Essential Skills for the Agile Developer

A Pareto Approach To Agile Technical Methods

Al Shalloway, CEO
Net Objectives

Al Shalloway
CEO, Founder of Net Objectives

alshall@NetObjectives.com
@AlShalloway

Co-founder of Lean-Systems Society
Co-founder Lean-Kanban University (no longer affiliated)
Contributor to SAFe
Purpose of This Tutorial

• Focus on the basics of Agile Technical Practices
• Assert straightforward, simple practices, can result in huge differences
• These practices are examples of the approach espoused by design patterns
What Tutorial Covers

- Good Code qualities
- Core (essential) Practices
  - Programming by Intention
  - Consider tests before writing code
  - Encapsulation as a design technique
  - Separate Use from Construction
  - Avoiding Duplication
- Essence of design patterns
- Core (essential) Practices Continued
  - Refactor to the Open Close
- Along the way we’ll learn null object, object-pool, strategy pattern

Programmer as Frog
Question to Ask

When working on a mature system consider when adding a new function,

which takes longer

writing the new function or

integrating it into the system?

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?
Code Qualities

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
• A goal: code is easily testable
• Pathology: writing tests is time consuming or a simple change in requirements has test changes ripple significantly
Programming by Intention

"Sergeant" Method

public void printReportOfEmployeesOfCustomer (String CustomerID) {
    Employee[] emps = getEmployees(CustomerID);
    if (needsSorting(emps)) sortEmployees(emps);
    printHeader(CustomerID);
    printFormattedEmployees(emps);
    printFooter(CustomerID);
    paginate();
}

*Note: These methods may not be literally private, as we may need to make some of them public or protected for testing. But we treat them as private from client objects, to limit coupling.
Alternative Methods (Separation of Concerns)

```java
public void printReportOfEmployeesOfCustomer(String CustomerID) {
    Employee[] emps = getEmployees(CustomerID);
    printReport(CustomerID, emps);
}
```

```java
private void printReport(String CustomerID,
Employee[] emps) {
    sortEmployees(emps);
    printHeader(CustomerID);
    printFormattedEmployees(emps);
    printFooter(CustomerID);
    paginate();
}
```

```java
private void sortEmployees(Employee[] emps) {
    if(needsSorting(emps)) doSortEmployees(emps);
}
```

---

Programming by Non-Intention

```java
public void printReportOfEmployeesOfCustomer(String CustomerID) {
    // Get Employees
    Employee[] emps
    ...Make series of calls to get employees
    // Sort Employees
    ... Series of calls to sort
    // Print Header
    ... Series of calls
    //... and so forth for lots of lines
}
```
What Programming by Intention Provides

Method Cohesion
Separation of concerns
  • "Sergeant" method calls other methods
  • Private methods implement details

Clarity
  • Clear code better than comments

Ease in finding/forming certain patterns
No extra work is required

Consider Tests Before Writing Code
Testability

Code that is difficult to unit test is often:

1. **Tightly Coupled**: "I cannot test this without instantiating half the system"

2. ** Weakly Cohesive**: "This class does so much, the test will be enormous and complex!"

3. **Redundant**: "I'll have to test this in multiple places to ensure it works everywhere"

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
Avoiding Duplication

Shalloway’s Law
Shalloway’s Principle

Shalloway’s Law

If ‘N’ things need to change and ‘N>1’, Shalloway will find at most ‘N-1’ of these things
Shalloway’s Principle

Avoid situations where Shalloway’s Law Applies

Example – Null Object Pattern

Motivation
When a behaviour may or may not happen, create an object that does nothing and make it substituteable for one or more other objects that implement actual behavior. Essentially, this redefines "do nothing" as one version of the behavior.

Encapsulation
Null Object encapsulates the fact that nothing will happen under some circumstances.
Encapsulation as a Design Method

The Problem

circa 1999 (when this was avant garde)

Personal investment system.

- See status of stocks, investments, etc.
- Enter buy and sell orders for these
- Done remotely over the web
General Architecture

Mandates and Problems

MANDATE
  • Throughput was the primary concern.

PROBLEM
  • The middleware was likely to be either the bottleneck or the manager of what was the bottleneck.
  • The DEV team had no experience on how to handle loads for an application of this type (meaning TCP/IP)
  • That might not have done me any good anyway as this application would likely have different load requirements anyway.
Questions

What were the right number of TCP/IP connections?
• > 1
• < 100

What would the error rate on be on the TCP/IP connections?
• How often would they go down?

Possible Approaches

Figure it out upfront
• Do lots of analysis
• Maybe even lots of simulations

Try something, learn something, fix it
Pros and Cons of Two Approaches

Figuring it out upfront

• Maybe we wouldn’t be looking at the right things
• Maybe we would overkill it
• Couldn’t show anyone anything
• The Mainframe folks were a known delay. The more time we spent figuring out what was needed the less time we could give them to get us what we needed
• We weren’t sure what we needed to know. Only by getting our feet wet, so to speak, would we know what we needed to know

Pros and Cons Continued

Try something, get feedback, fix it.

• Would discover the problems as we went – would not solve problems we didn’t need to.
• Would give maximum time to Mainframe people to give us what we needed.
• Would know true rate of progress.
• Would appear to the outside world that we were making progress.
Our Approach

Incremental, Iterative, Integrated Development Practices:
• Tackle high-risk issues quickly
• Isolate them if we can’t solve them
• Build an end-to-end application with minimal functionality (proof of concept)
• Defer anything we could defer that wouldn’t be risky

Consistent with:
• Focusing on the most important stuff
• Risk Aware
• Disciplined
• Realistic

Connection Requirements

Establish a robust TCP/IP connection.
Determine the # of TCP/IP connections to be used.
Establish methods to load balance the connections.
Handle errors on the TCP/IP connections.
Risk Issues

1) How much time would we need to become competent with TCP/IP?
2) If we changed the number of connections, would that affect the using code?
3) Would load balancing affect the using code?
4) Would error handling affect the using code?

Mitigating Risk

Wasn’t much we could do about learning TCP/IP – we had to! Issues 2-4 could be mitigated by encapsulating them from the using code.

• Main business logic needed to deal with one TCP/IP connection.
• Issues to isolate:
  1. how it functioned
  2. how many connections were involved
  3. how it handled errors

If we could isolate these, we could solve issue #1 without fear of causing other problems.
There Is No Hope

We were not going to rely on hope. We were not going to guess right (at least I don’t have a track record of doing this).

Had to make my system so we could change the number of connections and add error handling without effecting my code (except maybe in the smallest way).

This implied my client code (the user of the TCP/IP connection) should not be required to know anything about:

• the number of connections
• the error handling being used

Enter “someone else”

If my client code didn’t know, who did?

• “someone else”

We needed a TCP/IP manager. Didn’t want TCP/IP to self manage:

• would have weakened cohesion
• would have coupled management to the connection
The Magic Consultant Card

Instructions
1. Hold magic card face down in front of you
2. State aloud three times what your problem is
3. Flip the card over
4. Read the card
The Magic Consultant Card

**The Good News**
The magic card is always right!

**The Bad News**
It doesn’t tell you how to do what it tells you to do!

This is where design patterns come in.
In many cases a pattern exists to help you see how to solve your problem.

---

**Separate Use from Construction**
Follow Key Principles/Practices
What you hide you can change
Encapsulate construction

What You Hide You Can Change
If one object is coupled to another, then the second object cannot be changed without affecting the first. We say that the second object cannot be freely changed. Cannot be changed without regard to the objects that depend on it.
The nature of the coupling determines the nature of the freedom.
What You Hide You Can Change

We'd like to keep things as clear and simple as we can, by limiting coupling as much as possible.

One way to do this is to try and limit relationships to a single perspective.

One example:

• The perspective of using the object vs.
• The perspective of creating an object

This implies the use of factories to make instances.

A New Principle Emerges

The relationship between any entity A and any other entity B in a system should be limited such that

A makes B or
A uses B,
and never both.
The Players

Port – the class containing the connection logic.
PortManager – the class that will manage Ports

Although there could be many Ports there would be only one PortManager

Interfaces For The Players

Port
• open()
• close()
• send()

PortManager
• getInstanceOfPort()
• returnInstanceOfPort()
Client Code

```cpp
// do stuff before communication
myPort = PortManager::getInstanceOfPort();

myPort->send(message);
...
...

// done with port
PortManager::returnInstanceOfPort(myPort);
```

How Much Design’s Needed For PortManager?

As little as possible

**We could start with just one Port in the PortManager**

- We chose to have an *array of 5 (defined by MaxPorts)*
- We instantiated these in *PortManagers constructor*
- having more than one raised the issues of collections – but we didn’t concern ourselves much with these
The Solution

The Object Pool Pattern

Error Handling

Individual error handling turned out to be easy.

- **Ports** were in Try/Catch blocks.
- On error, ask for another **Port** and try again
- **Port** automatically marked itself as having an error so it wouldn’t get reused.
Lean Approach

Took minimal effort:

- Only needed to get connection going
- `PortManager` logic was trivial
- Error handling control in `Port` was a difficultly – but most of that was TCP/IP learning curve which we couldn’t avoid

Enabled us to get requirements to the Mainframe folks quickly – *without hampering any future work*

Cost of Adding Changes

Since we had been very disciplined about avoiding redundancy, there was only one place each change affected

Since we had encapsulated the implementations to achieve loose coupling, there were no side-effects when changing the code
How Much Extra Work Did This Take By Delaying It?

NONE!

Was This Luck?

NO!
Years Later

Why didn’t you multi-thread?
Cause didn’t know about multi-threading
Note we could change object pool into singleton pointing to a single multi-threaded object.

Advice from the Gang of Four
Gang of Four Gives Us Guidelines*

Design to interfaces

Favor object delegation over class inheritance

Consider what varies in your design … and “encapsulate the concept that varies” (allows for dependency injection)

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

Design to Interfaces: Methods

Craft method signatures from the perspective of the consuming entities

Hides the implementation of your methods

Programming by intention is a systematized way of designing to interfaces
Favor Aggregation Over Inheritance

Define a class that encapsulates variation, contain (via delegation) an instance of a concrete class derived from the abstract class defined earlier.

Find What Varies and Encapsulate It

A varying anything:

- Varying design
- Varying object creation
- Varying relationships (1-1, 1-many)
- Varying sequences and workflows
- Etc...

Can be variations that show up in the future.
Find What Varies and Encapsulate It

Base classes encapsulate their implementing subclasses
This encapsulates varying behavior

Chip → Encryption
   64  128

Refactor to the Open Closed
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


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

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

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

Refactoring to the Open Closed Example

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)
Start with ChipTCPIP Requirement

How would we design this if we designed for testability?

Solution Diagrammed

Note how had we design for testability in the first place we likely would have started with this.
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
First, Add Needed Interface and Factory

```
Client
+ getAndSendStatus()
+ encrypt()
+ send()

Chip
+ getAChipndStatus()
+ getAndSendStatus()
+ encrypt()
+ send()

Encrypt
+ encrypt()

Encrypt64

TCPIP
+ transmit()

Used by Client or Chip

Config
+ getEncrypt()
```

© Copyright 2015 Net Objectives, Inc. All Rights Reserved.
Then, Add New Implementation

Conclusions & Summary

Patterns are More than “Solutions to recurring problems within a context”.
“At this final stage, the patterns are no longer important: the patterns have taught you to be receptive to what is real”
- both from Christopher Alexander

Design to interfaces
• Programming by intention
• Consider tests before writing code

Encapsulate what Varies
• Encapsulation as a design technique
• Separating use from construction
• Refactor to the open closed

Patterns can be used to avoid duplication
The Net Objectives Patterns Repository

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Strategy</th>
<th>Bridge</th>
<th>Template Method</th>
<th>Null Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Decorator</td>
<td>Chain of Responsibility</td>
<td>Template Method</td>
<td></td>
</tr>
<tr>
<td>Workflow</td>
<td>Template Method</td>
<td>Visitor</td>
<td>Bridge</td>
<td>Null Object</td>
</tr>
<tr>
<td>Cardinality</td>
<td>Decorator</td>
<td>Chain of Responsibility</td>
<td>Proxy</td>
<td>Observer</td>
</tr>
<tr>
<td>Construction</td>
<td>Singleton</td>
<td>Abstract Factory</td>
<td>Builder</td>
<td>Factory Method</td>
</tr>
<tr>
<td>Selection</td>
<td>Chain of Responsibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Composite</td>
<td>Template Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity</td>
<td>Facade</td>
<td>Adapter</td>
<td>Proxy</td>
<td></td>
</tr>
<tr>
<td>Relationships</td>
<td>Observer</td>
<td>Command</td>
<td>Mediator</td>
<td>Visitor</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Mock Object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Net Objectives Pattern Repository


Lean for Executives
Product Portfolio Management
Business Product Owner
Product Owner

Lean Enterprise

Kanban / Scrum
ATDD / TDD / Design Patterns
Lean Management
Project Management

Onsite SPC
Leading SAFe
SAFe Architecture
PM/PO

Q&A

ASSESSMENTS
CONSULTING
TRAINING
COACHING

Net Objectives SPC Training in Seattle, Oct 27-30
www.netobjectives.com/events