

Refactoring, Design Patterns and Extreme Programming



www.netobjectives.com
info@netobjectives.com 425-313-3065

5/1/02

Copyright © 2002 Net Objectives

1

Why We Do This

- Adds value to the community we are a part of.
- We believe people should know what they are getting
 - tonight is a free taste
 - gives you a chance to see our values, personalities, skills
- It's more fun than marketing
- This is a technical session up until the last 10 minutes.
- What we ask:
 - if you stay past the break, please stay to the end
 - fill out our evaluation
 - let us know if you want to work with us on improving your skills or that of your team

5/1/02

Copyright © 2002 Net Objectives

2

Intent of Seminar

- Participants:
 - understand refactoring
 - improve their understanding of object-oriented design
 - make better decisions about how to structure their code so it is better the first time
 - see how two design patterns are implemented
 - see how using good locally-verified rules can lead to good global designs
 - explore the relationships between refactoring, design patterns and extreme programming

5/1/02

Copyright © 2002 Net Objectives

3

Outline of Seminar

- Maintainability
- Cohesion/Coupling/Refactoring
- Rules for writing quality code.
- Using nouns and verbs vs Emergent Design
- Case study - one story at a time
- Strategy Pattern
 - brief description
 - case study
- Bridge Pattern
 - brief description
 - review of Commonality/Variability Analysis
- Appendix

5/1/02

Copyright © 2002 Net Objectives

4

What Would It Take to Have Easily Maintainable Code?

- Code must be clear.
 - can see what individual things do
 - can see how individual things react
- Can change code in one place without it adversely (or unknowingly) changing behavior in another place.
- When need to make a change, only need to change it in one place.
- In other words: clear, independent, efficient

5/1/02

Copyright © 2002 Net Objectives

5

Cohesion

- Cohesion refers to how “closely the operations in a routine are related.” I have heard other people refer to cohesion as “clarity” because the more operations are related in a routine (or class) the easier it is to understand the code and what it's intended to do.
 - Steve McConnell, Code Complete, 1993, pg 81 (note: McConnell did not invent these terms, we just happen to like his definitions of them best)

5/1/02

Copyright © 2002 Net Objectives

6

Coupling

- Coupling refers “to the strength of a connection between two routines. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a routine are related to each other. Coupling describes how strongly a routine is related to other routines. The goal is to create routines with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines (loose coupling).”
- Tight coupling is related to highly interconnected code.
- Steve McConnell, Code Complete, 1993, pg 81 (this concept was first described by Larry Constantine in 1975, but we like McConnell’s definition best)

5/1/02

Copyright © 2002 Net Objectives

7

Refactoring

- “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”
- Now that we know how our code is to be, refactoring gives us a way to get there if it isn’t.
- Martin Fowler- Refactoring: Improving the Design of Existing Code, AWL, 1999

5/1/02

Copyright © 2002 Net Objectives

8

Types of Refactoring

Refactoring Bad Code

- Code "smells"
- Refactor to improve code quality
- A way to clean up code without fear of breaking the system

Refactoring Good Code

- Code is tight, because we followed good rules in the first place
- A new requirement means code needs to be changed
- A way to make this change without fear of breaking the system

5/1/02

Copyright © 2002 Net Objectives

9

Forces In Design

- Cohesion:
 - Put different behavior in different methods.
 - Mixing different behavior in the same method lowers cohesion.
- Coupling:
 - design to interfaces
 - use top-down design (design by intention) when defining methods
- Redundancy:
 - eliminating redundancy increases efficiency
- Refactoring is a way to achieve these when we don't have them.

5/1/02

Copyright © 2002 Net Objectives

10

Rules For Writing Good Code

CODE MUST (stated in order of importance)

1. Run all the tests
2. Contain no duplication (once and only once)
3. Loose coupling, high cohesion and clarity
 - When writing, act as if needed methods already exist - that is, just refer to them and implement them later. This keeps all implementation of a method at the same conceptual level.
 - Methods and classes should be implemented so they can be understood totally from their public interfaces. This not only allows for up-front testing, but decreases coupling.
 - Only put the implementation of related ideas in the same method.
 - Classes should organize ideas in a readily understandable way.
 - Use appropriate names so you don't have to explain method, member or class names with additional documentation
4. Minimize classes and methods. This is actually redundant, but is a reminder that we are trying to make our code simple and concise

5/1/02

Copyright © 2002 Net Objectives

11

Sometimes Not Enough

- We now know something about what we want to achieve, but that is not necessarily enough to know how to do it. (kind of like “buy low, sell high” in the market).
- Some things will be self-evident, some will not.
- Let's proceed until we get stuck, then we'll look at some strategies to follow.

5/1/02

Copyright © 2002 Net Objectives

12

What Are Patterns?

- Patterns are best-practice solutions for recurring problems in a particular context.
- Patterns have been classified as being:*
 - Architectural
 - Design
 - Idiomatic
- Design patterns can also be thought of as describing the relationships between the entities (objects) in our problem domain.
- Patterns have led to new modeling techniques:
 - handling variations in behavior
 - new ways of using inheritance to avoid rigidity and aid testing and maintenance issues.

* Pattern Oriented Software Architecture, Buschmann, et. al.

5/1/02

Copyright © 2002 Net Objectives

13

Different Approaches to Design

- Nouns and Verbs - problematic. Requires specialization as things change.
- Thinking in Patterns - works only when know a lot of patterns.
- Commonality / Variability Analysis - an alternative to Nouns and Verbs. Gives a better way of decomposing problem domain. Works both initially and as requirements change.
- Emergent design - rely on good coding principles and design will emerge.

5/1/02

Copyright © 2002 Net Objectives

14

CASE Study

- We're going to work a case study, but do it a requirement at a time.
- We will state all of the requirements up front, as each of these would be represented by stories in a real project.
- XP (extreme programming) tells us that we can implement one story at a time and get a good result.
- We'll take this approach and refactor as new requirements come along.

5/1/02

Copyright © 2002 Net Objectives

15

Complete Requirements

- We have to monitor both chips and cards. We want to write a program that can request the status of both of these types of hardware and then sends that status over either a TCP/IP connection or via e-mail.
- These messages may be optionally encrypted with either PGP64 bit encryption or PGP128 bit encryption.
- When sending status out for a card, we want to queue the information to send it out no more than every 10 minutes unless there is an error. Chips, on the other hand, send immediately.
- A configuration file will contain information about which transmission method to use.

5/1/02

Copyright © 2002 Net Objectives

16

Using Nouns And Verbs

- Nouns and verbs has us focus on the things we have. We handle different behaviors through the use of polymorphism.
- Can result in tall class hierarchies.

5/1/02

Copyright © 2002 Net Objectives

17

Problem Domain Nouns and Verbs - Ignoring Encryption Issue

- Chips and Cards
 - Chips sending with TCP/IP
 - Chips sending with SMTP
 - Cards sending with TCP/IP
 - Cards sending with SMTP
- We end up making four classes for each of the above to get polymorphism.

5/1/02

Copyright © 2002 Net Objectives

18

Have a Class for Each Type of Hardware (Textual)

Class Hardware

Method: getAnd SendStatus()

Class Chip

Method: getAnd SendStatus()

Class Card

Method: getAnd SendStatus()

Class ChipTCPIP

Methods:

getAnd SendStatus()

sendWithTcpi()

Class CardTCPIP

Methods:

getAnd SendStatus()

sendWithTcpi()

Class ChipSMTP

Methods:

getAndSendStatus()

SendWithSMTP

Class CardSMTP

Methods:

getAndSendStatus()

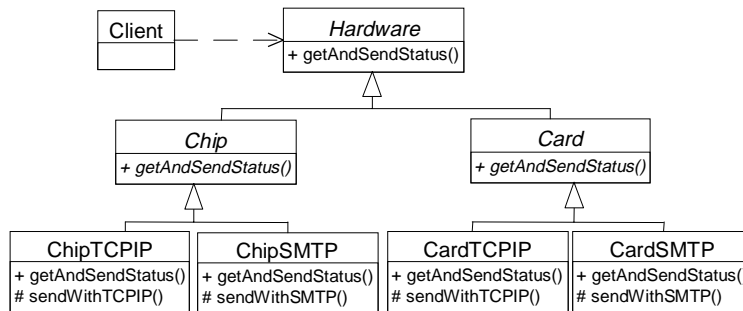
SendWithSMTP

5/1/02

Copyright © 2002 Net Objectives

19

Have a Class for Each Type of Hardware (UML)



+ = public # = protected

5/1/02

Copyright © 2002 Net Objectives

20

Relating the Diagram to Code

Java

```
class Client{
    private Hardware myHardware;
}

abstract class Hardware{}
abstract class Chip extends
    Hardware{}
abstract class Card extends
    Hardware{}
class ChipTCPIP extends Chip{}
class ChipSMTP extends Chip{}
class CardTCPIP extends Card{}
class CardSMTP extends Card{}
```

C++

```
class Client{
    Hardware *myHardware;
}

class Hardware{}
class Chip : public Hardware{}
class Card : public Hardware{}
class ChipTCPIP : public Chip{}
class ChipSMTP : public Chip{}
class CardTCPIP : public Card{}
class CardSMTP : public Card{}
```

5/1/02

Copyright © 2002 Net Objectives

21

Accommodating Change With Specialization

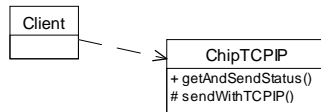
- Let's walk through a potential way our problem could evolve. We start with Chip and TCP/IP and add function one step at a time. We accommodate this through specialization.
- The result is not pretty (except as in pretty common).

5/1/02

Copyright © 2002 Net Objectives

22

Start With Chip Using TCPIP

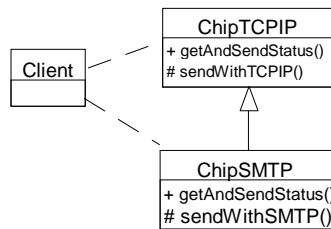


5/1/02

Copyright © 2002 Net Objectives

23

Add SMTP Capability



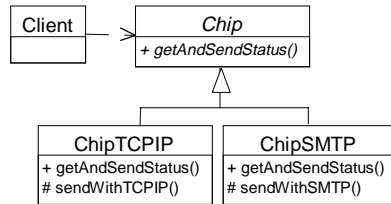
We could just derive a new Chip that uses SMTP, but ...

5/1/02

Copyright © 2002 Net Objectives

24

Add SMTP Capability



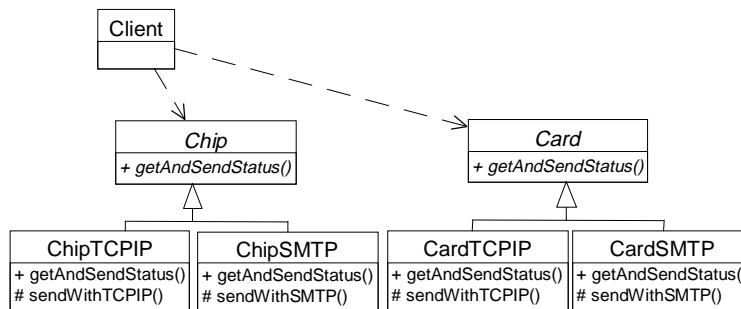
We could just derive a new Chip that uses SMTP, but... we've learned it's better to have an interface or an abstract class.

5/1/02

Copyright © 2002 Net Objectives

25

Add Cards

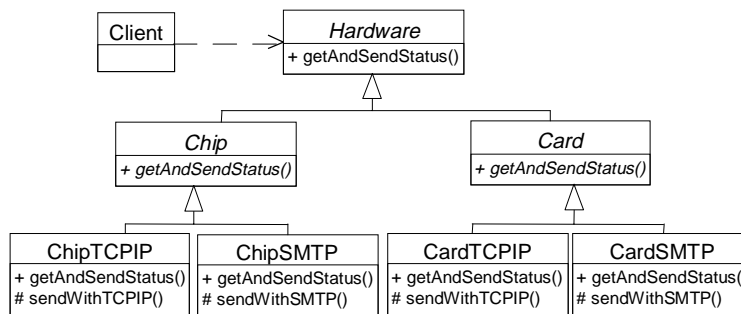


5/1/02

Copyright © 2002 Net Objectives

26

Create Hardware Class For Polymorphism



5/1/02

Copyright © 2002 Net Objectives

27

Problems With This

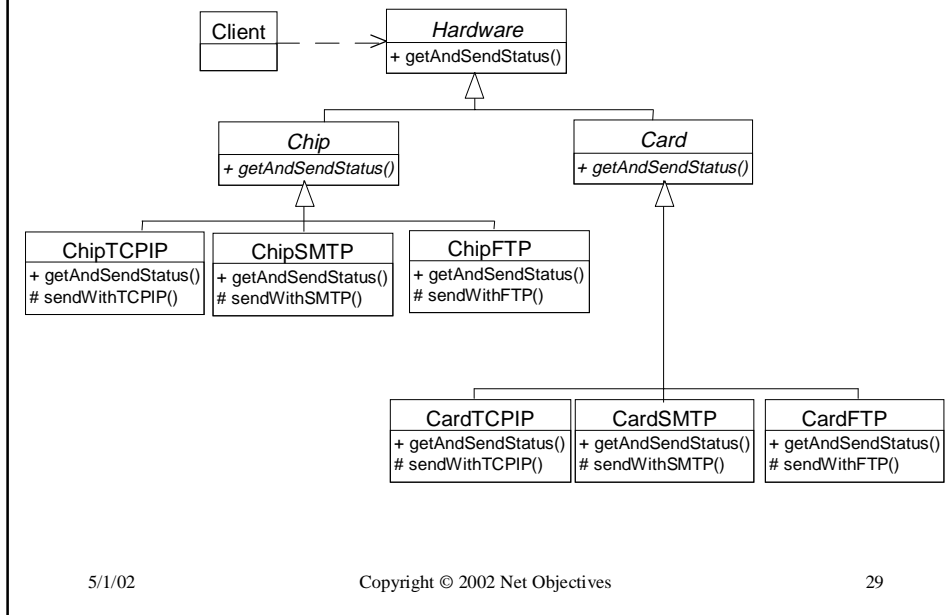
- Brittle – Will not easily allow for new derivations of Card and Chip, or new transmission types.
- Coupled – If you make a change to `sendWithTCPIP()` in `ChipTCPIP`, you also need to change it in `ChipSMTP`.
- Multiple variations will cause combinatorial (class) explosion, which increases maintenance problems

5/1/02

Copyright © 2002 Net Objectives

28

Combinatorial Explosion



Requirements as Stories

- Story 1: Request the status of a chip, encrypt it with PGP64 bit encryption and send that status out via TCP/IP.
- Story 2: Allow for not encrypting the status or using either PGP64 bit or PGP128 bit encryption. A configuration file will determine what (if any) encryption is needed.
- Story 3: Support transmission via an e-mail connection. A configuration file will determine which type of transmission to use.
- Story 4: Support getting and sending the status for a card as well.
- Story 5: When sending out the status for a card, if there isn't an error, queue the results for 10 minutes before sending in case get multiple status requests.

5/1/02

Copyright © 2002 Net Objectives

30

Starting With Story 1

- Request the status of a chip, encrypt it with PGP64 bit encryption and send that status out via TCP/IP.
- We will follow our rules attempting to implement the simplest solution possible. This really means:
 - no extra function
 - design for full system will emerge due to refactoring

5/1/02

Copyright © 2002 Net Objectives

31

Different Approaches

- One class:
 - pro: one class
 - con: low cohesion (could be hard to understand)
- Different classes for different responsibilities
 - pro: high cohesion
 - con: multiple classes

5/1/02

Copyright © 2002 Net Objectives

32

One Class Approach

See Code Example 1.

In this case, we simply write the code using methods sometimes and just writing the code sometimes.

This is not programming by intention.

Principles followed	
class cohesion?	N
method cohesion?	N
implementation encap?	Y

Problems With This

- The method – `getAndSendStatus()` is hard to follow because sometimes we have to look at the detailed code to see what is happening (getting and sending status) and sometimes we have to look at a method signature.
- Method cohesion requires that essentially all code in the method be at the same level of perspective.

“Simplest” Isn’t Always Best

- Some people will argue that simpler means having few methods (BTW: the XP gurus are not among these, they vote for few methods after consideration of coupling and cohesion issues).
- The getAndSendStatus() method would be clearer if we added getStatus() and sendStatus() methods.
- We can see this after the fact easily.
- We could also see it before the fact if we followed “programming by intention.” In this case, instead of writing an implementation, we would assume we have a method that does the job and put in a call to that. Then we would write this method.

5/1/02

Copyright © 2002 Net Objectives

35

“Programming by Intention”

See Code Example 2.

Here, we have moved the code that was embedded in the getAndSendStatus() into their own methods.

This is “programming by intention” and results in method cohesion.

Principles followed	
class cohesion?	N
method cohesion?	Y
implementation encap?	Y

5/1/02

Copyright © 2002 Net Objectives

36

Cohesion Leads to Cohesion

- Now that I've made the `getAndSendStatus` method more cohesive, I might notice that I can further increase this by moving the `openPort()` and `closePort()` methods to the `sendStatus()` method.
- I show this in Code Example 2A.

5/1/02

Copyright © 2002 Net Objectives

37

Types of Cohesion

- Method cohesion: a method is cohesive when the actions of the method are highly related.
- Class cohesion: when the methods of a class are highly related.

5/1/02

Copyright © 2002 Net Objectives

38

Possible Improvement

- We currently have good method cohesion, but poor class cohesion.
- The Chip class has stuff about the Chip, encryption and transmission in it.
- Although this seems reasonably clear here, in the real world, issues like whether some function belongs in a class are not always clear.
- We'll leave the class cohesion issue for now. It doesn't impact us negatively yet. When it does, we'll see how we can fix it later with refactoring without causing a lot of problems.

5/1/02

Copyright © 2002 Net Objectives

39

Transition

- Remember: Two Kinds of Refactoring
 1. *Refactoring Bad Code*: to improve code compliance to the principles of loose coupling, high cohesion, and no redundancy
 2. *Refactoring Good Code*: to implement a new/changed requirement, leading to emergent design
- We've been doing the first kind thus far
- Now we're going to do the second.

5/1/02

Copyright © 2002 Net Objectives

40

Issue - How Much Design Is Enough?

- Too much design is wasteful.
- Not enough design is wasteful
- Maybe the cost of not enough design can be lowered by following certain rules.

5/1/02

Copyright © 2002 Net Objectives

41

Story 2: Multiple Encryptions

- Allow for using no encryption, PGP64 bit encryption or PGP128 bit encryption.

5/1/02

Copyright © 2002 Net Objectives

42

Refactoring Says Restructuring Before Adding

- Refactoring tells us we should change our code before adding new function.
- That is, we keep function the same, but restructure our code to improve it.
- This has the following advantages:
 - if we've been doing up-front testing, our tests don't need to change (we're doing the same things)
 - we always have something that works
 - if something breaks, we are more likely to know what caused it (one step at a time)

5/1/02

Copyright © 2002 Net Objectives

43

One Class With Improved Cohesion

See Code Example 3

Principles followed	
class cohesion?	N
method cohesion?	Y
implementation encap?	Y

5/1/02

Copyright © 2002 Net Objectives

44

Now We Can Put In the New Function

- To put in new function someone must determine which encryption type to use.
- Let's say we have a configuration object that can do this for us.
- Client object can ask configuration object which to use.
- Chip object can then be told what behavior is needed.
- Now add encrypt128 and no encrypt options.
- NOTE: Chip object could be responsible for talking with configuration object, but then Client must give Chip all the information the configuration object needs.
- See Code Example 3A.

5/1/02

Copyright © 2002 Net Objectives

45

Challenges Remain

- The problem with this solution is that every time we get a new encryption algorithm, we need to change both the Config object and the Chip object.
- If we want to be able to change the encryption without changing the Chip object, we should use a strategy.

5/1/02

Copyright © 2002 Net Objectives

46

How To Improve Things

- Clear up which encryption behavior to use.
- Have Client use Chip the same way regardless of encryption type.

5/1/02

Copyright © 2002 Net Objectives

47

How To Implement This

- Case 1 (Specialization):
 - have different Chip objects, one for each different type of behavior
 - by deriving from a common class, Client can treat them the same
 - factory object can instantiate proper Chip object, simplifying Client
- Case 2 (Composition):
 - have one Chip object, but it is passed different types of encryption objects
 - each sending object works the same, simplifying Chip
 - Chip must be given proper encrypting object - can use factory

5/1/02

Copyright © 2002 Net Objectives

48

Looking at the Cases

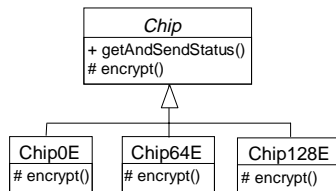
- Case 1 (Specialization):
 - slightly simpler from number of classes perspective
 - encrypting code is part of Chip so easy access to info
 - works well only if nothing else in Chip varies
 - works well only if Chip doesn't have to switch between encryption methods
- Case 2 (Composition):
 - adds an extra class, but improves cohesion
 - can vary Chip without changing encryption objects
 - can handle different encryption types in one session

5/1/02

Copyright © 2002 Net Objectives

49

Case 1: Specialize Chip



JAVA

```
class Chip0E extends Chip {
}
class Chip64E extends Chip {
}
class Chip128E extends Chip {
}
```

C++

```
class Chip0E : public Chip {
}
class Chip64E : public Chip {
}
class Chip128E : public Chip {
}
```

5/1/02

Copyright © 2002 Net Objectives

50

Advice From the Gang of Four*



* Gamma, Helms, Johnson, Vlissides - the authors of Design Patterns: Elements of Reusable Object-Oriented Design.

5/1/02

Copyright © 2002 Net Objectives

51

Gang of Four Gives Us Guidelines*

- Design to interfaces
- Favor object composition over class inheritance.
- Consider what should be variable in your design ... and “encapsulate the concept that varies.”
 - 1. Find what varies and encapsulate it in a class of its own
 - 2. Contain this class in another class to avoid multiple variations in your class hierarchies
- * Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995, pg 18, 20, 29.

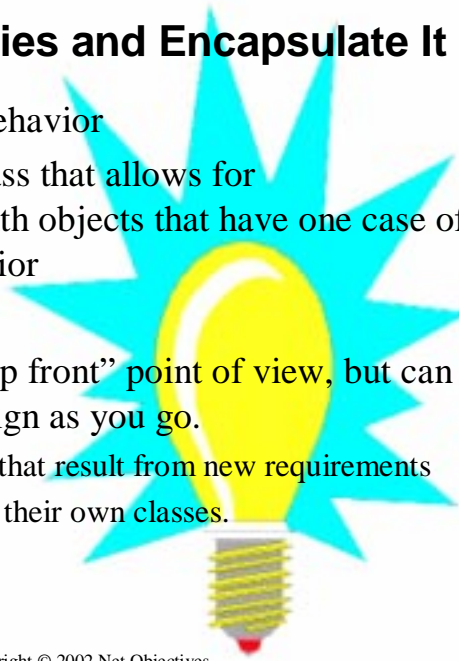
5/1/02

Copyright © 2002 Net Objectives

52

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
- This is a “design up front” point of view, but can be tailored for design as you go.
 - Extract variations that result from new requirements
 - Pull these out into their own classes.



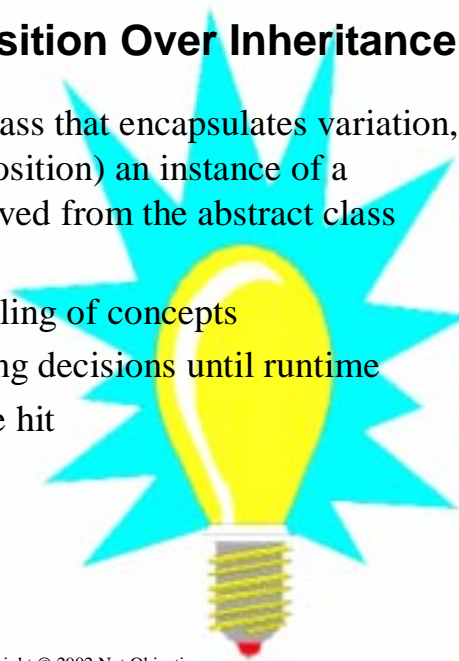
5/1/02

Copyright © 2002 Net Objectives

53

Favor Composition Over Inheritance

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



5/1/02

Copyright © 2002 Net Objectives

54

Handling Variations

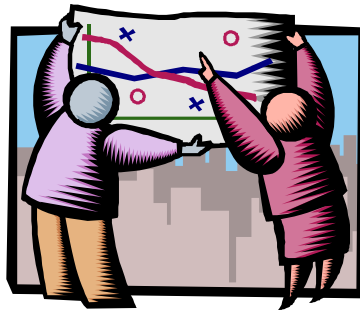
- It is virtually always easier to handle different types of instantiation rather than different types of usage.
- That is, a factory can create different objects that are handled in a similar way with polymorphism.
 - the factory is very cohesive (only deals with instantiation)
 - handling the different objects is easy (polymorphism)
- Using a single object with different behavior (handled with a switch) may be more difficult.
 - poor logic
 - low cohesion

5/1/02

Copyright © 2002 Net Objectives

55

The Strategy Pattern



5/1/02

Copyright © 2002 Net Objectives

56

The Strategy Pattern

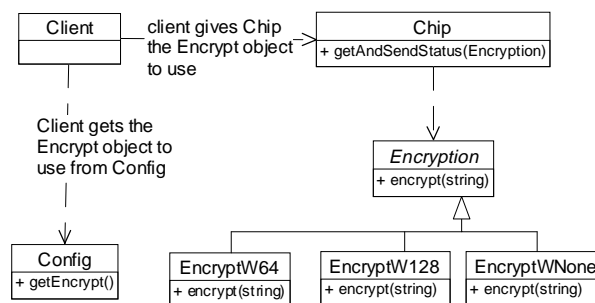
- GoF Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. ³
- We need different behavior at different times; either for different things we are working on or for different clients asking us to do work.
- ³ Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

5/1/02

Copyright © 2002 Net Objectives

57

UML Diagram of Solution



5/1/02

Copyright © 2002 Net Objectives

58

Using the Strategy Pattern

- We can make the Chip independent of the encryption method by:
 - have the Client decide on encryption method to use
 - give an encryption object encapsulating this method to the Chip
 - the Chip uses it
- See Code Example 4 for Implementation

5/1/02

Copyright © 2002 Net Objectives

59

Chip Is Independent of the Encryption Algorithm

- New encryption algorithms can be added without affecting Chip.
- Control of which one to use is done outside Chip (either in Client itself, or as shown, in Factory).

5/1/02

Copyright © 2002 Net Objectives

60

What Happens Now With Changing Requirements

- If we get a new encryption requirement, we only need to
 - derive the new class from the Encryption class
 - modify the Factory
 - no other objects change

5/1/02

Copyright © 2002 Net Objectives

61

Why Bother?

- What happens if many things vary?
- In other words, in addition to the encryption rules, could we have the following variations?
 - different sending methods
 - compression rules
 - different components (other than Chips)
- keeps things decoupled
- keep them cohesive

5/1/02

Copyright © 2002 Net Objectives

62

What We've Done

- At this point, we've implemented the Strategy pattern.
- We refactored once:
 - 1) to get method cohesion
 - 2) to get class cohesion
- Going so far as class cohesion was not obvious at the start -- certainly not when had only one encryption method.
- Method cohesion, however, is typically always a good idea.

5/1/02

Copyright © 2002 Net Objectives

63

Story 3

- Support transmission via an e-mail connection. A configuration file will determine which type of transmission to use.
- The GoF tells us to find what varies, encapsulate it and contain it in our class.
- Let's say we didn't do this. We'll see the result is lower cohesion.

5/1/02

Copyright © 2002 Net Objectives

64

Not Encapsulating the Implementation

- If we don't even encapsulate the implementation of the transmission, we could have a switch that indicated which transmission to use:

```
public void sendStatus(String anInfo) {
    if (transType == TCPIP) {
        // Code to send via TCPIP goes here
        // . . .
        // . . .
    } else {
        // Code to send via SMTP goes here
        // . . .
        // . . .
    }
}
```

- This lowers cohesion of the method this code is in.

5/1/02

Copyright © 2002 Net Objectives

65

Putting it in a Method Is Not Enough

- We can improve things a little by putting the transmission code in its own method, like we did for the encryption function. That is:

```
private void sendStatus(String anInfo) {
    if (transType == TCPIP) {
        sendStatusTCPIP(anInfo);
    } else {
        sendStatusSMTP(anInfo);
    }
}
```

- This is better. However, this approach still requires the Chip class to remember even more stuff about how to transmit. It's already remembering how TCP/IP works, now it'd have to remember SMTP stuff as well -- this erodes cohesion even more.

5/1/02

Copyright © 2002 Net Objectives

66

Want to Follow Gang of Four Advice

- Because transmission is going to vary, we want to encapsulate it and contain it in the using class (the Chip).
- To implement this, first pull out the existing transmission functionality into its own class.
- Note that there is no extra cost caused by us doing this now instead of when we first noticed it was possible.

5/1/02

Copyright © 2002 Net Objectives

67

A Note About Class Names

- It's easy to mis-name objects.
- People assume something does what it sounds like until they learn differently.
- In this case, we could call our transmission object Trans.
- However, we should keep in mind that although we only have one way to transmit now, the way we are doing it is via TCPIP. A name that reflects this will be more useful if we ever get a new method of transmitting.
- In this case we know we will. But even if we didn't, a simple rule to follow is:
 - give specific type names to concrete type classes.
 - give generic type names to abstract classes.
- See Code Example 5.

5/1/02

Copyright © 2002 Net Objectives

68

Still Can't Add New Functionality

- This was a good first step.
- However, now we must make things use an abstract class and then we can add the new code.
- We do this first, then add the new function SMTP.
- See Code Example 6.

5/1/02

Copyright © 2002 Net Objectives

69

Story 4: Support Retrieving and Sending the Status for a Card.

- First, note how this job is going to be easier because the Chip class has nothing about encryption or transmission in it except for the hooks to the objects that actually do the work.
- This means when developing the Card, we can use these objects as well.
- Code Example 7 actually combines two steps:
 - 1) Create HWComp class and have Chip derive from it
 - 2) Derive Card from HWComp class

5/1/02

Copyright © 2002 Net Objectives

70

Interesting Note

- We have moved much of the code in the Chip and Card class up to the HWComp class, since it is the same.
- However, when we add the function to the Chip and Card that is different, we have a place to put it.
- In Java, this was easy, and a good thing to do. We could have just used an Interface instead, but now we've eliminated a great deal of redundancy.

5/1/02

Copyright © 2002 Net Objectives

71

Story 5: Queue Info For Cards

- When sending out the status for a card, queue the results for 10 minutes before sending if there is no error, in case get multiple status requests.
- We now must make a decision about where to put the queuing. It can go on the card, since that is who is using it, or it can go as an option for transmission.
- The question can be rephrased as: is queuing an inherent characteristic of Cards or is it a behavior Cards may do?
- If queuing is a behavior of Cards, it may be that other HWComps will also have this behavior. That is, when transmitting other types of information, we may want to do queuing. Putting a Queue ability on the Trans class may lead to more reuse.

5/1/02

Copyright © 2002 Net Objectives

72

Adding Queue to Trans

- Where do we put the queue?
 - The Card class?
 - The Trans class?
- Consider where we might need it in the future.
- All things being equal now, do what might help us in the future.
- Admittedly, queuing in either class lowers cohesion, but we've seen this doesn't hurt us at this point.
- Code Example 8 adds the queue functionality to the Trans class.

5/1/02

Copyright © 2002 Net Objectives

73

A Few Comments

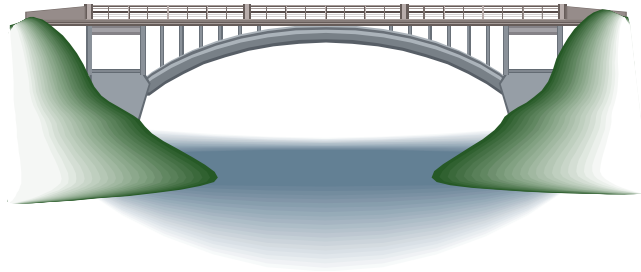
- It is not a coincidence, or just good luck that we could make these changes easily.
- It happened because we made sure there was no redundancy and because we kept unrelated things in different classes.
- These low level distinctions are easy to see, even if their immediate benefit is not.
- We should use them because they represent very low cost and will result in significant gains if things change -- which they almost certainly will.

5/1/02

Copyright © 2002 Net Objectives

74

The Bridge Pattern



5/1/02

Copyright © 2002 Net Objectives

75

The Bridge Pattern

- GoF Intent: De-couple an abstraction from its implementation so that the two can vary independently ³
- The normal reaction to reading this is:
 - huh?
- ³ Design Patterns, Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides

5/1/02

Copyright © 2002 Net Objectives

76

The Bridge in Our Case

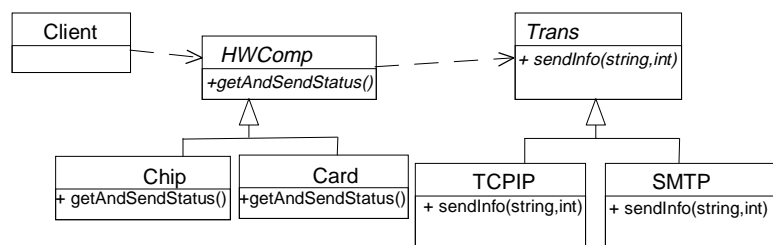
- The Bridge pattern in our case means we separate our abstractions (different kinds of hardware - Chips and Cards) from our implementations (TCP/IP communication and e-mail communication).
- The way to do this is by having all of our implementations look the same to all of our hardware components.

5/1/02

Copyright © 2002 Net Objectives

77

Bridge Would Tell Us to Build It This Way

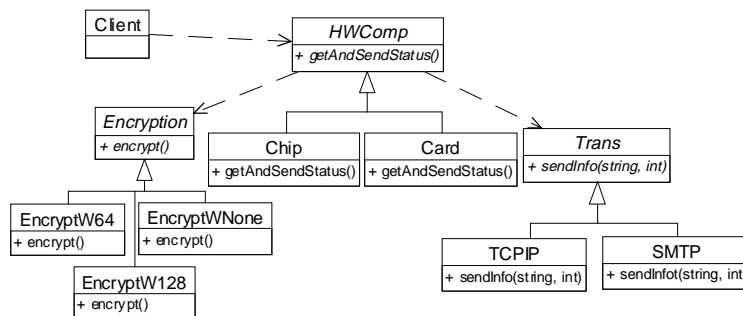


5/1/02

Copyright © 2002 Net Objectives

78

Bridge and Strategy Together



5/1/02

Copyright © 2002 Net Objectives

79

Can We Refactor a Design?

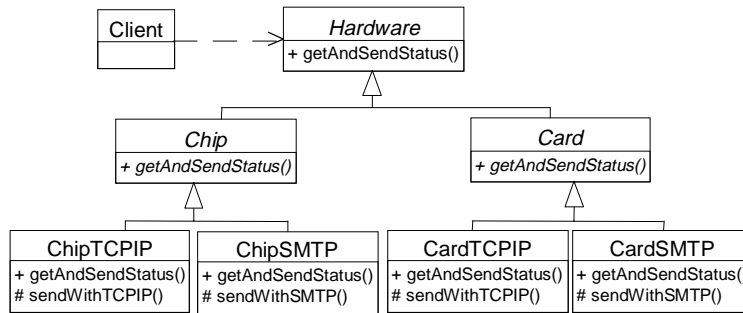
- Let's say we took the original "nouns and verbs" approach.
- Although we wouldn't recommend getting into this trouble in the first place, let's see if we can get out of it by refactoring.

5/1/02

Copyright © 2002 Net Objectives

80

Poor Design as a Result of Using Nouns and Verbs

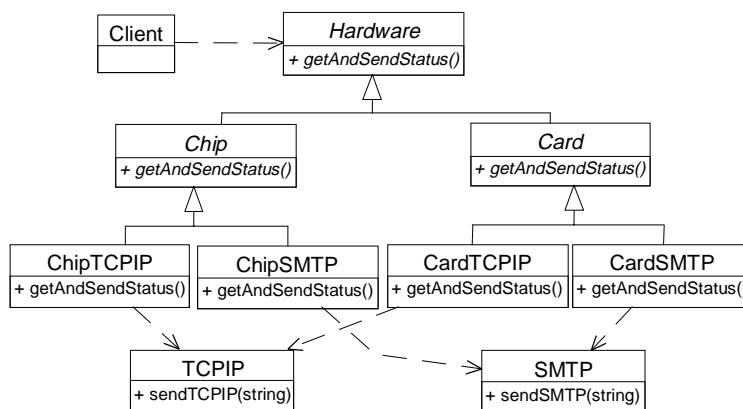


5/1/02

Copyright © 2002 Net Objectives

81

Pull Out Duplication of Transmission

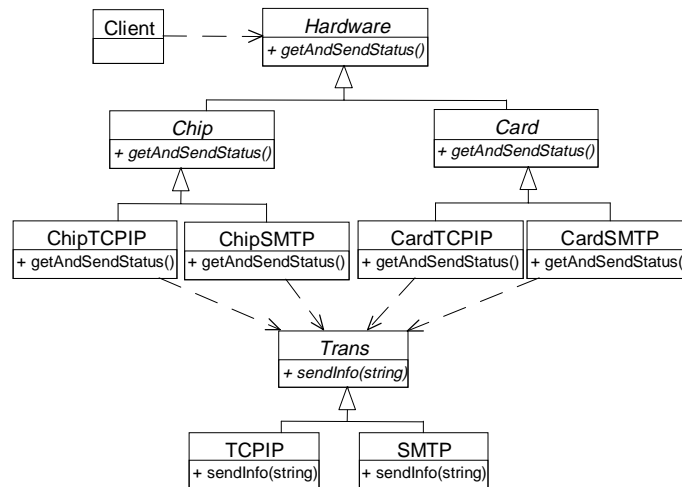


5/1/02

Copyright © 2002 Net Objectives

82

Create *Trans* Class To Simplify Things

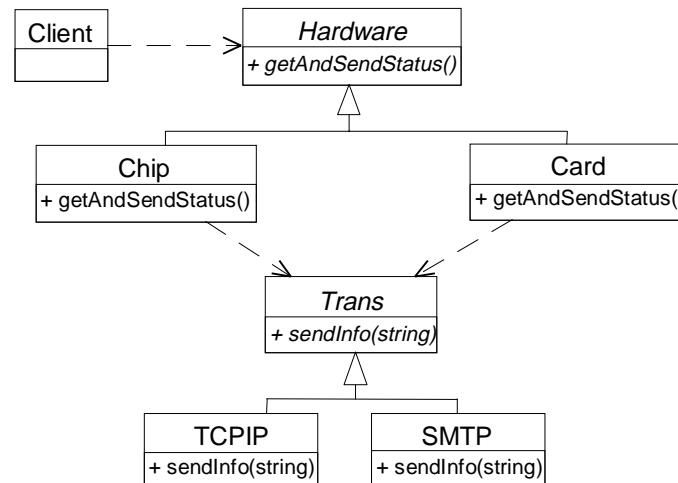


5/1/02

Copyright © 2002 Net Objectives

83

Derived Classes Now Not Needed

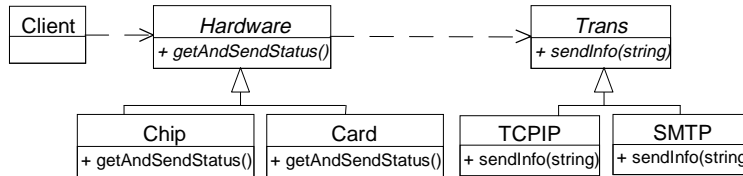


5/1/02

Copyright © 2002 Net Objectives

84

Can Keep Reference In Hardware

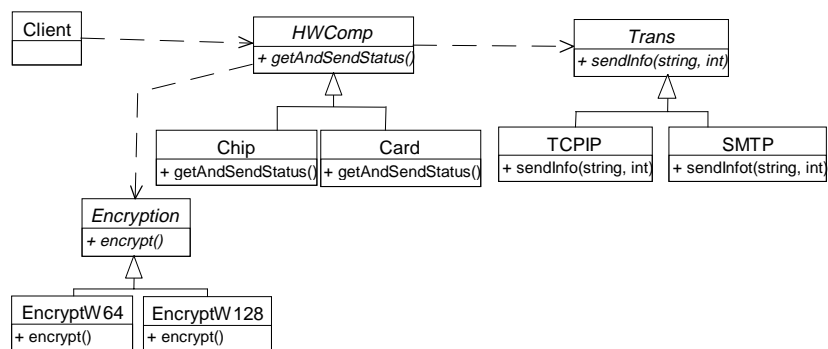


5/1/02

Copyright © 2002 Net Objectives

85

The Full Design: Bridge and Strategy



5/1/02

Copyright © 2002 Net Objectives

86

Using Commonality / Variability Analysis

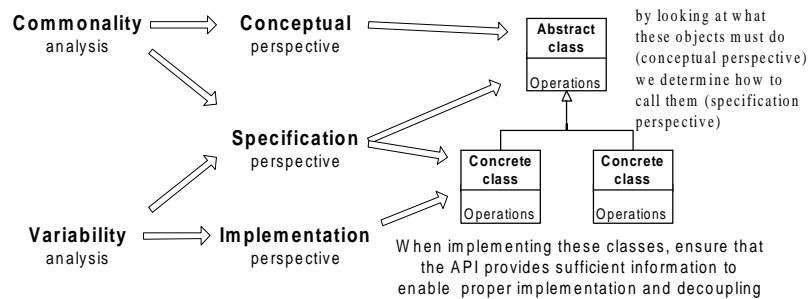
- CVA tells us to first find:
 - those things that don't change
 - the commonality of things that do change
- Define interfaces/abstract classes for these commonalities
- Define the variations of these commonalities (i.e, the specific cases)
- See how these classes relate to each other.

5/1/02

Copyright © 2002 Net Objectives

87

Using Commonality/Variability Analysis to Design Class Structures



Relationship between specification and conceptual: what interface do I need to handle all the cases (i.e., conceptual perspective)

Relationship between specification and implementation: given this specification, how can I implement this particular case (i.e., this variation)

5/1/02

Copyright © 2002 Net Objectives

88

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

5/1/02

Copyright © 2002 Net Objectives

89

CVA: Step 1 - Identify Fixed and Commonalities

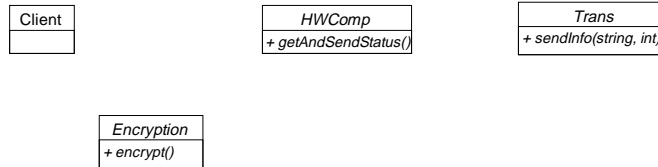
- In our problem domain, we have:
 - a Client
 - Hardware
 - Encryption methods
 - Transmission methods

5/1/02

Copyright © 2002 Net Objectives

90

CVA: Step 2 - Define Classes For Them

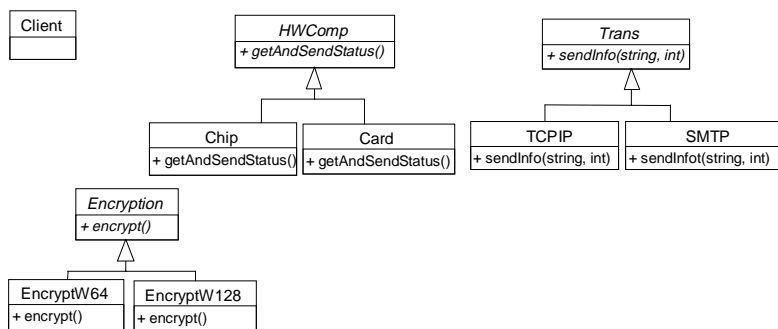


5/1/02

Copyright © 2002 Net Objectives

91

CVA: Step 3 - Define Variations

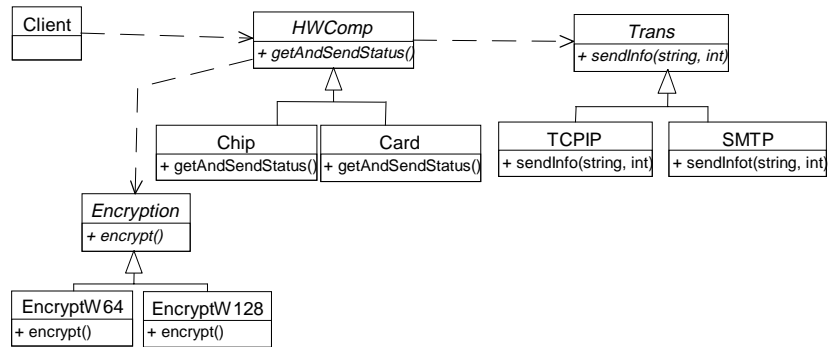


5/1/02

Copyright © 2002 Net Objectives

92

CVA: Step 4 - Show Relationships



5/1/02

Copyright © 2002 Net Objectives

93

Conclusions

- Design emerges from:
 - Refactoring, with adherence to good principles
 - Thinking in Patterns, which reflects past adherence to good principles
 - Commonality/Variability Analysis, which leads to good principles
- "Good Principles" are:
 - High cohesion (method, class)
 - Loose coupling
 - No Redundancy

5/1/02

Copyright © 2002 Net Objectives

94

Conclusions

- Agility in the development process requires flexibility in terms of the methodologies employed.
- Such flexibility comes from, among other things:
 - Adherence to Good Principles, while
 - Maintaining an awareness of emergent design, by
 - Understanding the forces of change, and
 - Recognizing applicable patterns in the design
- We *can* create maintainable code

5/1/02

Copyright © 2002 Net Objectives

95

Bibliography

- Agile Software Development, Alistair Cockburn
- Design Patterns Explained: A New Perspective on Object-Oriented Design, Shalloway, Trott
- Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helms, Johnson, Vlissides (still the best reference book on patterns)
- Extreme Programming Explained, Beck (best introduction to XP, see http://www.xprefined.com/xpr_xp2read.htm for minimal amount to read).
- Multi-paradigm Design for C++, Coplien (excellent description of commonality/variability analysis -- I even recommend the first part for Java programmers. See <http://www.netobjectives.com/download/CoplienThesis.pdf> for the books equivalent on the web)
- Refactoring, Martin Fowler (most in depth book on this old practice)
- Timeless Way of Building, Christopher Alexander (a classic. The only book in my top ten personal and top ten business)
- UML Distilled, Martin Fowler (easiest and best way to learn the UML)
- Writing Effective Use Cases , Alistair Cockburn
- Best place to find where to buy books: www.bestbookbuys.com

5/1/02

Copyright © 2002 Net Objectives

96

Net Objectives - Who We Are

- We provide training, mentoring and consulting for all phases of object-oriented software development.
- We assist companies transitioning to object-oriented development by providing mentoring throughout the entire development process.
- This enables our clients a cost-effective way to gain experience.
- We offer courses in OOA, OOD, Design Patterns, XML, XSLT, Schema, .NET, the Software Development Process, the UML, the Unified Process, XP, Java, and C++.
- **To subscribe to our e-zine, send an e-mail to info@netobjectives.com with “subscribe” in the subject line**

5/1/02

Copyright © 2002 Net Objectives

97

Upcoming Seminars and Courses in the Seattle Area

Free Seminars:

The Need For Agility. May 21, Seattle. Sponsored by the Association for Women in Computing

See http://www.netobjectives.com/pr_future.htm for more information and instructions on how to register.

Courses (all on Eastside):

- Pattern Oriented Design: Design Patterns From Analysis to Implementation. June 3-4.
- Agile Development. July 22-23.
- Design Patterns Applied with Bruce Eckel and Alan Shalloway. September 16-20

See http://www.netobjectives.com/courses/c_pubsched.htm for more information.

All courses have early registration discounts available.

5/1/02

Copyright © 2002 Net Objectives

98

Net Objectives' Design Patterns Community of Practice

- This community of practice centers on the way Net Objectives suggests using design patterns. A description of this approach is in Alan Shalloway and Jim Trott's *Design Patterns Explained: A New Perspective on Object-Oriented Design*. This site, and its corresponding discussion group, help facilitate understanding of the book as well as providing developers with a place to continue learning about design patterns. As a development community, it is comprised of developers who are committed to honing their development skills and see that design patterns can be useful in doing that.
- Located at:
<http://www.netobjectives.com/dpexplained/index.html>

5/1/02

Copyright © 2002 Net Objectives

99

Selected On-Site Courses Available

- **An Introduction to Object-Oriented Programming in Java.** Teaches good object-oriented concepts to those already familiar with the Java language but not expert in OO. (2 days)
- **Pattern Oriented Design: Design Patterns From Analysis to Implementation.** Introduces design patterns and creates a new paradigm for analysis and design. (2 days)
- **Design Patterns Lab.** Hands on programming experience with design patterns. (1 day)
- **Advanced Object-Oriented Design Methods Using Design Patterns.** Takes design patterns and commonality/variability analysis to the next level. (1 day)
- **An Introduction To Agile Methodologies.** Use an agile process to increase efficiency and effectiveness (1 day)

5/1/02

Copyright © 2002 Net Objectives

100

Appendix

Extra Material For Further Thought

5/1/02

Copyright © 2002 Net Objectives

101

A Note About Refactoring

- Although no XP expert would say so (and certainly not Martin Fowler), in practice I have heard refactoring used as a kind of excuse to not do good coding in the first place.
- In other words, the logic is something like - “let’s do it quick now and fix it later”
- This is not what refactoring is for.
- Refactoring, theoretically anyway, should not be needed except when a requirement comes in that forces a redesign.

5/1/02

Copyright © 2002 Net Objectives

102

Synergies

- Design Patterns
- Coding Rules
- Agile Methods (e.g., eXtreme Programming)

- These live in different domains and would seem to only indirectly affect each other.
- This is not the case.
- There are definite, synergistic benefits, to these working together.

5/1/02

Copyright © 2002 Net Objectives

103

Design Patterns

- Design Patterns:
 - teach us the importance of commonality/variability analysis
 - teach us the importance of:
 - loose coupling
 - high cohesion
 - no redundancy
 - give us strategies to follow when requirements change or there is variation in our problem domain
 - give us a common language

5/1/02

Copyright © 2002 Net Objectives

104

Coding Rules

- Loose coupling to eliminate side effects
- High cohesion to increase clarity
- No redundancy to improve efficiency

- These tell us in concrete terms how to code.
- Enhances common terminology.

5/1/02

Copyright © 2002 Net Objectives

105

Agile Methods

- Agile methods give us a way to manage the business forces.
- Incremental, iterative, integrated design allows for quick feedback and close customer interaction.
- Has us work on the right stuff.
- Has us eliminate waste.
- Requires a development methodology that can adapt to change quickly.

5/1/02

Copyright © 2002 Net Objectives

106

What Is Extreme Programming (XP)?

- Known best for:
 - paired programming
 - up-front testing
 - short iterations
- Based on the notion of doing a little at a time, getting feedback, and doing some more.
- Do design as you go - with proper coding techniques, good design will emerge.
- If you aren't familiar with XP, don't worry, but if you are, you can learn a bit about why emergent design works.

5/1/02

Copyright © 2002 Net Objectives

107

Design Patterns - Agile Methods

- Design patterns tell us to focus on commonalities and variations of these commonalities. In other words, we learn what is important and what can be deferred.
- The common terminology of design patterns facilitates the extra communication used in agile methods.
- Agile methods tell us to focus on the issues at hand.
- Bottom line: DPs handle the development forces of change, AM gives us a way to manage it while including the business forces. Also, DPs handle changes well - a fallout of AM.

5/1/02

Copyright © 2002 Net Objectives

108

Agile Methods - Coding Rules

- Agile Methods imply constant changing of code.
- Coding rules give us a way to write our code so it can be changed.

5/1/02

Copyright © 2002 Net Objectives

109

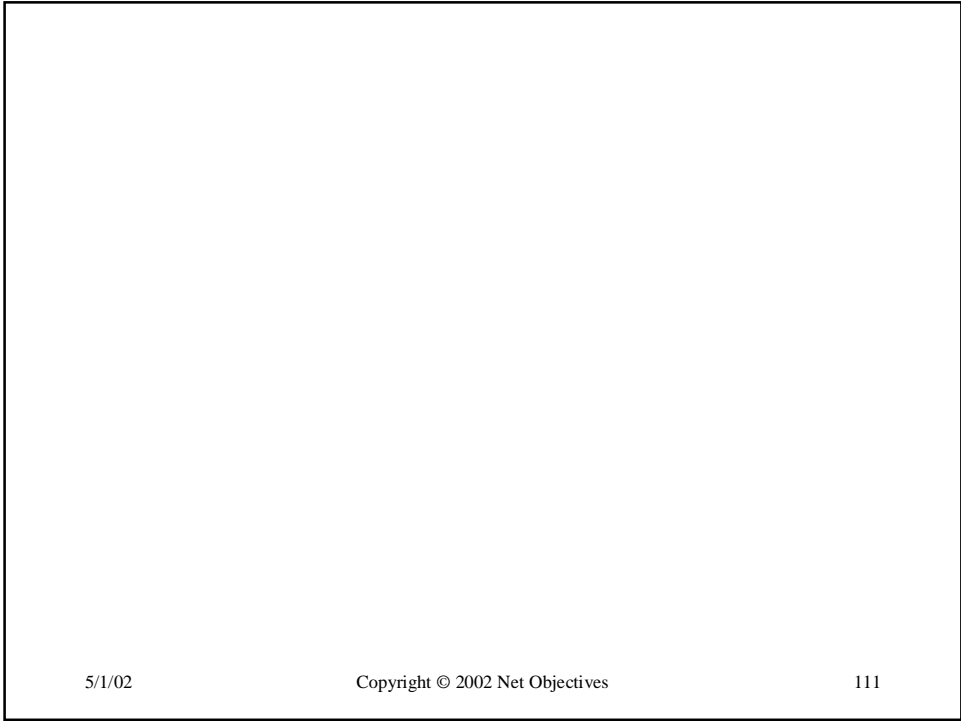
Coding Rules - Design Patterns

- Design patterns are good examples of good coding rules.
- Design patterns give strategies to follow to achieve good coding rules in the face of changing requirements.

5/1/02

Copyright © 2002 Net Objectives

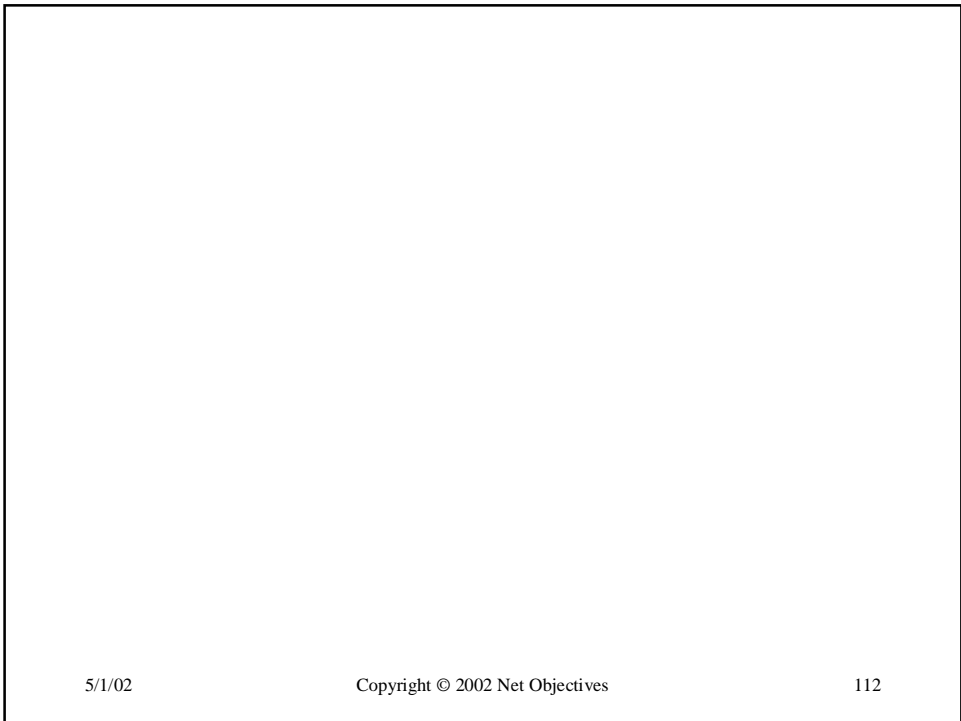
110



5/1/02

Copyright © 2002 Net Objectives

111



5/1/02

Copyright © 2002 Net Objectives

112