



Code Confidence Strategy

Prepared By:
Kenneth J. Pronovici, Cedar Solutions, Inc.
<http://cedar-solutions.com/>

License Terms

This document is copyright © 2016 Kenneth J. Pronovici. It is distributed under the terms of the Creative Commons Attribution 3.0 United States license (CC BY 3.0 US), which can be found online at this URL:

<https://creativecommons.org/licenses/by/3.0/us/>

In short, you are free to share and adapt this material as long as you give appropriate credit as described in the license.

Revision History

The latest version of this document can always be found at the following URLs:

<http://cedar-solutions.com/docs/confidence/CodeConfidenceStrategy.doc>

<http://cedar-solutions.com/docs/confidence/CodeConfidenceStrategy.pdf>

Below is the revision history for the document.

Revision	Date	Author	Reason
1.0	28 June 2011	Kenneth J. Pronovici	Initial revision.
1.1	26 Sep 2011	Kenneth J. Pronovici	Reference external files (i.e. Checkstyle) by URL.
1.2	31 Oct 2011	Kenneth J. Pronovici	Updates after soliciting feedback from colleagues.
1.3	07 Oct 2016	Kenneth J. Pronovici	Clarify license terms per client request.

Table of Contents

License Terms	2
Revision History	2
Table of Contents	3
Code Confidence	4
Test Strategy Should Influence Software Design	5
Layers and Interface Boundaries	5
The Container	6
Singletons are Evil	7
Code-Cleanliness Tools	8
Test Coverage	8
Coding Conventions	9
IDE Features	9
External Tools	10
About Warnings	10
Testing with Mocks	12
Interfaces	12
Mock Objects	12
Code Example	13
Other Topics to Consider	15
Other Tools	15
Continuous Integration	15
Code Reviews	15
Patterns and Frameworks	16
Appendix A: Eclipse Plugins	17
Checkstyle plugin	17
Emma plugin	17
AnyEdit Plugin	17
Appendix B: External References	18
Mockito	18
Spring	18

Code Confidence

When software is being built, developers tend to focus their effort on near-term goals, such as scheduled deliverables. Near-term goals are important, but what often gets lost in this thinking are longer-term concerns, such as how the software will be maintained once the initial (sometimes-frenzied) development push is complete.

My view is that maintenance considerations should be at least as important as the obvious near-term goals, especially since maintenance represents such a large portion of the total cost of a project over its lifetime. I personally approach all software projects with long-term maintenance as a major priority.

One of the ways to make long-term maintenance more manageable is to focus on what I think of as “code confidence”. When we have confidence in code we can:

- Make changes and deploy new releases without being afraid
- Bring on new developers quickly, without huge ramp-up times
- Comprehend the code and understand what it was intended to do
- Generally spend less time pulling out our remaining hair

This document discusses my strategy for creating code confidence. My strategy has the following major tenets:

- Rely on code-cleanliness tools to keep everyone honest
- Design your code to have a sensible set of layers
- Test in isolation at your interface boundaries
- Prefer solutions that do not depend on the container
- Avoid singletons in favor of dependency injection whenever possible
- Diligently refactor your code at every opportunity, so it stays focused your current needs

I started formally developing this strategy in 2007, and I have refined it while working on a variety of projects at several different customers. While this document focuses on Java, the strategy is generally applicable to any modern language, and I personally have validated it in both Java and C# .NET.

I do not consider this strategy (or document) complete. The strategy certainly does not represent a perfect solution. I plan to continue revising the document as I have time. However, I hope that anyone reading this document can find something useful to take away.

Test Strategy Should Influence Software Design

My theory about software development is pretty simple:

If you can't test your code in isolation, it isn't structured properly.

Yes, of course, that's a bit of a utopian statement. There will certainly be pieces of code that can't be tested easily. However, if you're careful, you really *can* test almost everything you write. To do this properly, you need to think through your application architecture ahead of time, to facilitate testing. The sections below discuss some factors that impact how easy it is to test your code.

Below, you'll see references to mock objects and mocking. For more information about mocks, see the section *Testing with Mocks*.

Layers and Interface Boundaries

The best way to write testable code is to create a set of layers with clear interface boundaries. Then, as you write your unit tests, you can focus on the interface boundaries. You don't have to write massive end-to-end tests. Instead, you write smaller tests that can safely make assumptions about the code on the other side of an interface boundary. Specifically, these tests can assume that the code on the other side of a boundary is working "properly".

If you choose this sort of strategy, you end up with more tightly-focused code. This strategy forces you to write both code and unit tests that know what they are supposed to know, and nothing else. Your code naturally becomes more modular, less brittle, and easier to refactor. Test cases become easier to create, because now you just need to set up conditions for one layer at a time, not conditions that make sense holistically through the entire system.

Let's take an example of a hypothetical batch application that also has a web interface. I would structure this application with the following layers:

Batch Jobs
Web Pages

Services

Data Access Objects
(DAOs)

Batch jobs and web pages are implemented exclusively in terms of the service layer and never interact with the DAO layer. Services are implemented in terms of other services, as well as DAOs. All database access happens in the DAO layer. Besides these three layers, utilities (i.e. string utility methods) are mixed in arbitrarily in any layer. We would use a dependency-injection framework (like Spring) to inject DAOs into services, and to inject services into batch jobs or web pages.

Now, when unit-testing this code, we only have to test at the interface boundaries. DAOs do have to be tested against the database. However, batch jobs, web pages, and services can (and should!) be tested without any database connection at all. Any given layer is tested by mocking the other layers it interacts with. Rather than injecting real objects, we inject mocks, and the class we're testing doesn't know the difference.

To prove that a batch job has been implemented correctly, you simply have to verify that the correct service methods are called with the correct data in the correct order. The batch jobs *do not care exactly what the service methods accomplish*. What's important is the interface, not the underlying behavior.

Likewise, you don't need a real DAO to prove that the service layer works properly. You just have to verify that your service calls the correct DAO methods properly. Tests for the service layer can assume that the DAOs work properly.

Depending on your application needs, you can create DAO-like abstractions for other third-party interfaces, too. For instance, I would argue that SOAP web service calls should always be abstracted behind an application-specific façade, which in this design would live in the service layer. Building an application-specific façade – including a set of interface objects that are independent of the WSDL – helps make it obvious which parts of the SOAP interface are important, and makes it easier to test the rest of the application. It also gives you an obvious place to handle any SOAP-related idiosyncrasies that you might find, without having to propagate that knowledge into the rest of your application.

Using well-defined interfaces like this can have other advantages outside the unit test realm, too. For instance, you could run the web pages against a “dummy” database to prototype certain user interface features. All you would have to do is stub out the DAO layer, and the rest of your code would continue to work properly.

There's a bit of an art in developing an interface like this, and you shouldn't be surprised if you have to revise it as time goes on. One technique that seems to work well is to develop the interfaces vertically – for instance, as you build the batch job, stub in calls to (nonexistent) service methods. Once your batch job is “done”, these method calls imply the service interface, which you can then create and mock to test the batch job. Then, you can move on and implement the service layer (mocking the DAOs), and finally implement the DAOs.

The Container

I have a love-hate relationship with the J2EE container. On the one hand, it does a lot of useful things for me. On the other hand, it conspires at every turn to make me dependent on it. Good application design exists at the fuzzy boundary where you rely on the container enough to get benefit from it, but not too much that it starts to control your life in unexpected ways.

I am happy to have the container manage certain things on my behalf. I like that there's a standard way to deploy applications, fine-grained control over who can start/stop/configure the application, a standard log directory, etc. I like having the container manage my database connections, queues, and topics. It's great when tedious administrative tasks (like updating SSL certificate stores) can happen outside of my application. These are all good things.

Unfortunately, having the container there also tends to encourage application design that *relies* on the container being there, and this is an absolute disaster from the perspective of code testing.

The classic example of this pattern is an older application based on EJB 2.0 session and entity beans. These EJBs can't be instantiated except when the container is running. Unless care is taken, you're usually stuck testing by hand (I've never seen a straightforward way to unit test an EJB running live in a container). Theoretically, EJB 3.0 fixes this problem, because it is possible to instantiate an EJB 3.0 bean outside the container. However, in practice, most EJB 3.0 applications I have seen are so tightly coupled that they can't be tested without the entire container being up and running. Strike one against testing in isolation.

A similar problem arises when you rely on features that are only available in the container. For instance, it's OK to wire things together using internal message-passing queues. However, if a side-effect is that you can't test anything unless your entire application is running in the container, then you've shot yourself in the foot. It's incredibly frustrating to have to deploy to the 'real' development environment just to test a 2-line code change.

Whenever possible, I think you're better off using a dependency-injection framework rather than a container-based framework. Dependency-injection frameworks are easier to deal with, and the resulting POJO (plain old Java object) classes are easier to test. Plus, without EJBs in the mix, you can avoid the temptation to rely too heavily on the container.

The one place where I think EJBs are worth it is for MDBs (container-managed message-driven beans). The container does a lot of work on your behalf when you deploy an MDB, and this is a big advantage. However, I still think that your MDB should be as thin as possible. I favor doing a minimum amount of JMS-specific work in the MDB, and then calling out to a Spring-managed service for all of the business logic.

Singletons are Evil

Ok, singletons aren't evil. But using them is.

The problem with singletons (and to a lesser extent, static utility methods) is that they contaminate your testing. Once you start to rely on a singleton, you're no longer testing in isolation. Your test always relies on the behavior of the singleton, which can make it extremely difficult (and sometimes impossible) to test certain scenarios – especially if the singleton or static method maintains some sort of state.

If you are using a dependency-injection framework, the best solution is to inject the singleton instance. That way, when you are testing, you can mock the singleton interface like any other DAO or service dependency. (Spring has utilities that are smart enough to instantiate your singleton via its factory method.)

Most of the time, you won't have problems with static utility methods, because they're usually simple enough that you'll never have to mock them. However, the more complicated your utility method gets, the more likely it is that you'll need to mock it to adequately test your code. This might be a hint that you should refactor your utility class into a service of its own. However, if you want to keep it as a utility class, then another solution is to turn the static utility class into a singleton, i.e. reference `MyUtils.getInstance().theMethod()` rather than `MyUtils.theMethod()`. Then, you can inject or mock the singleton instance.

Code-Cleanliness Tools

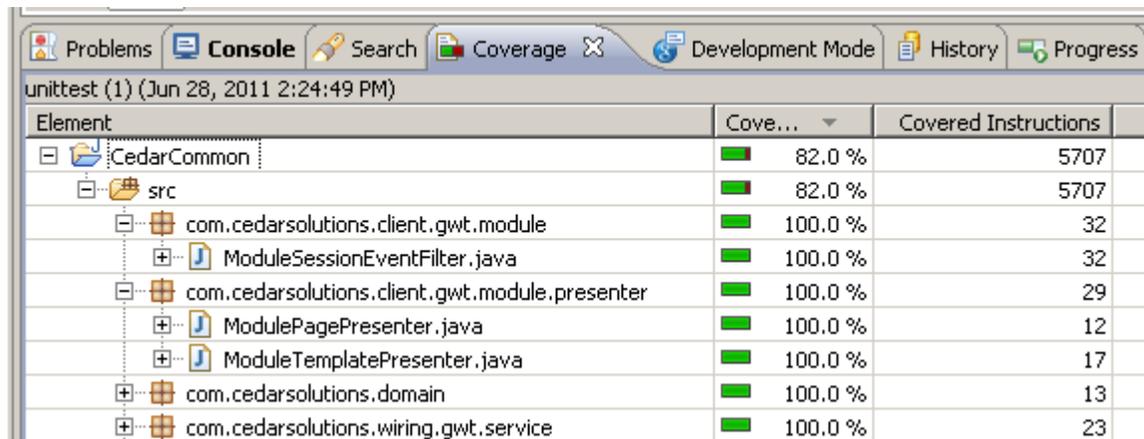
In the Java ecosystem, there are many tools that can make your life easier. I think you should focus on two major types of tools: tools that help you enforce a coding convention, and tools that help you understand test coverage.

Test Coverage

When I'm developing new code, I rely on test coverage tools to help me ensure that I have adequately tested the new code. Test coverage tools let you run your unit test suite and see how much of the code you have exercised.

Don't delude yourself into thinking that 100% coverage means that your code is bug free. Any coverage tool is counting which lines of code have been executed as part of the test run. It's quite easy to execute a line of code without fully testing its behavior. *A test coverage tool is probably best used to find code that you have forgotten to test, rather than to prove which code has been fully tested.*

For Eclipse, I like the Emma plugin. See *Appendix A* for more information about where to find this plugin. Once you have installed Emma, you can run your tests with **Coverage As > JUnit test** instead of the normal **Run As > JUnit test**. When you do this, Emma gathers coverage information and shows it to you in an interactive dialog:



Element	Cove...	Covered Instructions
CedarCommon	82.0 %	5707
src	82.0 %	5707
com.cedarsolutions.client.gwt.module	100.0 %	32
ModuleSessionEventFilter.java	100.0 %	32
com.cedarsolutions.client.gwt.module.presenter	100.0 %	29
ModulePagePresenter.java	100.0 %	12
ModuleTemplatePresenter.java	100.0 %	17
com.cedarsolutions.domain	100.0 %	13
com.cedarsolutions.wiring.gwt.service	100.0 %	23

You can even drill down into the code by clicking on individual files. Inside a source file, lines will be highlighted in either green or red depending on whether they've been covered by a test.

Coding Conventions

I am a strong believer in coding conventions. Code that is written consistently is much easier to read and makes it much easier for new developers to transition onto a project.

Pick a set of conventions, and stick with them. That means putting the braces in a consistent place, naming things consistently, and deciding whether to use tabs or spaces for indenting. Ultimately, it doesn't really matter whether everyone likes the conventions – you'll never please everyone. What's important is that everyone actually makes the effort to adhere to the conventions. Individual judgment is good – after all, it's what you're paying for when you hire experienced developers. However, you want to avoid pointless differences just for the sake of individuality.

I recommend that you enforce coding conventions using two different mechanisms. First, leverage the features in your integrated development environment (IDE). Then, rely on external tools to add on functionality that your IDE doesn't provide.

IDE Features

The most important thing you can do is to make sure that everyone's IDE is configured to use the same runtime environment, compiler options and coding style. In an Eclipse-based IDE, it's a good practice to configure these settings on a per-project basis. That way, anyone who checks out your project from revision control automatically gets the settings your project prefers rather than whatever settings are in their workspace.

I recommend that you configure project-specific settings (check **Enable project specific settings**) for at least the following project properties:

- **Properties > Java Compiler**
- **Properties > Java Compiler > Annotation Processing**
- **Properties > Java Compiler > Errors/Warnings**
- **Properties > Java Compiler > Javadoc**
- **Properties > Java Compiler > Task Tags**
- **Properties > Java Code Style**
- **Properties > Java Code Style > Formatter**

Some project teams choose to automatically re-format the code (using the IDE formatting rules) whenever a file is saved, or whenever a file is checked into revision control. I recommend against this. Consistent formatting is important, but legibility is more important. Sometimes, hand-formatting can result in code that is more legible than that produced by the automatic formatter, and I prefer to let developers make that judgment for themselves. You can always apply the automatic formatter to a piece of code by hand if you want to (**Source > Format** from within a source file).

It's important that everyone is using the same runtime environment. I've been on projects where one person was using JRE 1.5 and another was using JRE 1.6. This led to more than a little confusion. If at all possible, you also want to make sure that developers are using the same runtime as in your production environment. This will make it easier to reproduce problems that might be runtime-specific.

You configure the runtime environment via **Properties > Java Build Path > Libraries**. I recommend that you choose a standard environment (i.e. "Java2SE-1.6") rather than some

environment-specific runtime environment. Developers then can use workspace settings (**Window > Preferences > Java > Installed JREs > Execution Environments**) to point this standard environment at whatever specific version of the JRE is installed on their PC.

External Tools

I recommend two Eclipse plugins: Checkstyle and AnyEdit. See *Appendix A* for more information about where to find these plugins.

I use AnyEdit to convert tab characters to spaces and to trim trailing blanks. I have the plugin configured to do this conversion automatically whenever a file is saved. Although I generally prefer not to apply IDE formatting on save (see above), I think it's worth doing for this kind of formatting.

The Checkstyle plugin integrates the Checkstyle style-checker into Eclipse. Violations show up as warnings alongside compile errors and other problems. There are literally hundreds of available checks, some of which are more useful than others.

Over time, I have developed a set of the checks that I think are most important. In *Appendix A*, you can find the Checkstyle configuration that I use for the open-source Cedar Solutions Java projects. My Checkstyle rules are focused around naming conventions, indenting, maximum line length, brace location, etc. I skip the checks for things like cyclomatic complexity, nested depth of if statements, etc.

Note: If you decide to rely on plugins like this, you need to make sure that everyone on your team installs them. Otherwise, some people will see behavior that others won't. That causes confusion. Further, you need to manage the process of upgrading the plugins. Remember that each tool you adopt raises potential version upgrade problems.

About Warnings

The primary way that conventions are enforced is via code warnings. My recommendation is that you strive for zero code warnings when code is checked into revision control. My experience has been that this is the only way to keep the warnings under control. On projects where some warnings get ignored, things tend to snowball out of control, and pretty soon someone has to spend half a day fixing 400 of them all at once.

There are two things you can do to make warnings more manageable. First, take advantage of compiler and tool features to suppress warnings that you know are not applicable. Both the Eclipse compiler and tools like Checkstyle let you suppress warnings. When used appropriately, this can be a useful feature. The important thing is that you don't just willy-nilly ignore inconvenient warnings. Set an expectation up-front: it's OK to suppress a warning, but only if you have carefully evaluated the consequences first.

Second, leverage features of your IDE to trim down the warnings that you see at any given time. In Eclipse, what seems to work best is to organize projects by working set. Once you do this, you get the opportunity to filter errors and warnings in useful ways. First, create a working set (**New > Java Working Set**) and add your projects to it. Then, in the Package Explorer, click the down arrow  and select **Top Level Elements > Working Sets**.

Now, here's the big advantage. In the Problems view, click the down arrow and select **Show > Errors/Warnings on Selection**. Once this is configured, you'll only see errors and warnings for whatever element you have clicked on in Package Explorer. So, if you click on the working set,

you'll see errors and warnings for all code in the working set. Or, if you click on an individual package or file, you'll see errors and warnings down at that level. This can really be a time-saver.

Testing with Mocks

Interfaces

In Java, an interface boundary should usually be defined by an actual `interface`, not an abstract or concrete `class`. This allows for greater flexibility later: consider that not everyone will necessarily want to extend your abstract class just to so they can use your methods. The methods in a service (or DAO) interface are usually constructed in terms of domain objects or simple classes such as `String` or `Integer`.

I usually set up a package structure like this:

```
com.cedarsolutions.example.domain
com.cedarsolutions.example.service
com.cedarsolutions.example.service.impl
com.cedarsolutions.example.dao
com.cedarsolutions.example.dao.impl
```

The `domain` package contains domain objects, which are simple objects that contain very little or no business logic. Domain objects have absolutely no outside dependencies. If they do contain business logic in some methods, these methods should operate only on their instance variables and method arguments. Domain objects should be simple enough that they never need to be mocked, and should be very easy to unit test.

The `service` package contains the service interfaces. I prefer a naming convention that identifies interfaces, so you might see something like `ITemplateService`. The `service.impl` package contains the service implementations. Each service implements a service interface. Usually, the naming is consistent, so `TemplateService` implements `ITemplateService`.

Likewise, DAO interfaces go in `dao` with implementations in `dao.impl`. DAO interfaces like `IRegisteredUserDao` are usually implemented by a concrete DAO `RegisteredUserDao`.

The concrete implementation of any service or DAO will probably have some external dependencies. I usually treat both services and DAOs as Spring beans. For production code, dependencies are injected via Spring configuration. For unit tests, I typically inject mock dependencies by hand.

Mock Objects

Imagine that the concrete class `TemplateService` depends on a DAO of type `IRegisteredUserDao`. In order to test `TemplateService`, we need to give it a DAO to operate on. Since all that `TemplateService` cares about is the interface, it doesn't matter what type of DAO we provide – our tests will be equally valid regardless of whether we pass in a “real” DAO, or a “fake” DAO of some sort. A mock object is simply a “fake” object that satisfies an interface.

Historically, I have written mock objects by hand. Eclipse makes it easy to auto-generate a class that implements an interface. I would implement whatever methods I needed for my test, and throw a `NotImplementedException` from any other method. Typically, I would build in mechanisms to return sensible data and to confirm that methods were getting called correctly.

This does work, but it takes a lot of effort. On a large project, I could end up with dozens of hand-written mocks, if not more. Worse, the implementation of a mock can get kind of confusing, and you can run into issues when more than one test needs to rely on the same mock.

In early 2011, I finally found a mocking framework that I like a lot: Mockito. Mockito can mock any interface automatically, without you having to create any code by hand. It can even partially or completely mock concrete classes, allowing you override some or all of the class's functionality for your unit test. The only real limitation is that static and final methods cannot be mocked (but that's a JVM restriction).

The best part about Mockito is that every test case can now be independent of every other test case, and expectations can be set up right in the test case. It's very obvious which method calls are being stubbed, which in turn makes it easier to understand how dependency classes are used.

Code Example

Let's take a very simple example to demonstrate how Mockito works. Rather than following the service/DAO scenario above, I have taken a real example from the SantaExchange GWT web application:

Below, `testReplaceModuleBody()` is a JUnit4 unit test intended to validate the `ModulePagePresenter.replaceModuleBody()` method. The method we're trying to test is really simple:

```
protected void replaceModuleBody() {
    this.eventBus.replaceModuleBody(view.getViewWidget());
}
```

What want to do is confirm that the right `eventBus` method gets called with the right arguments. I think the result is pretty legible.

```
@Test public void testReplaceModuleBody() {
    Widget viewWidget = mock(Widget.class);
    ModulePagePresenter presenter = createPresenter();
    when(presenter.getView().getViewWidget()).thenReturn(viewWidget);
    presenter.replaceModuleBody();
    verify(presenter.getEventBus()).replaceModuleBody(viewWidget);
}

private static ModulePagePresenter createPresenter() {
    EventBus eventBus = mock(EventBus.class);
    IMyView view = mock(IMyView.class);

    ModulePagePresenter presenter = new ModulePagePresenter();
    presenter.setEventBus(eventBus);
    presenter.setView(view);

    assertSame(eventBus, presenter.getEventBus());
    assertSame(view, presenter.getView());

    return presenter;
}
```

First we use `createPresenter()` to create the presenter we'll test with. `ModulePagePresenter` has two dependencies, an event bus and a view. Both of these dependencies are mocked and then injected into the presenter. The returned presenter is fully operational, and it doesn't know that it's operating on mocks rather than real MVP4G objects.

Once the presenter is available, the test case sets up expectations via Mockito using the `when()` method. This line says, "hey, mock view: when someone asks for your view widget, return this particular instance." Once expectations are set, the test case invokes the presenter method. After the method returns, we check to make sure that the right event bus method was called with the right argument.

This is a very simple example. However, Mockito scales to much more complicated scenarios very easily. For instance, you can capture the arguments to a method call, and add other validations to make sure that the method works as expected. I have already forgotten how I lived without it.

Other Topics to Consider

When I discuss this document with colleagues that I trust, other topics are often raised. I don't have time right now to do any of these topics justice, but I do want to mention them briefly.

Other Tools

While my discussion above tends to focus on JUnit, this isn't the only unit testing framework out there (for instance, there's also TestNG). Do your research and choose the tool that is most appropriate to your needs.

Don't limit yourself to only unit-testing frameworks. In the last few years, a lot of acceptance test frameworks have started to hit the market. A good example is FitNesse. If you're writing a web application, you should be considering tools like Selenium, HtmlUnit (free) or LoadRunner (commercial).

Think broadly about how you are going to validate your software, and consider automating as much of this validation process as possible. Any tests that are automated can be run more frequently than tests which are run by hand. You might even consider writing a test suite that you can run against your production systems after a cutover. Think about how nice would it be to go to sleep at 3:00am with some confidence that things are actually working properly!

A word of caution: sometimes, automated tests built quickly or naively end up being brittle. These will not do you a lot of good. You are better off writing fewer solid, flexible test cases than more brittle test cases that have to be reworked almost constantly to stay relevant.

Continuous Integration

If you have a team of more than one person, you should investigate Continuous Integration. I have found that installing a continuous integration build server like Cruise Control or Hudson can help teams stay connected with each other. When you run a continuous integration server, the code that is checked into revision control is periodically re-built. The team is notified of build errors. If your tests are structured properly, the build environment can also run one or more test suites, and can notify the team of test failures as well. This way, problems are caught early.

Code Reviews

Taken seriously and structured properly, a good review process can help improve code confidence and quality. Not only do poor design decisions tend to pop out as a result of the review process, but your developers get the opportunity to learn from each other.

Serious developers know that they're either growing, or they're on the path to obsolescence. I aim to learn something new on every project I'm a part of. I often learn from looking at how someone else has solved a problem which I might have solved differently. This is why I think that code reviews can be an incredibly helpful learning experience both for the reviewee *and* the reviewer.

Patterns and Frameworks

I will admit that I don't use a lot of design patterns in the formal sense. However, what I do find useful is to look for patterns *in my own code*. Once those patterns pop out, I can create frameworks that simplify the use of those patterns. This seems like it should be obvious, but I'm always surprised when I see the same boilerplate code repeated over and over again in a given legacy application.

Take the time to look for patterns, and refactor common code out into frameworks. This gets you two advantages: first, the frameworks can be tested once, and bugs can be fixed in one place. Second, the code that uses a framework is naturally more consistent than cut-and-paste code, because every piece of code doing some task does it the same way. If your framework has been well-designed, the resulting code will not only be more consistent, but probably more intelligible too.

There is a bit of an art in designing a framework for this purpose. You have to choose the right level of abstraction, and you need to be careful in choosing your interface boundaries. Often, you not only have to implement framework code itself, but you need to factor out one or more new interfaces as well. Otherwise, the framework ends up tightly coupled to the original code, which doesn't do you a lot of good.

Appendix A: Eclipse Plugins

Checkstyle plugin

Integrates the Checkstyle style-checker into Eclipse.

Project homepage: <http://eclipse-cs.sourceforge.net/>

Automatic update URL: <http://eclipse-cs.sourceforge.net/update>

This plugin enforces a coding style. Violations show up as warnings alongside compile errors and other problems. If you keep on top of the warnings, this will help keep your code more consistent and easier to read.

Below are links to the Checkstyle rules used for Cedar Solutions Java projects, along with the related Eclipse formatting rules. There is also an example suppressions file. The suppressions file is used to suppress specific warnings in specific classes.

Rules: <http://cedar-solutions.com/docs/confidence/CheckstyleRules.xml>

Suppressions: <http://cedar-solutions.com/docs/confidence/CheckstyleSuppressions.xml>

Formatting: <http://cedar-solutions.com/docs/confidence/formatting.xml>

Emma plugin

Integrates the EMMA code coverage tool into Eclipse.

Project homepage: <http://www.eclEmma.org/>

Automatic update URL: <http://update.eclEmma.org/>

Right click on a project, package, or class and use **Coverage As > Junit Test** to run coverage for the entire project. Once you do a coverage run, you can open files from the coverage results. Covered lines will be highlighted in green; non-covered lines will be highlighted in red.

AnyEdit Plugin

Plugin that enhances most of Eclipse's editors:

Project homepage: <http://andrei.gmxhome.de/anyedit/>

Automatic update URL: <http://andrei.gmxhome.de/eclipse/>

When you use the auto-update URL, you'll have to dig around until you find AnyEdit. It's in there somewhere.

I use AnyEdit to convert tab characters to spaces and to trim trailing blanks. I have the plugin configured to do this conversion automatically whenever a file is saved. The plugin also supports some other useful behaviors. For instance, if you highlight a filename in a config file and choose **Open File Under Cursor**, the plugin will find the file for you and open it.

Appendix B: External References

Mockito

Mockito is a mocking framework for Java

Project homepage: <http://mockito.org/>

Mockito is the first mocking framework I have found that I actually like using. The Mockito folks claim that Mockito “lets you write beautiful tests,” and they’re right. All of the examples in the *Testing with Mocks* section use Mockito.

Spring

Spring is an application development framework.

Project homepage: <http://www.springsource.org/>

For my purposes, the most important feature Spring provides is called the IOC (inversion of control) container. By using IOC (also called dependency injection), you can *inject* dependencies into your code rather than writing code that is aware of where to find its dependencies.

The use of dependency injection greatly simplifies the task of organizing, configuring and testing your code. To extend the earlier example, you can use a completely different mechanism to get a data source for unit testing than for deploying your code to an application server. However, because that data source is injected, *you don’t have any additional code paths to test in your code*. You simply have to make sure that Spring is configured properly.