72

Following OCP When Inheritance Doesn't Work

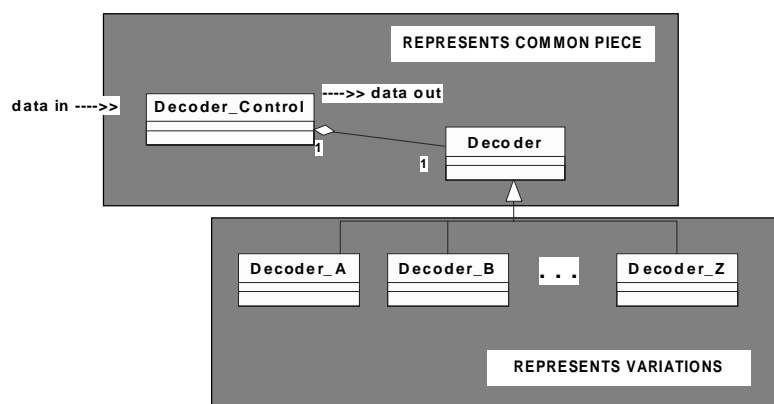
- Inheritance doesn't work well here because we need to have one object that handles all encoding.
- If we follow the strategy of finding what varies and encapsulating it, we realize we should encapsulate the different decoding strategies.
- Our decoder control object can use these decoding objects.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

73

Encapsulating the Encoder Variation



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

74

Not Quite Done

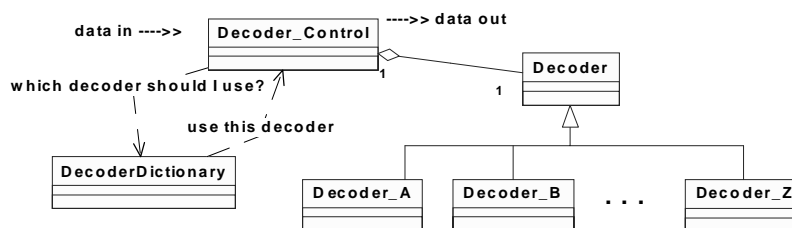
- We still have the problem of how does the Decoder_Control object know which decoder to use.
- There are several solutions to this.
- We will show two.
- Clearly someone must know which decoder to use.
- This is a differentiator, and something to encapsulate.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

75

Class Diagram With Factory Object



Our Decoder_Control object gives the header to the DecoderDictionary which gives back the decoder to use.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

76

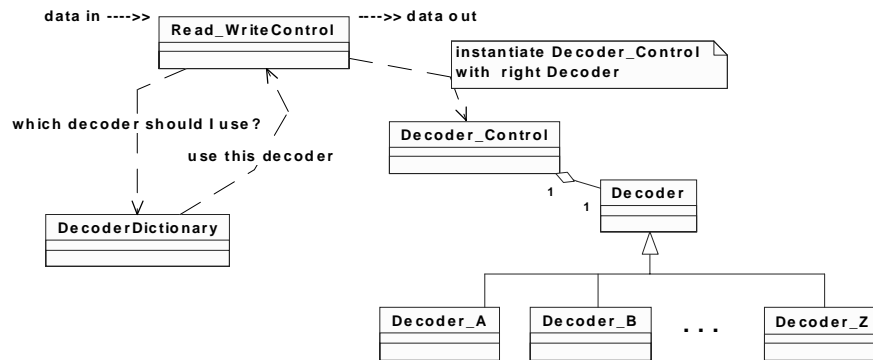
What Happens Now With Changing Requirements

- If we get a new decoder/encoder requirement, we only need to
 - derive the new decoder from the Decoder class
 - modify the DecoderDictionary
 - this may, in fact, work off a data dictionary so even it doesn't need to change
- No other objects change

Putting the Knowledge in the Client

- We can also add the knowledge of what to use in the client object.
- Let's reorganize our class diagram to show a little more detail.

Client Control of Decoder to Use



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

79

Value of Strategy

- Conceptually it is cleaner
- Each object worries only about one function (increasing cohesion)
- Better independence of different actions (decreasing coupling)
- If have more variations, have a better way of handling them (e.g., routing of message)

9/8/99 v.2

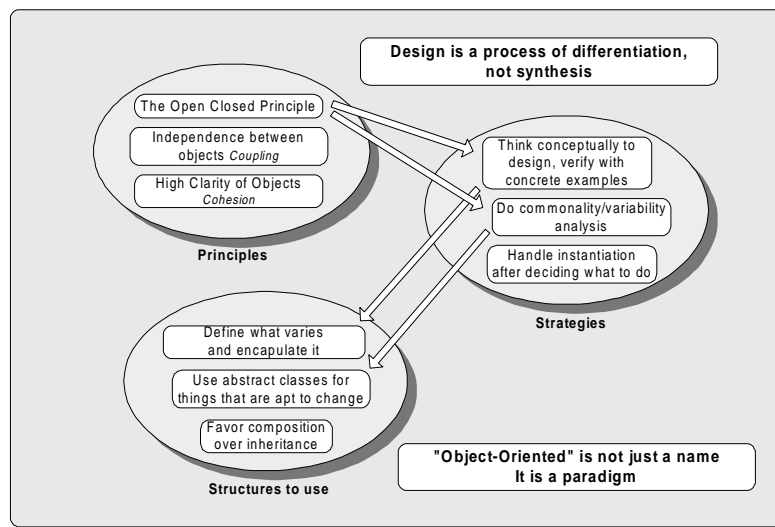
(1) Copyright © 1998-1999 Net Objectives

80

A Note About Encapsulation

- We often think of encapsulation as something we do on an object-by-object level.
- On reflection, however, we can see that we have used encapsulation to hide a set of classes (all the derived classes).
- Abstract classes essentially encapsulate all of their derivatives -- no one need know they exist.

Our Full Principles-Strategies-Structures Road Map



The Observer Pattern

- Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. ³
- Also known as publish-subscribe

- ³Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

The Differentiator for the Observer Pattern

- An event occurs and several objects need to know about it.

Situation

- We are dealing with monitoring devices
- These devices need to be able to 'inform' other devices of certain events occurring
- We know new devices will need to be developed in the future
- How can we design an architecture which will allow for a consistent way for the measuring device to know who to communicate with?

Problem

- We need a consistent, loosely coupled way of binding the object that triggers an event, with the objects that must be notified when the event occurs
- Tight binding will cause problems and violate the open-closed principle

The Solution

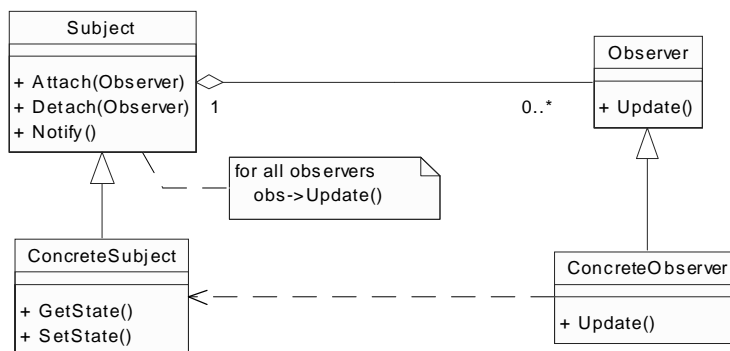
- Have a consistent method for observers (the objects to get notified) to register (and unregister) with the subject (the object causing an event to trigger or measuring an event that is triggered).
- Have a consistent method for the subjects to notify the observers

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

87

The Pattern



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

88

Essence of Observer

- The 'pattern' could conceptually be said to be:
 - observers register with the subject
 - subject looks for or is triggered by events
 - subject notifies observer

Many Variations of Communication

- There are many ways communication of the information between the subject and the observer could be done
 - Subject could pass on information
 - Subject could just let observer know that it changed and have observer poll it
 - Observer may request all information or just limited information

Not All Observers Are the Same

- Remember, the pattern is just a starting point
- It is just a guide
- Different observers may want different information
 - for example, in an accounting package, it may be that at the end of the day, all new accounts are to be sent to different objects for different reasons. These could be reports, welcome letters, auditing, ...
 - each of these different areas might want different information about the same, newly added customers
 - how to send the information is therefore dependent upon the problem domain

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

91

Other Uses of Observer

- The observer pattern is at the heart of the Model View Controller Pattern
- Let's look at many of the problems associated with GUIs

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

92

Design According To Christopher Alexander

- Design is often thought of as being a process of synthesis
- The best designs cannot be made this way
- The best designs are a process of differentiation

Excerpt from The Timeless Way of Building by Christopher Alexander (italics his).

Differentiating Space

- More from Alexander
- *... every individual act of building is a process in which space gets differentiated. It is not a process of addition, in which pre-formed parts are combined to create a whole: but a process of unfolding, like the evolution of an embryo, in which the whole precedes its parts, and actually gives birth to them, by splitting.*
- *Each part is slightly different, according to its position in the whole.*
- *Design is often thought of as a process of synthesis, a process of putting together things, a process of combination.*

Differentiating Space - Cont'd

- *According to this view, a whole is created by putting together parts. The parts come first: and the form of the whole comes second.*
- *But it is impossible to form anything which has the character of nature by adding preformed parts.*
- *When parts are modular and made before the whole, by definition then, they are identical, and it is impossible for every part to be unique, according to its position in the whole.*
- *It is only possible to make a place which is alive by a process in which each part is modified by its position in the whole.*

Differentiating Space - Cont'd

- *In short, each part is given its specific form by its existence in the context of the larger whole.*
- *This is a differentiating process.*
- *It views design as a sequence of acts of complexification; structure is injected into the whole by operating on the whole and crinkling it, not by adding little parts to one another. In the process of differentiation, the whole gives birth to its parts: the parts appear as folds in a cloth of three dimensional space which is gradually crinkled. The form of the whole, and the parts, come into being simultaneously.*
- *The image of the differentiating process is the growth of an embryo.*

Differentiating Space - Cont'd

- The unfolding of a design in the mind of its creator, under the influence of language, is just the same.
- Each pattern is an operator which differentiates space: that is, it creates distinctions where no distinction was before.
- And in the language the operations are arranged in sequence: so that, as they are done, one after another, gradually a complete thing is born, general in the sense that it shared its patterns with other comparable things; specific in the sense that it is unique, according to its circumstances.
- The language is a sequence of these operators, in which each one further differentiates the image which is the product of the previous differentiations.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

97

Pattern Oriented Design

- Pattern oriented design is a process in which we can create our application's design based on the entities and design patterns present in our problem domain.
- Pattern oriented design focuses on identifying relationships in our problem domain and using these relationships to further identify areas of potential changing requirements.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

98

Relationships and Design Patterns

- Many of these relationships can be described in terms of design patterns.
- At a local level, this enables quicker implementation with greater robustness than creating implementations ourselves.
- However, by initially focusing on relationships, instead of classes, we can separate complex relationships into their sub-relationships until we can describe our problem domain in a highly cohesive, loosely coupled manner.

Going From Relationships to Design

- According to Christopher Alexander, these relationships are relatively constant -- it is the entities that are embedded in the relationships that are changing.
- Alexander states that we should not design our entities until after we can see what the context is in which they will exist.
- By focusing on the relationships between the entities and not the relationships themselves, we can create this context.

From Context to Classes

- Once we have discovered the global structure for our application, we can define the local (class) structure.
- When patterns are present, our job is easy.
- Even when they are not, we have at least localized the problem we have to solve.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

101

Pattern Oriented Design: The Steps to Follow

- Identify the entities in our problem domain.
- Identify any relationships that are readily apparent.
- Do not worry about how to implement anything at this point.
- Identify behaviors that are required.
- If these behaviors belong to one entity, mark them as belonging to that entity.
- Often these behaviors will be a relationship between entities, if so, mark it as a relationship.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

102

Do This Visually

- Use a visual way to represent these. That is, lay out the entities and the relationships in a 2 dimensional diagram.
- Circle entities and draw lines between the entities of a relationship -- labeling the line with a description of the relationship.

Iterative Approach

- As you draw out this model, keep going back to your functional requirements and see if you have left anything out.
- Also, look at what you are drawing and see if any new relationships or entities become apparent.
- Sometimes behavior that appeared to belong to a entity will now appear to be a relationship between two entities. If so, update your drawing to illustrate that.

Stay At a High Level

- Stay at this high, conceptual, level until all entities and relationships have been put down.
- Your target is to get all:
 - entities down to simple, single item, entities
 - relationships down to simple, single-dimensional, relationships
- Keep update your diagram until you get things as orthogonal as possible. ***Do not worry about figuring out how to implement anything at this point. Doing so will be counter-productive.***

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

105

Once Orthogonality Is Achieved

- Once you have gotten things as orthogonal as possible, then you can go further.
- You should discover that several of the relationships sound like design patterns.
- Identify those that are.
- For those that are not design patterns, be extra careful that you are fully, and concisely, describing the relationship present.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

106

Going to Specification

- We are now ready to define our classes.
- However, we will do this in a broad, shallow approach as we just laid out our entities and design. That is, don't go any further than necessary. We are not specifying implementations yet, just specifications.

Getting to Implementation

- Once all of our classes are specified, we can finally get to implementation.
- First step is physical design -- precisely and completely describing our methods (in particular, method names and parameter lists).
- Once all physical design is done, we can do implementation of the methods.

Our Problem

- We are writing an inventory control system.
- Each of our customers can use one of 3 inventory costing algorithms.
- When we update our inventory, we must also immediately update a complex financial management system.
- Additionally, some of our customers handle sensitive or dangerous items. Whenever one of these items is affected we must notify certain government agencies with the inventory update information.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

109

Summary of Steps

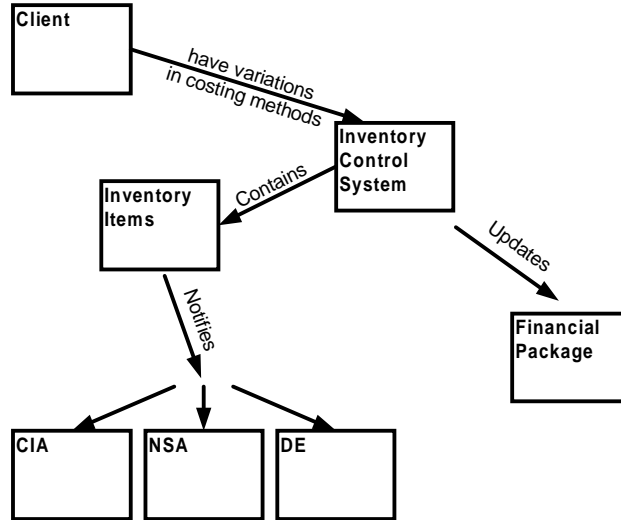
- 1) Identify the entities, behaviors and relationships in the problem domain by drawing them in a 2 dimensional diagram
 - 2) Associate behaviors with entities.
 - 3) Associate relationships with entities.
 - 4) Verify completeness by referring to functional specs
- Do not go down a level conceptually, until entire model is complete at higher level.***
- 5) Identify design patterns where present.
 - 6) Clarify relationships that aren't design patterns.
 - 7) Use design patterns and relationships to specify classes.
 - 8) Do implementation.

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

110

Steps 1-4: Identify Entities, Behaviors and Relationships in the Problem Domain

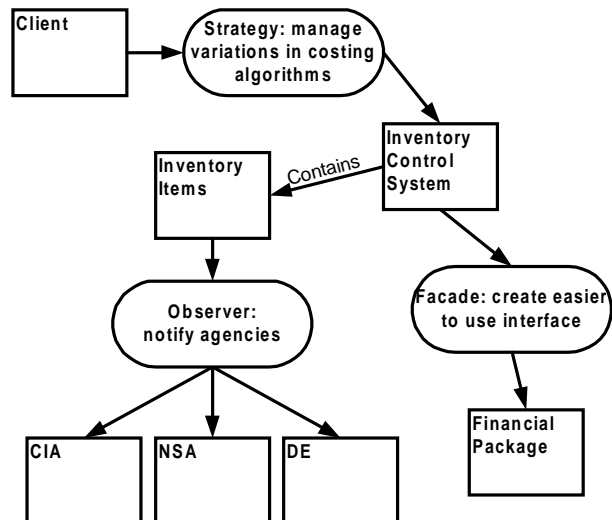


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

111

Step 5: Identify design patterns where present

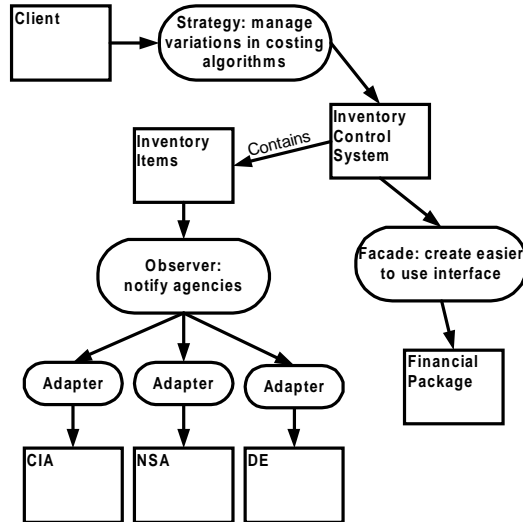


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

112

Expanding Relationships/Patterns

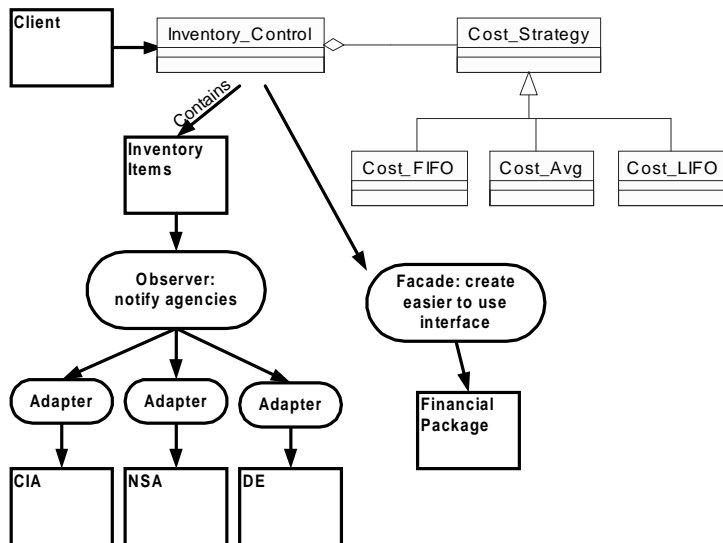


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

113

Defining Classes: Step By Step

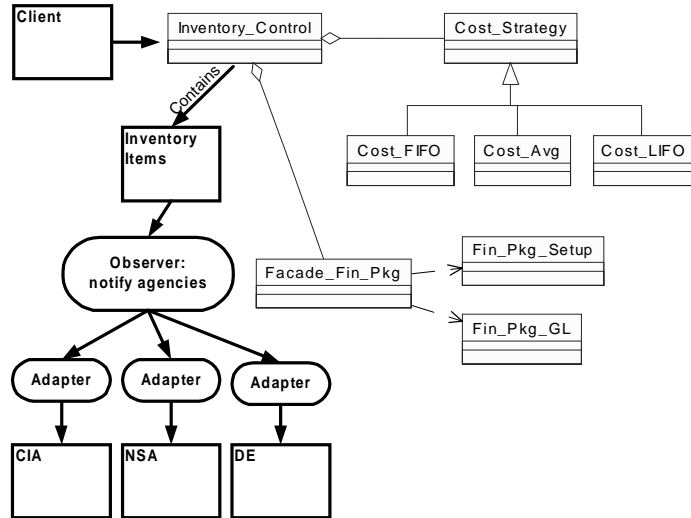


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

114

Continuing Specification

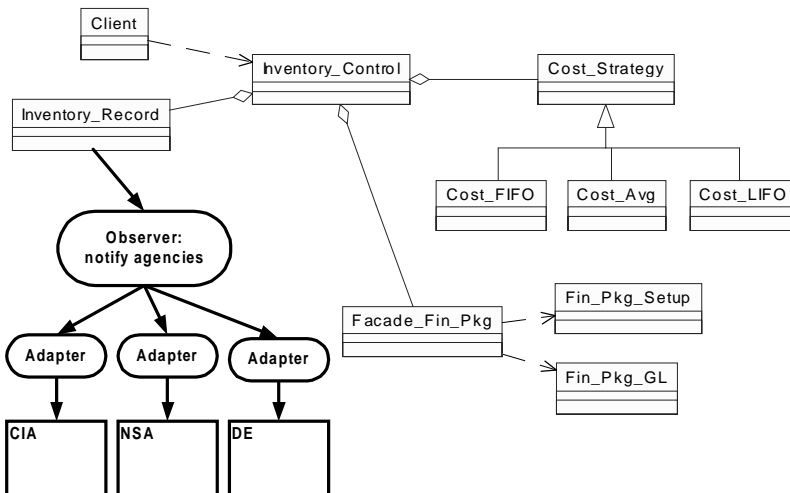


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

115

Continuing Specification

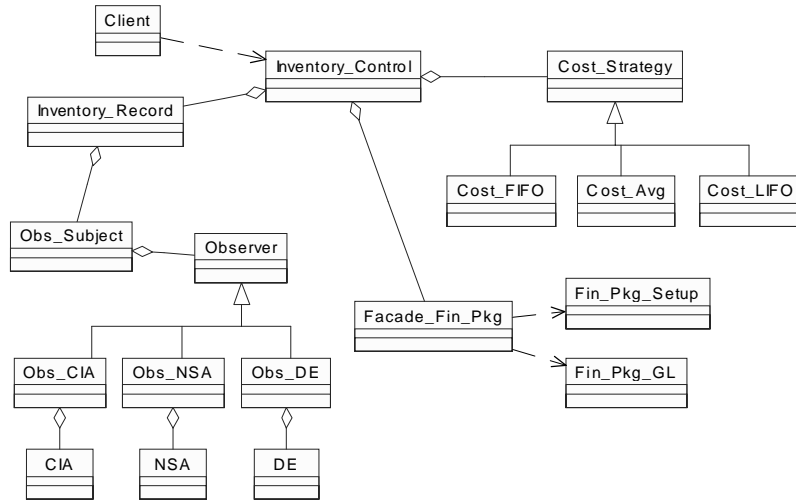


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

116

Continuing Specification

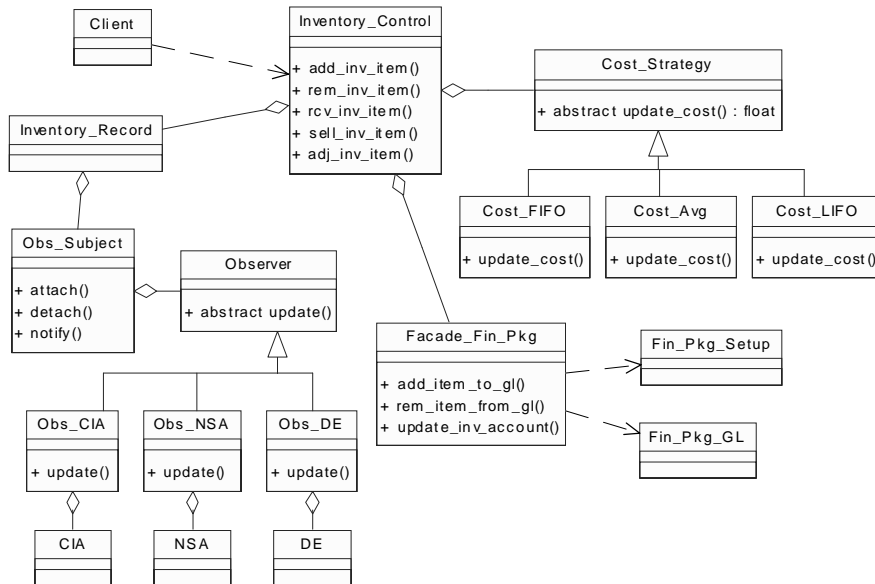


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

117

Finishing Specification



9/8/99 v.2

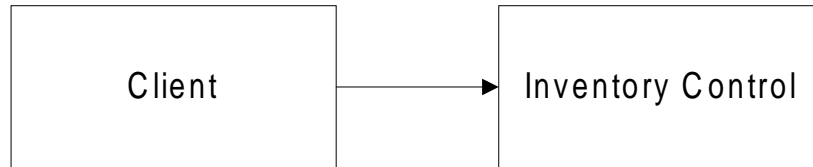
(1) Copyright © 1998-1999 Net Objectives

118

Alternative Way

Start with patterns.

Put down highest conceptual pieces and see what patterns are present.

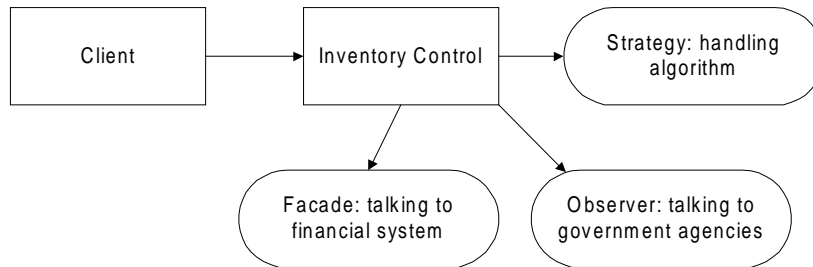


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

119

Add the Patterns

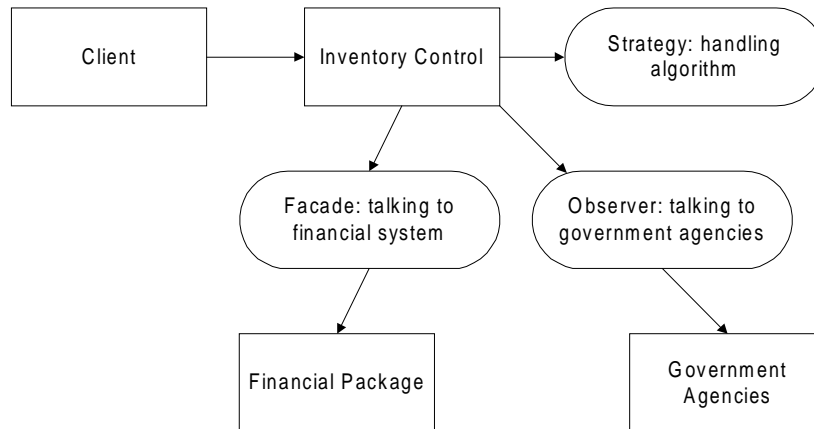


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

120

Add the Rest of the Entities

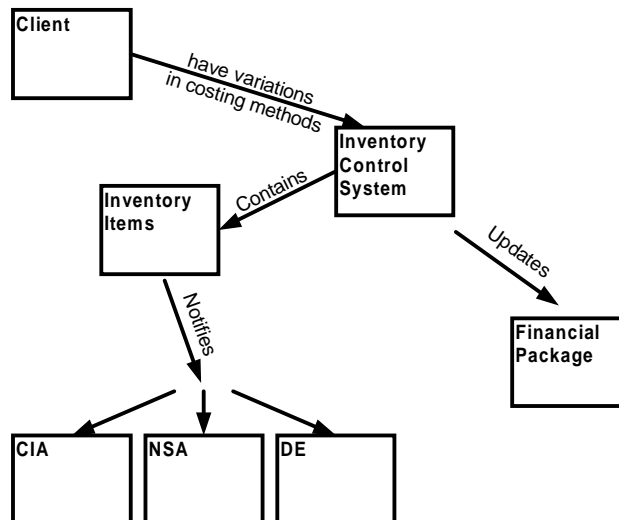


9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

121

Compare to Original



9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

122

How This Applies to Java

- Favoring composition over inheritance means containing 'polymorphic' structures
OR
- Using interfaces

- Use abstract classes when things have some common implementation
- Use interfaces when they do not

(sorry, this slide is not in your notes)

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

123

How To Learn More

- Net Objectives offers training in Pattern Oriented Design both for those new to object-oriented technology and those experienced in OO.
- Training also available in OOA, Java and C++.
- Mentoring and consulting can be provided.
- On-site courses available.

- www.netobjectives.com
- alshall@netobjectives.com -- 425-260-8754
- subscribe to e-zine (info@netobjectives.com with 'subscribe' in subject line) to get soft copy of talk

9/8/99 v.2

(1) Copyright © 1998-1999 Net Objectives

124