Growing Object-Oriented Software Guided by Tests
By Steve Freeman and Nat Pryce

Chapter 1: What is the point of Test Driven Development •

Paul Grenyer, 13 Feb 2011

Nat and Steve start off by telling us that software development is a learning process and of course it is. Every single one of us will be learning something on a daily basis. They go on to tell us that feedback is a fundamental tool for developers and give some good examples of feedback in the development process.

Next they get into the heart of the chapter and start describing not only the mechanics of unit testing, but the driving forces behind it and benefits: catching of regression errors and simplification of code. Pretty soon they get to the TDD cycle: Write a failing unit test, Make the test pass, Refactor , Repeat. And of course the state the golden rule of unit testing:

"Never write new functionality without a failing test"

I was a little disappointed at this stage. I have often questioned whether this approach is necessary to be doing TDD and I have frequently been told that it is. How many of us actually do it? I see myself as writing software guided by tests (sound familiar?), where I have excellent test coverage, loosely coupled and cohesive classes, but rarely write a test first. It's guided by tests as I have to think how I'm going to test the code as I write it. Maybe they'll discuss something similar later in the book.

Nat and Steve explain Refactoring very well. Far too many people use it to describe changing the behaviour code, which of course it is not.

Many of the early TDD books got hung up on unit testing. They had to really as they were introducing what for some was a radically new concept. Nat and Steve expand the sphere of testing to include acceptance testing, integration testing and end-to-end testing as well as unit testing.

External and internal quality is something I have been aware of for sometime, but I have never before seen it described so well. We all know that it's external quality that pays the bills, but without internal quality, external quality is more difficult to achieve consistently over the lifetime of a project.

Finally they discuss coupling and cohesion. I always have difficulty remembering exactly what cohesion is, although I strive for it every time I program. The description here has cleared this up. I will always think of washing dishes and clothes in the same machine!

This is a great introductory chapter to what promises to be a great book.

Chapter 2: Test-Driven Development with Objects •

Ian Bruntlett, 17th Feb 2011

1. A Web of Objects (liked this).

This section describes OOD (Object-Oriented Design). When working on software, I try to make it weakly coupled and highly cohesive (the most important thing Reg my Software Engineering lecturer ever told me). Just before this chapter starts, there is a good explanation of what coupling and cohesion are.

However, when describing OOD - "Object-Oriented Design focuses more on the communication between objects than on the objects themselves", I feel they are throwing out the baby with the dirty water. Objects need to be designed well – 1) "An element's cohesion is a measure of whether its responsibilities form a meaningful unit" and 2) "Features with 'high' coherence are easier to maintain".

I find some of the terminology unhelpful. I'm happy enough with calling object's member function as "methods" but when its referred to as "message passing" it makes me think of networked systems communicating over a network.

2. Values and Objects.

I seem to remember Kevlin Henney (http://www.curbralan.com/) giving a talk about value-based programming at an ACCU conference. IIRC, he said that an object lies along 3 dimensions – identity, state and behaviour. This paragraph declares "distinguish between values that model unchanging quantities or measurements and objects that have an identity". I'm troubled with the "unchanging quantity" - if I implement a class to model some kind of measurement/attribute I don't think it would be helpful if it could not have its value changed during the runtime of a system.

3. Follow the Messages.

This is about the dynamic behaviour of a system. Previously people have written about CRC diagrams (Class, Responsibility, Collaborations) – for an example, see http://accu.org/index.php/journals/477 for my take on CRCs.

While developing systems from a dynamic point of view, you use something similar - "Candidates, Responsibilities and Collaborators". That is covered elsewhere - "Object Design: Roles, Responsibilities and Collaborations" (Wirfs-Brock).

4. Tell, Don't Ask.

"the calling object should describe what it wants in terms of the role that its neighbour plays, and let the called object decide how to make that happen". I agree completely.

5. But Sometimes Ask

"We try to be sparing with queries on objects (as opposed to values) because they can allow information to "leak " out of the object, making the system a little bit more rigid". I agree completely.

6. Unit-Testing the Collaborating Objects.

This is the core aspect of TDD and I'm curious about the effect of TDD when implementing business-domain logic. To quote the chapter, “One option is to replace the target object's neighbours in a test with substitutes, or mock objects”. My question is: given a set of interacting objects that interact via methods (in C++), should those interactions be modelled as methods or a class hierarchy of some kind? Or is a better approach to say “this group of objects is a fundamental unit and we will implements mock objects at the borders at that unit”?

7. Support for TDD with Mock Objects.

This is intriguing and I am looking forward to the rest of the book.
Conclusion
Despite the Handwavium, overall, I like this chapter :)

Chapter 3: An Introduction to the Tools •

Andrew McDonnell, 24 Feb 2011

This provides a general overview of JUnit 4, Hamcrest, and jMock2. There's a good amount of code in this chapter, which is important for understanding how things work, but I'm not going to reproduce it here. I thought I had lucked out when I saw how short the chapter was, but it turns out to have a lot of information, in addition to two related appendices! However, we're told the technical details are less important than the concepts and motivation at this point.

JUnit

To get the basic definitions out of the way, JUnit is a unit testing framework for Java. You write code that does something to the object under test (OUT for brevity) and assertions describing the results; it runs the code and tells you when assertions fail.

For C++ purposes, one can expand this definition to allow for non-member functions with only primitive input and output, but that's not particularly relevant to the whole object-oriented thing we're doing here.

The test fixture is the fixed state at the start of the test. This must be reproducible for the test to be reproducible.

Technical Details:

- A new instance of the test class is created for each test. You can do extra initialization with a setup method, annotated with @Before, and cleanup work with a teardown method (@After annotation).
- Tests are methods annotated with @Test in a test class, with no return value and no parameters.
- The @Test annotation can also be given the "expected" argument, which causes the test to fail if the expected exception is not thrown.
- The test runner takes care of finding and running tests, and is specified with the @RunWith class annotation.

Hamcrest

Hamcrest matchers are objects that describe certain criteria, determine if a test object meets it, and if not can explain what was expected. Hamcrest (I love the name, it makes think of a huge bacon wave) matchers make life more pleasant on both sides of the test: 1) naturally readable test code, and 2) naturally readable test failure messages.

1. JUnit has an `assertThat()` method that takes the test object and a matcher. Because matchers typically come with a factory method named so that it fits grammatically in an assertThat, you can write code that reads like a sentence:
   ```
   assertThat( testString, containsString( "bananas" ) );
   ```
   instead of coding the string comparison directly. This is especially nice with more complicated criteria.
2. They provide readable failure messages, showing the intended and actual outcome ("Expected … but it …").

Technical Details:

Hamcrest comes with a number of ready-made matchers, but see Appendix B for how to write your own.

jMock

jMock is a mock object framework. It allows you to create mock objects, and then describe how the OUT should invoke them when the OUT is exercised in a certain way (expectations). JUnit together with the jMock test runner will then exercise the OUT and check if the expectations are met. The key concepts are mockery, mock objects, and expectations:

- The mockery is the context of the OUT: the mock objects it will send messages/tell (not ask!) things to.
- Mock objects are fake objects, specifically fake objects that track how they're invoked during the test.
- Expectations are declarations of how the mock objects should or should not be invoked. There's a rich set of expectations you can describe.

So a mock object test is a jUnit test that defines mock objects, does things to the OUT that will exercise the mock objects' interfaces, and describes your expectations for how it should exercise them. The test checks your expectations - there's often no need for any JUnit assertions.

Technical deatils:

See this chapter and Appendix A for more details. Here are the types of expectations you can define:
Invocation count: How many times a method was called
Arguments: What arguments the method received, as values or matchers
Sequence: The order in which methods were invoked
Action: what the mock object should do on invocation, for example return a value to the OUT
State: define a state machine and when transitions will occur, and say what state the machine should be in when a method is called

jMock also comes with some ready-made matchers for arguments, such as not(Matcher), which negates a matcher. I like the attention to creating a readable flow, for example providing identical `a(Class<T>)` and `an(Class<T>)` matchers.

Chapter 4: Kick-Starting the Test-Driven Cycle •

Joes Staal, 28 Feb 2011

This chapter is about how to get the system set up so that test-driven development can start. The main point is to get the infrastructure up and running so that end-to-end tests can happen. The first test should be 'a walking skeleton', i.e. as minimal as possible, because focus is on getting the automatic build, deployment and testing working. After the skeleton has started walking the first failing acceptance test for a meaningful feature can be written.

The important thing of the end-to-end process is that it closes the feedback cycle. Everything that is not understood will be exposed. Since it is often impossible to answer most questions raised so early in the development cycle, the infrastructure can only implement what is currently understood. That is fine, as long as it is acknowledged that changes may be needed later on when more information is gained so that true end-to-end testing can be achieved.

To get an idea of what is needed to set up the walking skeleton some idea is needed of the high-level structure of the application. A broad-brush picture of major components is needed for the first release. As rule of thumb, it should be able to sketch the design of the walking skeleton in a few minutes on a whiteboard. It can be a good idea to have a public drawing of the system's structure in the development team's work area.

Sketching out an early structure of the application serves only to get the skeleton walking, it is not meant as 'Big Design Up Front' (BDUF). Most ideas drawn up are likely to be wrong and will be adjusted while going along in the development cycle. The idea is to make as few decisions as possible to kick-start the TDD-cycle.

Having the walking skeleton allows early feedback from real customers and other teams giving better understanding of the problem domain.

Setting up the infrastructure can take more time than anticipated and stakeholders may have the opinion that nothing has been, since a lot of work is gone into building something that doesn't do a lot. Furthermore, if the stakeholders are used to late integration they may be worried about the amount of difficulties that have to be solved so early, while they are used to problems arising later at the end of the project.

However, the walking skeleton approach solves a lot of problems that will not have to be addressed later on. After the first features have been implemented the work should get into a routine where things seem just to work.

Main conclusion of the chapter is that this approach gives a sense of direction and concrete implementation to test assumptions underlying the software. An advantage of setting up the whole cycle so early is that it addresses issues at the start of the project when time, budget and willingness to solve them are still available.

The chapter ends with a side box on 'Brownfield Development', where a system has to be built on top of an already existing system without tests (or has to extend such a system). The book by Feathers (Working effectively with legacy code) is mentioned, which gives a lot of information on how to get such systems under test. It is important to get automatic builds and deployment working. The next step is to find the simplest path through the system. When that has been identified an acceptance test can be added that plays the role of the walking skeleton (but is more meaningful, though probably not as simple).

I found this a very interesting chapter with a lot of sensible information. Especially the (social) aspects regarding the stakeholders are in my opinion often overlooked.

Chapter 5: Maintaining the Test-Driven Cycle •

John Penney, 3 Mar 2011

This chapter takes us beyond the initial phase of development and into the cycle of iterative software development, maintaining the test-driven approach as our software grows. It covers a lot of ground and makes lots of good points which are expanded upon later in the book. I'll pick out those points that particularly interested me.

Just as we started our walking skeleton with an acceptance test, so we continue to grow our software by starting each iteration with a failing acceptance test. The authors emphasise the importance of writing the test using application domain language. I'd certainly agree that having a test that a user/customer can read and understand is a powerful benefit.

Given our recent discussions about what constitutes a regression test, there's a useful perspective on this in Chapter 5. As development progresses, acceptance tests turn from red to green, providing a satisfying measure of progress for developer and customer. Green acceptance tests continue to be run regularly in order to catch regressions. The authors suggest organising the tests into a "in progress" suite which reflects the up-coming and ongoing work, and a "regression" suite.

When we first write the test it will fail: you can learn a lot from the way the test fails, and the authors emphasize this. Rather often in my experience the test will actually pass! It's possible that this is because the feature is already there, but rather more likely that it's because the test was wrong. Equally, the test may be failing because the test data was set up wrongly, or the comms had failed, or the SUT wasn't actually in the state you expected. And the authors rightly point out that this is all a great opportunity for clarifying the tests, failure messages and diagnostics. I'll certainly be giving this more attention in the future.

So having got a failing acceptance test, where next? The advice is avoid the temptation to unit test domain model objects and services, but instead to let the acceptance test guide the development of the code, working from the inputs defined by the test, until eventually you've generated the outputs expected by the test. This "outside in" approach is really quite alien to me: but I have been trying it on my most recent project, and so far so good! I'm looking forward to the worked examples in later chapters of GOOS.

The chapter concludes with a discussion of how to achieve a balance between unit tests and integration tests, admitting that there may not be a single "right" answer. The best approach is to "regularly reflect on how well TDD is working": IMO every team should be doing regular retrospectives, and consideration of the team's TDD practises should be part of this.

Chapter 6: Object Oriented Style •

Timothy Wright, 7 Mar 2011

This chapter discusses how the authors think about the object oriented design they are growing. They outline heuristics used to guide this structure.

There are two principle heuristics:

1) Separation of Concerns:

This is common from Uncle Bob's books. A class should have one and only one reason to change. I have always understood this conceptually but harder to implement in practice. I can separate code that unpacks code from an Internet standard protocol and the business code that handles these messages. They both won't change for the same reason. The internet protocol code won't change because the standard is fixed, not likely to change. But the business code could change for many reasons. I think it is working with the domain experts and the history of the domain to know where changes could come from. This is an educated guess and best we can do.

In the code I work on, the standard model of the accelerometer has always been one input and one output for 20 years. Now, we have the requirement that the accelerometer could, but not always output two data channels instead of one. This takes more work, since my code did not anticipate this and it shouldn't have. Of course I only have a few weeks to make these changes :-). I guess my question is: is there a way to separate concerns that deals with today's problems that make the future problems easier, not easy but easier. I think there is but still hard in practice.

2) Higher Levels of Abstraction:

This is the one I am personally working on. It is hard on legacy code. The idea is that code should be written in layers. Each layer should have a specific level of abstraction. If one layer talks about streets and houses, the next layer could talk about cities, power grids etc. Each layer only uses its domain language.

Later in the chapter, the book uses the description of nested Russian dolls as a metaphor for abstraction levels.

I think the key is the domain language. Having a set of terms that everyone agrees to the meaning helps define the domain and the naming of the objects in the code. We have code we are constantly rewriting partially because it is hard to come up with terms without ambiguous meanings to define the layers. Every rewrite works okay but the code is still hard to read and understand. I do think in each rewrite, it does get a little better.

These two heuristics should push the structure toward Cockburn's ports and adaptors architecture. We write interfaces for the application model and adaptors that communicate between the application and the technical concepts like databases etc.

The next heuristics help flush out the behavior in the interfaces.

Side Bar Encapsulation and Information Hiding:

Encapsulation and information hiding are not the same thing. Encapsulation puts all behavior behind the APIs of the class. The class state can only change with these published methods. Information Hiding keeps all the inter workings of the class within the methods, so other objects that use it won't make bad assumptions. Keeps the abstraction level. I have to admit that I think I understand the point but a little confused at a gut level.

Standard practices to keep encapsulation: no global variables, no singletons, copy collections and mutable value objects between objects.

Internals vs. Peers:

We communicate between objects by passing messages. The objects that communicate directly are its peers. This helps maintain encapsulation and an appropriate abstraction level. How it communicates is very important because it can directly affect it's coupling with other objects. If an object exposes too much, other objects could do some of its work creating unnecessary coupling. We could get something like the train wreck example.

```
((EditSaveCustomizer).master.getModelisable().getDockablePanel().
getCustomerizer().getSaveItem().setEnabled(Boolean.FALSE.booleanValue())
```

This calls to other objects creates too much coupling, making changes very hard.

Side Bar: Different Levels of Language

The authors talk about creating small helper functions to make the code clearer and more readable.

In their experience, their code tends to use message style passing between objects and more functional style within a object.

I use similar small helpers in C++. From a constructor like Model::Model(int volts, int amps) to a construction like Model::Model(Volts, Amps) where Volts and Amps are simple classes that enforce the order and make the code easier to read. I thought that in Java or C# with a form of intellisense, you can see the signature easily but there is no compile time enforcement. It is interesting to see this type of helper classes in other parts of the code too. I think making the code easy to read passively without IDE support has value.

No Ands, Ors or Buts:

If a description of a class's responsibility must include an 'and', 'or' or 'but' then we know that there are too many responsibilities. There are more than one reason to change the class.

Object Peer Stereotypes

There are three general catagories of relationships that are used.

1) Dependencies:
An object is dependent on other objects to provide services required to do it's job. These objects are required.

2) Notifications
An object that is notified when a state or performs an action. These notification objects more independent, an object does not know about who is listening. These notifications decouple objects from one another.

3) Adjustments
Objects that alter behavior for the rest of the system. They modify output.

These stereotypes are only guidelines not hard rules to help develop the design.

Side Bar: New or not new, there is no try

The authors recommend to create a valid object at construction. It becomes problematic if the object is partially created then completed later with set methods. It is hard to maintain, the user must remember to finish the object.

Composite simpler than its parts:

As the system is built up from objects, each composition of objects should be simplier than its parts. The book uses an great example of the mechanical watch. There are a lot of moving parts in side but outside there are only the clock hands and stem.

Context Independence:

Each object should only know enough to get its job done. It should not know about the bigger context in which it lives. It is on a need to know basis. This creates classes that can be used in different contexts and allow for reconfiguration.

If the object uses language from multiple domains, then unless it is a bridging class, it does not have context independence.

Hiding the right information:

We want to be careful how and where we hide information. If done incorrectly, it could make the code more difficult to understand. It is important to understand the difference between encapsulation and data hiding. The book lists some examples, they sound good using the term encapsulation but don't if you use the phrase data hiding. One example in the book is:

"Encapsulate the data structure for the cache in the `CachingAuctionLoader` class"

Rephrase it to:

"Hide the data structure used for the cache in the `CachingAuctionLoader` class"

In the second sentence, you can tell that the concepts are mixed, causing coupling. The cache should be an object that has a job. The `CachingAuctionLoader` class does not need to know the details of the cache.

One passing thought about object oriented design that I thought of while working on this summary and doing my job. Currently I am trying to refactor existing legacy code to prepare for a new feature with help from the "Working with Legacy Code" book.

I have a class like:

```
class B  { ... }
class A
{
public A() {}

void method()
{
    B b;
    do something with b
}
};
```

As I understand it and maybe it will become clearer in chapter 7, I should not have been able to create this class from tests. Class A has this dependent class B that the test can't mock. But how/where would this class be created? It needs to be created somewhere. If I always pass in all dependencies ideally in the constructor or if it is a transient in by a method, then it must be created outside class A. We can call this outside class class AA. Since we need to mock out all peers in class AA, class B can't be defined in class AA. The chapter did mention having factories within abstraction levels to create the objects necessary for that level. But not how this will work.

Maybe that question is a teaser for Chapter 7 which I will also review in a few days. Maybe after rereading the chapter, I might have an answer.

Chapter 7: Achieving Object Oriented Design •

Timothy Wright, 10 Mar 2011

This chapter discusses how writing tests first helps the design and how the design ideas from Chapter 6 are realized.

We want to build classes that work well with its neighbors and simple and coherent so the system can be created by building on these classes. When requirements change, these classes can be rearranged to provide the new features.

The three aspects of TDD that help us are tests that describe what before how, understandable tests enforce simple tests, and the tests require we pass in the dependencies, they are made explicit. These constraints in TDD help create the right level of abstraction, separation of concerns and context independence.

Communication over Classification

The system the book describes focuses on the communication between objects and considers this very important. In Java, we can use interfaces to define the messages. We must also define the communication protocol.

An interface describes whether two components will fit together, while a protocol describes whether they will work together.

TDD is used with mock objects to make this communication explicit. These tests help us create the object dependencies, notifications and adjustments.

Value Types:

Value types are used to represent domain concepts. They can just be defined as a int or string type at first but later can be expanded if necessary. Value types are usually created with one of the following methods. These methods are very close to the methods to create objects created later. The book is a little confusing on this. Maybe the examples later will help.

Breaking out:

If the code is becoming complex, it is probably time to break it up. Create helper types to make things simpler.

Budding off:

Start a new concept with just a simple type and grow it as needed, adding more fields. These types raise the level of abstraction.

Bundling up:

If a group of values are always used together, maybe this is a sign to combine them. The combined value type should be simpler that its parts.

Where to objects come from?:

Objects come more from the tests which tell us when new objects are needed.

Breaking Out: Splitting a Large Object into a Group of Collaborating Objects:

Sometimes we must just write code to see what we can learn about the structure. The problem is when to stop and clean up. The code can get complex and some point we must stop and refactor. We can put out smaller objects to unit test separately reducing complexity.

Sometimes the code is too complex and it might be better to throw it away and start new. It will take less time since more is known about the domain.

Budding Off: Defining a New Service That an Object Needs and Adding a New Object to Provide It:

Often new types are discovered by pulling them into existence. The initial feature is added to an existing object but decided that it adds too much responsiblity. So this feature is spun off.

The interface is created by defining a service that the object under test needs from it's point of view. These supporting objects are first mocked to help further define the relationship between the object under test and the service required. Once the service is understood, then the object that provides the service is created and tested on its own. That object might need something, creating a chain of objects performing services. This is all pulled from needs. Or another way to put it "Develop from the Inputs to the Outputs".

Bundling Up: Hiding Related Objects into a Containing Object:

This is where the "composite simpler than the sum of its parts" rule come in. When we see a cluster of objects working together, we can bundle them together creating a higher level of abstraction. It also creates a name for this new abstraction, better defining the domain. It also allows us to scope the dependencies clearly. We can test the new composite directly with mocks.

Identify Relationships with Interfaces:

The authors use Java interfaces liberally to define relationships between objects. These interfaces should be as narrow as possible, and that means we will have more of them. Narrow interfaces are easier to read and understand. And extra convenience methods don't need to be carried around adding confusion. "Pulling" out interfaces helps keep them small, since there is no pressure to create more just in case.

Impl Classes are Meaningless:

Using descriptive suffixes like Impl for implemented interfaces or Interfaces in the names of interface classes is not helpful. If there is a naming problem that might mean there is a design issue. I agree wholeheartedly with the authors here. But my code is full of these bad names, labeling the code for what it is in the program rather than what it is in the domain. I am in too much of a hurry and don't take the time needed. I need to work on this.

Refactor Interfaces Too:

As the system continues to build, we can look for signals that the interface needs changing. There could be multiple interfaces that really represent the same thing. They can be combined. There could be similar interfaces that really represent different concepts. Or the interface could be unclear and need rewriting.

Compose Objects to Describe System Behavior:

I can't summarize better than this quote from the book:

"TDD at the unit level guides us to decompose our system into value types and loosely coupled computational objects". I also think that the TDD also helps create the abstraction layers too and not just the low level computational objects.

The low level objects can be used to build higher abstraction levels to create a web of objects. This theme of composition of objects into levels of abstraction is a major one in the book. The generated structure is very flexible to change because it is made up of well documented, well understood simple components that can be rearranged as needed. I would love to have my code be in this state, unfortunately everyday work gets in the way so the progress is slow. I do think there is big payoffs in maintainability but there are real costs to moving a legacy system to this ideal.

Building up to Higher-Level Programming:

Many times the infrastructure of the programming language can get in the way of clarity. All the keywords and punctuation obscure what is going on. The authors resolve this by separating the code into two layers: implementation layer and declarative layer. The implementation layer is the graph of objects that provides behavior. The declarative layer adds syntactic sugar to clarify, so the code can be read easier. The declarative layer is the what and the implementation layer is the how.

The coding style is different between the layers. The implementation layer follows the standard object oriented rules. The declarative layer is more flexible. It can use constructs like the train-wreck example. It's job is to present information to the reader.

What is less clear in the book is now these layers relate to the different abstractions in the system. When are the abstractions more object oriented programming and when is the declarative layer introduced? The book uses the example of jMock. In this case the declarative layer is between systems. jMock provides services for testing. And a business system would use these services through the declarative layer to make the juxtaposition of the two systems clearer.

And What about Classes?

Chapters 6 and 7 focused on the relationships between classes and how to build them with TDD. It does not, as the author's point out, address the internals of the class and how and when to use inheritance. This is considered an implementation detail, it does not define the needed types. The authors prefer delegation over inheritance which is a common recommendation. The authors point to other

books like Evans, Fowler and Kerievsky for more information on how to work with classes.

Summary:
There are a lot of ideas from Chapter 6 and 7 about object oriented design. I almost felt that ideas were thrown from all directions and not sure when to duck and when to catch. Many were I think glossed over, not because they weren't important but they are hard to explain without many examples which are later in the book. I am still working on how to integrate these ideas of object composition, abstraction layers, objects and value objects, growing the code from the inputs to outputs and letting the objects tell us what they need. How do you go from one unit test to another unit test to another unit test and keep everything in sync? I have a copy of the goos code and keep referring to it for examples. I am looking forward reviewing the actual code as a group.

About my question on keeping the unit tests narrow in scope. It seems to me that the unit tests must be coupled with the interfaces of its dependents. The interfaces are the property of the users not of the implementers. So a given unit test must include or import the interfaces of supporting classes but it should not include implementation of these interfaces. (Actually these interfaces should be grown from the unit tests). In my C++ example, I created an instance of the B class as a default. But won't that mean that I need to include that implementation of B in the unit test of A so it will compile? All dependencies must be passed into the class under test. The question of where to create these dependencies is still unanswered for me. I think is from factories at the correct abstraction level but the book does not address this directly yet.

Chapter 8: Building on Third Party Code •

Peter Hammond, 14 Mar 2011

This is quite a short chapter, which deals with what happens when the ideas from the previous chapters run into an API that we cannot change.

The first piece of advice is to not to mock any type that you can't change. I found this a little unexpected, as I have found it very useful in the past. The reasoning is (as I see it) that

1. we are unlikely to create all th enuances of the behaviour of the third party API, so we will discover issues at integration time anyway
2. Third party APIs will not meet the domain abtractions that we have arrived at, so we will want an adapter anyway.

Taking the two together, there is no value in mocking the API.

The authors recommend instead to create adapters, which implement the interface we would like in terms of the API that we have. Then we can mock the adapter. Value types from the library will be used directly, but generally should not propagate far into the code unless they happen to match the domain abstractions. If we need application layer objects as callbacks, observers etc in the adapter's tests, then these would be mocked.

A few tests will require mocked third party APIs, such as injecting error cases to test the adapter.

Chapter 9: Commissioning an Auction Sniper •

Peter Smith, 17 Mar 2011

This is a short chapter that describes the Worked Example that will be the basis for Part III of the book.

The example is an Auction Sniper. A program that will automatically bid on an online auction. The audience for the program are the staff of an Antique Buying company.

The example will require the authors to build a program that:

- Uses the network
- Communicates with a server using a communications protocol
- Has a graphical user interface

This allows the authors to show how TDD can be used in an application that is likely to have external dependencies on third-party libraries and potentially unreliable network communications. Moreover the customers for the example are non-technical which brings ample scope for changed requirements and miscommunication.

I think that this example is a good choice for a wide audience, and will map to problems that many readers will face. Whilst there are many integration difficulties, the test results should be predictable in advance.

As an aside I have often wondered how to apply the techniques when the results are not always simple to predict. For example a program where the low level parts are quite simple, but in combination they can combine to form complex output. I suspect - must confess that I haven't tried the TDD approach - that writing the bits, trying some input and verifying the output will be quicker than trying to predict the output correctly enough that I have confidence that the code is right. I guess writing the test so that it approximates the answer and can be refined over subsequent iterations. **End aside**.

The remainder of the chapter describes the problem, for those without access to the book I have written a summary as an appendix. I think that the details are best read from the chapter.

Key Points of the approach to solving the problem Forming a dictionary of definitions. This can be used to communicate with the customer, within the team and will likely be used in the implementation

Writing a state machine to describe the communication protocol with the server required for the initial prototype. This will be vital to understand the problem.

Agreeing to an incremental delivery with a simple first prototype. This will enable the functionality to be delivered in coherent vertical slices, starting with a walking skeleton. This should allow the customer to influence the product as it develops

and flush out integration problems early. The authors stress that it is important to get size of the slice right. It has to be large enough to be obviously done, yet small enough to be coherent and deliverable quickly.

Prioritising the UI over more complex functionality such as stop price. This decision has a human factor - communication with a non technical client, and a technical factor of a UI making it easy to set up multiple items each with a stop price. I think that the inclusion of the UI gave a larger vertical slice than a system with stop price and no UI.

The authors conclude the chapter with an appeal to the reader to accept the limitations of the example and not dismiss it as unrealistic.

Appendix:
The customers have agreed to incremental delivery, with a simple first prototype delivered first. The prototype will have the functionality:

- Allow the user to bid for multiple items
- Will show the identifier, stop price, current price, and status for each item it is sniping
- The user interface will allow the user to add new items to snipe
- The user interface will update in response to events from the auction house

In addition
The application will be a Java Swing desktop application
The application will communicate with the Auction House via the XMMP protocol

XMMP protocol features:
Client Server Architecture
Auction House will be running the server
Auction Sniper will be a client
All participants have a unique id
Users can connect from multiple applications, only the highest priority receives messages

Auction Protocol:
Bidders (Auction Sniper is a Bidder) send commands:
Join ; join the auction for an item
Bid ; send a price to the auction
Auctions send events:
Price ; report the price of an item, the minimum bid increment and the name of the accepted bidder. Sent to all new bidders and when a bid is accepted
Close ; Announce the winner of the bid

Representation:
The format of the message is:
protocol version number; Message Key: Message Value;
For example: SOLVersion: 1.1; Command: JOIN;

Chapter 10: the Walking Skeleton •

Alexandros Bantis, 22 Mar 2011

First, I want to preface my remarks by stating that I know nothing (I'm a very new newbie). In terms of Java, I know even less. So I would like to thank Nat and Steve in advance for their patience and understanding.

They begin by stating that the purpose of developing a walking skeleton: to clearly understand the requirements of a project. They speak of this both in terms of proposing and validating the structure.

Right from the outset, they emphasize how developing a walking skeleton front-loads the effort of the software development process. This initial process uncovers many issues (both technical and organizational) and centers the focus on the end user (cf. http://xkcd.com/844/).

They note that the very first stage of the Agile software development process is iteration zero, before "functional development starts in iteration one", and a big part of this iteration zero is developing the walking skeleton to "test-drive the initial architecture".

They suggest writing this first test of the program (which does not yet exist) as an example of "programming by wishful thinking" [Abelson96]. This first test helps to focus on the overall objectives from the user perspective rather than on writing "good code" (which is a means to an end rather than an end in itself). From the xkcd "good code" paradigm, this would mean getting the project done before the requirements have changed (implying that in some cases the project is thrown out because it never really met those requirements to begin with).

As this first test is end-to-end, it may be necessary to create a stub (a fake auction program that our program will interact with) in addition to the first working version of the program.

This end-to-end test will require some important initial choices. The project will need to be placed in a version control system that includes "an automated build/deploy/test process." It will be necessary to write several major components of the viper application to accomplish this (XMPP message broker, stub action & GUI testing framework.

This end-to-end test will also need to "cope with a multithreaded, asynchronous architecture." This aspect of synchrony is in contrast to unit testing because it will run in parallel to and interact with another application (the action site). This creates additional complexity because this interaction with another application may be difficult to capture through an automated build. Thus, this testing will be slower and more brittle, requiring some analysis and interpretation.

This chapter really helped me to understand the concept of test-driven development by walking me through an actual scenario. I also really appreciated

the extra time spent to explain and illustrate the concept of asynchrony, which I'm sure more experienced readers already understand.

Chapter 11: Passing the First Test •

Ian Miller, 24 Mar 2011

This chapter illustrates the process the authors recommend for getting the project underway. Iteration 0, covered in the previous chapter, decided what technology would be used and wrote the first test. The goal in this chapter is to make that test pass: to join a single auction and lose it without bidding.

Building the Test Rig

The emphasis is on doing the necessary minimum. We decide to work within a single host. Networking can come later. (For a professional set up, the authors recommend to have a more representative distributed test environment as well.)

The end to end test script always sets up an Openfire XMPP server - the Message Broker - from scratch. It creates one user account for the sniper to log into, and a couple for auctions. Once it has done that, it runs the tests. These don't get very far because we have not written `ApplicationRunner` nor `FakeAuctionServer` yet.

The Application Runner

Naturally, the purpose of this object is expressed in its name. The application itself has a Swing GUI. ApplicationRunner uses WindowLicker to drive that GUI so we don't have to. It drives GUI components using ComponentDriver objects.

We write the first code for ApplicationRunner and its `AuctionSniperDriver`. The former runs the sniper in a separate thread within the same JVM (so the driver can control it). The code asserts that the GUI reports that it is joining the auction shortly after being told to do so. Another function is provided to check for a Lost status report, to be called later.

The Fake Auction

To facilitate testing, we mock the real auction site. We write `FakeAuctionServer` to connect to the XMPP broker, to exchange messages with our sniper in accordance with the Southabee's On-Line specifications (which may or may not be accurate, so we'll still need to test against the real system eventually), and to test expectations about what the sniper will do.

We are introduced to the event driven XMPP client library, Smack. We are going to handle two types of event: auction management events such as joining requests, and auction partiticipation events such as bids.

Quietly, auctions have become chats. I suppose this is the XMPP domain language slipping in.

We write the first code for `FakeAuctionServer`: a minimal implementation of startSellingItem which starts an auction and waits for the sniper to connect using a `MessageListener`. The content of the join auction message is ignored for the time being. Establishing communications at all is our acceptance criterion at this stage.

Failing and Passing the Test

Write a script to build and deploy the software and then to run the tests. Run that script and start listening to the tests.

First User Interface

WindowLicker reports that it cannot find the top level JFrame it is expecting, so we write that and try again.

Showing the Sniper State

Now WindowLicker reports that it cannot find the Joining status string it is expecting so we add that label to our new Window.

Connecting to the Auction

The next failure is that `FakeAuctionServer` complains that it has not recieved a Join request within its timeout period. We arrange for the sniper to send a message.

Receiving a Response from the Auction

to change its status from Joining to Lost but the wait for that to happen times out. More chat plumbing takes care of that and finally the whole test passes.

The Necessary Minimum

"The point is to design and validate the initial structure of the end-to-end system... to prove that our choices... actually work." It may have been minimal but there was plenty to sort out to get this far.

Chapter 12: Getting ready to bid •

Ed Sykes, 29 Mar 2011

So this chapter is where we really get into the minds of Steve and Nat and how they approach things. For me personally this is exactly the kind of stuff i like in a book. Reading through this (and the next few chapters) reminded me a bit of readng Kent Beck's book 'Implementation Patterns'. In that book, as in here, you really need to read the chapter a few times to understand it properly. And the examples are in java. Deep thinking is definitely required.

For me, the fact that the examples are in Java made it a bit tougher as I had multiple learning dimensions while reading.

So what are we trying to achieve in this Chapter?

At the beginning of the chapter the authors state that they are focusing on 2 items from their todo list:

- Single item: join, bid, lose
- Single item: join, bid, win

Unless I missed something this chapter focuses only on the first of these two items. I suspect that this and the next chapter were originally part of one megachapter addressing both. I found I was a little confused at the end of the first read through for this reason.

First of all an end-to-end test is created that describes the bidding process and how our application will participate. I found the steps slightly confusing due to the naming. Below i will list the steps in the authors words and then in italics why this confused me.

this is the form:

code: meaning: *my confusion*

1. `auction.startSellingItem()`: Start the auction
2. `application.startBiddingIn(auction)`: Make our application join the bidding:
   at this point our application has not submitted a bid. I think saying startBiddingIn implies that a bid has been submitted. However at this point the application is just listening to what's going on. In my mind, this is more like when a person enters the bidding hall in a real auction. They are particpating only in a passive sense. So anyway the naming doesn't work for me here. I wonder if they will change this later...
3. `auction.hasReceivedJoinRequestFromSniper()`: Ensure that the auction received the join request: *This seems to back up the statements made above, it's like the language from the applications point of view is different to the auctions point of view. Perhaps this is intentional for a reason i don't realise*

4. `auction.ReportPrice(1000, 98, "OtherBidder")`. This means that the auction will announce that "OtherBidder" is the current highest bidder with a bid of 1000 and that anyone wishing to make a bid must increment their bid by 98. So the next bid should be 1098: *On the first read through the chapter I was confused about the bid increment. I though that the bid increment was the increment made by the current highest bidder to win the bid. I would have chosen to make the line auction.ReportPrice(1000, 1098, "OtherBidder"). The value of 1098 would be the bid offered to the floor. Again this is influenced by my only knowledge of auctions which is watching them on television where the auctioneer offers a price to the floor. I wonder whether it was a conscious decision on the part of the authors to send the minimal information needed, i.e. just the increment.*
5. `application.hasShowSniperIsBidding()`: Make sure our application has shown that it is trying to make a bid, triggered by the current price being sent to the sniper.
6. `auction.hasReceivedBid(...)`: Make sure that the the sniper tells the auction, as well as the user, about its bid.
7. `auction.announceClosed()`: The auction should now close.
8. `application.showsSniperHasLostAuction()`

I think the main thing that I take away from this part is that this is going to be tricky to write a really clear test when the changes in state and so on are event driven. This makes me wonder whether an event based application was a good choice of example. I've done a couple of years of the acceptance test driven style. I wonder how reading and understanding this is for someone with less experience.

The authors make the point that they have simplified the example to use integer based money. This choice hadn't occurred to me as something to highlight. I expect however that they don't want to fall into the trap of providing an example that is then taken by someone to mean that money is best represented by an integer when a fixed decimal would be more appropriate for most currencies. So it's a good thing to point out.

Extending the fake auction

In order to meet the needs of the end-to-end test the fake auction needs to be extended to report prices to the chat (reportPrice()) and to check whether our sniper has sent a bid. They implement these functions and take an opportunity to remove some duplication in the fake server. This illustrates the point nicely that test code (in this case inside a fake object) is a first class citizen and should be refactored in just the same way as 'production' code.

They make a point about using the same constant in creating the join message and checking its contents. I think here they are referring to `Main.JOIN_COMMAND_FORMAT`. They make a good argument for sharing this constant given the simplicity of the example.

I've seen a few occasions where information like this constant is repeated in test code. This doubles the overhead of a change for not much benefit. Like the authors, I don't think it's warranted in cases like this either. If you're worried about a bug then tests aren't the only method of defect prevention, visually inspecting

the constant and comparing it against the known message format is a valid technique here. As is calling over a colleague and getting them to do a sanity check for you.

They also change the method `hasReceivedJoinRequestFromSniper` to `hasReceivedJoinReuestFrom(String sniperId)` which means they have to change the end-to-end test. They don't really explain why this is, I think it's so that the main sniper id is shared from the main application to the fake auction rather than having to repeat it.

A Surprise failure

The authors write the method on the `ApplicationRunner` that checks whether the application showed the sniper status as bidding. In doing so they discover that there is an interaction between tests. They fix the interaction and then they are left with a failing test that fails in the way that they expect.

Outside in development

The authors describe their approach to breaking a problem down. They fix what can loosely be described as the inputs and outputs that they need as represented as an end-to-end test. This then leaves them with the freedom to write an initial implementation that passes the test. Then they will be able to explore alternative ways of making the test pass. They will do this by using TDD to create the pieces of behaviour they need. Hopefully by doing this the code will be simpler and more expressive.

Infinite attention to detail

In this section the point is made that the they were a bit lucky to find the bug in 'a surprise failure'. They ask the question 'how can we make sure we always find all the bugs straight away?'. I've exaggerated their point a little here. Something that I see developers worrying about when writing unit tests is over specification of tests. They try to mitigate against any defect that could ever possibly appear in the future. This leads to poor tests that are tightly coupled and brittle. On a well run project there are multiple feedback loops that will discover bugs. I agree with their point about professional testers. We have a saying on our team amongst the developers, 'We love our QAs'. They are a crucial feedback loop for us.

They describe a guiding principle of keeping component quality high and pushing to simplify. This, along with pushing to exercise as much of the system as early as possible, allows them to deal with complexity in the system.

The AuctionMessageTranslator

This is good section that I found quite easy to follow. I would say that this point in the outside-in process is a sweet spot for me. They discover a need for a class which translates auction messages to auction events. They name it `AuctionMessageTranslator`. They create the class and start writing unit tests for behaviour they need from it. They start at the outside and create an interface when they need some behaviour that doesn't belong in `AuctionMessageTranslator`. This is a true use of the single responsibility principle. Whilst I like this style of breaking behaviour down I know that some developers don't like such a granular

level of behaviour spread amongst classes. I would urge people to try and and see what happens. There are some surprises.
Two of them are:

1. Tests that are very expressive and easy to understand
2. Design patterns start emerging from the code as you create tests (rather than being designed in explicitly). Now, instead of saying 'let's use design pattern X' you'll find that you'll start saying 'this is beginning to look a lot like an X pattern'.

The authors also make a point of not mixing UI with logic to avoid a dependency between layers. Good advice.

In this case the main pattern is a listener pattern. Something that implements a `AuctionEventListener` will eventually be on the receiving end of events generated by the `AuctionMessageTranslator`. In the unit tests a mock object is used to detect the behaviour that is expected. These expectations are specified precisely. I think this sections about expectations might be quite difficult for someone who hasn't used a mocking library before.

Unit Tests

They then build their unit tests up. I'm reading on a kindle and i've just realised that the kindle mixes in ideas boxes with the main narrative. This has caused quite a bit of my confusion I think. On the kindle the sections 'Put Tests in a DIfferent Package' and 'Use null when it doesn't matter' is mixed in with the part on unit tests. I'll come back to this sections later.

They build up the unit tests slowly doing just enough to get the tests passing. It takes a lot of confidence to do this. When you begin TDDing in this way it can feel like you aren't creating a real implementation. However, what you find I think are simpler solutions to make the tests pass. First they make the test fail, then pass and then refactor. The usual TDD stuff.

Now some of those ideas boxes:

Put Tests in a different package

I've never seen anyone do anything else. I thought that this was pretty obvious as you don't want to ship code that is not actually generating business value directly. Does anyone have a different opinion?

Use null when it doesn't matter

Interesting, because in our C++ code we've moved to using references wherever possible. This means that we have to create all the dependencies whether we use them or not. I'm going to re-examine this decision and see if it would make the tests clearer and simpler to write if we went back to using pointers.

Simplified Test Setup

This is more for the java people i think. Although i do like making sure that an object is completely ready upon construction. We have readonly fields in .net but

you initialise in place so initialisation is common across different constructors. I didn't really follow the part about circular dependencies.

Closing the interface loop.

This is a really interesting place where the design just clicks in place. Now, the main object become an `AuctionEventListener` which is pretty neat!

Unpacking a price message

This second end-to-end test forces us to address the earlier decision not to unpack the message. The point here is that we need now to unpack messages in order to distinguish between them. This may seem a bit strange at first that we have to go and change behaviour generated for the first end-to-end test. However, the fact that we have all these tests supporting us allows these kinds of changes to be made quickly and with confidence. The authors choose a simple unpacking approach that turns out to be parsing the messages into key:value pairs.

Of course, they write a failing test first. Then they get the `AuctionMessageTranslator` to unpack the messages. Just to be clear, the `AuctionMessageTranslator` is one of the components that will ship with the software. Once it has unpacked them it can differentiate between them which allows us to get the test passing.

Discovering further work:

The messages come from an outside system and need to have error handling. Instead of breaking off from the flow of their current work the authors add that work to the end of the todo list. I have to admit that I often find myself breaking off from my main flow to do this kind of thing. I am going to try not to do that and to keep a list of things that need to be done. On the occasions when i don't break off i normally rely on an self-code-review to catch this kind of thing. It's much better to keep a list. I think using the list approach must be essential if you are pair programming too so that you stay on the main work track while tending to things that the non-typer will pick up on.

Finishing the job:

The authors finish the chapter by emphasising that this is a starting point. Many people would consider this design to be perfectly fine for the job. Something that I always encourage my team to do is to spend time thinking about expressivity and readability of the code. I find that when developers are under pressure to deliver they can cut corners on refactoring and thinking. This of course just means that when they come back to the code it is harder to understand and change. This in turn means that they can deliver less which starts a vicious circle.

Eds Summary:

This chapter was the toughest so far. I don't feel like you can skim at all at this point. I often skim a book for 4-5 chapters to get the gist and then come back. That hasn't really worked with this and the last chapter. There is a lot of details and it takes a long time to think about it and absorb it. I'm enjoying it though and looking forward to the next few chapters.

I found that trying to read the chapter in multiple sittings doesn't really work. It's very easy to get lost in which test feedback loop you're working at. Even reading on the computer didn't really help with this. I wonder whether reading the book would help with this. There are a few copies at work so I may try to give that a go for the next couple of books.

Something that might help the reader is to always prefix the word 'test' with what kind of test it is, unit or end-to-end. I found myself having to think a lot about which kind of test was currently under discussion.

Chapter 13: The Sniper Makes a Bid •

David Pol, 31 Mar 2011

1. Introducing AuctionSniper

The authors want the application to interpret the Price events sent by the auction, and they do several things in order to achieve that:

- First, they create a new class, `AuctionSniper`, that implements the `AuctionEventListener` interface in order to respond to the Price events. Some of the functionality in this class is currently written in Main --where it does not belong naturally--, so a little bit of refactoring is in order. The authors make a good point here about separating the user interface concerns in the `auctionClosed()` method from the new `AuctionSniper` type, which should only care about bidding matters. Single responsibility principle[1] to the rescue! Which brings us to the next point...

- A new interface, `SniperListener`, is created. The `AuctionSniper` will notify this new class of changes in its status. A first test saying that Sniper should report it has lost if it receives a Close event from the auction is made to pass by making `AuctionSniper` implement the `AuctionEventListener` interface appropriately. Also, the authors make Main implement the `SniperListener` interface, completing the circle.

At this point, the ratio of successful/failed tests is the same as before but the structure of the code is better, both for expressing concepts in the problem domain and for unit testing. The authors stress here that they don't see this whole refactoring process as a waste of valuable time, but rather as a specially useful endeavour to clarify the code at an early stage in development. They also point out the significance of the single responsibility principle as a "very effective heuristic for breaking up complexity" and tell us we don't have to be shy when it comes to creating new types. I can't agree more with this, and the philosophy behind this piece of advice makes me think of domain specific languages: the more expressive power we have to talk in terms of the problem, the easier it is to say useful things.

2. Sending a Bid

The authors introduce a new class here, Auction, that takes the role of accepting bids for items in the market. They make a point about not extending `SniperListener` for this purpose, as that class is all about tracking the status of the Sniper, not making commitments to an auction (again, SRP). In order to start bidding, the first step is to implement the response to a Price event, so they create a new unit test for the `AuctionSniper` that says when the Sniper receives a Price update, it sends an incremented bid to the auction and notifies its listener that it's bidding. This test features a more relaxed clause for the listener's expectation, atLeast(1); which, while probably won't be happening in practice, expresses more clearly the intent of the design: we don't care if the Sniper notifies its listener morethan once that it's bidding.

Two additional notes regarding this test are that the calculation of the expected bidding value is written directly into the test just because it is trivial (but this is not

a generally good practice), and that jMock expectations don't need to be matched in the order they appear in code although a certain ordering may be imposed by using a sequence clause.

Once this test is made to pass, the next goal is displaying the bidding status in the user interface and sending the bid back to the auction. In order to get the code compiling as soon as possible, the `AuctionSniper` is passed a null implementation of `Auction` in Main. The authors note here that by null implementation they mean a temporary empty implementation intended to be replaced, as opposed to the well-known null object pattern[2]. A cyclic dependency is discovered when thinking about the Auction implementation. The chat needs a translator, which needs a Sniper, which needs an auction. The authors break this dependency by noting there is no need to pass a `MessageListener` when creating a Chat. This discovery is only possible because of the availability of the source code for the Smack library. I agree; having access to the source code of a library usually helps a lot (that is, if the code is any good).

At this point the end-to-end tests pass and we have crossed off two items on the TODO list:

- Single item - join, lose without bidding
- Single item - join, bid & lose

One thing is left, though: we need to implement proper handling of XMMPExceptions when sending messages to auctions. No problem, the authors say, the idea was to get some structure into the code and see the tests pass. They just add another item to the list and wait until they know better. This is a prime example of deferred decision in order to keep going forward, a concept introduced later in the chapter.

3. Tyding Up the Implementation

The authors have made the end-to-end test pass now, but the implementation feels messy. They introduce several modifications to the code in order to improve it:

- They extract a `XMPPAuction` class from the `joinAuction()` method in Main. Its responsibility is to unify the functionality related to sending commands to an auction (join, bid). As we have seen several times now at this point in the book, this makes the language of the code closer to that of the problem.

- They extract a `SniperStateDisplayer` class from the event responding code in Main. Its responsibility is to translate Sniper events into a Swing representation.

This changes reduce Main to be the class that just creates the various components of the system and introduces them to each other --as it should be--. I've always liked nice little 'mains' that quickly delegate control of the application to the appropriate subsystem(s), it strikes me as 'the right way' to do it.

Finally, some cleaning is done to the `AuctionMessageTranslator` class and an `AuctionEvent` class is extracted from the code manipulating the auction messages. Here the authors recommend packaging common Java types, such as collections, in domain-specific classes. The idea being (again) working on the language of the problem so that there is less of a gap between the domain and the code. I think this is obviously good general advice. I'm not well-versed in Java, but this seems to be the natural equivalent of typedef'ing your types around in C++.

4. Defer Decisions & Emergent Design

These two short, final, sections summarize the most important lessons to take from the chapter:

- Deferring decisions is good to keep us focused on the task at hand. Null implementations of types and methods are useful decision deferrers, as we saw with the null Auction before.

- Minimizing the time between failed compilations is good. Keeping changes incremental allows us to have a better understanding of the scope of our changes. As a nice side effect, it is a commit-often-friendly practice.

The chapter ends with the authors reminding us what is "hopefully becoming clear": that the design is evolving from an unpromising beginning. I think they did a good work stressing this write-refactor (and defer as needed) cycle throughout the chapter. In line with this, I'd like to end this review with my favourite quote from this chapter:

"One of the great discoveries of test-driven development is just how fine-grained our development steps can be".

How true this feels now, working through the example! I'm enjoying these practical chapters a lot. They are challenging but at the same time reading them feels like reading a story. Going through the authors' mental process and understanding the reasoning behind their decisions is a mind-opener.

[1] http://en.wikipedia.org/wiki/Single_responsibility_principle
[2] http://en.wikipedia.org/wiki/Null_Object_pattern

Chapter 14: The Sniper Wins the Auction •

Chris O'Dell, 4 Apr 2011

First, A Failing Test

As a new feature is being added, the authors start by writing an end-to-end acceptance test of the desired behaviour. The test is based upon the existing '`sniperMakesAHigherBidButLoses`' test. This new test includes a fake auction price update with the Sniper winning and the expectation that the UI displays a 'Winning' message, followed by the fake auction reporting the auction to be closed with the expectation that the UI displays a 'Won' message.

Of course this test fails, but the error output guides us - we need to get the Sniper to report that it is winning.

Who Knows about Bidders?

To allow for the Sniper to determine if it is the current highest bidder, the authors add an enumeration to the `AuctionEventListener` interface. They explain that an enumeration was chosen as this would make its purpose clearer than a boolean when used in other parts of the codebase. The setting of this value was decided to be a part of the `AuctionMessageTranslator's` responsibilities and as such a new unit test is added (and an existing one amended to clearly state it focussed on 'another' bidder). They also take this opportunity to pass the Sniper's identity in the constructor of the Translator which is used in the new test as a constant to create the fake Price message.

(There is a footnote about Java developers and nested types here, but Java is new to me so I cannot comment. In C# I have frequently seen enums nested in situations like this.)

The test fails as the `AuctionMessageTranslator` does not yet use the Sniper Identifier, so the code is amended to compare the given Id with the one from the event message. This actual comparison is done inside `AuctionEvent` to keep the Translator clean and readable.

The Sniper Has More to Say

The end-to-end test is still failing as the code does not yet update the UI to show the Sniper is winning. The authors begin with a Unit Test to around the `AuctionSniper` to interpret the `isFromSniper` parameter. The test expects a sniperWinning() method on the `SniperListener` interface, so it is added along with an empty implementation on the SniperStateDisplayer, which allows us to compile and run the test.

As JMock appears to use strict mocking semantics by default (at least in the examples), the test throws an error when `auction.bid()` method is called unexpectedly. I don't want to derail the review too much, but in my experience strict mocking semantics lead to brittle tests with the tests expressing too much knowledge of the object under test - it is no longer a black box. It can also create a refactoring nightmare where a considerable amount of time is spent fixing up seemingly unrelated tests when new functionality is added.

Anyway, the test is fixed by adding a switch to the `AuctionSniper` which tells the `SniperListener` that it is winning when the price update is from the Sniper. This fixes the first part of the end-to-end test.

The Sniper Acquires Some State

To be able to display whether or not the Sniper it has won or lost, it must know if it is currently bidding or winning. The authors begin by amending the name of the existing test '`reportsLostIfAuctionCloses`' to include the word Immediately so that it is obvious to the reader that if the action is closed when no action has been taken that the default state is to report a Loss.

A new test is added to describe that the Sniper will report a Loss when the Auction closes with the Sniper in a 'bidding' state. To track the state of the Sniper, the authors introduce another feature of JMock - a context placeholder which is setup to listen for the 'sniperBidding()' method being called on the fake `sniperListener`. I am not keen on this approach.

As the authors state in the sidenote 'Representing Object State' they wished to make assertions about the object's behaviour without exposing how the state is managed for fear of breaking encapsulation, but I feel that 'keeping tabs' on the state using a separate context object is open to failure with the two going 'out of sync' if any refactoring takes place that does not affect the state, such as a call to display an additional message. However, in tandem with the strict mocking semantics of JMock described above, I would guess that this is unlikely, but the test would need to be fixed with the refactoring. Regardless, to me it feels like a brittle test based upon the object performing specific internal actions.

I would probably expose the state of the Sniper via a readonly property and split this test out into two tests - one to test that when the Sniper receives a currentPrice message where it is not the winner that it sets the state to 'bidding'. I can then Assert the Sniper's state explicitly. A second test would be the same but including a call to `auctionClosed()` and I would add an expectation on the mock `sniperListener` for the call to sniperLost(). Together the two tests cover the desired behaviour and at a more fine-grained level for greater feedback. A more flexible approach would be to not test on how the Sniper manages it's internal state, but rather that the behaviour is as expected, e.g. a Loss is reported if the Auction closes immediately after receiving a `currentPrice()` update where it is not the winner.

The Sniper Wins

A near duplicate of the above test is added except with a `currentPrice` where the Sniper is the highest bidder and an expectation on a call to a new `sniperWon()` method on the `sniperListener` (note, the `currentPrice` is called in the code sample with 'true' for the bidder, which I would assume is a typo as it should be the enum). The test fails as the Sniper is only able to report a Loss. A flag is added to hold the Sniper's state and updated to true when the Sniper is the highest bidder in the `currentPrice()` call, and the `auctionClosed()` is updated to call

`sniperWon()` when the Sniper's state is winning. Finally, the SniperStateDisplayer is amended to display the `STATUS_WON` message in the UI.

The end-to-end test should now pass and which enables us to cross off an item on the task list.

Chapter 15: Towards a Real User Interface •

Richard Barrett, 7 Apr 2011

Overview

Often the client has an idea about the user interface for an application, and, in our example, the Sniper UI has been roughed-out previously. However, instead of creating that UI up-front, our authors have kept it to the bare minimum, implementing just enough to prove the structure of the app. However, the client wants to *see* what he's paying for. Sound familiar?

The UI needs to change from a simple label to a table - perhaps the roughed-out UI suggested that?. How that change is made, guided by the tests, is the subject of this chapter, and it promises to be interesting.

Disclaimer: I've no great Java experience, certainly little with UIs. Please excuse any faux pas.

Replacing JLabel

So, we update the existing WindowLicker test to look for a cell in a table rather than the label,and check that it fails. Then the JTable and its associated TableModel is added, but, as always, only just enough functionality is added to get the code to compile and the test to pass.

By now, we are used to this minimalist approach;...

Displaying Price Details

...however, now it is decided to display what the user really wants to see - what they're bidding on, how their bids are progressing, and, what they might ultimately be paying! Therefore, what's required first is to update the acceptance tests with some dummy prices and bids.

To get the modified acceptance tests to pass, the item ID, price and bid need to be passed and these are bundled-up into a Value Type, `SniperState`. Interestingly, the fields of the value type are declared initially as public final with an "ambition" to replace this access later-on. That doesn't seem unreasonable at this stage, but how do we ensure that the type is revisited at some point? Also, I'm not sure why the need for a value type similar to this wouldn't have been apparent, or at least hinted-at, earlier.

To test the new methods and value type, an `AuctonSniper` test is updated to expect the object's listeners to be updated with the new `SniperState`; `AuctionSniper` is updated to make the test pass.

To get the `SniperState` displayed, "larger steps" are taken: the MainWindow has a new method to receive the new `SniperStatus`. and, having worked our way in, we start thinking about how the UI will be updated to display a `SniperState`, and the unit test to support that.

A need is identified for an enum to represent the table columns (writing the unit test would have hinted at this). The `SnipersModelTable` test checks (1) that `SnipersTableModel` retains the new state change, and (2) that the table is notified of the change.

In testing the second, a mock `TableModelListener` is attached: there is a comment about breaking the "only mock types that you own" rule. Mocking a Swing interface seems very low risk. I'd be interested in other's thoughts about this.

The implementation is updated to make the unit test pass, but the acceptance test passes the Bidding check, but fails now because the last price columns has not been updated.

Simplifying Sniper Events

Up until now, only a Bidding sniper event has been handled. There are the other events to consider: Winning, Lost and Won. Clearly, it would be possible to repeat what has been done so far, for the other events.

The decision to have a type to represent the sniper's status leads to a re-branding of the `SniperState` created earlier to `SniperSnapshot`. This allows the creation of new type to represent `SniperState`, reflecting the terminology a the state machine modelled earlier. One of the nice things about this book is that way that the design process is described as it, the design, evolves. This section isn't so much about tests, but about making small steps, allowing change to happen; the tests are there, providing a safety net.

Our new `SniperState` enum is added to the re-branded `SniperSnapshot`. And the `SniperListeners` are updated with a new `SniperSnapshot` when the sniper state changes. Much more elegant.

The table model is updated to translate the `SniperState` enum into real text. At this point, I wondered what had happened to `SniperStateDisplayer`; aren't there some changes to be made there but not described in the narrative? I resolve to actually try and follow the worked example in real code.

The tests are updated, including a 'little helper' method to repackage JMock's `FeatureMatcher` to use the terminology of our domain.

Having implemented the sniper state display in the UI, we now look to get price there too (remember, the acceptance test is failing that). The `AuctionSniper` test is updated to expect an update to `AuctionSniper`'s listeners when bidding and winning. `SniperSnapshot` is now expected to convey all state changes from `AuctionSniper`; to enable this in a clean way, three constructor methods are added to `AuctionSniper` to create the different snapshot states of bidding, winning and joining. This alludes to a state machine for `AuctionSnapshot`.

`SniperListener` is updated, and the acceptance test passes.

Follow Through

Further code simplification is is possible when the `SniperState` enum includes "won" and "lost" as states. `AuctionSniper` is changed to change the `AuctionSnapshot` when a new price is received and the power of the `AuctionSnapshot` concept proves itself. `AuctionSniper` looks lean and mean.

The idea of a Defect exception - thrown by "programming errors" - is presented. It's good to separate these errors from failures in the runtime environment.

The `TableModel` is updated to display the changed `AuctionSnapshot`, with a neat way of picking the `AuctionSnapshot` fields for the the Table columns.
Further simplification occurs when `SnipersTableModel` is changed to be a `SniperStateListener`, bypassing MainWindow, and `SniperStateDisplayer` becomes a "Decorator" for pushing updates on to the Swing update thread.

Final Polish

The Table's column titles are still missing from the UI, and so, of course, a unit test is written first before the `TableModel` is updated to provide the column titles to the table. The unit test for `TableModel` is updated to check that it provide the correct column names.

Finally, the acceptance test passes.

Observations

The chapter ends with some general thoughts...

SRP

I think the point about not putting business logic in the user interface is well said. I've often seen a UI implemented first and then the business logic being developed later; the business logic gets interwoven with the UI, or at the best, leaches into the UI implementation. The approach taken in the worked example, of doing just enough to get the test pass, and delaying a fuller UI implementation until the internal structure of the application is better known, leads almost naturally to this separation. It's that rather good notion of implementing a "slice" of the complete functionality, from earlier in the book

Keyhole surgery

As in rock climbing, taking small careful steps will get you to the final goal, protected by the rope (or in our case, tests). (A nod to Roger Orr's ACCU2009 "Refactoring" talk is in order here)

Programmer hyper-sensitivity

...to the waste of their own time. The authors are using the 'royal we' here, of course. Many programmers seem totally oblivious of the value of their time! There is a note about this requiring skill and experience: this reminds me of the 'programmer as craftsman'.

Celebrate changing your mind, and

This isn't the only solution.

We need to embrace change as we learn more about the problem, and our skills improve. In the general case, one's solution isn't not just the only solution, it's also probably not the best (and almost certainly not the worst).

In summary, this Chapter has shown how the UI and overall design have grown; there's not a lot of being guided by tests.

I'm enjoying reading this book, the worked example in particular. I feel that I've gained a lot from reviewing this chapter. I hope you do, too.

Chapter 16: Sniping for Multiple Items •

Jacob Metcalfe, 18 Apr 2011

In this chapter, we extend our system to support sniping for multiple items and adding items from the user interface, allowing us to tick two more items off the todo list. With the restructuring work from previous chapters, this proves surprisingly simple, but leaves the code in a bit of a mess, ready for further refactoring.

Supporting multiple snipers:

The authors begin, as usual, by adding an end to end test for multiple auction support, adapted from our previous single auction successful bid test. A few small changes are required in the `ApplicationRunner` in order to prepare for the test. The item ID is extracted into a function parameter in the `hasShownSniperIsBidding` method and we modify `startBiddingIn` to take a variable number of auctions and pass these through to the application under test. We can now write the rest of the failing test, which joins 2 auctions, begins bidding and wins both. The authors note that the ordering/interleaving of statements is important here and that by grouping the two `showsSniperHasWonAuction` calls, we gain more confidence about their validity. I wondered here whether grouping assertions like this was really a good general guideline, or whether it just comes down to a per-test-case judgement call on what will provide the strictest requirements.

We note here that the test failure message is not very useful, just telling us that an unexpected `NULL` value was received, and improve it by combining two assertions (the non-`NULL` assertion and the real value assertion) in to a single assertion using a PropertyMatcher - `assertThat(message, hasProperty(body, msgMatcher))`. The reporting could have been further improved by extending FeatureMatcher or writing a custom matcher, but the error is now clear enough.

It is now obvious that our test is failing because `Main` is currently ignoring all but the first auction, so the JOIN request for the second is never received. We modify main to hold out to its `XMPPConnection` and to loop through the arguments, joining all the auctions. This gets us past the first error and shows us that the test is now failing because our table model still only supports one hard-coded row.

To begin supporting multiple table rows, the table model is extended with `addSniper`, which is called from `Main.joinAuction` within an invokeAndWait wrapper, as we are modifying the Swing UI state from outside the Swing thread. We then write a failing unit test for the `SnipersTableModel`, to show the addition of a single row to the table model with the new method and the expectation of an insertion in the first row. In order to get this test working we move from a single `SniperSnapshot` to a collection and trigger the new table event. The new event breaks one of our existing tests, but it can simply be modified to ignore the event using a new allowing clause as it is not relevant to the test. We add another unit test to check the ordering of snipers as they are added and make it pass by looking up the row to be updated (using new methods `rowMatching` and

`SniperSnapshot.isForSameItemAs`) when we receive snapshot updates. Our end to end test now passes and we can move on to adding items through the GUI.

The authors note here that TDD can be very helpful with avoiding off-by-one errors by defining boundary conditions and double checking the implementations correctness.

Joining auctions from the GUI:

The client now comes back with a simplification to their original design, moving the join auction popup in to a top bar. The authors make a quick point that in their experience it *is* possible to make development progress while the design is still in progress. As they say, I think the important point is to maintain a flexible approach, to expect (and code for) change and, when design modifications demand structural code modifications, to accept change and invest the time to improve overall code quality.

We first modify our test infrastructure to support the new method of adding auctions, by introducing a new method, `startBiddingFor`, to the `AuctionSniperDriver`, which will use the text field and button to enter a new auction, and modifying the `ApplicationRunner` to use this method rather than passing the auctions through to `Main` as arguments. Our tests now fail because neither of the UI components exist. We correct this by creating a `makeControls` method in `MainWindow` to create the new top bar, so the test now fails because clicking the new button does not add the expected sniper row.

We now obviously need to associate our behaviour (add a new row, create a new chat) with the button, but it feels wrong to put work relating to these concepts in `MainWindow`, whose responsibility is the management of UI components only. To get around this, we add a new collaborator, `UserRequestListener`, with a single method, `joinAuction(String itemId)`.

The next test we would like to write will test that the `UserRequestListener` method is called when the join button is clicked. Swing threading makes this slightly complicated, but we can use WindowLicker to create a integration test, waiting for the tested code to stabalise. WindowLicker probes, which are repeatedly checked until the condition is satisfied or the operation times out help us here. We use a `ValueMatcherProbe` to compare to a Hamcrest matcher. To make the test pass, we fill in a simple listener infrastructure using `Announcer`, a utility class that manages a collection of listeners, and hook it up to the Swing `ActionListener` for the button. In order to get the full acceptance test working, we fill in an implementation of the `UserRequestListener` by simply inlining our previous `joinAuction` method in to the listeners `joinAuction` method. As this is being called from the Swing thread, we can also drop the invokeAndWait requirement that was needed earlier. At this point I wasn't really sure why the `Announcer` was required, as we currently only have a single `UserRequestListener`, created for a single connection in `Main.main`.

The chapter finishes with a quick look back at our changes. `Main` is looking a bit messy, as we put new functionality from various areas of the application in there to

get the tests passing. This code is not protected by unit tests and, in it's current form, is not really testable by anything but the end to end tests. Our next objective will be to clear this up.

It was nice to see the progress in this chapter. At least for me, it went a lot smoother than expected, despite the confused state of `Main` after the changes. I was struck by how simple it seemed to add in the multiple item support while keeping things compiling and only taking small steps towards the goal.

Chapter 17: Teasing Apart Main •

David Leftley, 21 Apr 2011

Following several chapters in which new functionality has been added to the system, in this chapter there is no new functionality. Instead it returns to the Main class, identifies some areas in which best practice (such as separation of concerns) is not being followed, and refactors to leave a much more pleasing and maintainable structure to the code.

Finding a Role

The previous chapter ended with the observation that while all the existing tests now pass, class `Main` has room for improvement in a number of areas:

- "design mess": a jumble of functionality that ought to be separated out.
- the code is not testable except through the end-to-end tests.

I was interested to see this latter point brought out, because it's something that has crossed my mind from time to time. I'm afraid I have yet to get round to embracing TDD for the entirety of a project, but I have wondered how it applies to those bits of the code that are there to tie together other units. This code is generally not too complex so one would like to think it's difficult to get it wrong, but as the amount of code grows throughout the project it would be nice to have some reassurance that it is still correct. I look forward to getting some ideas from this chapter about how this can be accomplished.

Having decided to revisit `Main`, the authors observe that it currently acts as a "matchmaker" bringing together other components, but that it also implements some of the components. A clue that highlights the mixture of responsibilities is the long list of imports: in particular `Main` depends on the Swing and Smack libraries, and it is the factoring out of these dependencies that drives the changes in this chapter.

Extracting the Chat

Firstly, our attention turns to the Chat functionality. `UserRequestListener.joinAuction()` refers to a `Chat` object on numerous occasions, hence the dependency on Smack. As a first step in tackling this, the dependency loop between `XMPPAuction`, `Chat` and `AuctionSniper` is broken down by introducing an `Announcer` to bind together the `AuctionMessageTranslator` and `AuctionEventListener`. This paves the way for moving all the chat-related code into the `XMPPAuction`, leaving `joinAuction` much shorter and tidier. More importantly, `joinAuction` is now written only in terms of `Auctions` and `Snipers`, with none of the chat-related implementation details visible at this level.

A new integration test is added at this stage to show that `XMPPAuction` can create a chat and attach a listener. It isn't clear to me here whether this test was written following the usual rule of "write a failing test, then make it pass" or whether this test is simply added to reassure us that code that already works won't be broken in

future. I think probably the latter, but the authors are moving at a slightly faster pace now, and glossing over some of the incremental steps. The golden rule of TDD is of course "no new code without a failing test", but is there any equally memorable guidance about when to add extra tests, and how many to add?

Now that all the chat functionality is encapsulated within `XMPPAuction`, this class and `AuctionMessageTranslator` are moved into a package of their own and their tests can similarly be moved into a new test package. That ties up the changes to this aspect of the code except for a note in the sidebar that the constructor ought only to set fields rather than including real behaviour as at present. For now, this is justified as being unavoidable when creating a veneer over an external library (Smack) that has similarly complicated constructors. For me, these discussions about when and why it is acceptable to depart from best practice make the book much more believable: authors who lay down rules that must not be broken under any circumstances often seem to me to be living in a different to this one where deadlines and distractions mean you can rarely go on refining your design until you are completely happy with it.

Extracting the Connection

Next, a factory class is introduced to create an instance of an `Auction` for a given item. The aim behind this is to get rid of the direct references to `XMPPConnection` within `Main`. The authors decide that this new type should be named using the language of auctions so it is called `AuctionHouse`. To me, this raises the question of how to decide which of your types should have names from the problem domain and which should be named after their function or the pattern they implement. My first impression is that it would be easier to get up to speed with the code if the actual concrete items in the problem domain (e.g. `Auction`) had names from that domain, and abstract objects within the implementation had names such as `AuctionFactory` that reflect their function. What do others think?

Extracting the SnipersTableModel

Having dealt with the XMPP dependency, Swing is the next dependency to be targetted. This looks like being a fairly big change, but it is tackled in a number of steps with fully working code between each of them.

The anonymous implementation of `UserRequestListener` now becomes a proper class with the name `SniperLauncher`. Once this class has been separated out, the authors note that they identify the next area for attention by observing the duplication in the code. The `SnipersTableModel` is attached to both the `Sniper` and the auction, and an initial `SniperSnapshot` is created both here and in the `AuctionSniper` constructor. The approach chosen to tidy up the `SniperLauncher` code is to introduce a `SniperCollector` that accepts the new sniper into the application, so now the `SniperLauncher` has the single function of setting up a new `AuctionSniper` and no longer needs to refer directly to Swing. (I am reminded again at this point that I ought to be referring to the code alongside the book, because as far as I can see the book doesn't show what the `SniperCollector` looks like).

Creating a `SniperPortfolio` to maintain the list of `Snipers` takes away one of the responsibilities from `SniperTableModel` and allows `Main` to be further simplified: now it only has to create the `SniperPortfolio` and `MainWindow` and pass the former to the latter. Now that there is a class that maintains the list of snipers, this also eliminates the need for the nasty hack `notToBeGCd`, which was introduced simply to ensure a reference to the chat object is kept as long as it is needed. The authors do however note a little later that this was not the only possible solution, and may indeed not be the best one. The code as it stands relies on the `SniperPortfolio` hanging on to the chat reference: an assumption that may prove incorrect. The alternative suggested is that as the issue stems from the XMPP code, it should be solved in the XMPP layer, perhaps by introducing a lifecycle listener to notify us when each `Auction` is no longer needed and can be released.

Observations

The chapter ends with a few general points about how the refactoring process has gone so far, and the architecture that has evolved.

Having restructured `Main`, the application structure now matches the "ports and adapters" architecture described earlier in the book, with classes representing core concepts in the application domain, others handling the technical domain and wrapping external dependencies, and a layer of adapter classes bridging between these two domains. I find it reassuring to see that a well-structured architecture can emerge out of the much more messy structure that was being presented just a few chapters back - it's always at the back of your mind when starting a new piece of work that if you don't get the structure completely right from the word go, it will haunt you for the lifetime of the code.

The authors also give a reminder that while refactoring naturally draws your attention to the static structure of the code (classes and interfaces), it is important not to lose sight of the dynamic structure (instances and threads). For example, it may be useful to draw an interaction diagram to get a different viewpoint and perhaps find a solution that isn't clear just from looking at the classes.

Chapter 18: Filling in the details •

Michael Baker, 25 April 2011

Chapter 18 begins by highlighting the fact that development has so far been aimed at attracting interest by offering an idea of the application's overall feel. It's certainly true that whilst some basic sniping functionality can be demonstrated, the current implementation has shortcomings that would limit its usefulness as a real tool: In particular, there is no way for the user to specify an upper limit for bidding on an item. Without such a mechanism, the sniper would continue bidding until it won the auction, whatever the cost. With this in mind, the goal of the chapter is to implement a "stop price" which will represent an upper limit on the snipers ability to bid.

The authors chose to add a new state (losing) to the model on the basis that the user will want to know the final auction price after they have lost. I can't quite see the link between this user requirement and the need for a new state. Looking at the code as it stands, I'm also not certain that there is any material difference in behaviour that would back up the decision. Maybe I'm missing something here or maybe it will come out in a subsequent chapter.

Implementing a Failing Test

In keeping with their general approach the authors proceed to define a failing end to end test before implementing the the new feature. In the interests of brevity, the text concentrates on the use case where the sniper bids on an item but loses because because the item's bid price exceeds the sniper's stop price.

Writing the test code for the new feature guides us to the introduction of two new member functions on the `ApplicationRunner` class: `startBiddingWithStopPrice` and `hasShownSniperIsLosing`. There's not much to add here as both functions are concise and pretty much more of the same that we've seen already. However, at the risk of being picky, I'd personally rather see prefixes such as 'has' and 'is' reserved for predicates and so a name like `verifyShownSniperIsLosing` would have made more sense to me. YMMV.

With the changes to the test infrastructure complete, the end to end test is run. Unsurprisingly, the resulting failure message indicates that the the main window contains no 'stop price' field. This prompts us that it's time to write the code that will enable the test to pass.

Implementing the feature

In order to address the failure, the code for the main window is modified by adding a new `JFormattedTextField` which is constrained to accept only integer values. The tests are run again but fail again. This time because there is as yet no logic to force the sniper to cease bidding once the stop price is reached.

To address this, the stop price must firstly be passed from the user interface down to the `AuctionSniper`. Faced with a choice of either widening interfaces to accommodate the stop price or implementing a new type, the authors opt for the

latter, implementing the `Item` value type. With the `Item` class in place, the `UserRequestListener` interface is suitably modified and the compiler is enlisted to identify the necessary changes.

Once the code is compiling again, the unit tests are augmented with checks to verify the new state transitions that result from the addition of the 'losing' state. Getting these tests to pass leads us to the changes that must be made to the logic in the `AuctionSniper` class. with those changes complete, all unit and end to end tests pass and we're done.

Chapter 19: Handling Failure •

Joes Staal, 28 April 2011

This last chapter on the implementation of the sniper application focuses on error handling. All the chapters so far have assumed a perfect world in which everything goes according to plan.

What If It Doesn't Work?

Southabee's On-Line, the product people say, occasionally fails and sends out messages that are not structured as specified. Since the Southabee's system is an aggregator for multiple auction feeds, the failure of an individual auction does not imply the whole system is unsafe. A decision is being made that if the software cannot decipher a message it will mark the corresponding auction as being 'Failed' and any further updates will be ignored. No recovery attempts will be made.

The authors do remark in a footnote that an auction site with such behaviour will probably not survive for long. However, for didactic reasons they want to work through this (contrived?) example.

The mechanism by which a message failure is shown is that the price and bid values are flushed (zeroed) and the status of the item is marked Failed. The event is recorded somewhere as well, so that it can be dealt with later.

An end-to-end test is written with a Sniper that receives a bad message. The Sniper then displays and records the failure and ignores further updates from that auction. The implementation of a function called `reportsInvalidMessage()` is being parked until later in the chapter, where (I believe) it actually doesn't report but records and therefore a better name might have been `recordInvalidMessage()`. I would prefer to call it `invalidMessageIsRecorded()`, in a similar vein as the reviewer of Chapter 18, but that is more a personal preference than a criticism.

Testing That Something Doesn't Happen

The end-to-end test makes a call to `waitForAnotherAuctionEvent()` method that forces an unrelated Sniper event to work through the system. This is needed to make sure the system catches up with so that the relevant price event has been processed. Chapter 27 is going to say more on asynchronous testing.

To make the test fail (as usual) the FAILED value is added to the `SniperState` enumeration with a text mapping in the `SnipersTableModel`.

Detecting the Failure

The `AuctionMessageTranslator` will be the place where the runtime exception occurs. The Smack library drops exception thrown by `MessageHandlers` and therefore the handler needs to catch everything. Furthermore, it is discovered that a new auction is needed: `auctionFailed()` is added to the `AuctionEventListener` interface. The unit test for a 'bad message' fails with an `ArrayIndexOutOfBoundException`. It doesn't really matter in this example which exception is thrown, because the message is either parsed successfully, or it isn't.

So the test is made to pass by putting the message parsing in a translate() method that is wrapped in a try/catch block in `processMessage()`.

The authors use the occasion to check for another failure mode (why they choose this one is unclear to me, probably just to show how they can check for multiple failures). They argue that a message may be well-formed but incomplete, such as a missing event type or current price (on a meta-level, isn't that an ill-formed message?). A couple of tests is written to show these failure can be caught and a `MissingValueException` is implemented for this purpose.

Displaying the Failure

The failure needs to be displayed in the application. A few tests are added and it turns out that the only method that needs to be added to `SniperSnapshot` is failed(). The application indeed displays the failure, but the end-to-end test still fails, because the Sniper hasn't been disconnected from the auction and can still receive further events.

Disconnecting the Sniper

A Sniper is disconnected by removing its `AuctionMessageTranslator` from its Chat. This can be done safely, because Chat stores it listeners in thread-safe 'copy-on-write' collection. This can be achieved by modifying `processMessage` in `AuctionMessageTranslator`, but the authors give two reasons for not doing this. First, construction of a Chat is painful, even though it could be mocked. But mocking this class would define a relationship with an implementation, not a role. Second (and it would be my first reason) it violates the Single Responsibility Principle (SRP): `AuctionMessageTranslator` would translate messages and decide what to do when it fails. Therefore, a disconnection policy object is defined and attached to the translator. This makes the end-to-end test pass.

The Composition Shell Game

A side box is giving a bit more background on the choice of adding an extra listener. It might be considered more complicated than just disconnecting the chat within the translator. However, the SRP seems to be the better design decision (with which I agree). The authors remark that putting behaviour somewhere else (ther is no there there) can be frustrating if one is not used to this style. Personally, not using Java but C++ I must admit that I find Java code sometimes unwieldy, mainly because I am not used to work a lot with listeners and observers.

Recording the Failure

It is now time to look at `reportsInvalidMessage()`. The requirement is that the application logs a message about failures in order to let the user recover from the situation. So, the test must look for a log file and check its contents.

Filling In the Test

The check is implemented. The log is flushed before each test, delegating log file management to an `AuctionLogDriver` class (using the Apache Commons IO library). The driver class cheats a bit by resetting the log manager in order to avoiding cached loggers not finding a deleted log file. The tests tell the pieces fit together, they don't check for content of log records. The end-to-end test fails since there is no log file to read.

Failure Reporting in the Translator

The first change is in `AuctionMessageTranslator`. Things to record are the auction ID, the message and the exception that was thrown. Again, by the SRP, the translator should not be responsible for how to report the failure and a `XMPPFailureReporter` interface is erected. All existing failure tests are amended and helper methods are created to wrap up message creation and common expectations. The new reports is fed through the constructor of the translator and called just before notifying any listeners. Since `message.getBody()` doesn't throw it is left outside the catch block.

Generating the Log Message

An implementation of `XMPPFailureReporter` is needed and a class `LoggingXMPPFailureReporter` is constructed. It is decided to use Java's built-in logging framework. Since file access and is covered by the end-to-end test, the unit-tests will run all logging in memory, reducing dependencies. If the example wouldn't be so simple, the authors would have written integration tests. Due to Java's logging framework lacking interfaces, the authors need to be more concrete than they would like. They decide (against their own rules) to use class-based mock to override the relevant Logger method. The test is passed with an implementation calling the logger with an invalid message.

Breaking Our Own Rules?

The authors defend their decision to mock classes, which they will discuss further in Chapter 20 (so stay tuned!). I think there are two reasons why the authors deviate from their best practice. First, they want this example to end and don't want to implement a Facade or Decorator around the Java logging system. Secondly, and I think they have a point, though a very strong one, they argue that the Java logging API is unlikely to change. But, I wonder, what would happen if a different logging strategy is demanded by the users? Now the application is tied to the Java built-in system. Wouldn't it have been better to have that extra level of indirection?

Closing the Loop.

All the bits are put together to make the end-to-end test pass. `LoggingXMPPFailureReport` is plugged into an `XMMPAuctionHouse`, so that its `XMPPAuctions` instruct each `AuctionMessageTranslator` to use its reporter. An extra exception (`XMPPAuctionException`) is added to deal with any failures within the package. The test passes and the next item is checked on the to-do-list.

Observations

'Inverse Salami Development'

The authors reflect on adding functionality adding in thin, but coherent slices. Write some tests for each new feature to show what it should do. Work through the tests, make them pass, refactor to open up space for new functionality or to reveal new concepts and ship. Use the compiler to navigate the chain of implementation dependencies.

The message to take home here is to keep the code well structured so that it can be taken wherever it is needed to go (most often unknown beforehand).

Small Methods to Express Intent

The authors reflect on writing helper methods to wrap up small amounts of code. I think it is in general a good idea, because it shows what bit of code responsible for which behaviour. It makes the code more readable.

Logging Is Also a Feature

The authors note that many teams would regard writing `XMPPFailureReporter` overdesign and just write the log message in place. Again, the SRP is put forward why one shouldn't do this. Furthermore, they argue, production logging is an external interface that should be driven by requirements of those depending on it and not by the current implementation. I fully agree.

Chapter 20: Listening to the Tests •

Andrew McDonnell, 2 May 2011

This chapter is about "test smells": common patterns that are difficult to write tests around, and how to change them. The main message is: when you find a feature that's difficult to write tests for, think about why it's difficult, not just how to write the tests. There's quite a bit of information in this chapter, so this review has become fairly lengthy also.

Lest we forget, one of the Ds in TDD is for Driven - those tests are there to help clarify and guide the structure of the code. At the risk of sounding incredibly redundant, the Tests are Driving your Development. That's true in the obvious sense of "write a failing test that describes new behavior, implement it (and then refactor)", but also in the sense of using the act of writing tests to learn about and improve the general structure. When something is hard to write a test for, it often means there is room for improvement in the design.

The authors distinguish between two kinds of test smells. The tests themselves may be unclear or brittle, in which case you should put this book down and go to the Test Smells chapter of Gerard Meszaros's book "xUnit Test Patterns". Alternatively, the test difficulty may be pointing out trouble in the target code - in which case you should keep reading.

1. I need to mock an object I can't replace (without magic)

a. Singletons are dependencies

Commonly-used objects or resources are often accessed through a global structure, usually a singleton. The example given is the Java Date object, which uses the singleton System to get the current time. This makes it difficult to test behavior of a class that depends on the period of time between two events (in the example, rejecting a second request that occurs on a different day from the first request).

```
final Date now = new Date();
// compare now to dateOfFirstRequest
```

The point here is that Date is a dependency, but it's not obvious because it doesn't appear in the interface. The solution is to make the dependency explicit by requiring a Clock object in the class's constructor. Now you can mock the clock so testing is easy, and the dependency is clear.

```
final Date now = clock.now();
```

I don't totally understand why this applies to singletons in particular. It seems like the real litmus test is if an object can change state independent of the class under test. These may often be singletons in practice, but that's more a symptom than the cause. I guess the point is that use of singletons (any non-value type that isn't specified in the class interface, even?) is the smell.

b. From procedures to objects

We listened to the tests enough to write the test we wanted, but they're still talking. Now that there's a Clock class, we can put the logic that checks if two dates are different in there rather than in the first class. This makes perfect sense, because comparing dates isn't part of the original domain, but fits perfectly into a clock domain. `clock.dayHasChangedFrom(dateOfFirstRequest)`

But we're still keeping around a clock object and explicitly comparing the dates with it. Even better would be to use a class that encapsulates the actual comparison we want. So instead of using the clock to get a date and compare it, create a class that does exactly what we mean. `sameDayChecker.hasExpired()`

Now Date and date logic has been completely removed from the object under test (this seems like budding off, in terms of Where Objects Come From on page 60), so it's free to concentrate on its true responsibility. And just think about how easy it will be to change the behavior if you suddenly need to support requests within two days of one another! Instead of having a clock dependency, the class has a time policy dependency, similar to Item::allowsBid that I liked so much from chapter 18.

c. Implicit dependencies are still dependencies

Don't use global values, even if they're just sitting there for you. What you're doing is hiding your dependencies, which leads to anger, then hate, then suffering. And as the sidebar notes, even though there are tools to break dependency problems for tests, you're better off listening to the tests and fixing the root of the problem.

2. Logging is a feature

There are two basic reasons for logging: support (errors and info, for the benefit of operators and support on a running system), and diagnostic (debug and trace, for the benefit of programmers while they develop). This implies we should treat these differently - support logging is part of the project and should be test-driven according to someone's requirements, but diagnostic can be more casual.

a. Notification rather than logging

In the code example, the authors note the shift in vocabulary and style between the functional and logging parts. This makes it obvious it's doing two things and thus breaking the Single Responsibility Principle. Instead, you can break logging into a notification class. This makes testing easier, of course, and makes the code read more clearly.

b. But that's crazy talk...

Well, the authors make a compelling argument that it isn't crazy to encapsulate support, for all the same reasons you encapsulate anything else - write code in terms of intent, easier to change, easier to be consistent, forces clearer thinking about the intent. As for diagnostic logging, it may or may not be encapsulated, but either way you can make that decision based on your needs.

I think the smell to investigate here is the existence of logging code, period.

3. Mocking concrete classes

That is, as opposed to mocking an interface - inherit from the class and override the methods that will be called in the test. That's your smell. The major problem here is that your test is making the interaction between objects implicit. This is a subtle point (to me, anyway), but important. You're overriding a method in your mock object because you expect the object under test to call it. However, knowledge of that relationship is only expressed by the override. What you really want is an explicit way of defining what the OUT requires from the mock class - that is, an interface. In the example, we also find that the interface required by the OUT is much thinner than the one provided by mock object. According to Robert Martin's Interface Segregation Principle, the OUT should only depend on the interface it requires. So smelling the mock concrete class shows that we've overspecified the OUT by requiring this concrete object, but are also lacking information about what parts of that object the OUT really requires.

The solution is obviously to extract an interface that the mocked class will implement and the OUT will depend on. Beyond making it easier to test, this forces you to think about the relationship you discovered, which is a good thing.

a. Break glass in case of emergency

Sometimes you may have to put up with this. Legacy code or third-party code (although you can usually write an interface layer to it) are common examples, but you should still feel pangs of regret. Regardless, only override visible methods. If you can't get what you need, the tests are telling you to break the class up into smaller features.

4. Don't mock values

There's no reason to mock values, just create an instance and go. They should be immutable anyway. Some heuristics to see if a class is a value: Its own values are immutable (although it may be an adjustment object, a policy like sameDayChecker above). You can't think of a meaningful name for a class that implements an interface for the type (and InterfaceImpl doesn't count as meaningful!).

If your reason for wanting to mock a value is that it's too complicated to set up, see chapter 22 about builders.

This smell doesn't mean you should do anything in particular with the domain code, but it will make the tests easier to write.

5. Bloated constructor I

When you have a constructor with a lot of arguments (good, because you've been extracting the dependencies), you're going to have a lot of setup to do in your tests (bad, a pain and leads to brittle tests). When you smell a lot of constructor arguments, hopefully some of them can be combined as a coherent concept. Beyond things that obviously go together, look for objects that are always used together, and objects with the same lifetime. Then comes the hard part, figuring out a name that describes the combined concept.

6. Confused object (Bloated constructor II)

This smells just like a Bloated Constructor, but the root cause is different. The problem is a class with too many responsibilities and/or unrelated responsibilities (either one is breaking the SRP). Michael Feathers' "Working Effectively with Legacy Code" addresses how to break up classes like this. You'll also notice that the test suite for a confused object is confused itself. Different tests won't be related, and you can make major changes to one section without changing another. If you can slice your test class into independent units, you may want to do the same thing to the object under test.

7. Too many dependencies (Bloated constructor III)

Are all those constructor arguments really dependencies? Don't forget the Object Peer Stereotypes back on page 52 - some of them may really be adjustments, so you can set them to a default in the constructor (including a null object) and provide an interface to adjust them later. One hint something isn't a dependency is that it isn't Java-final.

8. Too many expectations

Throughout the text the authors have made a point of using allowing() for method calls that will happen but aren't really what's being tested. Expectations should be used to highlight what's important to the particular test - these are "assertions we want to make about how an object interacts with its neighbors." This is in opposition to stubs, which are "simulations of real behavior that help us get the test to pass." In addition to distinguishing between expectations and stubs, the sidebar recommends using few expectations in general. If a test has a lot of them, it's probably because the unit you're testing is too large (break it into several classes) or you're specifying too many of its interactions (use more allowing).

They also note that the example code given for this smell could be changed so that all the expectations made in the test aren't needed at all, even as stubs, which would be an even better solution for this particular case. The initial symptom is a hidden train wreck (sprawled over two lines) going through a chain of objects to get the one that's needed.

9. What the tests will tell us (if we're listening)

Here's what you'll get from listening to yourtests.

a. Keep knowledge local - compartmentalized objects are easy to swap in and out
b. If it's explicit, we can name it - being able to name both objects and relationships between objects mean having an explicit understanding of the structure
c. More names mean more domain information - your code will be written more in domain terms than in implementation terms
d. Pass behavior rather than data - "Tell, Don't Ask". This tends to favor using callbacks instead of passing values up the stack, and gives you better information hiding and code written more in domain terms.

In general, keep your unit tests tightly focused and small. Listen to tests, so both the tests themselves and the actual application remain manageable.

I liked the code examples in this chapter, and the descriptions of how to look at and think about code structure from the tests' perspective. Lots of good advice, and more importantly lots of insight into the thought process.

Chapter 21 : Test Readability •

Clive George, 5 May 2011

I was quite pleased to get this chapter because I'm keen on having readable code having experienced the opposite at various times in the past, so am interested to see how the same ideas apply to writing the tests.

The first interesting point was in the introduction, where it was explained that although both code and tests require readability, actually tests need a slightly different sort of readability.

We then have descriptions of ways this readability can be achieved.

Test naming

First is test naming - and naming is another thing I'm very keen on. Naming is a first step to control, and this echoes bits which have been talked about before - just as if you can't test it, maybe you should try a different approach, if you can't name it, maybe that's a clue that there's a problem.

The naming convention suggested, TextDox, reminded me of user stories (1) in that there's a standard sentence structure used for the test name. Using a full sentence may seem a little alien to people not used to writing tests, and I imagine nobody would do that for code (hence the different sort of readability), but the small amount of time I've spent on tests (2) has shown me it's worthwhile.

One useful practice suggested was extracting the documentation from the tests - if done appropriately, the fully named tests come out as standard readable language rather than `gertLumpsOfCamelCase`, and reading this can sometimes highlight problems in the original name.

Test structure

Second is test structure. I noticed it's second - although the test structure is where the work happens, the name comes first and is thus more important in some ways.

Like all idioms, patterns, etc, a standard test structure means people know what to expect - where to look for setup, teardown, etc. There are a couple of different structures required, depending on how the test is performed - eg with mock object expectations, these are set up at the beginning and the assertion becomes implicit, rather than the assertion being explicit.

Once again, as with names, the idea that the structure is also an early warning of problems is mentioned - if the test won't fit the structure, maybe something about the test needs to be changed.

There's a nice hint on how they actually write the tests - name, call to target code, then expectations and assertions, and finally the setup and teardown to support these.

We then have some detail on bits which will help implement the structure. I'm struggling a little bit to say anything about these - I suspect I'd do better if I were to actually try writing something. Ignoring exceptions when they're irrelevant to the test passing was a nice bit, and I suspect "Delegate to Subordinate Objects" will become much more important on a larger application project - the WindowLicker reference here supports that.

The assertions and expectations section is a couple of simple ideas - make sure they're what's required, no more, otherwise you get bogged down in detail and things become brittle (harder to change), and watch out for `assertFalse` - negatives can be a problem in language, especially if there's more than one, and they are here too. Even if as an American one could care less :-)

And finally a reminder that tests need concrete values, which means magic numbers and constants. Just like for every other magic number, giving it a name makes it more useful, and tells us what it actually means.

Summary

In summary, a lot of this is simply basic good programming practice - but it's got a slight change of emphasis because testing has slightly different requirements. Making the code describe what the test does is more important than creating a beautifully abstracted out extensible structure where you can't actually see what's happening. KISS?

(1 I reread the relevant paragraph and saw this was deliberate)
(2 I'm here to learn - I've never managed to get TDD into our apps, and the closest I've got to trying it is in Jon Jagger's CyberDojo at the conference)

Chapter 22: Constructing Complex Test Data •

Ed Sykes, 9 May 2011

The goal of this this chapter is techniques to address brittle tests. By this I mean tests that break easily when the code changes and are difficult to change to meet the requirements of the new code.

I think his is a really important goal since many people give up on testing because of pain when trying to later change the code. This is the reason for test suites being abandoned when the code changes.

Constructing the test data is the input face of the connection between code and tests. In this chapter they foes on this.

Introduction

The authors start out describing a pattern called Object Mother. This pattern is a simple way to remove some duplication but breaks down in the face of variations in the data needed. Each variation requires its own method definition. For example imagine a Car class with 3 fields Wheels, Colour and Sunroof. Each combination of field values requires a method definition Eg.

```
newCarWith4WheelsWithColourRedWithASunroof(){...}
newCarWith4WheelsWithColourWithoutASunroof(){...}
newCarWith4WheelsWithColourBlueWithASunroof(){...}
```

This explosion of combinations is a problem since it is a binding site between test and code. Each one is a place to change when the code needs to change. We have some test code that looks exactly like this. Luckily, in 2.5 years, the code hasn't changed. When it does need to change we're going to be stung by this.

Test Data Builders

An improvement to Object Mothers is to use Builders. These allow chaining of variations e.g.

```
new CarBuilder().withWheels(4).withColour(Red).withSunroof(true).
```

This moves away from cartesian product of values expressed in methods to a system that allows combinations around the fields. This reminds me of the bridge pattern in OO in the way that it reduces the combinations into the dimensions that produce those combinations.

It's important here to note that the objects constructor is referenced once only. A version of this pattern exists where each method calls the target object constructor. Each method copies values from the current object into the target object constructor. This pattern should be avoided. The single constructor pattern in the build method has much looser coupling.

Once the refactoring to Builders is done the authors demonstrate the reduced noise and increased clarity of the code. This is done by giving an example where calling the wrong chained method is easier to spot.

Creating Similar Objects

An example is given for the re-use of a Builder to create two target objects of the same type. This highlights the difference between the two objects which helps to create a test that communicates better. This technique works best when the variations are in the same field in the test data. The authors describe some techniques for reusing a builder when the test data must vary across different fields.

Combining Builders

The authors demonstrate that when you need to combine Builders to create complex test data you're better off passing Builders to each other rather than the target objects the Builders create.

Emphasising the Domain Model with Factory Methods

The authors condense the creation of Builders into factory methods. This changes the language from being about Builders to being about the domain. Eg:

```
Order order = new OrderBuilder().fromCustomer...
```

vs

```
Order order = anOrder().fromCustomer...
```

They then talk about using the type system to collapse all the 'with' methods into a single overloaded method. So, using the car example i used earlier, instead of withWheels(int), withColour(Colour) and withSunroof(bool) you would have with(int), with(Colour) and with(bool). This presents a problem though if the same type is used to represent multiple domain types. If there were another method withDoors(int) you can't use overloading to distinguish between with(int) referring to wheels and with(int) referring to doors. The suggested fix is to create the domain types in your code. So you should have with(Wheels) with(Colour), with(Sunroof) and with(Doors). This allows the syntax of a chained request to look something like this:

```
newCar.with(new Wheels(4)).with(Red).
       with(Sunroof.Automatic).with(new Doors(5));
```

It looks like there is an error in the kindle version of the book here. The two examples that they give look identical to me:

```
Order order = anOrder().fromCustomer(aCustomer().
                        withAddress(anAddress().
                        withNoPostcode())).build();
```

and

```
Order order = anOrder().
              fromCustomer(aCustomer().
              withAddress(anAddress().withNoPostcode())).build();
```

Removing Duplication at Point of Use

This section is about removing duplication in the use of the target objects created by the Builders. The previous sections focused on removing duplication in creating the target objects.

The authors show a refactoring that moves the following things to a helper method:

- Builder creation
- target object creation by the builder
- target object use

The helper method `submitOrderFor` hides the detail nicely. However, there is a downside. Any variation in the details of the order causes an explosion of combinations. A helper method is required for each combination of variations. This is the same problem as trying to combine Builders. We solved that problem by passing around Builders. Again, injection of the Builders is the key. Contrast the two different client code examples:

```
int customerRef = 1234;
submitOrderFor("DeerStalker Hat", "Tweed Cape");
submitOrderFor("DeerStalker Hat");
submitOrderFor("DeerStalker Hat", "Tweed Cape", customerRef);
submitOrderFor("DeerStalker Hat", customerRef);
```

For each new variation a new method definition is required (4 here). Now with injection of the builder:

```
submitOrderFor(anOrder().withLine("DeerStalker Hat", 1).
                         withLine("Tweed Cape", 1));
submitOrderFor(anOrder().withLine("DeerStalker Hat", 1));
```

Only a single helper method is needed. Note that in the examples in the book they also changed the name of the method to `sendAndProcess`. I've left it the same here to make the comparison.

I've seen this exact problem in our code base. Once or twice I have stumbled across a builder to solve the problem. I didn't understand what was happening though. This makes it very clear what the problem is and I think I'll be able to start removing this explosion of variations.

Raise The Game

The authors then do a very simple refactoring on the helper method name. They change it from `sendAndProcess` to `havingReceived`. They're demonstrating

something here which is quite profound, I think. They talk about it briefly here and expand on this idea later in the book. It's worth discussing here because it's at the crux of trying to communicate with the reader. Often when naming methods we do so to communicate from the methods perspective, from the implementation perspective. The method name 'sendAndProcess' communicates in the language of the implementation. The advice from the authors is we should instead name methods in the language of client.

The authors describe this as choosing a method name from the client's point of view. I think, more precisely, that the method name should be chosen from the reader's point of view as they read the client code. The name 'havingReceived' is not related to what they client is doing. It's related to the effect the client is having upon the domain. The distinction is subtle. It's an important distinction because it helps guide naming that is phrased in the language of the domain model. The client code, the test in this case, isn't expecting to receive the order. The domain model receives the order as a result of the test code. The name of the helper method reflects the need to communicate to the reader the effect of the client code (the test code) upon the domain model. It's fascinating that so many things come back to thinking about the domain model when trying to communicate with the reader.

I think there is a potential guideline here to help naming methods in general. The guideline is that you should think about the system (aka domain model) that the client code is trying to change. As an example, a class that manages a linked list: in order to fulfill some functionality the class needs to add an item to the linked list. The method name to do that is `linkedList.Add(item)`. This is phrased in the language of the linked list domain. It's what is happening from the point of view of the effect the client has upon the domain model. If we were to choose a method based on the language of the implementation it would be something like `setNextPointerInFinalItemToPreviousPointerInNewItem(item)`. In this example it's obvious what the method name should be and easy not to make that mistake. I think that's only because the language of the domain (linked lists) is so well established. In a domain where the language isn't as well established I think I make this mistake in choice of language. My current approach is to think of a number of method names and then pick the one that feels instinctively correct. I think this guideline can take me away from that scattergun approach to something more methodical. This is definitely something that I'll be trying and thinking about over the next two weeks. I'll report back any insights to the group.

Communication First

The authors round off the chapter with an explanation of how the techniques in this chapter reflect their obsession with the language of the code. They mention a couple of tools, FIT and LIFT. I'd never heard of LIFT, it stands for Framework for Literate Functional Testing. They also make a statement at the end of the summary that, like the LIFT team, they achieve much of what FIT gives you whilst staying within the development toolset. I might be reading a bit too much into that statement: it seems like a swipe at FIT. A question for Steve and Nat: Is there something behind that statement?

I really enjoyed this chapter. Especially the recognition of a number of pain points that I am grappling with and trying to explain to my team at the moment. Lots of good stuff to take back to them to help us get the most out of our unit testing.

Chapter 23: Test Diagnostics •

Ken Duffill, 12 May 2011

First let me point out that I like the little quotes that Steve and Nat have put at the beginning of each chapter. This one: "Mistakes are the portals of discovery. - James Joyce" reflects my own philosophy and echoes one my father always taught me "The man who never made a mistake never made anything". This philosophy underpins my dedication to iterative development practices, which in turn underpins agile and hence TDD.

The chapter starts off by pointing out that it is important to note that a failing test is actually doing its job! Provided the diagnostic messages that accompany the failure are helpful in identifying the cause of the failure it is actually GOOD to see a failing test.

Of course by the time the code is released, all the tests should pass and so it is very easy to not see a failure message for days, months or years. So the tests must be coded in such a way that when some regression failure later causes one or more to fail, the diagnostics point immediately to the cause of the failure without relying on any knowledge or context being in the head of the developer assigned to investigate the failure.

To me this seems to be one of the major reasons to adopt 'Test First' as opposed to adding tests to an already (apparently) working codebase. It is so much easier to see the diagnostic output when you are constructing a failing test than later when you may have to artificially make the code break in order to test the test.

Of course seeing the diagnostic output is one thing, ensuring that it will make sense to someone else months later is something different and isn't easy. It takes focus and some determination.

During the earlier chapters of this book the authors have been careful to evolve the tests in such a way that reading the code of the test explains the tests purpose well and this chapter reminds us of the importance of this as well as extending that idea to making the diagnostic output so meaningful that it might not even be necessary to refer to the code at all to understand what is going on when a test fails. The authors remind us that when changing both production code and test code, it is important to synchronize with the source control repository frequently. This provides the ability to roll back easily when the code starts to become less, rather than more, expressive. And it also makes us feel more comfortable if we need to 'experiment' as we know we can always go back to working code if we decide to throw away our experiment and start again.

The first step in making the test failures expressive is to ensure that the purpose of each test (the target failure mode we are trying to capture) is clear in our own mind and clear in the code of the tests, ensuring that the test is well-named is important here. The second step is to ensure that failures produce meaningful messages.

Help is available from JUnit's assertion methods that provide for an assertion failure message. Using this feature doesn't seem to be as common as is should be, they say, but its use can change an uninformative message like

```
    Comparison failure: expected: <[16301]> but was: <[16103]>
```

into something much more useful like

```
    Comparison failure: outstanding Balance expected: <[16301]> but was: <[16103]>
```

In the example environment, which was set up early in the book, there is additional help provided by using the Hamcrest Matchers. Throughout the book the authors have suggested ways to make the test code more and more expressive using these matchers and they refer in chapter 23 to a matcher set up on Page 252 as an example. Unfortunately, in my Kindle version of the book there are no page numbers, and in this instance there was no link, so I could not easily find the source for the matcher (I eventually found it towards the end of Chapter 21), but the output is shown and does explain the failure well. [In an email from John Penney on Tuesday in response to the Chapter 22 review he suggests that .NET extension methods might be useful in producing similarly expressive code in a C# environment.]

An alternative to using matchers (or rather an additional tool in the test writer's armory) is to replace hard coded values with 'Self Describing Values' though there is no indication of how they make the test report output 'a customer account id' instead of its value in the output:

```
    Comparison failue: expected: <[a customer account id]> but was: <[id not set]>
```

Maybe 'Self Describing Values' are a common idiom in Java, so they didn't feel the need to explain, but I got a little lost here.

It is clearer how we can do the same with reference types and the example producing the diagnostic report

```
    java.lang.AssertionError: payment type
    Expected: <startDate>
         got: <endDate>
```

Shows clearly the benefit of doing so.

The authors add that with tests written to produce such self describing diagnostics, there is probably no need to add the assertion failure message as well.

However they do point out that simple types may not be suited to being converted to Self Describing Values and so some 'Obviously Canned' values (that are just hard coded values that should not occur naturally) could be used in the tests, though the team should develop a conventional set of Obviously Canned values so they do indeed become, well, obvious.

Next they describe a Tracer Object, which is simply an object that is a dummy object with no supported behavior of its own, except to describe its role when something fails. They suggest using of Tracer Objects as substitutes for domain concepts not yet implemented during the code evolution.

Another useful tip is to make sure the tests explicitly assert that all the expectations were satisfied. In tests that have both assertions and expectations this is important so as to ensure that a failure due to a missing collaboration is not simply reported as a wrong value for some object that has not been modified by the collaborator properly.

The final section of this chapter reminds us that to achieve the desired effect of having failing tests easily indicate the source of the problem we need to treat Diagnostics as a First-Class Feature of our code. To this end Steve and Nat extend the simple Three Step TDD Process (fail, pass,refactor) into a four step process (fail, report, pass, refactor), whichthey then expand to a diagrammatic form of:
1. Write a failing test
2. Make the diagnostics clear
3. Make the test pass
4. Refactor
Go to 1.

[Ouch I have used a goto! - so that's why they did it in a diagram]

This was a relatively short chapter and highlighted, for me, some very useful insights into what makes good tests still good when returning to them some while after they were written.

Chapter 24: Test Flexibility •

Timothy Wright, 16 May 2011

This chapter continues with advice about tests. This time the focus is on test flexibility. We don't want the tests to be brittle and break with any small change of the code. I definitely relate to this, my tests are way too brittle, they break with nothing to do with its job.

We want the tests to only fail when its relevant code is broken. Reasons for this are 1) tests are too tightly coupled to unrelated parts of the system, 2) tests over specify the behavior, the tests are doing too much work and 3) duplication in the tests. Bad tests are also a reflection of the design of the system. If the system does not allow for well separated tests, then it can be improved.

A good test is one that is focused, easily setup, and has minimal duplication. It is easier to name and easier to understand.

The summary sentence for the chapter is: Specify Precisely What Should Happen and No More.

Test for Information, Not Representation

Tests should be written in terms of the information passed between objects not of how the information is represented. The example in the book uses a `CustomerBase`. Tests checks if null is returned of there are no customers. But that is testing the representation not the meaning of the information. Plus we have a rule not to pass null between objects. So a maybe object is created. If null is used everywhere, it is hard to replace the null with maybes. However, if the null was represented with a constant named `NO_CUSTOMER_FOUND`, then it would be easy to replace the constant with a `Maybe<Customer>` object.

Precise Assertions

We want precise assertions targeting only what's relevant to the scenario being tested. Testing for equality does not scale well as the value being returned might change and become a more complex value type. This can take various forms. First, it could be a complex value type. We can just assert the attributes we are interested in. In the following example, we are only interested in the strike price.

```
assertEquals("strike price", 92, instrument.getStrikePrice());
```

We can use Hamcrest matchers to make things more expressive. Sometimes we are only interested in relative size.

```
assertThat(instrument.getTransactionId(), largerThan(PREVIOUS_TRANSACTION_ID));
```

Another source of complexity is testing of strings. We need to test string messages. We don't want to be dependent on the actual message, but just look for keywords. The critical information.

```
assertThat(failureMessage, allOf(containsString("strikePrice=92"),
        containsString("id=FGD.430"), containsString("is expired")));
```

If we find a lot of string tests, the tests might be telling us we are missing an intermediate structure object like an `InstrumentFailure`. It can be tested for all the information but the actual string could be generated at the last possible moment.

Precise Expectations

We can also be precise with our expectations. jMock is designed for specifying this communication between objects.

Precise Parameter Matching

In the `AuctionSniper` we have

```
oneOf(auction).addAuctionEventListener(with(sniperForItem(itemId)));
```

Here we are only testing the `itemId` of the `AuctionSniper`, we don't care about the other states. We are focusing only on what we care about in the tests.

Allowances and Expectations

Sometimes we don't care about a particular object. We can with jMock, allow anything to happen to an object. Allowances may be matched or not. Expectations must be matched. Allowances support the interaction we are testing. They can be used as stubs to feed values into the object. This is still a little murky for me. We can ignore some behavior, allow some behavior that does not have to happen or we can require the behavior to happen. Why in a test are we ambivalent about a behavior?

I think that using allowing can create a stub that will return a known value like:

```
allowing(catalog).getPriceForItem(item); will(returnValue(74));
```

The value 74 will be returned when `getPriceForItem()` is called. This could be used in the test. However, this continues my confusion. Is allowing for optional stuff or for creating stubs we have control over in the test?

The book has a side note that groups allow commands with queries that don't change state, and expectations with command that do change state. This does not help my confusion about how to use `allowing()`.

Ignoring Irrelevant Objects

We can simplify tests by ignoring collaborators that we don't need. jMock will not check any calls to ignored objects. It will provide default or zero return values for ignored methods for convenience. There is a table of the return type and the zero value returned which I won't repeat here.

This is a powerful tool to keep the scope of the test narrow. However, too much use of the ignore feature of jMock could be a sign of something wrong, that functionality needs to be pulled out into collaborators.

Invocation Order

jMock allows invocations on a mock object to be called in any order. It also allows for specified order in sequence. If the invocations order is more complex, then the jMock state machine can be used.

The book notes that we should only enforce invocation order when it matters. Otherwise we are over specifying the tests causing it to be brittle.

This jMock feature is important to specify the protocol of the object interactions. The example in the book is the need to search for auctions with specific attributes. The order the auctions are found are not important. But when the search has inspected all the auctions (`searchMatched()` is called), then we need to call `searchFinished()`. The protocol is that `searchFinished()` needs to be called last.

We could start with a test that does not care the call order of the search methods but that does not help us. Does not constrain the design enough. We could also be very rigorous, specifying the first `searchMatched()` call, the second `searchMatched()` and lastly the `searchFinished()`. But this over specifies the tests. We don't care the order the matches are found just that `searchFinished()` is called at the end. This is where the state machine in jMock can help.

The Power of jMock States

jMock states can be used to model the three types of participants in a test; the object being tested, its peers, and the test itself.

When testing the object as in the discussion above, we are confirming the protocol. If there are events that do not follow protocol, they fail the test.

We can use states to represent how a peer will change state and affect the object under test.

jMock also has plug-in points to support the definition of arbitrary expectations. The example in the book allows for calling any method from `aPeerObject` that has a get in its name with no arguments. It can be used to accept calls from one or a set of objects.

"Guinea Pig" Objects

Sometimes we write generic adaptor objects the translate domain objects into the system's technical infrastructure. We might be tempted to use actual domain objects in the testing of these adaptors but that would be the wrong approach. It will tie the adaptor's test to the domain. If the domain changes, then the adaptor's test will also fail when it shouldn't. Instead an object should be created for the sole purpose of testing the adaptor, removing the coupling with the domain code. This class acts as a "Guinea Pig" allowing for testing before releasing to production code.

This style of TDD depends on the jMock and hamcrest tools. I know that there has been other discussions about the state of these tools in other languages. How easy is it to use this style in C#, C++, python, etc? I know that there is a version of the book example in C# and partially in ruby. I need to look at them.

Chapter 25: Testing persistence •

Olaf Schleusing, 19 May 2011

Chapter 25 is about testing code that handles persistence mechanisms such as Object/Relational Mapping (ORM), i.e. mechanisms that enable transformations of objects to persistent storage (e.g. database entries) and vice versa. As it was highlighted in Chapter 8, the link between the application domain and a usually given API of a persistence system should best be implemented by a thin Adapter layer (see also Adapter pattern of the GoF). Several details of this interface implementation should be tested:

- Correct issuing of queries
- Correct mapping between objects and relational schema
- SQL dialect is compatible with used database
- Adherence to integrity constraints of database
- Correct communication with transaction manager
- Correct and timely management of external resources
- Prevention of bugs tripping over from the database driver into application domain

Furthermore, our test code has to deal with some additional complexity. Since we are working with persistent data, we need to add some logic to isolate tests from each other. In particular, database entries should be deleted before each test in order to obtain a dedicated setup of the persistent store before the execution of each test. To achieve this, the authors propose to defer the code for erasing database entries into a subordinate object, which is called by any test before its execution.

Aside The purpose of cleaning up persistent data before each test is to have the ability to diagnose errors of previous tests with the aid of remaining database entries (see also "Recording the Failure" in Chapter 19). **Aside end**

Explicit Transaction Boundaries

The next section I had to read a few times but I am still not sure where I am missing something (**raising flag for help wanted**). Each test is embedded in a <transaction> in order to isolate it from other tests. Once a test is finished, the transaction is rolled back. The idea is to leave the persistent state the same after the test as it was before.

I think what we said is that this is a technique that people use and that we don't recommend it for exactly the reasons you describe. S.

This last statement is what confused me. It somewhat contradicts what had been stated before in terms of "Recording the Failure", i.e. to clean the persistent store only before each test and leave it as it is thereafter. As the authors point out, cleaning the database after the test from entries that changed during the test makes it also difficult to test certain ORM operations such as commit.

To continue, transactions are used to make the boundaries between tests stand out. They are implemented by a subordinate object called transactor. The transactor is passed a unit of work, which is part of the test itself. The transactor performs all the necessary background work to setup and clean the database before and after the test. This reminds me of the Strategy pattern of the GoF.

In their first example, the authors apply the above to a customer base by extracting customers that meet a search criterion from persistent storage. The test contains two methods; one creates customer objects with different values for the respective criterion and the other asserts the values under test. Each of those two methods makes use of the transactor, thereby cleanly separating the transactions and the actual test code that we are interested in.

Verifying that objects can be persisted

Another particularity of tests for persistence code is that it may be difficult at times to interpret its error messages. More tests may help here to pinpoint particular problems. For example, "Round-trip"-tests verify that objects are transformed into a persistent representation and then back to objects correctly. Obviously, the persistent fields of each object should be retained. In the book, a second example demonstrates a persistability test that makes use of the data builders back from Chapter 22.

This example is further extended, by allowing relationships between entities that might never be instantiated during a transaction. The solution to this problem is to wrap builders of objects possibly not in existence in a Decorator (Decorator pattern of GoF), which persists the respective objects whenever necessary.

A final remark addresses the issue of speed related to database tests. Since we have wrapped the database API in an Adapter layer, we can mock the persistence implementation and have fine-grained and fast unit tests of our interface implementation. End-to-end tests then will give us more feedback about the integration of our code with the external services such as databases that depend on additional configuration and typically execute slower.

Chapter 26: Unit Testing And Threads •

John Penney, 23 May 2011

Introduction

In this useful chapter, the authors start to consider how best to test concurrent code. This topic is continued in the next chapter.

The authors start with a useful introduction that basically summarises why testing a unit that has an element of concurrency can be tough.

- You have to be concerned about the synchronisation in the unit and between the unit and the test.
  Only last week I discovered a test that had been passing "by accident" - some missing interlock between the test threads meant that there was a window of opportunity for the test to fail.
- Exceptions can be swallowed by background threads.
  Yep: Our UT suite ran just fine when we ran it from the IDE, but failed when run from the CI server. An uncaught exception from a background thread was the cause.

They advocate designing in your concurrency, especially if you want to use a framework or library with threading built in. There are formal modeling techniques that can be used to prove your design is free of "certain classes of synchronization errors". However, no matter how carefully considered your design, unit testing can give us additional confidence that the code carries out its synchronization responsibilities.

Separating Functionality and Concurrency Policy

In this section, the authors explain that a class that has concurrency in mixes functional concerns with synchronization concerns. They show one approach for decoupling these concerns and testing them separately.

Searching for Auctions Concurrently

They use the Action Sniper example to show how a concurrent search facility can be added (the search runs concurrently as the user enters search words). In this example they show a key technique: isolate the concurrency to one unit. This means that the other functionality can be tested without being worried about concurrency issues.

Introducing an Executor

For this example they create a "task runner" to create threads for tasks, using a Java interface called Executor. Conveniently, this has a jMock implementation called `DeterministicExecutor` which allows us to "run the tasks on the **test thread** after the call to the tested object has returned".

I guess that if you don't have a `DeterministicExecutor` in your toolkit then it wouldn't be hard to write an implementation of Executable that saves up the executable tasks and then runs them deterministically when your UT asks.

Implementing AuctionSearch

There's a bug in `ActionSearch`: the `runningSearchCount` is not synchronized. The authors set about "driving out" the synchronisation issue with a test.

Unit-Testing Synchronization

As we don't have precise control over the thread scheduler, we can't guarantee that our tests will find synchronization errors. But stress testing our code by running the same code lots of times gives us a good chance!

One interesting approach to designing stress tests is to consider "the object's observable invariants with respect to concurrency". This is demonstrated by adding a stress test to find the `runningSearchCount` bug: they use jMock to help them write a test that runs the search multiple times, with the expectation that the auction will finish just once. Fixing this test required changes in more places than the authors expected!

Safety First (sidebar)

Steve and Nat suggest that both functional *and* stress tests should be written, then made to pass, before checking in, to avoid checking in code that passes functional tests but contains concurrency errors.

Stress-Testing Passive Objects

In this section, Steve and Nat point out that most objects concerned with threading don't actually control threads but have threads "pass through them". (In other words, they must be thread-safe!)

To stress test such objects, a unit test must start its own threads to call the object, and verify that the resulting state is the same as if the calls had been made sequentially. In the example shown, we use a helper object `MultiThreadedStressTester`. Having a suite of such utilities certainly seems to make life easier. I'm going to go a-hunting for a C# suite after writing this review!

Synchronizing the Test Thread with Background Threads

Suppose we're testing a unit that launches a thread to do some stuff. How do we ensure that the test thread waits until the stuff is done? The simplest mechanism is to add a sleep: however this is both tedious and fragile. Better to use a true synchronization object with a timeout (so the test doesn't hang in the event of failure). Steve and Nat use jMock's `Synchroniser`.

The Limitations of Unit Stress Tests

Stress tests can be helpful in pinpointing a defect if tests fail. I wish I'd written more of this kind of test in the past: this would give confidence not only on a change to the code, but an environmental change such as a new high- or low-powered machine or an upgraded or new OS. However, stress tests are no guarantee: the best you can do is run them often and in a range of environments.

Run unit tests *and* end-to-end tests. And in addition, modeling tools like LTSA and static analysis tools like FindBugs are valuable.

This chapter has considered unit-testing concurrent code: this sets us up for the next chapter to consider the much more complex world of large-scale testing of concurrent behaviour.

(I'd like to thank Northern Rail for giving me unexpected extra time to write up this review on my trains this morning...)

Chapter 27: Testing Asynchronous Code •

Paul Grenyer, 25 May 2011

Introduction

In this final chapter Nat and Steve describe asynchronous testing, that is how to write tests where the control returns to the test before the tested activity is complete. They explain asynchronous systems with an example of talking asynchronously to a server.

Sampling or Listening

There are two ways that a test can observe a system. It can either poll the system (sampling) or wait for an event (listening). It is often necessary to use both techniques to get good coverage of a system. When employing either technique you should make the tests detect success as quickly as possible to give rapid feedback. It is also important to make sure asynchronous tests can time out; or runaway tests can stop your test suit from ever finishing.

In the example of both techniques I, again, learnt that I'm not as good as I could be at separating out functionality into other classes. I wish I'd used something like the Probe interface in some of the projects I've been on.

There are a number of gotchas relating to asynchronous tests including tests that return the system to its original state and lost updates where a change in the system is missed by the test.

External Event Sources

I first heard Steve and Nat talking about an external event scheduler at their 2010 ACCU Conference session. I think it's a superb idea for system testing, especially as at the time I was working on a system where events can wait for several weeks to be triggered. However, I just couldn't convince my team. They felt, as is raised in the chapter, that it's not testing the real system and some other method such as changing some configuration setting was the way to go.

The chapter describes how externalising events can make testing simpler as the responsibility for firing events can be handled by the test when testing and an external event scheduler in production.

Nat and I exchanged some emails about an external scheduler at the time and I even knocked up most of an implementation. I've often thought that using a messaging service would be an excellent way to configure and trigger the events, but I've never got around it.