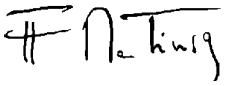
METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Spring 2012 (Volume 20 - number 1) www.methodsandtools.com

Who Are You Working For?

This can be considered as a simple question and you might start with some simple answers like "company X" if you are an employee or "myself" if you are freelance. If you work as a consultant or an outsourcer, you could also name the people that pay you or your company to develop their software. It is however less obvious that you will think of your "internal" customers and say that as a software developer you are working for the accounting department, unless your IT function is directly related to a functional department in your organization. Maybe you will also name somebody if you think that this person has a big influence on your work. Many people at Apple would think they were working for Steve Jobs. Even if you think that you are working for a single person, the answer to the initial question could be more complicated as you should also include your colleagues in it: project managers, business analysts, developers, testers or future maintainers. These people are also impacted by your work and attitude, even if they might not influence (appraisals set aside) directly your salary. The visions, time horizons and interests of these persons can be very different and sometimes conflicting. Usually, your customer wants a "cheap" solution quickly. Your project manager might value more you reliability. As long as you are sticking to your estimates, the internal quality of your code is not so important. Your colleagues are interested in the quality of your work, but they might also put a high value on your "social" skills, especially if you share some office space. Future maintainers of your code will value good design and readability. You have to deal yourself with your own conflicts of interest: deciding between a quick short term solution or a "cleaner" long term design that need more time to be realized. How much do you include in the current iteration, how much do you leave for future refactoring? A software developer works for many different people. You have to manage all these conflicts of interest and know that imperfect results are often part of our daily job. I personally have this type of feeling when I have to deal with code I wrote more than 6 months ago. The frontier between making people that you work for, including yourself, happy or unhappy is sometimes thin, but don't forget this rule that apply also to software development: "Don't do to others what you don't want others do to you". Know who you work for and try to be good with them.



Inside

The Most Popular Collaborative Models

Gojko Adzic, http://gojko.net/

This article is based on Specification by Example, published in June 2011. It is being reproduced here by permission from Manning Publications. Manning early access books and ebooks are sold exclusively through Manning. Visit the book's page for more information.

Although all the teams I interviewed collaborated on specifications, the ways they approached that collaboration varied greatly, from large all-hands workshops to smaller workshops, and even to informal conversations. Here are some of the most common models for collaboration along with the benefits the teams obtained.

Try big, all-team workshops

When: Starting out with Specification by Example

Specification workshops are intensive, hands-on domain and scope exploration exercises that ensure that the implementation team, business stakeholders, and domain experts build a consistent, shared understanding of what the system should do. The workshops ensure that developers and testers have enough information to complete their work for the current iteration.

Big specification workshops that involve the entire team are one of the most effective ways to build a shared understanding and produce a set of examples that illustrate a feature.

During these workshops, programmers and testers can learn about the business domain. Business users will start understanding the technical constraints of the system. Because the entire team is involved, the workshops efficiently use business stakeholders' time and remove the need for knowledge transfer later on.

Initially, the team at uSwitch used specification workshops to facilitate the adoption of Specification by Example. Jon Neale describes the effects:

> It particularly helped the business guys think about some of the more obscure routes that people would take. For example, if someone tried to apply for a loan below a certain amount, that's a whole other scenario [than applying for a loan in general]. There's a whole other raft of business rules that they wouldn't have mentioned until the last minute.
>
> Specification workshops helped them think about those scenarios up front and helped us go faster. It also helped the development team to interact with the other guys. Having that upfront discussion helped drive the whole process - there was a lot more communication straight away.

Implementing Specification workshops into PBR workshops

Product Backlog Refinement (PBR) workshops are one of the key elements of well-implemented Scrum processes. At the same time, I've found that most teams that claim to run Scrum actually don't have PBR workshops.

PBR workshops normally involve the entire team and consist of splitting large items on the top of the backlog, detailed analysis of backlog items, and re-estimation. In *Practices for Scaling Lean and Agile*, [1] Bas Vodde and Craig Larman suggest that PBR workshops should take between5and 10 percent of each iteration.

Illustrating requirements using examples during a Product Backlog Refinement workshop is an easy way to start implementing Specification by Example in a mature Scrum team. This requires no additional meetings and no special scheduling. It's a matter of approaching the middle portion of the PBR workshop differently.

The Talia team at Pyxis Technologies runs their workshops like this. André Brissette explains this process:

"This usually happens when the product owner and the Scrum master see that the top story on the backlog is not detailed enough. For example, if the story is estimated at 20 story points, they schedule a maintenance workshop during the sprint. We think that it's a good habit to have this kind of a session every week or every two weeks in order to be certain that the top of the backlog is easy to work with. We look at the story; there is an exchange between the product owner and the developers on the feasibility of it. We draw some examples on the whiteboard, identify technical risk and usability risks, and developers will have to make an evaluation or appraisal of the scope. At this time we do planning poker. If everyone agrees on the scope of the feature and the effort that it will take, then that's it. If we see that it is a challenge to have a common agreement, then we try to split the story until we have items that are pretty clear and the effort is evaluated and agreed to."

Large workshops can be a logistical nightmare. If you fail to set dates on a calendar up front, people might plan other meetings or not be readily available for discussions. Regularly scheduled meetings solve this issue. This practice is especially helpful with senior stakeholders who want to contribute but are often too busy. (Hint: call their secretary to schedule the workshops.)

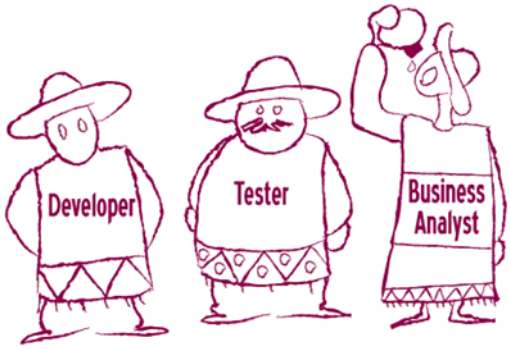
If you have a problem getting enough time from business users or stakeholders, try to fit into their schedule or work on specifications during product demos when they're in the room. This is also effective if the business users and delivery team don't work from the same location.

Large workshops are an effective way to transfer knowledge and build a shared understanding of the requirements by the entire team, so I highly recommend them for teams that are starting out with Specification by Example. On the other hand, they cost a lot in terms of people's time. Once the process matures and the team builds up domain knowledge, you can move on to one of the easier alternatives.

Try smaller workshops ("Three Amigos")

When: Domain requires frequent clarification

Having a single person responsible for writing tests, even with reviews, isn't a good approach if the domain is complex and testers and programmers frequently need clarification.



Run smaller workshops that involve one developer, one tester, and one business analyst.

A popular name for such meetings is Three Amigos. Janet Gregory and Lisa Crispin suggest a similar model for collaboration in *Agile Testing*, [2] under the name The Power of Three. (I used to call such workshops Acceptance Testing Threesomes until people started complaining about the innuendo.)

A Three Amigos meeting is often sufficient to get good feedback from different perspectives. Compared to larger specification workshops, it doesn't ensure a shared understanding across the entire team, but it's easier to organize than larger meetings and doesn't need to be scheduled up front. Smaller meetings also give the participants more flexibility in the way they work. Organizing a big workshop around a single small monitor is pointless, but three people can sit comfortably and easily view a large screen.

To run a Three Amigos meeting efficiently, all three participants have to share a similar understanding of the domain. If they don't, consider allowing people to prepare for the meeting instead of running it on demand. Ian Cooper explains this:

> The problem with organizing just a three-way is that if you have an imbalance of domain knowledge in the team, the conversation will be led by the people with more domain expertise. This is similar to the issues you get with pairing [pair programming]. The people knowledgeable about the domain tend to dominate the conversation. The people with less domain expertise will sometimes ask questions that could have quite a lot of interesting insight. Giving them an option to prepare beforehand allows them to do that.

A common trick to avoid losing the information from a workshop is to produce something that closely resembles the format of the final specification. With smaller groups, such as the Three Amigos, you can work with a monitor and a keyboard and produce a file. Rob Park worked on a team at a large U.S. insurance provider that collaborated using Three Amigos. Park says:

> The output of the Three Amigos meeting is the actual feature file -Given-When-Then. We don't worry about the fixtures or any other layer beneath it, but the acceptance criteria is the output. Sometimes it is not precise - for example, we know we'd like to have a realistic policy number so we would put in a note or a placeholder so we know we're going to have a little bit of cleanup after the fact. But the main requirement is that we're going to have all these tests in what we all agree is complete, at least in terms of content, before we start to code the feature.

Stuart Taylor's team at TraderMedia has informal conversations for each story and produces tests from that. A developer and a tester work on this together. Taylor explains the process:

> When a story was about to be played, a developer would call a QA and say, "I'm about to start on this story," and then they would have a conversation on how to test it. The developer would talk about how he is going to develop it using TDD. For example, "For the telephone field, I'll use an integer." Straightaway the QA would say, "Well, what if I put ++, or brackets, or leading zeros, etc."

The QA would start writing [acceptance] tests based on the business acceptance criteria and using the testing mindset, thinking about the edge cases. These tests would be seen by the BA and the developer. During showcasing we'd see them execute.

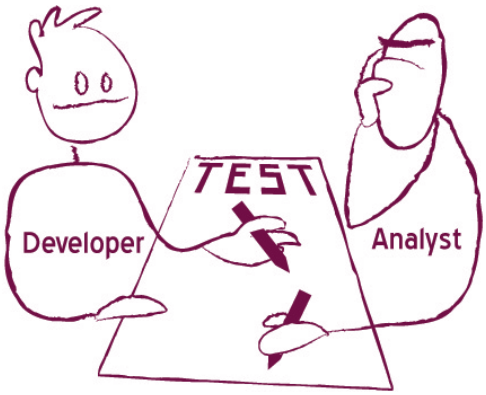
Producing a semiformal test collaboratively ensures that the information won't get distorted during automation later on. It also helps to share knowledge about how to write good specifications with examples; this is only feasible if the entire group can sit around a single monitor and a keyboard. Don't try to draft semiformal documents in an all-hands workshop, because it won't encourage everyone to get involved.

Teams that work on mature products and already have a good knowledge of the target domain don't necessarily have to run meetings or have separate conversations to discuss the acceptance criteria for a story. Developers and testers might not necessarily need to provide as much input up front into the specifications, and they can resolve small functional gaps and during implementation. Such teams can collaborate with informal conversations or reviews.

Pair-writing

When: Mature products

Even in cases where the developers knew enough to work without big workshops, teams found it useful to collaborate on writing specifications with examples.



Analysts can provide the correct behavior but developers know the best way to write a test so that it's easy to automate later and fits into the rest of the living documentation system.

Andrew Jackman's team at BNP Paribas works on a relatively mature product. They have experimented with different models of writing tests and concluded that they need to get both business analysts and developers involved in writing the tests. He says:

> When developers were writing the tests, it was easy to misunderstand what the story is about. If you don't have the interaction with the business analysts, it's only the developers' view of a thing. We moved to BAs writing the tests and that made a big

> difference. The challenge is when they write a story, that story might influence a number of existing tests, but they can't foresee that. The BAs like to write a test that shows a workflow for a single story. Generally that leads to a lot of duplication because a lot of the workflows are the same. So we move bits of the workflow into their own test.

Some teams - particularly those in which the business analysts cause a bottleneck or don't exist at all - get testers to pair with programmers on writing tests. This gives the testers a good overview of what will be covered by executable specifications and helps them understand what they need to check separately. The team at Songkick is a good example. Phil Cowans explains their process:

> QA doesn't write [acceptance] tests for developers; they work together. The QA person owns the specification, which is expressed through the test plan, and continues to own that until we ship the feature. Developers write the feature files [specifications] with the QA involved to advise what should be covered. QA finds the holes in the feature files, points out things that are not covered, and also produces test scripts for manual testing.

Pairing to write specifications is a cheap and efficient way to get several different perspectives on a test and avoid tunnel vision. It also enables testers to learn about the best ways to write specifications so that they're easy to automate, and it allows developers to learn about risky functional areas that need special attention.

Have developers frequently review tests before an iteration

When: Analysts writing tests

> Get a senior developer to review the specifications.

The business users that work with Bekk Consulting on the Norwegian Dairy Herd Recording System don't work with developers when writing acceptance tests, but they frequently involve developers in reviewing the tests. According to Mikael Vik, a senior developer at Bekk Consulting, this approach gives them similar results:

> We're always working closely with them [business users] on defining Cucumber tests. When they take their user stories and start writing Cucumber tests, they always come and ask us if it looks OK. We give them hints on how to write the steps and also come up with suggestions on how our Cucumber domain language can be expanded to effectively express the intention of the tests.

If developers aren't involved in writing the specifications, they can spend more time implementing features. Note that this increases the risk that specifications won't contain all the information required for implementation or that they may be more difficult to automate.

Try informal conversations

When: Business stakeholders are readily available

Teams that had the luxury of business users and stakeholders sitting close by (and readily available to answer questions) had great results with informal ad hoc conversations. Instead of having big scheduled workshops, anyone who had a stake in a story would briefly meet before starting to implement it.

SpiraTeam the complete Agile ALM suite - Click on ad to reach advertiser web site

Informal conversations involving only the people who will work on a task are enough to establish a clear definition of what needs to be done.

"Anyone who has a stake" includes the following:

- The analysts who investigate a story
- The programmers who will work on implementing it
- The testers who will run manual exploratory tests on it
- The business stakeholders and users who will ultimately benefit from the result and use the software

The goal of such informal conversations is to ensure that everyone involved has the same understanding of what a story is about. At LMAX, such conversations happened in the first few days of a sprint. Jodie Parker explains:

> Conversations would be done on demand. You've got the idea and your drawings, and you really understand how it is going to be implemented. If you've not already written down the acceptance tests, a developer and a tester can pair on this. If the conversations didn't happen, things would end up being built but not being built right.

Some teams, such as the one at uSwitch.com, don't try to flush out all the acceptance criteria at this point. They establish a common baseline and give testers and developers enough information to start working. Because they sit close to the business users, they can have short conversations as needed.

Some teams decide whether to have an informal discussion or a larger specification workshop based on the type of the change introduced by a story. Ismo Aro at Nokia Siemens Networks used this approach:

We have an agreement to have ATDD test cases [specifications], not necessarily a meeting. If the team feels it's coming naturally, then it's OK not to do the meeting. If it seems harder and they need input from another stakeholder, then they organize an ATDD meeting [Specification workshop]. This might be due to the team knowing a lot about the domain. When you are adding a small increment to the old functionality, it's easier to figure out the test cases.

Remember

Specification by Example relies heavily on collaboration between business users and delivery team members.

Everyone on the delivery team shares the responsibility for the right specifications. Programmers and testers have to offer input about the technical implementation and the validation aspects.

Most teams collaborate on specifications in two phases: Someone works up front to prepare initial examples for a feature, and then those who have a stake in the feature discuss it, adding examples to clarify or complete the specification.

The balance between the work done in preparation and the work done during collaboration depends on several factors: the maturity of the product, the level of domain knowledge in the delivery team, typical change request complexity, process bottlenecks, and availability of business users.

References

[1] Craig Larman and Bas Vodde, *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum* (Pearson Education, 2010).

[2] Lisa Crispin and Janet Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams (Addison-Wesley Professional, 2009).

Automated Testing of ASP.NET MVC Applications

Artëm Smirnov, Developer of Ivonna for Typemock Isolator, www.typemock.com

Introduction

For many years the developers who practiced unit testing were frustrated about numerous problems they had when trying to apply automated testing to ASP.NET sites, in particular, those that were built using the WebForms framework (which was, for many, a synonym of ASP.NET). Not so long ago, Microsoft developed a new ASP.NET framework, called ASP.NET MVC. One of the selling points of this framework was its testability. In this article, we're discussing what the problems are with testing ASP.NET sites in general, and how the ASP.NET MVC framework tries to solve them.

What's different about testing ASP.NET applications?

When you are developing a library, or an application, you "own" the code. It means that, provided you follow certain guidelines, you can automate testing all, or at least all worth testing, features of your product. While you usually use third party libraries, or system resources like database or Internet connection, they can be considered "enhancements" sitting on top of your code.

With ASP.NET, the situation is reversed. What you develop are, essentially, enhancements to the IIS (or another hosting process). This is not unique, of course, to ASP.NET – the same holds true for any plugin development. It is like your code is a "guest" in this system, and should behave accordingly. You can add as many classes as you wish, and make them testable, but there is always a boundary layer that communicates with the framework. The host system creates these boundary classes, and calls certain methods that you provide, so that you have a brief moment of control. In WebForms, this layer is made of ASPX pages and code-behind files. In ASP.NET MVC, it is Controllers and Views.

For testing, a developer can take two very distinct paths. A developer can either test the main system together with your custom code (integration testing), or test your code in isolation, somehow dealing with the absence of the main system (unit testing).

Integration Testing

End-to-end, or integration, tests verify the behavior of the system as a whole. One can start, for example, with a certain URL, and inspect the HTML output, plus the side effects (a DB record inserted, an email sent, some money written off of our account, etc). Based on this test, we can assume that everything works as expected, or something is wrong. Given the HTML response, for example, all we can is search for a certain string and hope that if we find it, then everything else is as expected as well. If not, we have an idea that something went wrong, but we don't know what. This is known as a "black box" testing – we provide an input and inspect the output, but we don't know what's inside.

We can push this even further, and test our client code together with the server code. While the former tests mimic a browser (set up an HTTP request, inspect the response), the latter ones mimic a user ("click" a button, inspect the content of a certain HTML element). Typically, we write code that "drives" a browser, automating all the clicking for us.

There are some important qualities that characterize such tests:

- We are testing the system behavior as it is presented to the end user, so if the tests pass, the system is expected to work correctly.
- If a test is broken, we don't know whether it is a problem with the client or the server code, or the incompatibility between these two.
- Such tests are really slow. The clicking happens faster than a human would do it, but still much slower than the code processing. A complex system can take several hours to test.
- Complicated setup and cleanup: Often we have to perform several steps before we even get to the page we want to test. For example, testing a password protected page would require visiting the login page first.
- Tests are hard to maintain, as there is a lot of test code involved that is not directly related to the system.

Whether we use client code testing or not, there are some general problems with such tests, and these problems do not depend on a particular framework or programming language:

- If something's wrong, we are not sure about what needs to be fixed.
- More often than not, some tests start to fail because of a non-functional change (such as changing an ID of an HTML tag).
- It is hard to control the external dependencies. For example, if we test an e-commerce application, we don't want to bill somebody's account each time we run a test, yet we need to get a positive response from the billing system, so that we can feed it into our system.
- Integration tests alone do not drive your system to a better design. After all, they don't care about the implementation.

So, while integration tests are very important, and should be a part of any test suite in order to ensure that your system works properly as a whole, it is essential to use unit testing in your development process.

Unit testing

As we saw, when working with a framework that hosts your code, you have to create a number of "boundary" classes that interface with that framework. It is these classes that are sometimes hard to test, and have to be designed carefully in order to keep both your hosting framework and your test runner happy. One of the biggest challenges is to test your boundary class without the "real" hosting system up and running. For example, the WebForms framework is responsible for parsing the ASPX pages, assembling the controls into a web page, and firing certain events now and then. In order to write a sensible test for a WebForm page, you need at least the page itself built up properly, meaning that you should do a lot of work that normally is done for you by the framework. Another problem is that often the tested code depends on certain objects that only the framework can provide. For example, the infamous HttpContext object, that cannot be created manually, and its "children" – HttpRequest, HttpResponse, etc.

Typically, the only way out of this situation is make the boundary classes to serve as mere adaptors, and put the real "meat" in other classes, those that do not interact directly with the framework, or at least are easy to setup manually and tested in isolation. This supposedly leads to a better code as well, since, for example, moving the business and DAL logic away from the code-behind leads to more maintainable applications. The boundary classes themselves are left untested.

One example is when, rather than put your data access in the code behind, you move it to DataObject classes that are used by ObjectDataSource controls. These classes, although being created and used by the framework, are relatively easy created and tested in isolation.

Here, the point whether the framework design cares for testability becomes important. A lot of people (me included) tried to follow this path for WebForms applications, trying to apply the MVP pattern. What happened is a lot of plumbing code that required an amount of effort far exceeding the testability benefits, and making the code harder, to maintain.

ASP.NET MVC: Testability Paradise

After listening to developers criticize the WebForms design and testability issues for several years, Microsoft gave us the ASP.NET MVC framework. It has been designed to solve many shortcomings of WebForms, including the testability problems. The main "boundary" classes are now Controllers. Although they are created by the framework, and have to conform to some rules, they are quite easy to setup in tests, and they can happily exist without the ASP.NET request pipeline. What's important, we are not limited to default constructors: we can inject dependencies into our controllers, making it possible to mock our services and test controllers in isolation. Another major benefit is that controllers are totally independent of the UI. It is widely agreed that the UI is hard to test, and Codebehind classes were suffering from this "burden". Now we've got Views, which are still hard to test (although some effort is made into that direction, one blog post from an architect on the ASP.NET team), but this does not affect testing our Controllers.

Another step toward testability is addressing the HttpContext nightmare. Microsoft took the "thin wrapper" approach, which, admittedly, is the only way when you deal with the vast amount of existing code which should be easily upgradeable to the new ASP.NET version. In terms of testability, I'd say it's moving from impossible to very hard. And if we recall the popular belief that testability equals good design, this case is a great example of the opposite. So, what we gained here is a possibility to write unreadable tests with chained partial mocks, plus having to use Http** wrappers in your production code without any real benefit to the application design.

Most developers tend towards one of two extremities. One, the "Big Ball of Mud" approach (that MVC does *not* save us from), is to put all the request handling code in the Action method of the controller. Writing tests for such methods is essential, since the code can become very complicated. That would be, of course, integration tests: each would test everything that happens during the request. However, the size of the "black box" is now much smaller, and we can even peek at what's inside it, since our test is now in the same process as the code being tested.

The other tendency, dubbed "Slim Controller", follows the idea that the sole responsibility of a controller is to wire up various services and model classes. Each Action method looks like a piece of configuration in an executable form. Typically, a test for such a method involves lots of mocking, and essentially duplicates the method itself (although in an unreadable form). Such tests should be generally avoided, since they don't add any value to the system, being only a maintenance problem. The rest of the classes involved in the Action method are either business layer services (which are testable or not regardless of the framework), or infrastructure classes like custom Action Filters, Model Binders etc., which are mostly testable, but with some considerable effort (see, for example, Scott Hanselman's post about unit testing ASP.NET MVC Custom Model Binders).

So, there are two extremities: bad design with integration tests, or good design with brittle and useless unit tests. This is why many developers that tend towards good design tend to not write tests for controllers at all, instead concentrating their efforts on testing the underlying classes.

How about test-driving our design?

Controller testability can be a good encouragement for a better Action method design. For example, it's a bad idea to use HttpContext.Current in your code. And while you can use HttpContextBase as a mockable dependency (see our discussion of the Http* wrappers above), the mere fact that your tests are hard to write and maintain should also give you an idea that you shouldn't do it either. Fortunately, you can use Model Binders, Action Method Filters, and other infrastructure classes to shield your code from the framework. That still leaves you with a piece not covered by tests, but at least this piece is part of the infrastructure now, and is not supposed to be changed often. Such classes are very simple, since their responsibility is just to extract the information from the framework specific classes and present it in a framework agnostic form.

Is it perfect?

As with most applications, unit testing is not enough. This is especially true when your code interface with a framework that does not participate in your unit tests. Since the boundary part is not tested for the most part, and can become rather wide (especially if you start using the infrastructure classes, like custom Route Providers and Model Binders), it is essential that you write a number of tests that make sure all your parts work together as expected.

Another potential problem is testing the Views. While this problem should discourage you from using considerable amounts of code in your Views, sometimes in order to make your Views maintainable you have to use Helpers, Partials, and even render Child Actions, and the result can become much more complicated than a simple piece of data bound HTML. Add to the mix a lot of "magic" that the framework conveniently does for you, and you'll find yourself in a situation when making a simple change suddenly breaks things in an unpredictable way. Having a safety net of at least simple smoke tests can save you from numerous regression bugs.

I guess the correct question would be not "is it perfect," but rather "could they make it better." My answer is, given the circumstances, not much. On one hand, looking at the FubuMVC framework which I started to use recently, I see that it *can* be better. On the other hand, the general idea of making a Ruby on Rails clone for ASP.NET must have forced a number of API decisions (e.g. returning an ActionResult from the Action method). If we go deeper and try to test some advanced infrastructure code (e.g. a custom convention for View location), we discover that the Microsoft's infamous preference of inheritance over composition does a poor job for writing such tests.

However, I should admit that for 99% of development efforts these issues are not a big problem. It is definitely good that Microsoft realized the importance of automated testing, and made testability one of the cornerstone principles of the ASP.NET MVC framework. Thanks to that decision, and its proper implementation, developers can make Web sites that are fully tested and have much less bugs. Because of that, the Web has become a better place.

The Risk-Driven Model of Software Architecture

George Fairbanks george.fairbanks [at] rhinoresearch.com
Rhino Research, http://rhinoresearch.com/

Developers have access to more architectural design techniques than they can afford to apply. The Risk-Driven Model guides developers to do just enough architecture by identifying their project's most pressing risks and applying only architecture and design techniques that mitigate them. The key element of the Risk-Driven Model is the promotion of risk to prominence. It is possible to apply the Risk-Driven Model to essentially any software development process, such as waterfall or agile, while still keeping within its spirit.

As they build successful software, software developers are choosing from alternate designs, discarding those that are doomed to fail, and preferring options with low risk of failure. When risks are low, it is easy to plow ahead without much thought, but, invariably, challenging design problems emerge and developers must grapple with high-risk designs, ones they are not sure will work. Henry Petroski, a leading historian of engineering, says this about engineering as a whole:

The concept of failure is central to the design process, and it is by thinking in terms of obviating failure that successful designs are achieved. Although often an implicit and tacit part of the methodology of design, failure considerations and proactive failure analysis are essential for achieving success. And it is precisely when such considerations and analyses are incorrect or incomplete that design errors are introduced and actual failures occur. [10]

To address failure risks, the earliest software developers invented design techniques, such as domain modeling, security analyses, and encapsulation, that helped them build successful software. Today, developers can choose from a huge number of design techniques. From this abundance, a hard question arises: Which design and architecture techniques should developers use?

If there were no deadlines then the answer would be easy: use all the techniques. But that is impractical because a hallmark of engineering is the efficient use of resources, including time. One of the risks developers face is that they waste too much time designing. So a related question arises: How much design and architecture should developers do?

There is much active debate about this question and several kinds of answers have been suggested:

- No up-front design. Developers should just write code. Design happens, but is coincident with coding, and happens at the keyboard rather than in advance.
- Use a yardstick. For example, developers should spend 10% of their time on architecture and design, 40% on coding, 20% on integrating, and 30% on testing.
- Build a documentation package. Developers should employ a comprehensive set of design and documentation techniques sufficient to produce a complete written design document.
- Ad hoc. Developers should react to the project needs and decide on the spot how much design to do.

The ad hoc approach is perhaps the most common, but it is also subjective and provides no enduring lessons. Avoiding design altogether is impractical when failure risks are high, but so is building a complete documentation package when risks are low. Using a yardstick can help you plan how much effort designing the architecture will take, but it does not help you choose techniques.

This article introduces the risk-driven model of architectural design. Its essential idea is that the effort you spend on designing your software architecture should be commensurate with the risks faced by your project. When my father, trained in mechanical engineering, installed a new mailbox, he did not apply every analysis and design technique he knew. Instead, he dug a hole, put in a post, and filled the hole with concrete. The risk-driven model can help you decide when to apply architecture techniques and when you can skip them.

Where a software development process orchestrates every activity from requirements to deployment, the risk-driven model guides only architectural design, and can therefore be used inside any software development process.

The risk-driven model is a reaction to a world where developers are under pressure to build high quality software quickly and at reasonable cost, yet those developers have more architecture techniques than they can afford to apply. The risk-driven model helps them answer the two questions above: how much software architecture work should they do, and which techniques should they use? It is an approach that helps developers follow a middle path, one that avoids wasting time on techniques that help their projects only a little but ensures that project-threatening risks are addressed by appropriate techniques.

In this article, we will examine how risk reduction is central to all engineering disciplines, learn how to choose techniques to reduce risks, understand how engineering risks interact with management risks, and learn how we can balance planned design with evolutionary design. This article walks through the ideas that underpin the risk-driven model.

1 What is the risk-driven model?

The risk-driven model guides developers to apply a minimal set of architecture techniques to reduce their most pressing risks. It suggests a relentless questioning process: “What are my risks? What are the best techniques to reduce them? Is the risk mitigated and can I start (or resume) coding?” The risk-driven model can be summarized in three steps:

1. Identify and prioritize risks

2. Select and apply a set of techniques

3. Evaluate risk reduction

You do not want to waste time on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build successful systems by taking a path that spends your time most effectively. That means addressing risks by applying architecture and design techniques but only when they are motivated by risks.

1.1 Risk or feature focus

The key element of the risk-driven model is the promotion of risk to prominence. What you choose to promote has an impact. Most developers already think about risks, but they think about lots of other things too, and consequently risks can be overlooked. A recent paper described how a team that had previously done up-front architecture work switched to a purely feature-driven process. The team was so focused on delivering features that they deferred quality attribute concerns until after active development ceased and the system was in maintenance [1]. The conclusion to draw is that teams that focus on features will pay less attention to other areas, including risks. Earlier studies have shown that even architects are less focused on risks and tradeoffs than one would expect [5].

1.2 Logical rationale

But what if your perception of risks differs from others' perceptions? Risk identification, risk prioritization, choice of techniques, and evaluation of risk mitigation will all vary depending on who does them. Is the risk-driven model merely improvisation?

No. Though different developers will perceive risks differently and consequently choose different techniques, the risk-driven model has the useful property that it yields arguments that can be evaluated. An example argument would take this form:

> We identified A, B, and C as risks, with B being primary. We spent time applying techniques X and Y because we believed they would help us reduce the risk of B. We evaluated the resulting design and decided that we had sufficiently mitigated the risk of B, so we proceeded on to coding.

This allows you to answer the broad question, "How much software architecture should you do?" by providing a plan (i.e., the techniques to apply) based on the relevant context (i.e., the perceived risks).

Other developers might disagree with your assessment, so they could provide a differing argument with the same form, perhaps suggesting that risk D be included. A productive, engineering-based discussion of the risks and techniques can ensue because the rationale behind your opinion has been articulated and can be evaluated.

2 Are you risk-driven now?

Many developers believe that they already follow a risk-driven model, or something close to it. Yet there are telltale signs that many do not. One is an inability to list the risks they confront and the corresponding techniques they are applying. Any developer can answer the question, "Which features are you working on?" but many have trouble with the question, "What are your primary failure risks and corresponding engineering techniques?" If risks were indeed primary then they would find it an easy question to answer.

2.1 Technique choices should vary

Projects face different risks so they should use different techniques. Some projects will have tricky quality attribute requirements (e.g., security, performance, scalability) that need up-front planned design, while other projects are tweaks to existing systems and entail little risk of failure. Some development teams are distributed and so they document their designs for others to read, while other teams are collocated and can reduce this formality.

When developers fail to align their architecture activities with their risks, they will over-use or under-use architectural techniques, or both. Examining the overall context of software development suggests why this can occur. Most organizations guide developers to follow a process that includes some kind of documentation template or a list of design activities. These can be beneficial and effective, but they can also inadvertently steer developers astray.

Here are some examples of well-intentioned rules that guide developers to activities that may be mismatched with their project's risks.

- The team must always (or never) build full documentation for each system.
- The team must always (or never) build a class diagram, a layer diagram, etc.
- The team must spend 10% (or 0%) of the project time on architecture.

Such guidelines can be better than no guidance, but each project will face a different set of risks. It would be a great coincidence if the same set of diagrams or techniques were always the best way to mitigate a changing set of risks.

2.2 Example mismatch

Imagine a company that builds a 3-tier system. The first tier has the user interface and is exposed to the internet. Its biggest risks might be usability and security. The second and third tiers implement business rules and persistence; they are behind a firewall. The biggest risks might be throughput and scalability.

If this company followed the risk-driven model, the front-end and back-end developers would apply different architecture and design techniques to address their different risks. Instead, what often happens is that both teams follow the same company-standard process or template and produce, say, a module dependency diagram. The problem is that there is no connection between the techniques they use and the risks they face.

Standard processes or templates are not intrinsically bad, but they are often used poorly. Over time, you may be able to generalize the risks on the projects at your company and devise a list of appropriate techniques. The important part is that the techniques match the risks.

The three steps to risk-driven software architecture are deceptively simple but the devil is in the details. What exactly are risks and techniques? How do you choose an appropriate set of techniques? And when do you stop architecting and start/resume building? The following sections dig into these questions in more detail.

3 Risks

In the context of engineering, risk is commonly defined as the chance of failure times the impact of that failure. Both the probability of failure and the impact are uncertain because they are difficult to measure precisely. You can sidestep the distinction between perceived risks and actual risks by bundling the concept of uncertainty into the definition of risk. The definition of risk then becomes:

$$\text{risk} = \text{perceived probability of failure} \times \text{perceived impact}$$

A result of this definition is that a risk can exist (i.e., you can perceive it) even if it does not exist. Imagine a hypothetical program that has no bugs. If you have never run the program or tested it, should you worry about it failing? That is, should you perceive a failure risk? Of

course you should, but after you analyze and test the program, you gain confidence in it, and your perception of risk goes down. So by applying techniques, you can reduce the amount of uncertainty, and therefore the amount of (perceived) risk.

3.1 Describing risks

You can state a risk categorically, often as the lack of a needed quality attribute like modifiability or reliability. But often this is too vague to be actionable: if you do something, are you sure that it actually reduces the categorical risk?

It is better to describe risks such that you can later test to see if they have been mitigated. Instead of just listing a quality attribute like reliability, describe each risk of failure as a testable failure scenario, such as "During peak loads, customers experience user interface latencies greater than five seconds."

3.2 Engineering and project management risks

Projects face many different kinds of risks, so people working on a project tend to pay attention to the risks related to their specialty.

Project management risks	Software engineering risks
"Lead developer hit by bus"	"The server may not scale to 1000 users"
"Customer needs not understood"	"Parsing of the response messages may be buggy"
"Senior VP hates our manager"	"The system is working now but if we touch anything it may fall apart"

Figure 1: Examples of project management and engineering risks. You should distinguish them because engineering techniques rarely solve management risks, and vice versa.

For example, the sales team worries about a good sales strategy and software developers worry about a system's scalability. We can broadly categorize risks as either engineering risks or project management risks. Engineering risks are those risks related to the analysis, design, and implementation of the product. These engineering risks are in the domain of the engineering of the system. Project management risks relate to schedules, sequencing of work, delivery, team size, geography, etc. Figure 1 shows examples of both.

If you are a software developer, you are asked to mitigate engineering risks and you will be applying engineering techniques. The technique type must match the risk type, so only engineering techniques will mitigate engineering risks. For example, you cannot use a PERT chart (a project management technique) to reduce the chance of buffer overruns (an engineering risk), and using Java will not resolve stakeholder disagreements.

3.3 Identifying risks

Experienced developers have an easy time identifying risks, but what can be done if the developer is less experienced or working in an unfamiliar domain? The easiest place to start is with the requirements, in whatever form they take, and looking for things that seem difficult to achieve. Misunderstood or incomplete quality attribute requirements are a common risk. You can use Quality Attribute Workshops [2], a Taxonomy-Based Questionnaire [4], or something similar, to elicit risks and produce a prioritized list of failure scenarios.

Even with diligence, you will not be able to identify every risk. When I was a child, my parents taught me to look both ways before crossing the street because they identified cars as a risk. It would have been equally bad if I had been hit by a car or by a falling meteor, but they put their attention on the foreseen and high priority risk. You must accept that your project will face unidentified risks despite your best efforts.

3.4 Prototypical risks

After you have worked in a domain for a while, you will notice prototypical risks that are common to most projects in that domain. For example, Systems projects usually worry more about performance than IT projects do, and Web projects almost always worry about security.

Project domain	Prototypical risks
Information Technology (IT)	Complex, poorly understood problem Unsure we're solving the real problem May choose wrong COTS software Integration with existing, poorly understood software Domain knowledge scattered across people Modifiability
Systems	Performance, reliability, size, security Concurrency Composition
Web	Security Application scalability Developer productivity / expressability

Figure 2: While each project can have a unique set of risks, it is possible to generalize by domain. Prototypical risks are ones that are common in a domain and are a reason that software development practices vary by domain. For example, developers on Systems projects tend to use the highest performance languages.

Prototypical risks may have been encoded as checklists describing historical problem areas, perhaps generated from architecture reviews. These checklists are valuable knowledge for less experienced developers and a helpful reminder for experienced ones.

Knowing the prototypical risks in your domain is a big advantage, but even more important is realizing when your project differs from the norm so that you avoid blind spots. For example, software that runs a hospital might most closely resemble an IT project, with its integration concerns and complex domain types. However, a system that takes 10 minutes to reboot after a power failure is usually a minor risk for an IT project, but a major risk at a hospital.

3.5 Prioritizing risks

Not all risks are equally large, so they can be prioritized. Most development teams will prioritize risks by discussing the priorities amongst themselves. This can be adequate, but the team's perception of risks may not be the same as the stakeholders' perception. If your team is spending enough time on software architecture for it to be noticeable in your budget, it is best to validate that time and money are being spent in accordance with stakeholder priorities.

Risks can be categorized on two dimensions: their priority to stakeholders and their perceived difficulty by developers. Be aware that some technical risks, such as platform choices, cannot be easily assessed by stakeholders. This is the same categorization technique used in ATAM to prioritize architecture drivers and quality attribute scenarios [3].

Formal procedures exist for cataloging and prioritizing risks using risk matrices, including a US military standard MILSTD882D. Formal prioritization of risks is appropriate if your system, for example, handles radioactive material, but most computer systems can be less formal.

Software engineering	Other engineering
Applying design or architecture pattern	Stress calculations
Domain modeling	Breaking point test
Throughput modeling	Thermal analysis
Security analysis	Reliability testing
Prototyping	Prototyping

Figure 3: A few examples of engineering risk reduction techniques in software engineering and other fields. Modeling is commonplace in all engineering fields.

4 Techniques

Once you know what risks you are facing, you can apply techniques that you expect to reduce the risk. The term technique is quite broad, so we will focus specifically on software engineering risk reduction techniques, but for convenience continue to use the simple name technique. Figure 3 shows a short list of software engineering techniques and techniques from other engineering fields.

4.1 Spectrum from analyses to solutions

Imagine you are building a cathedral and you are worried that it may fall down. You could build models of various design alternatives and calculate their stresses and strains. Alternately, you could apply a known solution, such as using a flying buttress. Both work, but the former approach has an analytical character while the latter has a known-good solution character.

Techniques exist on a spectrum from pure analyses, like calculating stresses, to pure solutions, like using a flying buttress on a cathedral. Other software architecture and design books have inventoried techniques on the solution-end of the spectrum, and call these techniques tactics [3]or patterns [12,7], and include such solutions as using a process monitor, a forwarder-receiver, or a model-view-controller.

The risk-driven model focuses on techniques that are on the analysis-end of the spectrum, ones that are procedural and independent of the problem domain. These techniques include using models such as layer diagrams, component assembly models, and deployment models; applying analytic techniques for performance, security, and reliability; and leveraging architectural styles such as client-server and pipe-and-filter to achieve an emergent quality.

4.2 Techniques mitigate risks

Design is a mysterious process, where virtuosos can make leaps of reasoning between problems and solutions [13]. For your process to be repeatable, however, you need to make explicit what the virtuosos are doing tacitly. In this case, you need to be able to explicitly state how to choose techniques in response to risks. Today, this knowledge is mostly informal, but we can aspire to creating a handbook that would help us make informed decisions. It would be filled with entries that look like this:

If you have <a risk>, consider <a technique> to reduce it.

Such a handbook would improve the repeatability of designing software architectures by encoding the knowledge of virtuoso architects as mappings between risks and techniques.

Any particular technique is good at reducing some risks but not others. In a neat and orderly world, there would be a single technique to address every known risk. In practice, some risks can be mitigated by multiple techniques, while others risks require you to invent techniques on the fly.

This frame of mind, where you choose techniques based on risks, helps you to work efficiently. You do not want to waste time (or other resources) on low-impact techniques, nor do you want to ignore project-threatening risks. You want to build successful systems by taking a path that spends your time most effectively. That means only applying techniques when they are motivated by risks.

4.3 Optimal basket of techniques

To avoid wasting your time and money, you should choose techniques that best reduce your prioritized list of risks. You should seek out opportunities to kill two birds with one stone by applying a single technique to mitigate two or more risks. You might like to think of it as an optimization problem to choose a set of techniques that optimally mitigates your risks.

It is harder to decide which techniques should be applied than it appears at first glance. Every technique does something valuable, just not the valuable thing your project needs most. For example, there are techniques for improving the usability of your user interfaces. Imagine you successfully used such techniques on your last project, so you choose it again on your current project. You find three usability flaws in your design, and fix them. Does this mean that employing the usability technique was a good idea?

Not necessarily, because such reasoning ignores the opportunity cost. The fair comparison is against the other techniques you could have used. If your biggest risk is that your chosen framework is inappropriate, you should spend your time analyzing or prototyping your framework choice instead of on usability. Your time is scarce, so you should choose techniques that are maximally effective at reducing your failure risks, not just somewhat effective.

4.4 Cannot eliminate engineering risk

Perhaps you are wondering why we should try to create an optimal basket of techniques when we should go all the way and eliminate engineering risk. It is tempting, since engineers hate ignoring risks, especially those they know how to address.

The downside of trying to eliminate engineering risk is time. As aviation pioneers, the Wright brothers spent time on mathematical and empirical investigations into aeronautical principles and thus reduced their engineering risk. But, if they had continued these investigations until risks were eliminated, their first test flight might have been in 1953 instead of 1903.

The reason you cannot afford to eliminate engineering risk is because you must balance it with non-engineering risk, which is predominantly, project management risk. Consequently, a software developer does not have the option to apply every useful technique because risk reductions must be balanced against time and cost.

5 Guidance on choosing techniques

So far, you have been introduced to the risk-driven model and have been advised to choose techniques based on your risks. You should be wondering how to make good choices. In the future, perhaps a developer choosing techniques will act much like a mechanical engineer who chooses materials by referencing tables of properties and making quantitative decisions. For now, such tables do not exist. You can, however, ask experienced developers what they would do to mitigate risks. That is, you would choose techniques based on their experience and your own.

However, if you are curious, you would be dissatisfied either with a table or a collection of advice from software veterans. Surely there must be principles that underlie any table or any veteran's experience, principles that explain why technique X works to mitigate risk Y.

Such principles do exist and we will now take a look at some important ones. Here is a brief preview. First, sometimes you have a problem to find while other times you have a problem to prove, and your technique choice should match that need. Second, some problems can be solved with an analogic model while others require an analytic model, so you will need to differentiate these kinds of models. Third, it may only be efficient to analyze a problem using a particular type of model. And finally, some techniques have affinities, like pounding is suitable for nails and twisting is suitable for screws.

5.1 Problems to find and prove

In his book How to Solve It, George Polya identifies two distinct kinds of math problems: problems to find and problems to prove [11]. The problem, "Is there a number that when squared equals 4?" is a problem to find, and you can test your proposed answer easily. On the other hand, "Is the set of prime numbers infinite?" is a problem to prove. Finding things tends to be easier than proving things because for proofs you need to demonstrate something is true in all possible cases.

When searching for a technique to address a risk, you can often eliminate many possible techniques because they answer the wrong kind of Polya question. Some risks are specific, so they can be tested with straightforward test cases.

It is easy to imagine writing a test case for “Can the database hold names up to 100 characters?” since it is a problem to find. Similarly, you may need to design a scalable website. This is also a problem to find because you only need to design (i.e., find) one solution, not demonstrate that your design is optimal.

Conversely, it is hard to imagine a small set of test cases providing persuasive evidence when you have a problem to prove. Consider, “Does the system always conform to the framework Application Programming Interface (API)?” Your tests could succeed, but there could be a case you have not yet seen, perhaps when a framework call unexpectedly passes a null reference. Another example of a problem to prove is deadlock: Any number of tests can run successfully without revealing a problem in a locking protocol.

5.2 Analytic and analogic models

Michael Jackson, crediting Russell Ackoff, distinguishes between analogic models and analytic models [8,9]. In an analogic model, each model element has an analogue in the domain of interest. A radar screen is an analogic model of some terrain, where blips on the screen correspond to airplanes - the blip and the airplane are analogues.

Analogic models support analysis only indirectly and usually domain knowledge or human reasoning are required. A radar screen can help you answer the question, “Are these planes on a collision course?” but to do so you are using your special purpose brainpower in the same way that an outfielder can tell if he is in position to catch a fly ball.

An analytic (what Ackoff would call symbolic) model, by contrast, directly supports computational analysis. Mathematical equations are examples of analytic models, as are state machines. You could imagine an analytic model of the airplanes where each is represented by a vector. Mathematics provides an analytic capability to relate the vectors, so you could quantitatively answer questions about collision courses.

When you model software, you invariably use symbols, whether they are Unified Modeling Language (UML) elements or some other notation. You must be careful because some of those symbolic models support analytic reasoning while others support analogic reasoning, even when they use the same notation. For example, two different UML models could represent airplanes as classes one with and one without an attribute for the airplane’s vector. The UML model with the vector enables you to compute a collision course, so it is an analytic model. The UML model without the vector does not, so it is an analogic model. So simply using a defined notation, like UML, does not guarantee that your models will be analytic. Architecture description languages (ADLs) are more constrained than UML, with the intention of nudging your architecture models to be analytic ones.

Whether a given model is analytic or analogic depends on the question you want it to answer. Either of the UML models could be used to count airplanes, for example, and so could be considered analytic models.

When you know what risks you want to mitigate, you can appropriately choose an analytic or analogic model. For example, if you are concerned that your engineers may not understand the relationships between domain entities, you may build an analogic model in UML and confirm it with domain experts. Conversely, if you need to calculate response time distributions, then you will want an analytic model.

5.3 Viewtype matching

The effectiveness of some risk-technique pairings depends on the type of model or view used. The module viewtype includes tangible artifacts such as source code and classes; the runtime viewtype includes runtime structures like objects; and the allocation viewtype includes allocation elements like server rooms and hardware. It is easiest to reason about modifiability from the module viewtype, performance from the runtime viewtype, and security from the deployment and module viewtypes.

Each view reveals selected details of a system. Reasoning about a risk works best when the view being used reveals details relevant to that risk. For example, reasoning about a runtime protocol is easier with a runtime view, perhaps a state machine, than with source code. Similarly, it is easier to reason about single points of failure using an allocation view than a module view.

Despite this, developers are adaptable and will work with the resources they have, and will mentally simulate the other viewtypes. For example, developers usually have access to the source code, so they have become quite adept at imagining the runtime behavior of the code and where it will be deployed. While a developer can make do with source code, reasoning will be easier when the risk and viewtype are matched, and the view reveals details related to the risk.

5.4 Techniques with affinities

In the physical world, tools are designed for a purpose: hammers are for pounding nails, screwdrivers are for turning screws, saws are for cutting. You may sometimes hammer a screw, or use a screwdriver as a pry bar, but the results are better when you use the tool that matches the job.

In software architecture, some techniques only go with particular risks because they were designed that way and it is difficult to use them for another purpose. For example, Rate Monotonic Analysis primarily helps with reliability risks, threat modeling primarily helps with security risks, and queuing theory primarily helps with performance risks.

6 When to stop

The beginning of this article posed two questions. So far, this article has explored the first: Which design and architecture techniques should you use? The answer is to identify risks and choose techniques to combat them. The techniques best suited to one project will not be the ones best suited to another project. But the mindset of aligning your architecture techniques, your experience, and the guidance you have learned will steer you to appropriate techniques.

We now turn our attention to the second question: How much design and architecture should you do? Time spent designing or analyzing is time that could have been spent building, testing, etc., so you want to get the balance right, neither doing too much design, nor ignoring risks that could swamp your project.

6.1 Effort should be commensurate with risk

The risk-driven model strives to efficiently apply techniques to reduce risks, which means not over-or under-applying techniques. To achieve efficiency, the risk-driven model uses this guiding principle:

Architecture efforts should be commensurate with the risk of failure.

If you recall the story of my father and the mailbox, he was not terribly worried about the mailbox falling over, so he did not spend much time designing the solution or applying mechanical engineering analyses. He thought about the design a little bit, perhaps considering how deep the hole should be, but most of his time was spent on implementation.

When you are unconcerned about security risks, spend no time on security design. However, when performance is a project-threatening risk, work on it until you are reasonably sure that performance will be OK.

6.2 Incomplete architecture designs

When you apply the risk-driven model, you only design the areas where you perceive failure risks. Most of the time, applying a design technique means building a model of some kind, either on paper or a whiteboard. Consequently, your architecture model will likely be detailed in some areas and sketchy, or even non-existent, in others.

For example, if you have identified some performance risks and no security risks, you would build models to address the performance risks, but those models would have no security details in them. Still, not every detail about performance would be modeled and decided. Remember that models are an intermediate product and you can stop working on them once you have become convinced that your architecture is suitable for addressing your risks.

6.3 Subjective evaluation

The risk-driven model says to prioritize your risks, apply chosen techniques, then evaluate any remaining risk, which means that you must decide if the risk has been sufficiently mitigated. But what does sufficiently mitigated mean? You have prioritized your risks, but which risks make the cut and which do not?

The risk-driven model is a framework to facilitate your decision making, but it cannot make judgment calls for you. It identifies salient ideas (prioritized risks and corresponding techniques) and guides you to ask the right questions about your design work. By using the risk-driven model, you are ahead because you have identified risks, enacted corresponding techniques, and kept your effort commensurate with your risks. But eventually you must make a subjective evaluation: will the architecture you designed enable you to overcome your failure risks?

About the article

This article is excerpted from Chapter 3 of the book Just Enough Software Architecture: A Risk-Driven Approach [6], available in hardback or as an e-book from http://RhinoResearch.com.

References

[1] Muhammad Ali Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009, September 2009.

[2] Mario R. Barbacci, Robert Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. Quality attribute workshops (qaws), third edition. Technical report, Software Engineering Institute, Carnegie Mellon University, 2003.

[3] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. Addison-Wesley, second edition, 2003.

[4] Marvin J. Carr, Suresh L. Konda, Ira Monarch, F.Carol Ulrich, and Clay F. Walker. Taxonomy-based risk identification. Technical Report CMU/SEI-93-TR-6, Software Engineering Institute, Carnegie Mellon University, June 1993.

[5] Viktor Clerc, Patricia Lago, and Hans van Vliet. The architect's mindset. Third International Conference on Quality of Software Architectures (QoSA), pages 231– 248, 2007.

[6] George Fairbanks. Just Enough Software Architcture: A Risk-Driven Approach. Marshal and Brainerd, 2010. E-book available from rhinoresearch.com.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-WesleyProfessional Computing Series). Addison-Wesley Professional, 1995.

[8] Michael Jackson. Software Requirements and Specifications. Addison-Wesley, 1995.

[9] Michael Jackson. Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, 2000.

[10] Henry Petroski. Design Paradigms: Case Histories of Error and Judgment in Engineering. Cambridge University Press, 1994.

[11] George Polya. How to Solve It: A New Aspect of Mathematical Method (Princeton Science Library). Princeton University Press, 2004.

[12] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

[13] Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

MVP Foundations for Your GWT Application

Adam Tacy, Robert Hanson, Jason Essington, Ian Bambury, and Christopher Ramsdale
http://www.manning.com/tacy/

This article is based on GWT in Action, Second Edition, to be published in August 2012. It is being reproduced here by permission from Manning Publications. Manning early access books and ebooks are sold exclusively through Manning. Visit the book's page for more information.

One of the main selling points of GWT is that it allows you to use an industry-grade language, with an industry-grade set of tools, to build...well...industry-grade web apps. But, as with any large scale development project, you can easily paint yourself into a corner. Far too many times when building GWT-based apps, we find ourselves slinging code wherever necessary to make the app work, and (sometimes more importantly) look good.

Fortunately, there is a well known solution to this problem: build your applications based on the model-view-presenter (MVP) paradigm.

Architecting your GWT-based apps to utilize the Model View Presenter (MVP) paradigm provides the foundation and rails necessary to avoid some common pitfalls.

Let's see first how the user sees the application and then what we will be doing from the MVP perspective.

From the user's perspective

Figure 1 shows the screens that can be found in our sample photo application.

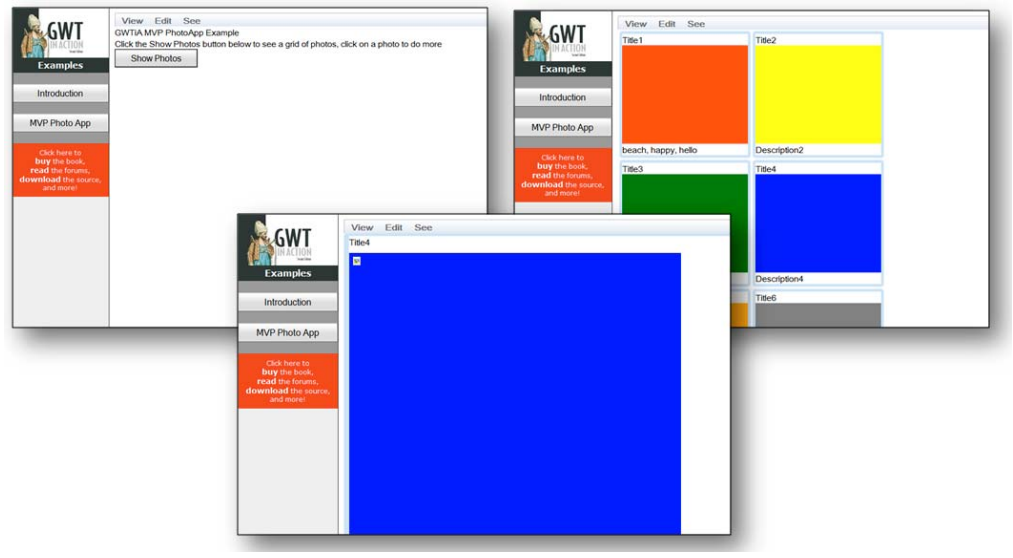


Figure 1 The three views shown by the photo application. The user starts at the WelcomeView, clicks a button to see the PhotoListView, then clicking a photo brings up the PhotoDetailsView. When you start the application, you are taken to a welcome screen, where you can request a list of photos by clicking the View Photos button or going through the menu bar options. While the list of photos is being retrieved, a busy message is displayed to the user.

Our photographer seems to have been in one of their abstract moods for this photo set because these are all just blocks of color. You can click on one of the photos, and a larger version is shown on its own, and here you can edit the title through an editable label where the new title is saved back to the server, if requested.

It really is a nice way to see the underlying implementation using MVP.

From the MVP perspective

At its core, MVP is a design pattern that breaks your app up into the components listed in table 1, the right-hand column of which explains how this is done for `PhotoApp`.

Table 1 Components of the MVP paradigm

MVP Component	Description
Model	Houses all of the data objects that are presented and acted upon within your UI. The number and granularity of models is a design decision. In our PhotoApp, we will have one very simple model: PhotoDetails - holds data about a photo, including the thumbnail URL, original URL, title, description, tags, and so on. (Our model will have data filled from a server component. We implement that as a very simple mock-up component that reads data from a file and. when "writing" just implements a delay before returning; you could extend that to read from a database, or make a call to a web service, and so on.)
View	These are the UI components that display model data and send user commands back to the presenter component. It is not a requirement to have a view per model—you may have a view that uses several models, or several views for one model. We will keep things simple for our Photo Application, and will have three views: WelcomeView - a very simple welcome page. PhotoListView - displays a list of thumbnail photos and their title PhotoDetailsView - displays the photo together with title and other data and allows the user to change some of those details.
Presenter	The presenter will hold the complex application and business logic used to drive UIs and changes to the model. It also has the code to handle changes in the UI that the view has sent back. Usually for each view, there will be an associated presenter. In our photo application, this means we will have the following three presenters: WelcomePresenter - pushes the welcome screen in front of the user, and handles the jump to PhotoListView. PhotoListPresenter - drives the thumbnail view. PhotoDetailsPresenter - drives the view of the original photo.

The concept is that a presenter creates its associated view and requests the view to be populated with appropriate data from the model. When UI elements in the view are updated, they notify the presenter to change the model. This way, the view doesn't need to know about the model, making it easy to swap and maintain.

A quick note on connecting views and presenters

The more contemporary MVP setup you may have read about in previous articles, blog posts, and so on, would indicate that presenters register to receive events raised from the view's widgets.

However, you will see that we specifically make the presenter register itself in the view, and that the view notifies the presenter by calling appropriate registered presenter methods when events are raised.

Navigation to a new view is performed either by the application updating the URL in the browser as a result of the user doing something, or the user clicking the forward/back buttons. The application will react to changes in the browser's URL and creating the appropriate presenter (which then creates the view). We will call this the Application Controller or `AppController` for short.

We will also make use of an `EventBus`. We'll use this to raise application-wide events. To keep to the MVP paradigm, we need to be careful that these events do not cause changes in the model. What is their use then? Well, in our application we will allow various presenters to raise `ApplicationBusy` and `ApplicationFree` events, for example, when starting and completing communication with the server. These events will be dropped onto the `EventBus` by the presenters, and the `AppController` will listen for them and react by showing and hiding a busy message.

Using a ClientFactory

Throughout the two photo app projects, you will see us use a `ClientFactory` object. It will provide access to singleton instances to common objects across the application. For example, it provides access to our views as singleton objects, and this improves efficiency because views contain DOM calls, which are expensive.

There's no requirement to use a `ClientFactory` in the MVP paradigm, but it is helpful if we are thinking of enterprise-wide development.

Between a presenter and its view, there is a strong connection, but loose coupling. Nice words, but what does it mean? Well, let's crack open some of the photo apps code and have a look.

Creating Views

Remember that our view should have no application logic in it, at all. It should be just UI components that the presenter can access to set or get values from. All of our views will be implemented as three separate items: a generic interface, a specific interface and an implementation. The next three sections will look at each of those in turn, starting with the generic interface.

The root of all views

Each of our views will be built upon a basic interface that we will call `View`. This is not a requirement to have, but suits us well as it creates a basic contract that all our views will adhere to, and that other components are aware of and can trust. Here it is:

```
public interface View extends IsWidget{
  void setPresenter(PhotoDetailsPresenter presenter);
}
```

There are two key things to note about this interface. Any implementation must implement the `setPresenter` method - which allows the presenter to register itself with this view (this is the topic of section 14.1.3 so we will leave that for now). The other is that all our views will extends the `IsWidget` interface. This means each view will implement the `asWidget` method, which we will have to implement so that it returns a representation of the view that is a widget. Luckily, since most views will be made up from a `Composite` which, from GWT 2.1 onwards, already implements `asWidget`, we are good to go with no extra work!

Each specific view needs to be described individually, and for that, we create first a new interface.

View specific interface.

For each view we will have, we create a new interface that extends `View`. These view specific interfaces define the methods that a presenter can call to get access to the UI components. As an example, our `PhotoDetailsView` will have a title that can be changed and so the value of needs to be retrieved. We can make the interface show that by saying it must implement a getPhotoTitle method that returns an object implementing the `HasValue<String>` interface:

```
public interface PhotoDetailsView extends View {
  HasValue<String> getPhotoTitle();
}
```

Our application has three views, so you can find the three interfaces in the source code, for example in the `com.manning.gwtia.ch14.client.mvp.views` package.

With the details of the views now specified in the interfaces, we need to implement them.
Implementing the views

Our view specific interfaces detail exactly what can be expected from the view. But without an implementation, they are not going to do much!

It turns out that our implementation of the `PhotoDetailsView` interface is pretty simple. That should be as expected as the view is dumb and contains no logic. The first two chapter examples are the same, except they differ in how views are implemented. In the first example, the implementations are verbose as we have a lot of boilerplate code; in the second, we have swapped the view implementation for UiBinder versions.

The only difference between the two versions is the use of UiBinder, and therefore how we bind events to actions. The actual actions do not change. Take the detailed view, each implementation implements the `setPresenter` method to register the parameter as this object's presenter:

```
public void setPresenter(PhotoDetailsPresenter presenter) {
   this.presenter = presenter;
}
```

Each view also implements a reaction to the title being changed. In the UiBinder approach it is neatly described as follows

```
@UiHandler("title")
public void changeTitle(ValueChangeEvent<String> event) {
   if (presenter != null){
      presenter.onUpdateTitle();
   }
}
```

In other words, if the `title` widget raises a `ValueChangeEvent`, then as long as the presenter is not `null`, call the presenter's `onUpdateTitle` method.

In the non UIBinder project, the same logic is there, but it is implemented by manually creating the title object and then adding a `ValueChangeEvent` handler to it. By convention, we tend to put all the binding code in a `bind()` method, if we are not using UIBinder.

So, our view is dumb and contains zero application logic. That is because we agreed that all of that should appear only in the presenter. And this brings us to the discussion of those presenters.

Presenters

Presenters are where all the application logic sits and will have no UI components. (Those are all in the view, as we've just seen.) In a similar way to views, we provide a generic presenter interface, a specific one, and an implementation for each presenter.

The root of all presenters

Just like views, we will create a basic interface that all of our presenters will implement. Again, there is no requirement to do this; it is just good to have. Here's our basic interface:

```
public interface Presenter{
  public void go(final HasWidgets container);
  public void bind();
}
```

The `go` method takes the widget in which we wish the presenter-associated view associated to be displayed. All we require of that widget is that it implements the `HasWidgets` interface; in other words, it is a panel of some kind.

A presenter will also implement a `bind` method. In our design, this is where the presenter will listen to any application-wide events it is interested in (for example, this is where it hooks into the event bus) as well as where it calls the associated views `setPresenter` method.

Each view will have an associated specific interface.

Presenter-specific interface

The specific functionality of the view is given in a new interface that implements `Presenter`.

In the specific presenter interface we wish to declare those methods that will be called from the view when things happen. Remember that we said the view is responsible for reacting to UI events within itself, but that it will then call methods on the presenter that was registered with it for the actual business logic. Here, we see the `PhotoDetailsPresenter` interface relating to the title:

```
public interface PhotoDetailsPresenter implements Presenter{
  public void onUpdateTitle();
}
```

What we are saying here is that we expect the onUpdateTitle method to be called when the title is updated in the view. We would add other methods to the interface based on the other UI components that are of interest.

Now we have the specific presenter interface in place, we should implement it.

Implementing specific presenters

Our presenters are just implementations of the specific presenter interfaces. For example, `PhotoDetailsPresenterImpl` implements the `PhotoDetailsPresenter` interface.

This means it needs to implement the `go` and `bind` method from `Presenter` as well as the `onUpdateTitle` from `PhotoDetailsPresenter`. Listing 1 shows the skeleton of our implementation.

Listing 1 PhotoDetailsPresenter

```
public class PhotoDetailsPresenterImpl implements PhotoDetailsPresenter {

   private ClientFactory clientFactory = GWT.create(ClientFactory.class);
   public PhotoDetailsPresenterImpl(final PhotoDetailsView photoDetailsView
                                    , final String id) {
      this.rpcService = clientFactory.getPhotoServices();        |#1
      this.eventBus = clientFactory.getEventBus();               |#1
      this.photoDetailsView = photoDetailsView;                  |#1
      eventBus.fireEvent(new AppBusyEvent());                     #2
      rpcService.getPhotoDetails(……)                              #3
      bind();
   }

   public void bind() {
      photoDetailsView.setPresenter(this);                        #4
   }

   public void go(final HasWidgets container) {               |#5
      container.clear();                                      |#5
      container.add(photoDetailsView.asWidget());             |#5
   }                                                          |#5

   public void onUpdateTitle() {                              |#6
      rpcService.savePhotoDetails(……);                        |#6
   }                                                          |#6
}
```

#1 Getting shared resources
#2 Firing application wide event
#3 Making a server call
#4 Binding to other items
#5 Implementing the go method
#6 Called from View when title updated

The first thing that is done in the constructor is to get hold of some common resources from the `ClientFactory` that we previously mentioned. This is not a requirement but makes our lives easier. For example, we grab resources such as the RPC service and the event bus - we might as well share the RPC service for efficiency, and we have to share the event bus; otherwise, it would not be system wide.

Once we have grabbed our resources from the factory, we make a call to get the details of the photo just after we have fired off a system-wide `ApplicationBusy` event. The intention here is that some other component will inform the user that the application is busy in some way—we don't care at this point how that is done. Not shown in listing 1 is the fact that an `ApplicationFree` event is fired within the RPC return handling code so that the user is notified the application is no longer busy.

Within the constructor, we also call the `bind` method from the `Presenter` interface. For this implementation, that simply calls the `view`'s `setPresenter` method to register this view with that presenter. Other implementations may register on the event bus if it has to handle application wide events.

In #5, we are implementing the `Presenter`'s `go` method. This clears the container widget passed in as the parameter and then adds the associated view as a widget. In the application controller, we will create presenters in the following manner:

```
new PhotoDetailsPresenterImpl(clientFactory.getDetailsView())
        .go(container)
```

Where `container` is the widget that we wish the presenter to present in the view.

That's it really for the views and presenters. Just before we jump to the AppController, which is responsible for tying everything together and starting presenters, we want to take a little deeper look at the relationship between presenter and views we have used.

The two-way presenter/view relationship

At the heart of the MVP pattern is the relationship between your presenters and views. Presenter classes should contain all of your complex application logic, and no widget code/references. Completely inverse to that, your views should contain nothing more than widgets, and should avoid any application logic wherever possible.

Why? For two reasons: 1) fast testing with increased code coverage and 2) maximum code reuse when porting to other Java based platforms, for instance Android (written carefully in the MVP paradigm, the application logic of a GWT application can be reused and all you need to do is just replace the view component).

If you look at testing, enforcing this split between presenters and views offers another way to save more time/money/development frustration. Whenever you find yourself needing to test functionality within your app that relies on widgets or some Javascript Native Interface (JSNI), you're going to need a browser context. This means you're going to need to run it within a `GWTTestCase`, which means - you guessed it - it's going to take a long time to run that test. So how do we fix this? Simple. Let's not test them at all - if you make the view as dumb as possible, by moving all of your app logic out into the presenter, you should be left with a view that contains nothing more than GWT widgets. Given that these are continuously tested by the GWT team, doing so in our own tests is simply redundant and not required. And, where you do need to test your views, those tests should be few and far between and, in many instances, simply integration testing being driven by Selenium (or some Selenium-like testing suite).

Now that we agree on moving the app logic out of the view, we have to move the view logic out of the presenter. More specifically, we have to make sure our presenters are solely relying on plain old Java code. This is to ensure that they can be tested using a vanilla JUnit test (which runs in ~1/1000th of the time of a `GWTTestCase`).

But, we are also making the relationship between view and presenters slightly differently to what you might expect. The more contemporary MVP setup you may have read about in previous articles, blog posts, and so on, would indicate that presenters register to receive events from the view's widgets.

However, you can see in the code above, we specifically make the presenter register itself in the `bind` method with the view via the view's `setPresenter` method, and that the view listens for events and then calls the appropriate method in the presenter.

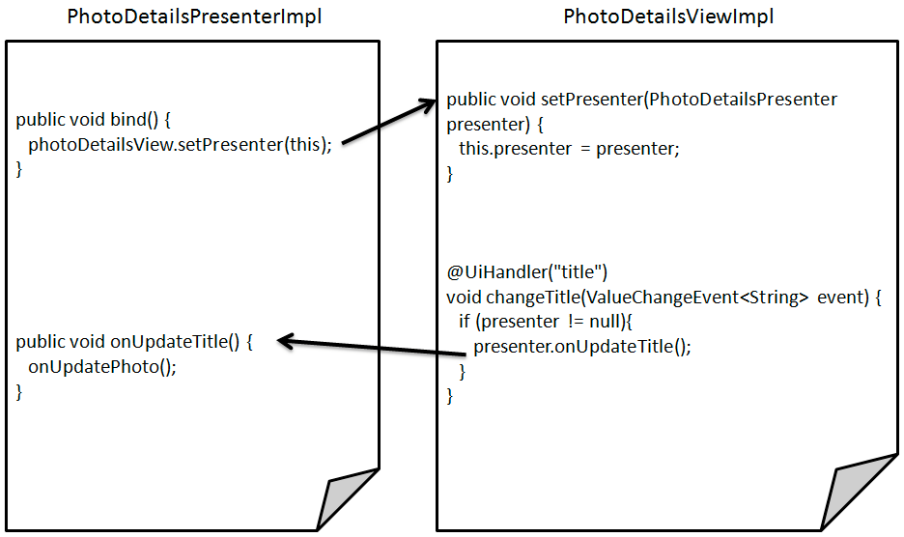


Figure 1 The coupling between presenter and view in our photo application.

Technically there is no reason not to make the presenter listen for events from the view widgets. But allowing the view to call back into the presenter makes using the UIBinder approach easy, as you can see in figure 1. Up until now we have views and presenters for our PhotoApp, but we have no way of knowing what presenter to request (so no idea what view to show), or react to the user changing views. For that, we need to control the application.

Controlling the application

We need a mechanism to change views in our application. The most common one is to indicate in the browser URL the new view required, usually via a token on the URL, and then to react to that.

In our presenters, if a view change is needed, we will change the history token to an appropriate value. This happens, for example, when selecting a photo in the Photo List view. The click on a photo is registered in `PhotoListViewImpl`, which then calls back to the presenter saying which photo is selected. In the `PhotoListPresenterImpl` code, we have:

```
public void onSelectPhotoClicked(String id) {
   History.newItem(Tokens.DETAIL + "&" + id);
}
```

All we are doing here is changing the history token to have a value defined in the `Tokens` class to represent Photo Detail view appended by the photo id the user is interested in. If we're interested in photo number 00004, then our development mode URL becomes as shown in figure 2.

Figure 2 Development mode URL after photo 00004 has been selected to be shown in the detailed view

If we're using history to store tokens, then we must set up our GWT history management to handle history changes and fire up the appropriate presenter. We do that in the `AppController` class, as shown in listing 2.

Listing 2 AppController showing bind() to History and other events

```
public class PhotoAlbumApp implements ValueChangeHandler<String> {      #1

  private void bind() {
    :
    History.addValueChangeHandler(this);                                #1
  }

  public void onValueChange(ValueChangeEvent<String> event) {           |#2
    String token = event.getValue();                                    |#2
    if (token != null) {                                                |#2
      if (token.startsWith("details"))                                  |#2
        doPhotoDetailsDisplay(token);                                   |#2
      else if(token.startsWith(”list”))                                 |#2
      :                                                                 |#2
    }                                                                   |#2
  }                                                                     |#2

  private void doPhotoDetailsDisplay(String token){}                    |#3
  private void doPhotoListDisplay(String token){}                       |#3
  private void doWelcomeDisplay(String token){}                         |#3
}
```

#1 Implementing the History Management
#1 Binding events to actions in the code
#2 Handling history changes
#3 Changing the view

The `AppController` binds itself to listen to listen to history events in #1 by implementing the `ValueChange` interface as well as adding this object as the `ValueChangeHandler` in the `History`.

We must implement the `onValueChange` method (#2), which is called when the history changes by GWT's history subsystem. This method we set up to simply parse the history token to determine what view is requested, and then is required, and will call one of the `do` methods in #3, if the history token is recognized and an action can be determined. These `do` methods are responsible for creating the view and the presenter, and then calling the presenter's `go` method. For example, if we have changed to the situation in figure 2, then the `onValueChange` method determines that the value `detail` means it needs to call the `doPhotoDetailsDisplay` method. This action shows the user the new photo details view with the requested image, while the `AppController` sits there waiting for the next history change.

Summary

The MVP pattern is extremely useful when building large, web-based applications with GWT. Not only does it help make code more readable, and subsequently more maintainable, it also makes it much easier to implement new features, optimizations, and automated testing. Speaking from experience, we can't stress the testing benefits enough. It's a fundamental part of writing real-world applications, but often times it's overlooked because it's left until the end, and is too much of a pain to integrate. When your app is developed using the MVP pattern, test cases are a piece of cake, so much so that you'll want to write them first. Remember, test driven development is your friend.

TestNG

Tomek Kaczanowski, kaczanowski.tomek [at] gmail.com
http://practicalunittesting.com

TestNG is general purpose open source Java testing framework, not limited to unit tests. As a natural competitor to JUnit, it trumps its predecessor with features better suited for integration and end-to-end testing, still being as strong as JUnit in the field of unit tests.

TestNG supports parameterized tests out-of-the-box in much more convenient way than JUnit does. It facilitates running multi-threaded tests, and allows to express dependencies between test methods.

Web Site: http://testng.org
Version Tested: TestNG 6.4 on Linux with Java 1.6.0_26
License & Pricing: Open Source (Apache 2.0, http://testng.org/license/)
Support: User mailing list (http://groups.google.com/group/testng-users)

Using TestNG

TestNG is distributed as a single JAR file, which must be available in the classpath in order to execute tests. It integrates very well with the most popular build tools: Ant, Maven (via Maven Surefire Plugin) and Gradle. TestNG is also supported by all major IDEs (in some cases the installation of additional plugin is required), can be used with different JVM languages (e.g. Java, Groovy, Scala) and cooperates with many quality and testing tools (e.g. code coverage tools, mocking libraries, matchers libraries). Some popular solutions - e.g. Spring Framework - provide means to facilitate testing with TestNG.

This introductory article concentrates on the TestNG features. Please consult TestNG documentation for installation instructions.

Developers Testing

The idea that developers have to test their own code has gained ground in recent years. Numerous examples of complex systems tested almost solely by developers prove the usefulness of developers tests. Unit tests help to maintain a sound codebase, which is crucial for its maintenance and further development. Integration and end-to-end tests allow to verify if the produced software meets client's requirements. TestNG can be used to implement all kinds of tests, thus improving both the internal and the external quality of the implemented systems.

Let the Code Speak

We start our journey into the world of TestNG with a very simple example. We have a class which represents money (named - surprise, surprise! - `Money`), and we want to test whether its `add()` method is properly implemented. A simple TestNG test could look as follows:

```
import org.testng.annotations.Test; [1]
import static org.testng.Assert.assertEquals; [2]

public class MoneyTest {
```

```
     @Test
     public void shouldAddSameCurrencies() {
     Money moneyA = new Money(4, "USD"); [3]
     Money moneyB = new Money(3, "USD"); [3]
     Money result = moneyA.add(moneyB); [4]
     assertEquals(result.getAmount(), 7); [5]
     }
}
```

Two imports are required:

[1] one for `@Test` annotation which marks test methods (that is, methods which should be executed by TestNG),

[2] assertion methods, which are later used to verify our expectations.

The test method itself is very simple: it constructs the required objects [3], invokes the method in question [4], and verifies its result [5].

If the assertion is not met then TestNG provides a detailed information on the cause, for example:

```
java.lang.AssertionError:
    Expected :7
    Actual   :6
```

TestNG also provides a stacktrace, which is not printed here for the sake of brevity. The stacktrace makes it clear which line of test caused the test to fail. This allows developer to quickly navigate to the failed test, and start fixing things.

Based on this example we can see that TestNG works similarly to other testing frameworks. Now let us take a look at some more advanced and interesting features of TestNG.

Parameterized Tests

Unlike other testing frameworks, TestNG allows to pass parameters to test methods. . The following snippet of code illustrates this idea:

```
@DataProvider [1]
private static final Object[][] getMoney(){
    return new Object[][] {
        {new Money(4, "USD"), new Money(3, "USD"), 7},
        {new Money(1, "EUR"), new Money(4, "EUR"), 5},
        {new Money(1234, "CHF"), new Money(234, "CHF"), 1468}};
}

@Test(dataProvider = "getMoney") [2]
public void shouldAddSameCurrencies(Money a, Money b,
        int expectedResult) {
    Money result = a.add(b);
     assertEquals(result.getAmount(), expectedResult);
}
```

The purpose of the first method - `getMoney()` - is to provide data for the actual test method - `shouldAddSameCurrencies()`. – When we run this test, we execute three tests, each with different set of parameters. The first test would be run with US Dollars, the second with Euros, and the third with Swiss Francs. This is achieved with two simple annotations: [1] the `dataProvider` attribute of the `@Test` annotation informs TestNG that it should ask the [2] `getMoney()` method to provide data. Such construct allows to execute many test cases at once without introducing repetition of code.

Data providers nicely decouple the "business logic of test" from the test data. The logic to be tested is enclosed within the test method, while the data is kept separately.

Test Fixture Setting

The setting of the test context (so-called test-fixture) is one of the challenges of tests writing. TestNG helps by providing many methods which allow to control when the objects are being created. This is very important especially for integration tests, which often use resources (e.g. web servers or database connections) that take time to create. An example is presented below. A server is created before the whole test suite is executed [1], while a client object is created before every test method [2]. This way a costly resource is set up only once, and client (a class we test) is in a clean state before any of its method is executed.

Additionally, the `timeOut` attribute of the `@Test` [1] annotation will cause the `setUp()` method to fail if it does not finish before `timeOut` (which is set to one second in this case). This is another useful feature of TestNG which allows to avoid tests hanging forever.

```
@Test
public class TestFixtureTest {

     private Server server;
     private Client client;

     @BeforeSuite(timeOut = 1000) [1]
     public void setUpServer() {
         server = new Server();
         server.start();
     }

     @BeforeMethod [2]
     public void setUpClient() {
         client = new Client();
     }

     // some test methods here

     @AfterSuite
     public void shutDownServer() {
         server.stop();
     }
}
```

Custom Listeners and Reporters

In case of all kind of tests, it is very important that developers get a very precise information on the results of test execution. TestNG not only provides a decent report by default, but also facilitates to tweak it. It is up to developers how they use this feature: printing some additional information during the execution of tests (e.g. time of execution of the last test method), taking screenshots after each failed Selenium test, or writing test results into an Excel spreadsheet.

Parallel Tests Execution

More and more of the code we write should work when accessed by multiple threads at once. TestNG provides some support for testing this type of code as we can see below. Using the `threadPoolSize` and the `invocationCount` attributes of the `@Test` annotation [1] we instruct TestNG to execute the same test method 100 times by 7 threads simultaneously. This way we can find out if our `IdGenerator` really generates unique IDs (at least within a single JVM). This test method relies on the specific set implementation provided by the `HashSet` class, which returns true only if an unique object was actually added to it [2]. The parallel execution of this test method by many threads is done with the use of a single annotation and does not require any additional coding.

```
public class IdGeneratorTest {

     private IdGenerator idGen = new IdGenerator();
     private Set<Long> ids = new HashSet<Long>(100);

     @Test(threadPoolSize = 7, invocationCount = 100) [1]
     public void idsAreUnique() {
     assertTrue(ids.add(idGen.nextId())); [2]
     }
}
```

Groups and Dependencies Between Tests

Grouping tests is another very useful feature of TestNG. Adding appropriate annotations to test classes or test methods, you could easily divide your tests into “fast” and “slow” groups), and execute them independently.

While it is usually advisable to keep unit tests completely independent from each other, this is not the case for integration and end-to-end test. TestNG recognizes this difference and provides convenient ways to implement such scenarios. It is possible to express dependencies between groups of tests and/or individual tests. The dependent test methods would be run only if all test methods, they depend on, succeeded. This could be used to implement many interesting scenarios. You could run integration tests only if unit tests passed or exhaustive system tests only if smoke tests succeeded. This helps to shorten the feedback loop, which in general is a good thing.

Conclusions

The old saying says that you should “use the right tool for the job”. In these days of strong focus on automated tests, and teams moving towards Continuous Deployment, it is even more important that the developers use the best tools available. I hope to convince you that TestNG is indeed the right tool with this article that presented only a subset of TestNG features.

Let us finish this short overview with a summary of TestNG capabilities. TestNG:

- supports all kind of developer tests: unit, integration and end-to-end tests,
- is robust and frequently released
- is well documented, and has a vivid users community,
- has a strong development leader - Cédric Beust - who makes sure the development of TestNG does not deviate from the right path,
- trumps JUnit in many aspects, especially when it comes to integration and end-to-end tests,
- has a flat learning curve,
- is well supported by the vast majority of tools.

Further Readings

- TestNG website: http://testng.org
- TestNG official documentation: http://testng.org/doc/documentation-main.html
- TestNG users mailing list: http://groups.google.com/group/testng-users
- Next Generation Java Testing: TestNG and Advanced Concepts, book by Cédric Beust http://testng.org/doc/book.html
- Practical Unit Testing with TestNG and Mockito, book by Tomek Kaczanowski, http://practicalunittesting.com

SBT Scala Build Tool

Evgeny Goldin, http://evgeny-goldin.com/

"sbt" is a Scala-oriented build tool, developed by Mark Harrah. Unlike most other general build tools, "sbt" is targeted to support only Scala and Java projects. In addition to traditional batch operation where a number of build tasks are executed in sequence, "sbt" provides a special *interactive mode*, making Scala builds significantly faster by using the same JVM, watching project sources for modifications and keeping the compiled classes "warm". As a build tool, it is capable of compiling Scala code, packaging archive artifacts, running Scala code, executing tests and many other build operations.

Web Site: https://github.com/harrah/xsbt/wiki
Version Tested: 0.11.0 on Mac OS X 10.7.3, Java 1.6.0_29 and Scala 2.9.1, March 2012
License & Pricing: https://github.com/harrah/xsbt/blob/0.11/LICENSE, Free
Documentation:
Getting started guide: https://github.com/harrah/xsbt/wiki/Getting-Started-Welcome
Detailed topics: https://github.com/harrah/xsbt/wiki/Detailed-Topics
Default settings and tasks: http://harrah.github.com/xsbt/latest/sxr/Keys.scala.html
Plugins list: https://github.com/harrah/xsbt/wiki/sbt-0.10-plugins-list
Support:
Mailing list: https://groups.google.com/forum/?fromgroups#!forum/simple-build-tool
Issue tracker: https://github.com/harrah/xsbt/issues
Videos:
"SBT Introduction & Cookbook for Scala": http://goo.gl/sR552
"sbt: Design and Implementation": http://days2010.scala-lang.org/node/138/165
"sbt 0.9: Why, what, how?": https://vimeo.com/20263617

Installation

One simple way to install "sbt" would be downloading a self-executable sbt-launch.jar file and running it with an "sbt" script:

```
java -Xmx512M -jar "sbt-launch.jar" %*
```

If you're a Mac user then "sbt" can be installed with MacPorts:

```
sudo port install sbt
```

When "sbt" is run for the first time all required dependencies are downloaded automatically. Make sure you have everything set up correctly by running "sbt" and then typing "about":

```
$ sbt
[info] Loading global plugins from /Users/evgenyg/.sbt/plugins
[info] Loading project definition from
/Users/evgenyg/Projects/scala-examples/project
[info] Set current project to scala-examples (in build
file:/Users/evgenyg/Projects/scala-examples/)
> about
[info] This is sbt 0.11.0
[info] The current project is
{file:/Users/evgenyg/Projects/scala-examples/}hello
```

```
[info] The current project is built against Scala 2.9.1
[info] Available Plugins: org.sbtidea.SbtIdeaPlugin
[info] sbt, sbt plugins, and build definitions are using Scala
2.9.1
>
```

If you're using IntelliJ IDEA then you can also install two additional plugins:

- sbt-idea plugin for auto-generating IDEA project files using your “sbt” Scala configs.
- SBT plugin for running SBT interactive console from inside IntelliJ IDEA.

“sbt” example project

“sbt” example is available at https://github.com/evgeny-goldin/scala-examples. This is a simple Scala project making use of Twitter's API to make a search request for user query. You can clone it and run as follows (commands typed are marked in **bold**):

```
$ git clone git://github.com/evgeny-goldin/scala-examples.git
Cloning into 'scala-examples'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 24 (delta 3), reused 24 (delta 3)
Receiving objects: 100% (24/24), 2.06 KiB, done.
Resolving deltas: 100% (3/3), done.
$ cd scala-examples
$ sbt
[info] Loading global plugins from /Users/evgenyg/.sbt/plugins
[info] Set current project to scala-examples (in build
file:/Users/evgenyg/Temp/scala-examples/)
> run something
[info] Running com.goldin.scala.Twitter4j something
...
[iam_imperfect]: [RT @EhIs4Adam: Beginning to feel the need to
work out or something... you know because of the whole 2012
thing. #EndOfDays #PumpItUp]
[bnsbbyx3]: [If you're truly in love, the other person should be
your bestfriend. If they're not.. there's something wrong with
your relationship.]
...
[success] Total time: 1 s, completed Feb 25, 2012 6:44:43 PM
>
```

With just a few keystrokes we've run a Twitter query for “something”. Try it out for any other query!

Interactive Mode

Java-based build tools operate in *batch mode*: you run the tool and provide it with goals or tasks to execute, such as “ant clean zip”, “mvn clean install” or “gradle test”. The build tool boots the JVM, processes the build file, reads the goals and executes them. Gradle is capable of running a background daemon process, saving on JVM boot up time, but the overall picture stays mostly the same.

"sbt" took a new approach and provided an *interactive mode* in which you only invoke "sbt" once, as we've just seen, and then "sbt" waits for your commands. "run something" in the previous example was a "run" command followed by the "something" argument, invoking the Scala code and passing it the argument. The reference to the commands provided by the tool is available at http://goo.gl/vZ73u and it includes the following commands, among many others: update, compile, package, test, publish, and doc. http://goo.gl/2y7bU provides a list of all "sbt" tasks available by default.

Two special commands "show" and "inspect" allow to examine the build environment of the project in more details:

```
> show sources
[info] ArrayBuffer(/Users/evgenyg/Temp/scala-
examples/src/main/scala/com/goldin/scala/QueryRunner.scala,
/Users/evgenyg/Temp/scala-
examples/src/main/scala/com/goldin/scala/Twitter4j.scala)
> show dependency-classpath
[info] ArrayBuffer(Attributed(/Users/evgenyg/.sbt/boot/scala-
2.9.1/lib/scala-library.jar),
Attributed(/Users/evgenyg/.ivy2/cache/org.twitter4j/twitter4j-
core/jars/twitter4j-core-2.2.5.jar),
Attributed(/Users/evgenyg/.ivy2/cache/org.slf4j/slf4j-
api/jars/slf4j-api-1.6.4.jar),
Attributed(/Users/evgenyg/.ivy2/cache/ch.qos.logback/logback-
classic/jars/logback-classic-1.0.0.jar),
Attributed(/Users/evgenyg/.ivy2/cache/ch.qos.logback/logback-
core/jars/logback-core-1.0.0.jar))
> inspect scala-version
[info] Setting: java.lang.String = 2.9.1
[info] Description:
[info] The version of Scala used for building.
...
> inspect clean
[info] Task: Unit
[info] Description:
[info] Deletes files produced by the build, such as generated
sources, compiled classes, and task caches.
>
```

"build.sbt"

"sbt" is guided by the "build.sbt" file stored in project's root:

```
organization := "com.goldin"
name         := "scala-examples"
version      := "1.0"
scalaVersion := "2.9.1"
showSuccess  := true
showTiming   := true
libraryDependencies += "org.twitter4j"  % "twitter4j-core"  %
"2.2.5"
libraryDependencies += "org.slf4j"      % "slf4j-api"       %
"1.6.4"
libraryDependencies += "ch.qos.logback" % "logback-classic" %
"1.0.0"
```

As you see, it contains the standard settings of a build model, probably familiar to you from Maven or Gradle. Dependency delegation is performed using Ivy and the language used to specify the model is Scala. See http://goo.gl/aW4cI for more details about settings available and http://goo.gl/2y7bU for a complete list of all default settings. Note that "sbt" follows the standard Maven conventions: sources are expected to be in "src/main/java" and "src/main/scala", tests are expected to be in "src/test/java" and "src/test/scala". It means very little needs to be done in order to build your Scala project: put your sources in "src/main/scala", add the corresponding "build.sbt" file and type "sbt package" or just "sbt" and then "package".

Why SBT?

What makes "sbt" different from other build tools? After all, one can compile Scala sources with Maven and Gradle, so what's the point of using a new build tool?

As it turns out, there are plenty of reasons for using "sbt".

First of all, "sbt" was developed as a Scala build tool from the very beginning. As I mentioned earlier, most Java-based build tools are general in their nature, i.e., they can work with a variety of JVM languages: Java, Groovy, Scala, Jython, JRuby, etc. "sbt" specializes in Scala which makes it more powerful in various Scala-related scenarios. For example, it is possible to switch the Scala version from "2.9.1" to "2.9.0" with the "++2.9.0" command in the interactive mode, which makes it especially convenient for running tests in various Scala versions. Another example of a Scala-specific command is "console" starting a familiar Scala REPL with project files compiled and available in the classpath.

The tool itself is written in Scala and all build files are written in Scala as well. It means that Scala knowledge is expected but it also makes the job easier for Scala developers. It also means Scala rigorous type safety is built into your build files.

One of "sbt" cornerstone principles is "same JVM execution" using powerful interactive mode. It eliminates JVM startup costs and keeps compiled Scala classes "warm", thus making project recompilation an immediate event – the only sources modified are actually compiled. A special "~ command" syntax watches sources for modifications and re-runs the command automatically whenever modifications are recognized, freeing you from a need to re-run it manually. This way, the "~ test" will re-run project's tests every time a new test is added. The interactive mode is also accompanied with tab completion when commands are typed. As you see, it provides a tremendous speedup and advantage over the traditional "type and wait" way of running build tools, with Gradle being one of the few tools that work hard to minimize the wait time with its background daemon. It's good to see "sbt" takes this issue into a whole new level.

"sbt" allows for parallel aggregation and execution of multi-project builds, providing yet another attractive speedup for your builds.

As you see, "sbt" is all about Scala, flexibility and performance, therefore it is a compelling alternative in the build tools arena. If your project is developed in Scala, I believe this should be the first tool to check out.

CUBRID – Open Source RDBMS Highly Optimized for the Web

Esen Sagynov, NHN Corp., http://www.cubrid.org

CUBRID is a comprehensive open source relational database management system highly optimized for Web applications. Implemented in C programming language, CUBRID provides great scalability and high availability features recognized by industry leaders, in addition to almost full SQL compatibility with MySQL.

Web Site: http://www.cubrid.org
Latest Version: CUBRID 8.4.1 Beta (2012/01/31)
License & Pricing: GPL v2+ (Server), BSD (APIs, Tools).
Community Support: http://www.cubrid.org/questions, http://www.cubrid.org/forum

Overview

CUBRID Database project was started in 2006 by NHN Corporation, the leading IT services provider in Korea. Developed from scratch the first public release was announced in October 2008. As of today there have been 15 releases announced in the interval of two to six months. It is one of the most active and dedicated open source projects available under the terms of GPL v2.0 and higher.

The primary goal of CUBRID project is to develop a relational DBMS optimized for Web services. Thus, Web optimizations are in the core of CUBRID which has the following key features:

- Fully transactional, ACID compliant
- ANSI SQL
- Referential Integrity
- B+tree, covering index
- Table joins (INNER, LEFT, RIGHT, OUTER, UNION, MERGE)
- Nested and Hierarchical queries
- LOB data support (BLOB/CLOB)
- True VARCHAR
- Conditional Regular Expressions
- Over 90% SQL compatibility with MySQL
- Query plan caching
- Cursors, Triggers and Stored Procedures
- Unicode support0-
- Cross-platform
- Multi-process architecture and multi-threaded server implementation
- Native API for PHP, PDO, Python, Ruby, Perl, ODBC, OLEDB, ADO.NET, C.
- Native Database Administration and Migration Tools.
- Command line database management utilities and SQL Interpreter.

- High-Availability feature for 24/7 uninterrupted service availability highly recommended by leading IT and Communication providers. (Master : Slave architecture)
- Native Connection Pooling (can also be used with Oracle and MySQL)
- Load balancing and distribution
- Database Sharding for automatic scale-out – coming in the next version.
- Synchronous/asynchronous/semi synchronous, one-way, transaction-level, schema independent, chained or grouped replication
- Data Import/Export
- Online/Hot, Offline, and Incremental Backup with an opportunity to schedule and adjust based on the needs.
- Unlimited databases, tables and rows

CUBRID is a fully featured Enterprise level RDBMS available completely for free and no licensing fees. It is a reliable solution ideal for Web services.

Installation

Installing CUBRID is pretty straightforward. Below you can see the installation instructions for Ubuntu. Other Linux, Windows, Shell and RPM installation instructions are also available on the community wiki site.

To install CUBRID using apt-get on Ubuntu, we need to add CUBRID's repository so that Ubuntu knows where to download the packages from, and then tell the OS to update its indexes.

```
sudo add-apt-repository ppa:cubrid/cubrid
sudo apt-get update
```

Now install the latest version of CUBRID:

```
sudo apt-get install cubrid
```

Done. This will automatically create *cubrid* user and a group to manage CUBRID services. To complete the installation and set the CUBRID PATH variables, restart your OS. CUBRID Service will be set to auto start on system reboot.

Authentication in CUBRID

CUBRID provides two levels of authentication: *host* and *database*.

Host Authentication is similar to those of other RDBMS. To administer CUBRID Host Server from CUBRID Manager (CM), you need to enter admin’s username and password. When you try to connect to a host from CM for the first time, you will be prompted to set admin credentials. Once you connect to a host you can manage the Server depending on the permissions you have.

Once you are successfully connected to a host, you cannot simply start looking into each database available on this host even if you are a top-level database server administrator. You need to login to any database you want to access. Thus, an administrator of the database A successfully connected to a host will not have access to a database B existing in the same host, unless granted. This provides independence and added layer of security.

Creating a Database

There are several ways you can create a database. You can use CUBRID Manager (CM), an open source GUI based database administration tool developed with DBAs in mind, or CUBRID Query Browser (CQB), a lightweight version of CM oriented for developers, or CQB plugin for Eclipse IDE. Other options can be using programming APIs like PHP, Ruby, Python, Perl, ADO.NET, etc.

In this tutorial we will use `cubrid createdb` utility, which provides the necessary means to create a database in the command line. Typing `cubrid createdb` by itself will display various options you can use with this command.

Now let's create a database called `world_database`.

```
cubrid createdb world_database
```

Notice, that a database name should not contain "@" symbol since later when referencing the database name, @ will be interpreted as if a host name is specified. So avoid this kind of scenarios.

The default owner of the database, when created, is *dba* with a blank password.

Starting a Database

CUBRID provides one more convenient feature. You can control which databases need to be started. If you do not use a particular database, you can stop it to save the system resources. This feature is very useful especially in the development environment when you work on several projects simultaneously which use different databases. In this case, when you work on one of the projects, you can start *that* project's database(s) only. No need to spare the resources with other databases if you do not use them.

To start a database type the following command in the terminal. *Remember* that all these operations can also be performed in GUI based CUBRID Manager, if you prefer.

```
cubrid server start world_database
```

SQL in CUBRID

SQL syntax in CUBRID is very alike to those of MySQL and even Oracle. You can execute most of your existing SQL statements in CUBRID without any modifications. Though, there are a few important facts to know about CUBRID.

CUBRID is *not* a fork of MySQL (or any other DBMS, for that matters), therefore it does not have anything similar like `ENGINE=InnoDB` part in `CREATE TABLE` statement. So, if your SQL includes it, just remove it.

Take a quick look at the reserved words in CUBRID. If your project uses some of them as table or column names, you need to always quote them using backticks (``), or double quotes (""), or square brackets ([]), whenever they are referenced.

If you ever use Large Objects (BLOB/CLOB), in CUBRID they are stored outside the database on the external storage while the references to those files are stored in database columns. You are highly encouraged to read how LOB data should be used and maintained in CUBRID.

As of version 8.4.1 CUBRID does not support `UNSIGNED, BOOL, ENUM` data types, if you want to use them. They will be implemented in the next 8.5.0 version. You can find more information on recommended data type mapping at CUBRID community site.

Also pay attention to permitted maximum and minimum values for data types.

CUBRID uses B+tree indexes to improve search performance

- Works great for equality operator (=).
- Since indexes are represented as trees, *Key range scan* allows to significantly improve the search performance if `WHERE` clause contains range conditions (`<, >, <=, >=, =`). If Range conditions are not defined, the Optimizer will attempt to perform sequential table scan.
- Try to avoid using irregular conditions such as `<>, !=, or NULL`, since the Optimizer will not be able to take the full advantage of the index tree, and instead will perform a sequential scan.
- Covering Index is magic. Use it to improve the search performance.
- If possible, utilize `LIMIT` clauses to take advantage of *Key limit optimizations*.
- Learn about *In-place Sorting* feature in CUBRID to improve the performance of `ORDER BY` statements.
- Use COUNT(*) instead of COUNT(col_name), unless you really know what you are doing.

Take advantage of CUBRID's unique Click Counter feature. For example, instead of executing two separate SQLs like:

```
SELECT article FROM article_table WHERE article_id = 130,987;
UPDATE article_table SET read_count = read_count + 1 WHERE
article_id = 130,987;
```

… in CUBRID you can execute only one:

```
SELECT article, INCR(read_count) FROM article_table WHERE
article_id = 130,987
```

This will help you avoid expensive lock of the working record generated by `UPDATE` operation.

There are many resources you can refer to on the community site to learn more about SQL in CUBRID and query optimizations. You are highly encouraged to refer to them.

CUBRID APIs and Tools

A large variety of drivers are available for CUBRID developers. Refer to CUBRID APIs Wiki to download the necessary driver. On the download page you can also find the installation instructions and quick start guides.

CUBRID Manager (CM) is a great and powerful tool to manage databases developed with DBAs in mind. You can use it for both local and remote database administration. If you usually perform just basic database management, you may want to use the lighter version of CM called CUBRID Query Browser (CQB).

Additionally, if you think of migrating your databases from other DBMS to CUBRID, consider using easy-to-use CUBRID Migration Tool (CMT), which provides a smooth bridge between your databases.

All these tools are open sourced and available for download at CUBRID Tools Wiki. You are highly encouraged to visit Resources for CUBRID Developers.

At this point it is fair to wrap up this introduction of CUBRID, an open source relational DBMS highly optimized for Web applications.

If you are interested what other users are asking about CUBRID, feel free to visit CUBRID Q&A site. If you encounter any issues, post your message to CUBRID Forum.

References

Community site http://www.cubrid.org

SF.net project site: http://sourceforge.net/projects/cubrid

GuaranáRafael Z. Frantz, University of Sevilla - Spain
http://www.tdg-seville.info/rzfrantz

Guaraná is a technology that can be used to design, implement, and run enterprise application integration solutions. In this article we provide an introduction to the domain-specific language that comes with Guaraná, as well as, the software development kit that can be used to implement and run the solutions.

Web Site: www.tdg-seville.info/rzfrantz/guarana
Version described: 1.4.0
License & Pricing: TDG's License 1.0 and free

1 - Introduction

Typical companies run software ecosystems [1] that consist of many applications that support their business activities. Frequently, new business processes have to be supported by two or more applications, and the current business processes may need to be optimised, which requires interaction with other applications. However, it is common that these applications were not designed with integration concerns in mind, i.e., they do not provide a programming interface. As a result, the interaction is not always a trivial task, and has to be carried out in most cases by means of the resources that belong to the applications, such as their databases, data files, messaging queues, and user interfaces. Recurrent challenges are to make the applications inter-operate with each other to keep their data synchronised, offer new data views, or to create new functionalities [2]. In this context, many companies rely on Enterprise Service Buses to develop their integration solutions, since they provide the necessary technology to integrate disparate applications by means of exogenous workflows [3, 4]. An integration solution is deployed to the software ecosystem as a new application that provides its users with a high-level view of the integrated applications with which they can interact.

Guaraná is a technology that can be used to design, implement, and run application integration solutions. Guaraná provides a very intuitive domain-specific language (DSL) that can be used to design solutions at a high-level of abstraction. It also provides a software development kit (SDK) that can be used to implement and run the solutions. In this article we will introduce the DSL and guide the readers in the use of the SDK for the implementation of the integration solution.

2 - The Domain-Specific Language

Guaraná DSL was designed aiming at provide support for the design of integration solutions at a high-level of abstraction, in a way that software engineers can concentrate efforts on the design of the solution without having to worry about the technical details of its implementations. The language provides large support for the integration patterns documented by Hohpe and Woolf in their book [2]. The main building blocks of Guaraná DSL are: communication ports, processes, tasks, and slots.

Communication Ports: Abstract away from the communication mechanism necessary to communicate with an application. In the current version of the language, four types of ports are supported:

Type of Port	Description
Entry Port	Enables a process to receive information from an application.
Exit Port	Enables a process to send information to an application.
Solicitor Port	Used to solicit information from an application.
Responder Port	Used to respond for request messages received from applications.

Table 1: Types of communication ports

Processes: Are intended to orchestrate the interactions with a number of applications. Processes rely on tasks to perform their orchestration activities. Simply put, a process can be viewed as a message processor.

Tasks: Are building blocks that allow executing an atomic action against messages that flow internally in a process. Guaraná DSL provides several different tasks organised categories. Table 2 shows some tasks that can be used to route, modify, or transform messages. Please, refer to Guaraná web site for a complete list and description of tasks.

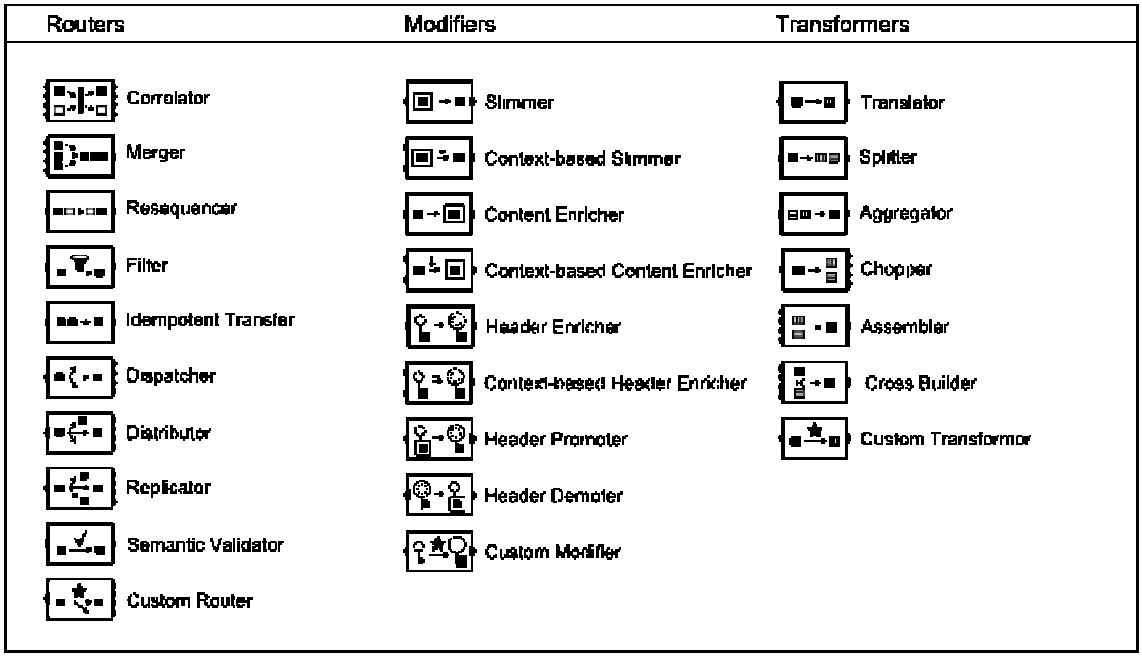


Table 2: Tasks that implement integration patterns in Guaraná DSL

Slots: Are used to connect and desynchronise the communication between two tasks or a task and a port. Every task must be connected to at least two slots, from which the task receive and send messages to other elements inside the integration process. Slots that connect ports and tasks are known as inter-slots.

3 - Travel Booking Example

Let us illustrate how Guaraná DSL can be used to design an integration solution for the well-known travel booking scenario. The goal of this integration solution is to take travel booking requests as input and book the flights and the hotel specified in the booking.

Integration Problem: The software ecosystem in this scenario contains five applications, namely: *Travel System*, *Invoice System*, *Mail Server*, *Flights Façade*, and *Hotels Façade*. The Travel System is an off-the-shelf software system that the travel agency uses to register information about their customers and booking requests. The invoice service runs on the Invoice System, which is a separate software system that allows customers to pay their travels using their credit cards. The Mail Server runs the e-mail service and is used for providing customers with information about their bookings. The Flights Façade and the Hotels Façade represent interfaces that allow booking flights and hotels. They both, in addition to the Mail Server, represent applications that were designed with integration concerns in mind. Contrarily, the Travel System and the Invoice System are software systems that were designed without taking integration into account, thus, the integration solution must interact with them by means of their data layer. The only assumption we make is that every booking registered in the Travel System contains all of the necessary information about the payment, flight and hotel, and a record locator which uniquely identifies the booking. The integration solution must periodically poll the Travel System for new travel bookings, so that flights and hotel can be booked, the customer can be invoiced and provided with a piece of e-mail with the information about his/her travels.

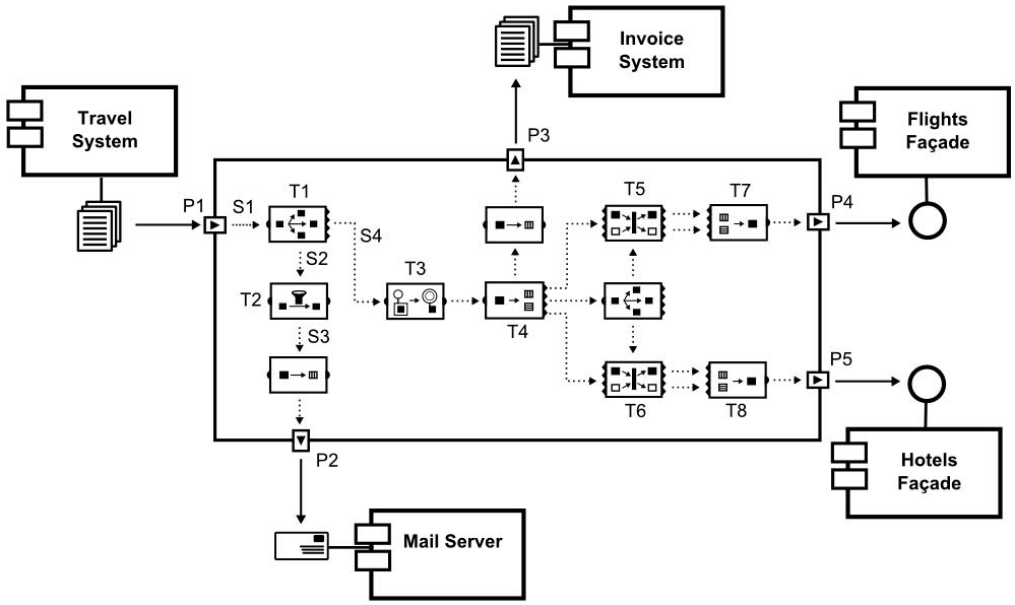


Figure 1: Integration solution for booking flights and hotels

Solution: The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, cf. Figure 1. Some ports have access to the Travel System and the Invoice System data layers by means of files. Translator tasks were used in the process to translate messages from canonical schemas into the schemas with which the integrated applications Invoice System and Mail Server work.

The workflow begins at entry port P1, which periodically polls the Travel System to find new bookings. Bookings are stored in individual XML files. For every booking, the entry port inputs a message to the process, which is in turn added to slot S1. Task T1 gets messages from this slot and replicates them, so that one copy is used to send the e-mail to the customer and the other is used to prepare the invoice and the booking. The first copy goes through filter task T2, which prevents exit port P2 from receiving messages without a destination e-mail address. Task T3 promotes the record locator from the body of the message to the header of the message, so that it can be used for correlation purposes in tasks T5 and T6.

Prior to the correlation, the second copy is chopped by task T4 into different messages, so that one outbound message with the payment information goes to the Invoice System and tasks T7 and T8 can assemble the messages used to book the flights and the hotel, respectively.

4 - The Software Development Kit

Guaraná SDK is a Java software tool that allows to implement and execute integration solutions designed with Guaraná DSL. The architecture of Guaraná SDK is organised in two layers. The first layer, referred to as the framework, provides an abstract implementation of all basic concepts of the DSL. The second layer, referred to as toolkit, provides a concrete implementation of the framework, as well as, a task-based runtime system. Below we will show how to implement some building blocks of the DSL using Guaraná SDK for the integration solution depicted in Figure 1.

To start you will have to create a Java project in your favourite IDE and import the jar file provided at Guaraná web site. Then you can start coding your solution. Since a process is only a container of tasks, you have to write:

```
import guarana.framework.process.Process;
public class BookingProcess extends Process {
      ...
}
```

Inside the process, you have to declare the ports, tasks, and slots of the process for your integration solution. Now, let us declare port P1 and task T1:

```
import guarana.framework.port.EntryPort;
import guarana.framework.task.Task;
...
private EntryPort p1;
private Task t1;
```

Since ports are responsible for the communication, they count on a special type of tasks: communicators. Every type of port has its corresponding communicator, thus Entry Ports use an `InCommunicator` task to read information from the application. Entry ports can be created as follows:

```
import guarana.toolkit.task.communicators.InCommunicator;
import guarana.framework.task.Slot;

p1 = new EntryPort( "Reads from the Travel System Application" ) {
@Override
      public void initialise() {

      setInterSlot( new Slot( "s1" ) );
      // Communicator
      c = new InCommunicator( ... );

      c.output[0].bind( getInterSlot() );


      setCommunicator( c );
}

};
addPort( p1 );
```

The code above creates an entry port and initialises it using its `initialise()` method. Inside this method, slot s1 is created and set to the port as the slot which is going to be used to pass inbound messages to the following task in the integration flow, the replicator task t1. The communicator task is set to the port, and, finally, the last line of code adds port p1 to the process.

Guaraná DSL allows software engineers to produce their models to solve the integration problem. Some tasks are very simple and software engineers do not have to specify any business logic to complement its semantics, such as the replicator task. However, other tasks have to be configured using low-level business logic, such as the filtering policy that must be applied to messages. This configuration is not possible if you are drawing your solution in a sheet of paper using Guaraná DSL, but can be done if you are using a software tool that allow to design and configure the integration solution using a visual designer. Below we show how to configure replicator task t1 from Figure 1.

```
import guarana.toolkit.task.routers.Replicator;
s2 = new Slot("s2");
s4 = new Slot("s4");

t1 = new Replicator( "t1", 2 );
t1.input[0].bind( p1.getInterSlot() );
t1.output[0].bind( s4 );
t1.output[1].bind( s2 );

addTask( t1 );
```

When configuring filter task t2, software engineers have to override the `doWork()` method to provide the filtering policy according to their solution. This task can be configured as follows:

```
s3 = new Slot("s3");

t2 = new Filter("t2") {
      @Override
      public void doWork(Exchange e) throws TaskExecutionException {

      Message<String> in = (Message<String>) e.input[0].poll();....
      String body = in.getBody();

      if ( body.length() != 0 ) {
          e.output[0].add(in);
      }
}
};
t2.input[0].bind( s2 );
t2.output[0].bind( s3 );
addTask( t2 );
```

Summary

In this article, we have introduced a domain-specific language (Guaraná DSL) to design enterprise application integration solutions at a high-level of abstraction. We have also introduced a software development kit (Guaraná SDK), which is a software tool to implement enterprise application integration solutions designed with Guaraná DSL.

Bibliography

[1] D. Messerschmitt and C. Szyperski. Software ecosystemm: Understanding an indispensable technology and industry. MIT Press, 2003

[2] G. Hohpe and B. Woolf. Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley, 2003

[3] J. Davies, D. Schorow, S. Ray, and D. Rieber. The definitive guide to SOA: Enterprise Service Bus. Apress, 2008

[4] D. Chappel. Enterprise Service Bus: Theory in practice. O'Reilly, 2004

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php