Net Objectives
ESSENTIAL WHITE PAPERS

Shalloway's Law

excerpted from **Essential Skills for the Agile Developer**
by Alan Shalloway, Scott Bain, Ken Pugh, and Amir Kolsky

Shalloway's Law

by Alan Shalloway, Scott Bain, Ken Pugh, and Amir Kolsky

A Net Objectives Essential White Paper

Net Objectives Press, a division of Net Objectives

1037 NE 65th Street Suite #362

Seattle, WA 98115-6655

404-593-8375

Find us on the Web at: *www.netobjectives.com*

To report errors, please send a note to *info@netobjectives.com*

10 9 8 7 6 5 4 3 2

Printed in the United States of America

This article is excerpted from *Essential Skills for the Agile Developer: A Guide to Better Programming and Design* by Alan Shalloway, Scott Bain, Ken Pugh, Amir Kolsky. Addison-Wesley Professional: 2011.

CHAPTER 4: SHALLOWAY'S LAW

A few years ago someone in one of my Design Patterns classes mentioned I should name something after myself since I had written a successful book on design patterns. I, of course, liked this person and his idea immediately. So I went about thinking about what would be appropriate. The best I could come up with was:

"When N things need to change and N>1, Shalloway will find at most N-1 of these things."

While I had hoped to find something complimentary, this was the most appropriate thing I could come up with. I point out that I didn't ask for this when I was born – I was given this "ability." Most people also have this trait. In other words, this isn't choice – it's how we are. This means we had better pay attention to it. Otherwise, we'll find that if we write code that requires finding more than 1 thing, we won't find them all, but our customers (or if we're lucky, someone else on our team) will.

While I am not particularly proud of Shalloway's Law I am proud of Shalloway's Principle which I came up with to deal with it. Shalloway's Principle states:

"Avoid situations where Shalloway's Law applies."

Kent Beck's famous "once and only once rule" is one approach to this – in other words, keep N at 1 – but not the only one. While avoiding redundancy is perhaps the best way to follow Shalloway's Principle, it is not always possible. Let's begin by looking at different types of redundancy and see how we might avoid them, or if not, how we can still follow Shalloway's Principle.

TYPES OF REDUNDANCY

Copy and Paste

This is the most obvious type of redundancy and probably easiest to avoid. Using functions is a common way to avoid this.

Magic Numbers

This is not quite as obvious as redundancy, but it is. Basically the redundancy is that what the magic number means must be known everywhere the magic number is used. How to avoid magic numbers is well known – just use "#defines "or "consts" or their equivalent depending upon your language of choice.

WHAT IS REDUNDANCY?

Redundancy can be much more intricate than what people initially think. The definition of redundancy I am referring to here is – "characterized by similarity or repetition".

I would suggest redundancy can be fairly subtle and to define it as duplication or repetition is not sufficient. Defining it as similarity, unfortunately, can be a bit vague – so perhaps it isn't that useful either. I propose a definition of redundancy in code that I believe is very useful –

"Redundancy is present if when you make a change in one place in your code, you must make a corresponding change in another place."

Alan Shalloway is the founder and CEO of Net Objectives. With over 40 years of experience, Alan is an industry thought leader in Lean, Kanban, product portfolio management, Scrum and agile design. He helps companies transition to Lean and Agile methods enterprise-wide as well teaches courses in these areas. Alan has developed training and coaching methods for Lean-Agile that have helped Net Objectives' clients achieve long-term, sustainable productivity gains. He is a popular speaker at prestigious conferences worldwide. He is the primary author of Design Patterns Explained: A New Perspective on Object-Oriented Design, Lean-Agile Pocket Guide for Scrum Teams, Lean-Agile Software Development: Achieving Enterprise Agility and Essential Skills for the Agile Developer. Alan has worked in literally dozens of industries over his career. He is a co-founder and board member for the Lean Software and Systems Consortium. He has a Masters in Computer Science from M.I.T. as well as a Masters in Mathematics from Emory University. You can follow Alan on twitter @alshalloway

A little reflection will tell us that redundancy, at least defined this way, is almost impossible to avoid. For example, a function call has redundancy in it. Both the calling and defined statement must be changed if either change. From this we can also see the relationship between redundancy and coupling. And, as with coupling, not all redundancy is bad or even avoidable. I would say the type of redundancy you must avoid is that redundancy that violates Shalloway's Principle.

Redundancy that doesn't violate Shalloway's Principle is likely to be a nuisance at most. For example, in the case above, I can have a function called from any number of places. Doing so has my system have a significant amount of redundancy. However, this doesn't violate Shalloway's Principle? Why? Because if I change the defining statement, the compiler will generate a "to do" list for me to change my calling statements. I still, of course, have work to make my changes, but that is considerably different from the dangerous situation I would be in if I had to also find the changes that were required.

OTHER TYPES OF REDUNDANCY

Given our new definition of redundancy, what are other common forms of it (and how do we avoid them)? Implementations are often redundant even if the code making them up are not duplicates of each other. For example, if a developer takes a function and copies it (clearly redundant at this point) but then changes all the code (presumably removing the redundancy) because the implementation of the new function is different – do you still have redundancy? I would suggest you do. Not of the implementation, but most likely the algorithm you are implementing. The second function was copied from the first one presumably because the flow of both algorithms were the same - only their implementations were different.

How do you remove this type of redundancy? I'll refer to Design Patterns Explained: A New Perspective on Object-Oriented Design's – The Template Method Pattern. Basically, it involves putting the algorithm in a base (abstract) class and having the implementations of each step be in derived (extended) classes.

THE ROLE OF DESIGN PATTERNS IN REDUCING REDUNDANCY

We often talk about the purpose of design patterns is to handle variation. Many patterns are readily identified as doing this:

- Strategy handles multiple algorithms
- Bridge handles multiple implementations
- Template Method handles multiple implementations of a process
- Decorator allows for various additional steps in a process

Most of the design patterns in the seminal work – Design Patterns: Elements of Reusable Object-Oriented Software are about either directly handling variations or support handling variations.

Another way to think of the use of design patterns is that they also eliminate the redundancy of having to know which implementation is being used.

Because design patterns handle variations in a common manner, they can often be used to eliminate redundant relationships that often exist in a problem domain. For example, a purchasing/selling system will have several types of documents and payment types. Each document type may have a special payment type but the relationship between them is probably similar to the relationship between any other pair. This sets up redundant relationships. By using abstract classes and interfaces, redundancies can be made explicit and allow the compiler to find things for you. For example, when an interface is used, the compiler will ensure that any new method be defined in all cases – you won't have to go looking for them.

FEW DEVELOPERS SPEND A LOT OF TIME FIXING BUGS

A common misconception amongst software developers is that they spend a lot of time fixing bugs. But on reflection, most realize that most of their time is spent in finding the bugs. Actually fixing them takes relatively little time. One of the reasons people spend a lot of time finding bugs is that they have violated Shalloway's Principle. If you can't find the cases easily, bugs will result.

A key to avoiding this problem is to be aware of when you are violating Shalloway's Principle. Here's an interesting case. Let's say you've been using an Encrypter class in your code. If you've been following our suggestion of separating use from construction you may have code that looks something like this:

```
public class BusinessObject {

   public void actionMethod() {

      AnotherObject aAnotherObject=

         AnotherObject.getInstance()

      String aString;

      String aString2;

      // Other things

      Encrypter myEncrypter=
         Encrypter.getEncrypter();

      //

      myEncrypter.doYourStuff(
         aString);

      //

      aAnotherObject( myEncrypter);


      //

      myEncrypter.doYourStuff(
         aString2);

   }

}

public class AnotherBusinessObject {

   public void actionMethod( Encrypter
      encrypterToUse) {

      // Other things

      //

      //

      encrypterToUse.doYourStuff(
      aString);

   }

}
```

Now let's say there become a case where we don't need to use the Encrypter. We might change the code from:

```
// Other things

   Encrypter myEncrypter= Encrypter.
      getEncrypter();
```

to

```
      Encrypter myEncrypter;

      If (<<need an encrypter>>)

      myEncrypter= Encrypter.
      getEncrypter();
```

Then, of course, we have to go through our code and see when we don't have an encrypter:

```
   if (myEncrypter != null)

   myEncrypter.doYourStuff( aString);
```

At some point we'll hit the second case of this. This means Shalloway's Law is in effect. By the way, a corollary to Shalloway's Law is "If you find two cases, know you won't find all of the cases." At this point, we should figure out a way not to have to test for the null case. An easy way is to put the logic in the getEncrypter method in the first place. In other words, have Encrypter's getEncrypter method have:

```
// NullEncrypter derives from
Encrypter but does no encryption

If (<<don't need an encrypter>>)
return new NullEncypter();
```

This first of all, keeps all the knowledge about the construction of the encrypter out of the calling class. It also eliminates the need for checking in the null condition – both avoiding Shalloway's Law and de-coupling the client code from the Encrypter object.

This, by the way, is the Null Object Pattern. I would suggest that anytime you find you are doing a test for null more than once, you should see if you can use this properly.

I suspect that many readers will think this example somewhat contrived because with a factoring making the Encrypter object it is pretty clear that the test for a null case should be handled in there.

But this is also my point – when you separate use from construction you are more likely to make better decisions later on. If a getEncrypter wasn't being used, and the client code had the rules of construction, never setting the myEncrypter reference would likely occur.

REDUNDANCY AND OTHER CODE QUALITIES

It's useful to note how redundancy is related to other code qualities. In particular, coupling and testability. Anytime you have redundancy, it is likely that if one of the occurrences change, the other one will need to change. If this is the case, these two cases are coupled. Coupling and redundancy are often different flavors of the same thing.

Note that redundancy also raises the cost of testing. Test cases can often be reduced if redundant relationships are avoided. Let's consider the following case. Note that each of the service objects are doing conceptually the same thing, but are doing it in different ways (e.g., different kinds of encrypting).

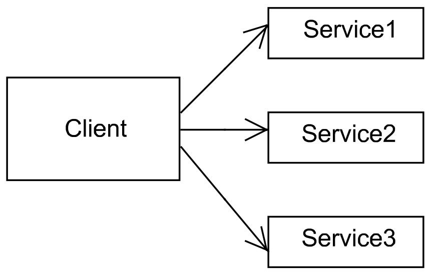


Figure 4.1. Testing in a 1 to many relationship

Note that we need to have the following for a full set of tests:

- Test of Service1
- Test of Service2
- Test of Service3
- Client using Service1
- Client using Service2
- Client using Service3

The need for testing Client using the services is because we have no assurance that we've abstracted out the service code. There may be coupling taking place – especially since each service interface may be different. Notice what happens when more clients become involved – this gets worse and worse.

Now, consider what happens if we make sure that all of the service objects work in the same way. In this case, we basically abstract out the service objects. If we put in an abstraction layer (either an abstract class or an interface that the services implement) we get what is shown in figure 4-2.

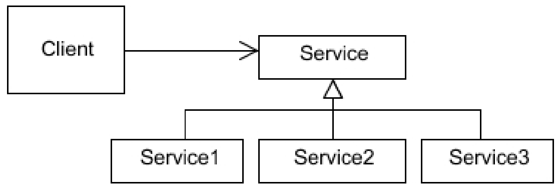


Figure 4.2. Creating a 1 to 1 relationship

While we still need to test each Service, we now only need to test the Client to Service relationship. Note that as we get more client objects the savings are even greater.

SUMMARY

Shalloway's Law is both humorous attempt at saying avoid redundancy while giving developers some guidance in how to do so – or at least to make it less costly not to do so. Understanding redundancy is key to Shalloway's Law and avoiding the cost of it is the essence of Shalloway's Principle.

A powerful question when programming that can be deduced from all of this is – "if this changes, how many places will I have to change things and can the compiler find those for me?" If you can't see a way to make it so the answer is either "1" or "yes" then you have to acknowledge that you have a less than ideal design. At this point you should consider an alternative – or, heaven help you – ask someone else to suggest an alternative.

OTHER ARTICLES OF INTEREST

Go to *www.netobjectives.com/articles* to see the following articles:

- The Business Case for Agility
- Can Patterns be Harmful?

Check out the Net Objectives Portal at *portal.netobjectives.com* where an extensive amount of online self-study is available.

Drive from Business Value

BUSINESS-DRIVEN SOFTWARE DEVELOPMENT

Business-Driven Software Development is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. This approach has a consistent track record of delivering higher quality products faster and with lower cost than other methods.

Business-Driven Software Development goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

Our approach integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. Here are some key elements:

- Business provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment.
- Teams self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed.
- Management bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality.

BECOME A LEAN-AGILE ENTERPRISE

Involve all levels. All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

WHY NET OBJECTIVES

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place, such as focusing on the team when that is not the main problem, or using the wrong method, such as using Scrum or kanban because they are popular.

Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban) and integrates business, management and teams. This lets us help you select the right method for you.

LEARN TO DRIVE DEVELOPMENT FROM THE DELIVERY OF BUSINESS VALUE

What really matters to any organization? The delivery of value to customers. Most development organizations, both large and small, are not organized to optimize the delivery of value. By focusing the system within which your people are working and by aligning your people by giving them clear visibility into the value they are creating, any development organization can deliver far more value, lower friction, and do it with fewer acts of self-destructive heroism on the part of the teams.

THE NET OBJECTIVES TRANSFORMATION MODEL

Our approach is to start where you are and then set out a roadmap to get you to where you want to be, with concrete actionable steps to make immediate progress at a rate your people and organization can absorb. We do this by guiding executive leadership, middle management, and the teams at the working surface. The coordination of all three is required to make change that will stick.

OUR EXPERTS

Net Objectives' consultants are actually a team. Some are well known thought leaders. Most of them are authors. All of them are contributors to our approach.

Al Shalloway

Alan Chedalawada

Guy Beaver

Scott Bain

Max Guernsey

Luniel de Beer

SELECTED COURSES

Executive Leadership and Management

Lean-Agile Executive Briefing

Preparing Leadership for a Lean-Agile/SAFe Transformation

Product Manager & Product Owner

Lean-Agile Product Roadmaps

PM/PO Essentials

Lean-Agile at the Team

Acceptance Test-Driven Development

Implementing Team Agility

Team Agility Coaching Certification

Lean-Agile Story Writing with Tests

Technical Agility

Advanced Software Design

Design Patterns Lab

Effective Object-Oriented Analysis and Design

Emergent Design

Sustainable Test-Driven Development

DevOps

DevOps for Leaders and Managers

DevOps Roadmap Overview

SAFe®-Related

Implementing SAFe with SPC4 Certification

Leading SAFe® 4.0

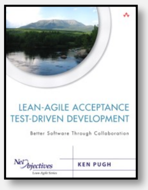
Using ATDD/BDD in the Agile Release Train (workshop)

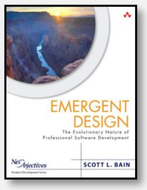
Architecting in a SAFe Environment

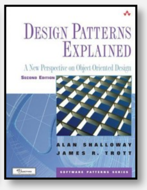
Implement the Built-in Quality of SAFe

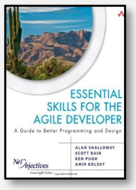
Taking Agile at Scale to the Next Level

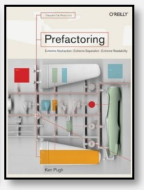
OUR BOOKS AND RESOURCES

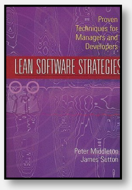


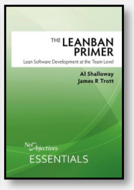


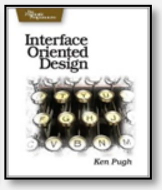












Net Objectives UNIVERSITY

CONTACT US
info@netobjectives.com
1.888.LEAN-244 (1.888.532.6244)

LEARN MORE
www.NetObjectives.com
portal.NetObjectives.com

Net Objectives