

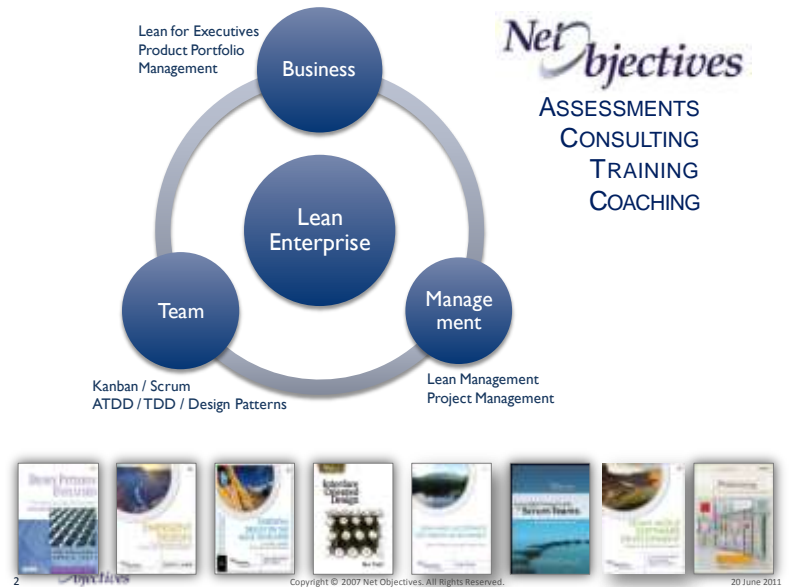
Design Patterns Explained



Net Objectives
info@netobjectives.com
www.netobjectives.com

Avoiding Over and Under Design

1 Copyright © 2008 Net Objectives. All Rights Reserved. 20 June 2011



Net Objectives
ASSESSMENTS
CONSULTING
TRAINING
COACHING

Business
Lean for Executives
Product Portfolio
Management

Lean Enterprise

Team
Kanban / Scrum
ATDD / TDD / Design Patterns

Management
Lean Management
Project Management

2 Copyright © 2007 Net Objectives. All Rights Reserved. 20 June 2011

Design Patterns Explained



Net Objectives' Talks at Conference

Monday

- 1:30-5:00pm Rob Myers. *Test Driven Development*

Wednesday

- 8:30-10:00am Ken Pugh. *Regular Expressions in C++*
- 3:30-5:00pm Ken Pugh. *Prefactoring*
- 3:30-5:00pm Rob Myers. *Principles and Practices of Scrum*

Thursday

- 10:15-11:45am Scott Bain & Rob Myers. *Emergent Design Demo through Unit-Testing, Refactoring and Pair Programming*
- 10:15-11:45am Alan Shalloway. *Lean Software Development: The Business Case for Agility*
- 1:30-3:00pm Alan Shalloway. *Design Patterns Explained*
- 1:30-3:00pm Rob Myers. *Business Value of Pair Programming*

Friday

- 10:15-11:45am Alan Shalloway. *Emergent Design: Design Patterns and Test-Driven Development*
- 3:30-5:00pm Scott Bain. *Mock Objects and Mock Turtles: The Role of Patterns in TDD*
- 3:30-5:00pm Rod Claar. *Dealing With Enterprise Data in an Agile Environment*

And don't forget to see us in the Expo!

Design Patterns Explained

Emergent Design Agenda



- Assumes knowledge of:
 - Code Qualities
 - Unit Testing
 - Refactoring
 - Test First
 - Test-Driven-Development:
 - Test as design
 - Emergent Design



Net Objectives
info@netobjectives.com
www.netobjectives.com

Emergent Design

Design Patterns and Refactoring
for Agile Development

6

Copyright © 2008 Net Objectives. All Rights Reserved. 20 June 2011

Design Patterns Explained

Test: What Does It Mean?

- **1 a chiefly British : CUPEL b (1) :** a critical examination, observation, or evaluation : **TRIAL**; *specifically : the procedure of submitting a statement to such conditions or operations as will lead to its proof or disproof or to its acceptance or rejection* <a test of a statistical hypothesis> (2) : a basis for evaluation : **CRITERION** c : an ordeal or oath required as proof of conformity with a set of beliefs
- Test can be thought of as what your specification is that says you are doing what you should be doing.

7

Net Objectives

Copyright © 2008 Net Objectives. All Rights Reserved.

20 June 2011 20 June 2011



Net Objectives

Copyright © 2007 Net Objectives. All Rights Reserved.

20 June 2011

Design Patterns Explained

Predictability



- We *can't* predict how our requirements are going to change
- We *can* predict how our code will adapt to unpredictable requirements changes
- How can we increase our prediction abilities of code quality?

Question to Ask



- When working on a mature system consider when adding a new function...
 - which task area takes more time
 - writing the new function or
 - integrating it into the system?

Design Patterns Explained

Qualities and Pathologies

- Strong cohesion
 - A goal: classes do one thing – easier to understand
 - Pathology: the “God object” is as bad as it gets
- Proper coupling
 - A goal: well defined relationship between objects
 - Pathology: side affects when have improper coupling
- No redundancy
 - A goal: once and only once
 - Pathology: a change in one place must be duplicated in another
- Readability
 - A goal: coding standards
 - Pathology: non-readable code
- Encapsulation
 - A goal: hide data, type, implementation
 - Pathology: assumptions about how something is implemented makes it difficult to change

What Are the Characteristics of Easily Maintainable Code?

- Code is readable.
 - can see what individual things do
 - can see how individual things interact
- 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: readable, independent, efficient

...and, of course, correct

Design Patterns Explained

Cohesion

- Cohesion refers to how “closely the operations in a routine [or class] 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. *
- Strong cohesion is related to clarity and understanding.
- “No schizophrenic classes”

* Steve McConnell, *Code Complete*, 1993, p. 81. Note: This concept was first described by Larry Constantine in 1975, but we like McConnell's definition best.

Coupling

- Coupling refers “to the strength of a connection between two routines [or classes]. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a routine [or class] are related to each other. Coupling describes how strongly a routine is related to other routines. The goal is to create routines [and classes] with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines [and classes] (loose coupling).”*
- Tight coupling is related to highly interconnected code.

* Steve McConnell, *Code Complete*, 1993, p. 81. Note: This concept was first described by Larry Constantine in 1975, but we like McConnell's definition best.

Design Patterns Explained

No Redundancy

- "One Rule in One Place"
- Redundancy is not just:
 - Redundant state
 - Redundant functions
- It can also be redundant *relationships*

Encapsulation

- Data
 - The data needed for a class to fulfill its responsibilities is hidden from other entities.
- Implementation
 - How a class implements a particular function, or whether it implements it itself or delegates to other objects, is hidden.

There are well-known, but also consider:

- Type
 - Abstract classes and interfaces can hide their implementing classes.
- Design
 - Assembling collaborating classes with an object factory keeps clients decoupled from the way they are designed.
- Construction
 - Encapsulation of construction means wrapping "new" in, at least, a separate method, giving you control over the return type.

Design Patterns Explained

Qualities and Pathologies

- **Strong cohesion**
 - A goal: classes do one thing – easier to understand
 - Pathology: the “God object” is as bad as it gets
- **Proper coupling**
 - A goal: well defined relationship between objects
 - Pathology: side affects when have improper coupling
- **No redundancy**
 - A goal: once and only once
 - Pathology: a change in one place must be duplicated in another
- **Readability**
 - A goal: coding standards
 - Pathology: non-readable code
- **Encapsulation**
 - A goal: hide data, type, implementation
 - Pathology: assumptions about how something is implemented makes it difficult to change

Testability and Design

- Considering how to test your objects before designing them is, in fact, a kind of design
- It forces you to look at:
 - the public method definitions
 - what the responsibilities of the object are
- Easy testability is tightly correlated to loose coupling and strong cohesion

Design Patterns Explained

Testability

- Code that is difficult to unit test is often:
 - Tightly Coupled:** "I cannot test this without instantiating half the system"
 - Weakly Cohesive:** "This class does so much, the test will be enormous and complex!"
 - Redundant:** "I'll have to test this in multiple places to ensure it works everywhere"

Unit Testing

- Tests at a low, or granular level
- Each test confirms that the code accurately reflects one intention of the system
- Test per class is often a natural fit
- A good test of our thought process:
 - If we've designed classes to do one thing (strong cohesion), then we should be able to test their functionality
 - If classes do not create side effects in other classes, (loose coupling), then we should be able to test them individually

Design Patterns Explained



Refactoring

- Refactoring: "Improving the Design of Existing Code"*
- It's actually more than that.
- Largely underestimated in importance
- Martin Fowler's book: "Refactoring" – an essential reference for any developer/team

Martin Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Design Patterns Explained

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.”*
- Assuming we know when we mean by "quality code", Refactoring gives us a way to get there if we're not already

Martin Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Types of Refactoring

Refactoring Bad Code

- Code "smells"
- Improve Design without changing Function.
- Refactor to improve code quality
- A way to clean up code without fear of breaking the system

Refactoring Good Code

- Code is "tight"
- A new Requirement means code needs to be changed
- Design needs to change to accommodate this.
- A way to make this change without fear of breaking the system

Design Patterns Explained



Net Objectives
info@netobjectives.com
www.netobjectives.com

Case Study

Monitoring Microwave Communications Hardware

25 Copyright © 2008 Net Objectives. All Rights Reserved. 20 June 2011

The slide features a blue background with a hand placing a puzzle piece into a larger puzzle. The puzzle pieces are light blue and white. In the bottom right corner, there is a small black silhouette of a satellite dish on a hill.

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 (SMTP)
- These messages may be optionally encrypted with either PGP64 bit encryption or PGP128 bit encryption
- When sending status out for a chip, we want to queue the information to send it out no more than every 10 minutes unless there is an error. Cards, on the other hand, send immediately
- A configuration file will contain information about which transmission method to use

Design Patterns Explained

Finding Entities and Their Behaviors



- This has us focus on the “things” we have, first. Then, we handle differing behaviors through the use of polymorphism
- If more than one behavior varies, can result in tall class hierarchies...
 - ...which are hard to test
 - ...which have unintended coupling
 - ...which have redundancies
 - ...which promote weak cohesion

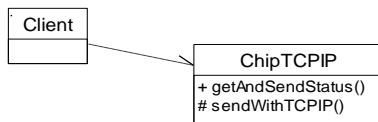
Accommodating Change with Specialization



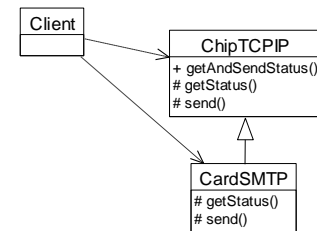
- Let’s walk through a potential way our problem could evolve. We start with Chip and TCP/IP and add function one step at a time. We accommodate this through specialization
- The result is not pretty (except as in pretty common)

Design Patterns Explained

Start with ChipTCPIP Requirement

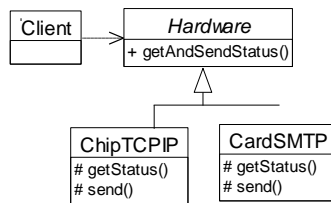


Now Get Card with SMTP



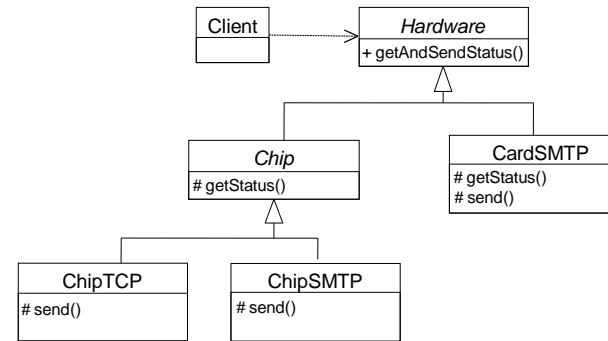
Design Patterns Explained

Better Way



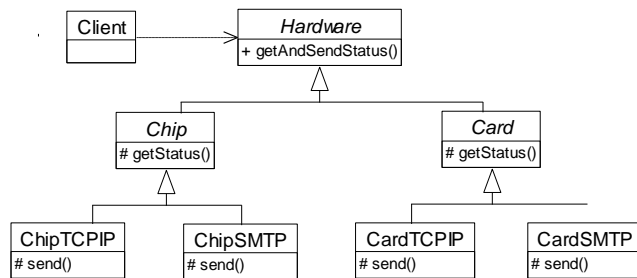
This at least avoids some confusion

Then Get Chip with SMTP Requirement



Design Patterns Explained

Finally Get Card with TCPIP Requirement



It Is, Of Course, *Worse*

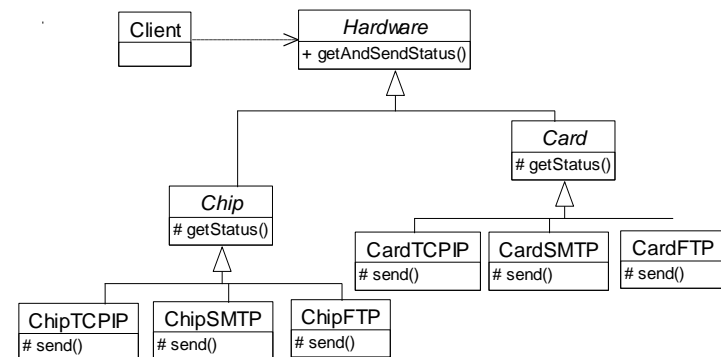
- Note that we haven't even been discussing the variations of encryption
- That will just make things worse
- If you use switches instead of inheritance you merely have coupled switches instead of a tall class hierarchy – it is still bad

Design Patterns Explained

Problems with This

- Brittle. Will not easily allow for new derivations of Card and Chip, or new transmission types
- Redundant. If you make a change to send() in ChipTCPIP, you also need to change it in CardTCPIP
- Weak cohesion. Concrete classes are about multiple things.
- Multiple variations will cause combinatorial (class) explosion, which increases maintenance problems
- In Short: Using inheritance for specialization *does not scale*

Combinatorial Explosion



Design Patterns Explained



Net Objectives
info@netobjectives.com
www.netobjectives.com

Advice from the Gang of Four

37 Copyright © 2008 Net Objectives. All Rights Reserved. 20 June 2011

Gang of Four Gives Us Guidelines*

- Design to interfaces
- Favor object aggregation 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

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, pp. 18, 20, 29.

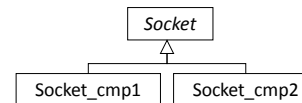
Design Patterns Explained

Design to Interfaces

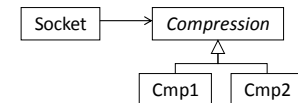
- Determine the proper interface for a class, and design to that, ignoring implementation details
 - If you ignore implementation details, you cannot possibly couple to them
 - What you hide you can change
- Allows you to stay at a conceptual level, mentally, while designing
 - Most people cannot think effectively on multiple levels without confusion
 - Promotes Cohesion

Favor Aggregation Over Inheritance

- We can define a class that encapsulates variation, contain (via aggregation) an instance of a concrete class derived from the abstract class defined earlier



Class Inheritance to Specialize



Object Aggregation

1. Allows for decoupling of concepts
2. Allows for deferring decisions until runtime
3. Small performance hit

Design Patterns Explained

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

What Are Design 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.

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

Design Patterns Explained



Emergent Design

- Test-Driven Development, integrated with:
 - the concepts of high quality code
 - the knowledge of design patterns
 - the attitude of building only what you need (Agile, YAGNI)
- Allows for designs to **emerge**
- We can take advantage of what we know without overbuilding or over-designing.

Design Patterns Explained

Requirements as XP-Style “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

Emergent Design: 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 via refactoring

Design Patterns Explained

Thinking from a Testability Perspective

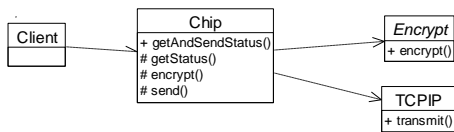
- What do we need to test?
 - getting status
 - encrypting a string
 - sending an encrypted string
- Writing the tests for the individual methods should be straightforward enough.
- Where should the methods lie?
- What's the easiest way?

Test-First Development Steps

1. Write a test that expresses an intent of a class in the system
2. Stub out the class (enough to allow the test to compile)
3. Fail the test (don't skip this)
4. Change the class just enough to pass the test
5. Pass the test
6. Examine the class for coupling, cohesion, redundancy, and clarity problems. Refactor.
7. Pass the test
8. Return to 1 until all intentions are expressed

Design Patterns Explained

Solution Diagrammed



Why Is this Better?

- It is clear what the pieces do and how they relate
 - Increases extensibility
 - Eases maintainability
- Sometimes requirements mis-lead us

Design Patterns Explained

Transition

- Remember: Two Kinds of Refactoring
 1. *Refactoring Bad Code*: to improve code compliance to the principles of loose coupling, strong 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

Story 2: Multiple Encryptions

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

Design Patterns Explained

Refactoring Says to Restructure 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)

How Can We Most Easily Test Things?

- Need to test every encryption
- Want to deal with using encryptions the same way.
- Easiest way is if we can have a common interface to the encryptions
- If interfaces aren't the same, then we have two choices:
 - Have Chip know differences and deal with them.
 - Use lessons from design patterns and hide the variations. This requires wrapping the differences. There are many ways to do this, even when it at first appears difficult.

Design Patterns Explained

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

For a good article on the Open-Closed Principle, see www.objectmentor.com/publications/ocp.pdf

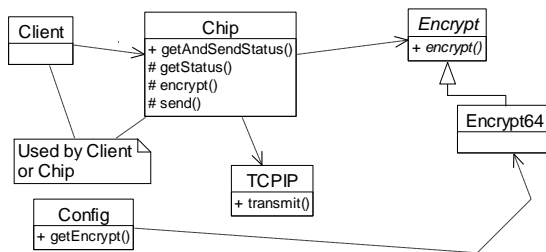
Refactoring to Open-Closed



- First refactor code so can add new function following OCP
- Then add new code

Design Patterns Explained

First, Add Needed Interface and Factory

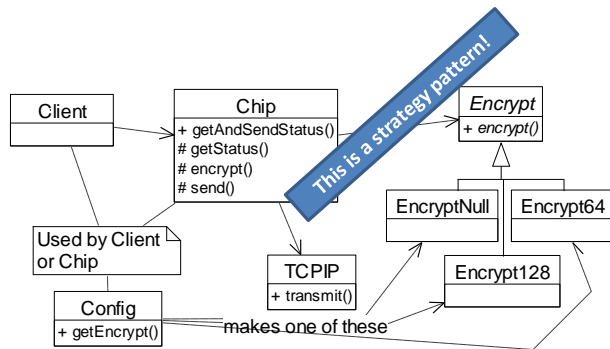


Now, We Can Put in the New Function

- 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

Design Patterns Explained

Then, Add New Implementations



Story 3

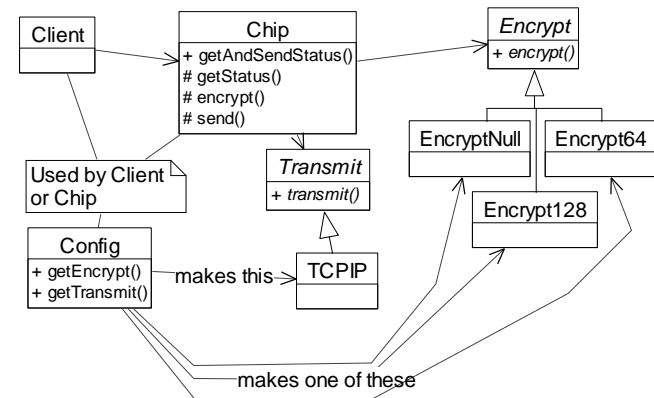
- Support transmission via an e-mail connection. A configuration file will determine which type of transmission to use
- Testability tells us to deal with concept of Transmission (don't have to test all combinations)
- Design Patterns tell us same thing

Design Patterns Explained

Want to Follow Gang of Four Advice

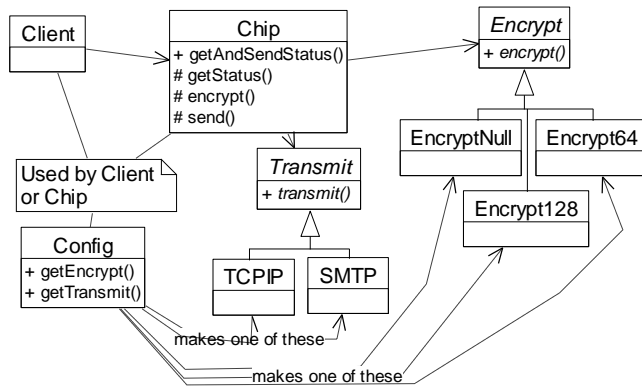
- Because transmission is going to vary, we want to encapsulate it and contain it in the using class (the Chip)
- To implement this, first pull out the existing transmission functionality into its own class
- Note that there is no extra cost caused by us doing this now instead of when we first noticed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

Refactor First



Design Patterns Explained

Add Other Transmitters

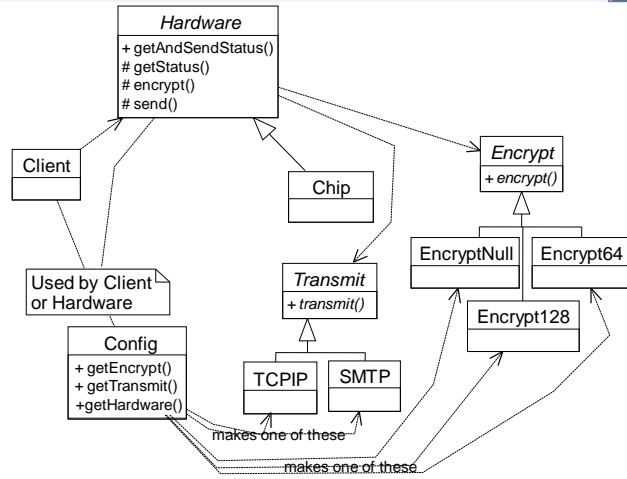


Story 4: Support Cards

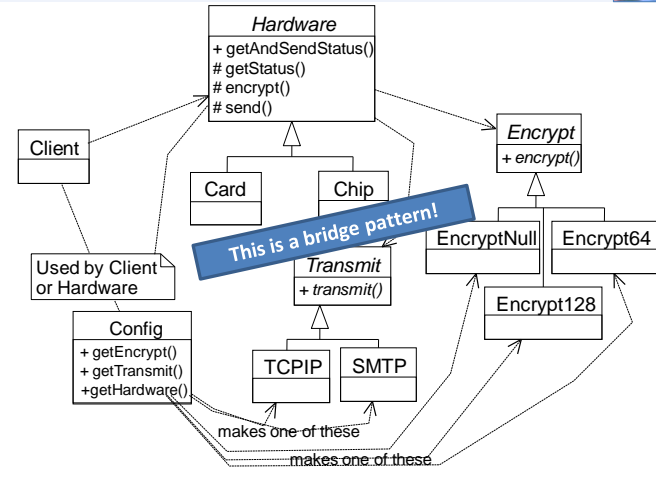
- For testing reasons, we don't want to have to handle different functional test cases when we have Cards or Chips – we want to handle them together
- We want Cards and Chips to appear to be the same to the Client
- We will implement the code with our two-step Refactor-OCP shuffle

Design Patterns Explained

First, Refactor So OCP Can Apply



Then, Add New Function



Design Patterns Explained

A Few Comments

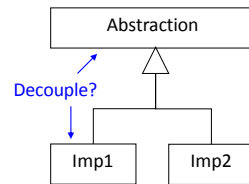
- It is not a coincidence, or just good luck that we could make these changes easily
- It happened because we made sure there was no redundancy and because we kept unrelated things in different classes
- These low level distinctions are easy to see, even if their immediate benefit is not
- We should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



Design Patterns Explained

The Bridge Pattern

- **GoF Intent:** “De-couple an abstraction from its implementation so that the two can vary independently”*
- This generally confuses people, because they think of “abstractions” as abstract super classes, and “implementations” as concrete subclasses
- This is not how the GoF is using these terms



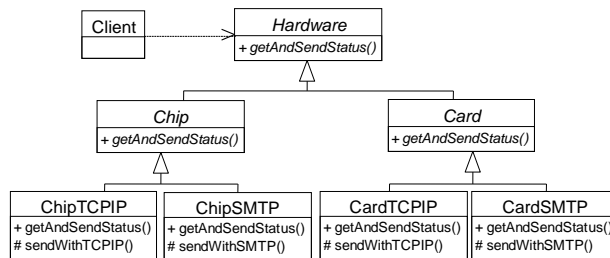
Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995.

The Bridge in Our Case

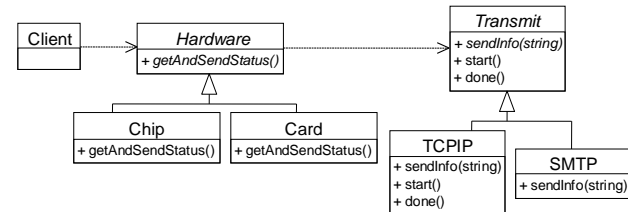
- The Bridge pattern in our case means we separate the abstraction of our main responsibility (different kinds of hardware - Chips and Cards) from **an** implementation of one aspect of its responsibility (TCP/IP communication and e-mail communication)
- The way to do this is by having all of our implementations “look the same” to all of our hardware components

Design Patterns Explained

An Abstraction (*Hardware*) Tightly Coupled to Its Implementation (*Transmission*)

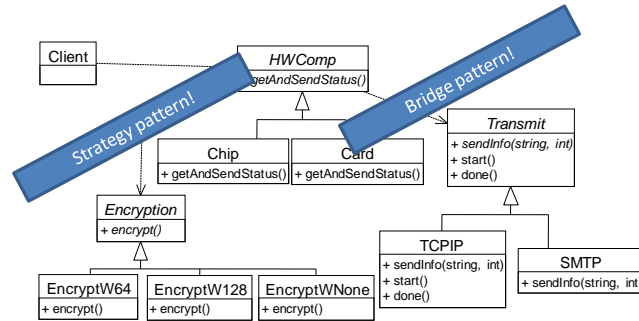


Bridge Would Tell Us to Build It This Way



Design Patterns Explained

Bridge and Strategy Together



Net Objectives
info@netobjectives.com
www.netobjectives.com

Refactoring a Poor Design

20 June 2011

774

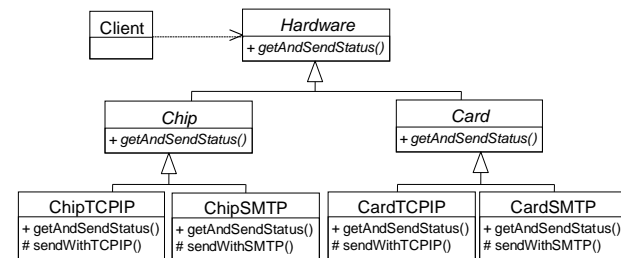
Copyright © 2008 Net Objectives. All Rights Reserved. 20 June 2011

Design Patterns Explained

Can We "Refactor" a Design?

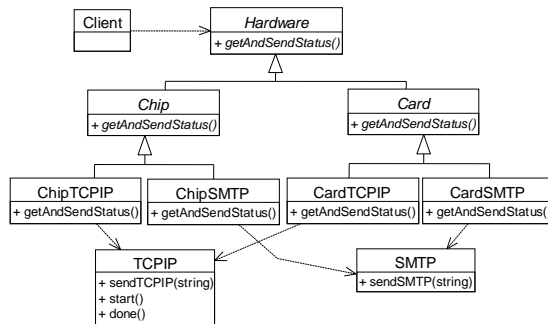
- Let's say we took the original "entities with behavior" approach.
- Although we wouldn't recommend getting into this trouble in the first place, let's see if we can get out of it by refactoring the *design*.

Poor Design as a Result of Using Entities and Behaviors / Specialization

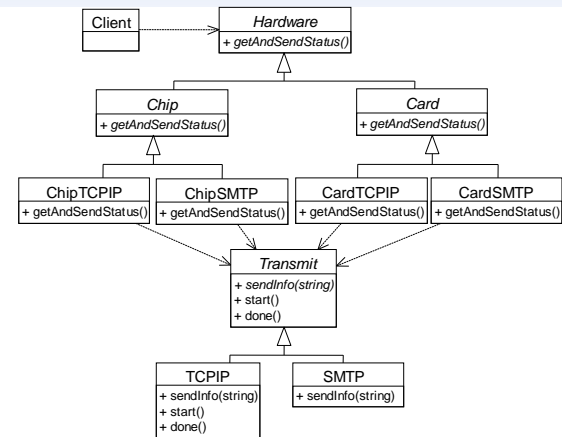


Design Patterns Explained

Pull Out Duplication of Transmission

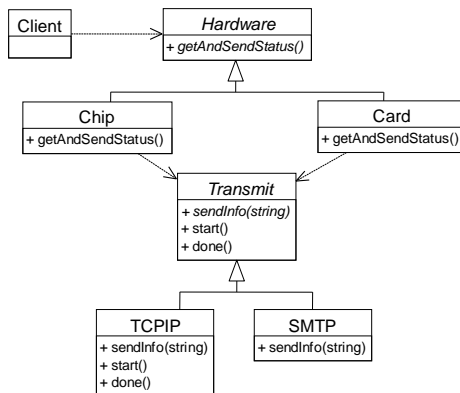


Create *Trans* Class to Simplify Things

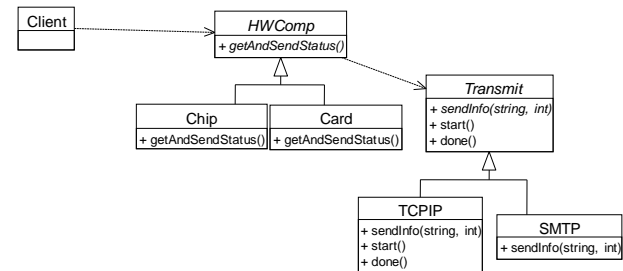


Design Patterns Explained

Derived Classes Now Not Needed

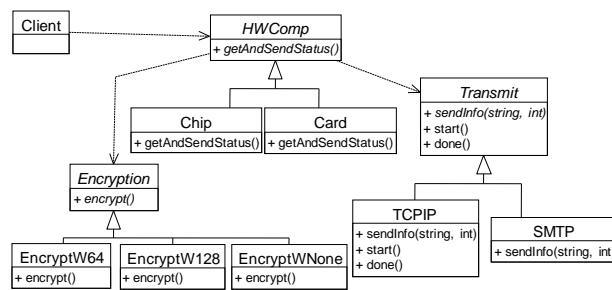


Can Keep Reference in Hardware



Design Patterns Explained

The Full Design: Bridge and Strategy

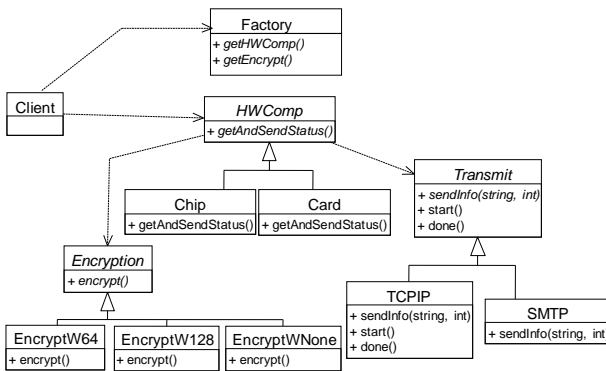


A Universal Context in Software

- What objects you want to use creates the context for the factories that will create those objects
- That is, before you can create something, you need to know what you want to create

Design Patterns Explained

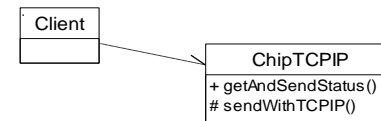
The Full Design with Factory



Results in another kind of encapsulation....

Using Coding Qualities

- You could actually get here by just following coding qualities.
- However, TDD, DPs and CVA are so related to code qualities (and are easier to use) that we recommend using them.
- However, to illustrate, say we had started with:



- and now got another transmitter.

Design Patterns Explained

Handling This in Chip Weakens Cohesion

- We can handle this variation as follows:

```
private void sendStatus(String anInfo) {  
    if (transType== TCPIP) sendStatusTCPIP(anInfo);  
    else sendStatusSMTP(anInfo);  
}
```
- However, this requires the Chip class to remember even more detail 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.
- I might have not worried about where I was, but I won't tolerate getting worse – I'll split out Transmitter.

Using Encapsulation

- Consider how you would break up the functionality into classes where you encapsulated as much as possible
- Implementation encapsulation means you need to pull out encryption
- Design encapsulation means you don't want the Chip to even know there are multiple encryptions (hence, you need to use polymorphism)

Design Patterns Explained

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 code qualities
- "Good Code Qualities" are:
 - Strong cohesion (method, class)
 - Loose coupling
 - No Redundancy
 - Encapsulation
 - Testability
 - Readability
 - Focus

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!

Design Patterns Explained

Other Net Objectives' Talks



Monday

- 8:30-12:00pm Alan Shalloway. *Scaling Agile with the Lessons of Lean Product Development Flow*

Tuesday

- 8:30-12:00pm Alan Shalloway. *Design Patterns Explained: From Analysis Through Implementation*

Wednesday

- 2:30-3:45pm Ken Pugh. *Prefactoring: Extreme Abstraction, Extreme Separation and Extreme Readability*
- 2:30-3:45pm Alan Shalloway. *Getting Executive Management on the Agile Bus*
- 4:00-5:30pm Alan Shalloway. *Avoiding Over and Under Design*

Thursday

- 10:15-11:15 Alan Shalloway. *Effective Agility Across the Enterprise: Patterns of Successful Adoption*
- 2:30-3:45pm Alan Shalloway. *Lean Development Practices for Agile Enterprise*
- 4:00-5:30pm Alan Shalloway. *Developing Competence in Agile Teams*

Come see 3 talks and get a t-shirt And don't forget to see us in the Expo!

Register at www.netobjectives.com/register