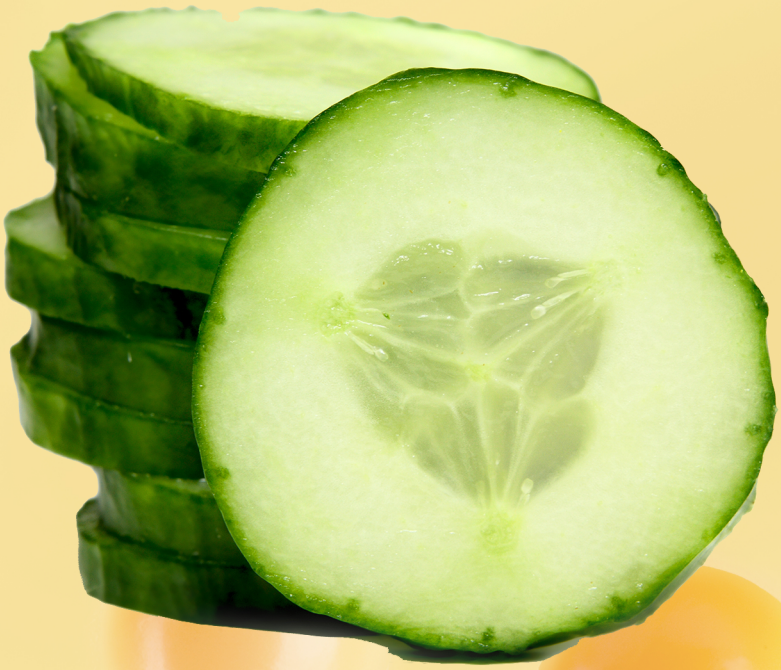
Cucumber & Cheese



A Testers Workshop

By Jeff Morgan

Cucumber & Cheese

A Testers Workshop

Jeff Morgan

This book is for sale at http://leanpub.com/cucumber_and_cheese

This version was published on 2013-08-11

Tweet This Book!

Please help Jeff Morgan by spreading the word about this book on Twitter!

The suggested hashtag for this book is #cucumber_and_cheese.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#cucumber_and_cheese

Contents

Change History

This is a beta book. There will be several updates to the content on the way to a finished book. This section lists the updates.

B9.0 - August 11, 2013

- Spelling fixes
- Updated images for pretty_face report
- Completed Chapter 11

B8.0 - July 20, 2013

- Spelling and grammatical fixes
- Completed the chapter on Building your own Ruby Gem

B7.0 - July 9, 2013

- Fixes numerous spelling and grammatical errors

B6.0 - June 10, 2013

- Completed the section on Surrogate Services which completes chapter 7

B5.0 - June 1, 2013

- Fixed some of the examples
- Completed the section in Chapter 7 on testing Web Services

B4.0 - May 16, 2013

- Updated Chapters 4 and 5 to reflect enhancements in page-object
- Completed Chapter 8 on testing Mobile Applications
- Started the section on Testing Web Services to Chapter 7
- Completed Reporting the results section in Chapter 11
- Minor cleanup on the Ajax examples
- Updated Appendix B

B3.1 - December 1, 2012

- Cleaned up existing sections of Chapter 11
- Added new Continuous Testing section to Chapter 11

B3.0 - November 25, 2012

- Fixed a lot of minor errors
- Added first half of Chapter 7
- Added the section on rake to Chapter 11
- Added the section on using tags to Chapter 11

B2.0 - August 5, 2012

- Numerous small changes to fix typos and grammatical errors
- Added Chapter 10
- Added first section of Chapter 11

NOTE: You will need to update the puppy application for the exercises that I added in this release.

B1.1 - July 28, 2012

- Addresses several spelling and grammatical errors.

B1.0 - July 26, 2012 (Happy Birthday Kim)

- This is the initial public release of the book.

  It contains the first six chapters which walk you through testing simple web applications and incorporating database access to your test.

Acknowledgments

I got the idea to write this book nearly two years ago. The truth is that the book wasn't ready to reveal itself to me back then. Also, I needed more experience and trials in order to get me ready for what you are about to read. Have no doubt, all of the things you will read about and learn in this book came out of my drive to serve my customers while I was helping them to adopt Agile and Lean software development processes. I was constantly pushing to see how far I could take my efforts to simplify the automation approaches we taught and how far I could work with my teams to streamline our development processes.

I have learned a lot from a lot of people along the way. It would be nearly impossible for me to list them all here but I do have a few I'd like to call out directly.

I'd like to thank Dan North, David Chelimsky, and Aslak Hellesoy for their work on Behavior Driven Development and the tools that lead to and drove the development of Cucumber. We all owe you a lot.

I'd like to thank all of the teams I've coached over the past eight years. While I was helping you I received so much in return. When I'd show up with crazy ideas to raise the bar even higher you were there to answer the call. You truly have helped me grow and made me the developer and coach I am today. Thanks.

I'd like to thank all of you that have contributed to some of my Ruby gems. Whether it be reporting defects, requesting new features, writing documentation, or contributing code, your help has been greatly appreciated. For the rest of you, if you have not done so please find a worthy open source project and contribute.

I'd like to thank Patrick Welsh for his help editing this book. I am not a great writer and he has helped me grow in this area. A lot of what you will read here is written the way it is because of him.

Last but certainly not least I'd like to thank my family. They are my inspiration and drive to keep going. Kelley, Kim, Katie, Jared, and Joseph - Thank you for all of your help and love.

Preface

A few years ago I set out to master Acceptance Test Driven Development (ATDD, also variously known as Behavior Driven Development or Storytesting). After trying several of the available tools I settled on Cucumber. For Web applications, I also used the most popular Ruby tool for page traversal and verification, Watir.

As I started working with teams to implement this practice, I discovered that the ATDD community was writing more and more new Ruby gems to empower and simplify this kind of testing. I also found that some of the things I personally wanted to simplify were not supported by any existing Ruby gem, so like the crazed developer I am, I set out to build my own. To my surprise, others started using some of the gems I created and began asking a lot of questions about their usage.

This book is my attempt to share as much of my experience using Cucumber and Watir as possible. I will cover many patterns, practices, tools, and (Yes) Ruby gems that make testing applications (particularly but not limited to Web applications) easier including a few of my own. I will also cover the proper way to structure and write your test automation code so that it is less brittle, simpler, better organized, more expressive, and therefore easier to change over the lifecycle of your application. My goal is to help you see and understand the benefits of ATDD and learn how to use Cucumber and Ruby to adopt (and help us refine!) this amazing practice.

This book is full of hands-on programming exercises and I strongly suggest you do them all. Even if you are not testing applications that are like the ones showcased in this book, the knowledge of how to write sound robust tests will transfer to whatever you have to test.

Who is this book for

This book is for anybody that has to write or wants to learn how to write Functional Automated Tests. In the past this has typically been left to individuals who have the role of Developer. This is quickly changing and people who have traditionally had the role of Quality Assurance are discovering it is necessary to produce test automation.

Today's tools make it much easier to add automation as part of our development process. This book will show you how. But we have learned that it is also possible to take these test automation efforts further and write the tests before or during application development. This has the effect of finding defects in requirements and code much sooner and thereby reducing the costs associated with fixing them. This book will show you how you can do this as well.

How this book is organized

This book starts off with a description of the state of traditional testing and what can be done about the multitude of problems testers typically face in Chapter one. Chapter two walks you through the process of installing and configuring all of the software you will need for the workshops and examples that follow.

The remainder of the book is a series of workshops and examples that take you through the process of writing tests against several example applications. I strongly suggest you follow them sequentially as each chapter builds upon the work of the previous chapter.

Getting help

If you wish to make a comment about this book, need help with some of the examples, or wish to report some problem with the book please do so on the Cucumber & Cheese forum[1].

[1] https://groups.google.com/forum/?fromgroups#!forum/cucumber-and-cheese

1. Let's get it started

Cucumber is a testing tool that facilitates writing the requirements for a system in plain English and then automating those requirements turning them into functional tests. Having the ability to read, understand, and collaborate during the creation of the functional tests for a system has the potential for profound changes and improvements in the way we work and the quality of the software we deliver.

To understand how profound this potential is, I would like to start off by describing the challenges that exist with the way we typically test software in today's world. Some of what I am describing in the next few sections is taken from an Agile point of view. If you are not fortunate enough to be working in this manner I think it should be easy to adapt these ideas to your own circumstances to see if they apply.

What's wrong with the way we test software?

The software testing world is in a state of transition. The way we have historically tested software has proven to be inefficient and ineffective. It is inefficient because many of the practices we follow do not scale as the software we are working on becomes larger and more complex and also because there is a large amount of waste[2] in our processes and workflows due to rework caused by late defect detection. It is ineffective because, despite our best efforts software development teams regularly deliver software that is of low quality and contains many defects.

About eleven years ago a group of individuals got together and created the Manifesto for Agile Software Development[3]. They had spent the proceeding years testing new ideas on how teams can work together to produce higher quality software in a more predictable fashion. In the early days of the Agile movement[4] much of the writing was focused either on software developers or on roles that are more management focused. There was very little written about how somebody that comes from a more traditional testing background should participate in the development process.

Since there was so little written about the role of software testers, many teams working in an Agile fashion assumed there was no need for a tester on the team. Others brought along their testers but had them continue to work using their traditional approach to testing software. Both of these approaches led to a lot of problems that we will discuss in this chapter.

Teams that worked in a more traditional methodology like waterfall[5], RUP[6], or a more generic

[2] http://en.wikipedia.org/wiki/Lean_manufacturing#Types_of_waste

[3] http://agilemanifesto.org/

[4] http://en.wikipedia.org/wiki/Agile_software_development

[5] http://en.wikipedia.org/wiki/Waterfall_model

[6] http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process

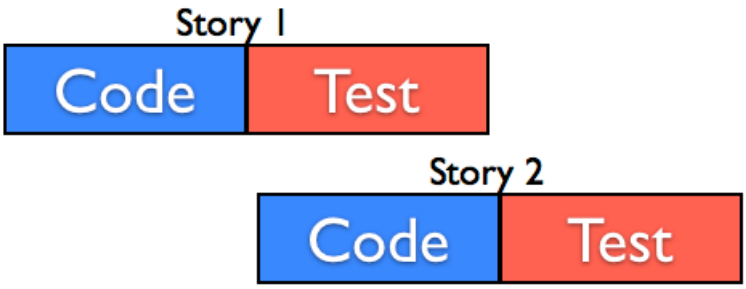
iterative[7] approach to building software typically waited until the development phase was completed before the software testing phase began. This approach lead to a large amount of rework in the form of defect fixing and also caused our delivery schedules to be very unpredictable.

The following sections are a list of many of the common problems we have seen when working with teams that follow a fairly traditional approach to testing software.

A Game of Ping Pong

At the end of most software development efforts there is a handoff from developers to testers. The developers have completed some functionality and the testers need to check that functionality for correctness. In non-agile development efforts the developers may have been coding for weeks or even months prior to this handoff. With agile teams this handoff takes place at the end of each story which is typically a day or two of coding. This seems logical enough so what could possibly be wrong with these pictures?

Waterfall Testing



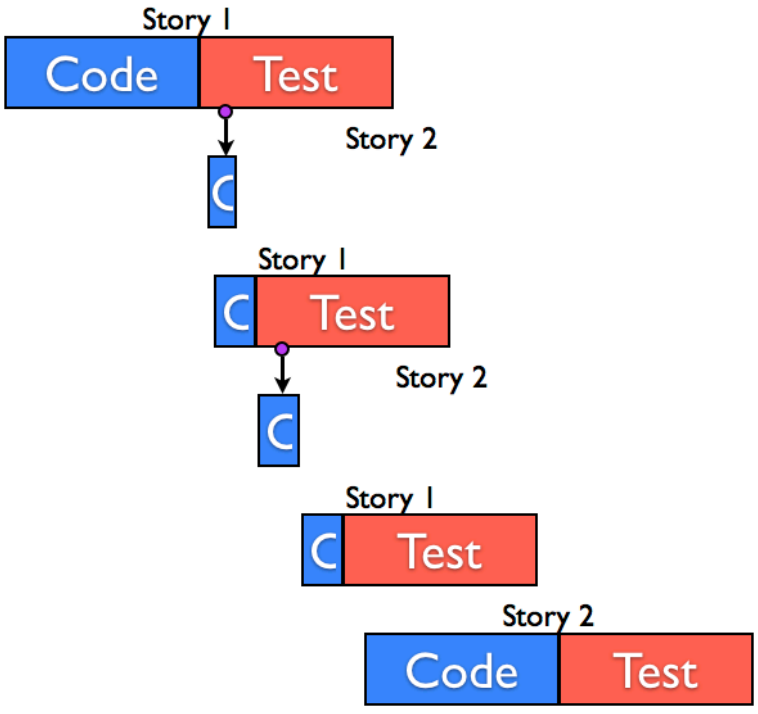
Agile Testing

The problem arises when the tester find defects in the system. When a defect is found a lot of unexpected activities begin.

1. First of all the tester will typically have to log the defect in their tracking system with enough details so it can be reproduced by a developer at a later time. Enough detail often means a lot of detail including the exact data used when the defect occurred.
2. Often, they must stop working on the test plan in which they found the defect or focus on some other area of the application. It often does not make sense to continue a workflow in which the defect occurred. There is no way to be sure that the defect does not affect later behavior. Also, we know that the code is going to be touched (when the fix is applied) so we will have to run the entire test case again later.

[7] http://en.wikipedia.org/wiki/Iterative_and_incremental_development

3. Many organizations have so many defects that they need to have somebody prioritize the defects and determine when it will fit into the busy schedule. Often low priority defects are defered until some later date. When defects are defered the waste is even more profound. The developers have a much harder time understanding the context of the code to be fixed. Also, low quality (due to a high number of defects) causes the entire team to move more slowly in many of their development activities.
4. Eventually the defect makes it into the work stream and is picked up by a developer. The developer needs to stop what they were working on in order to address the defect. Hopefully they can reproduce the defect but often this will take some time and may also require the help of others. If the defect had specific data requirements (as noted in the report) the developer might need to copy some data from the testing database to their development database. If a lot of time has passed the data in the testing database might not even exist any longer and the task of reproducing the defect becomes even more complicated.
5. Once the defect is reproduced the developer begins the process of determining the resolution. If the code he needs to update is not very clean (as is the case for most code) he will have to move very slowly and carefully to ensure he does not introduce a new defect while fixing the current one.
6. Eventually the defect is fixed. The next step is to check in the updates and then deliver a new build of the software for the testers so they can begin the whole process all over again.



What happens when we have defects

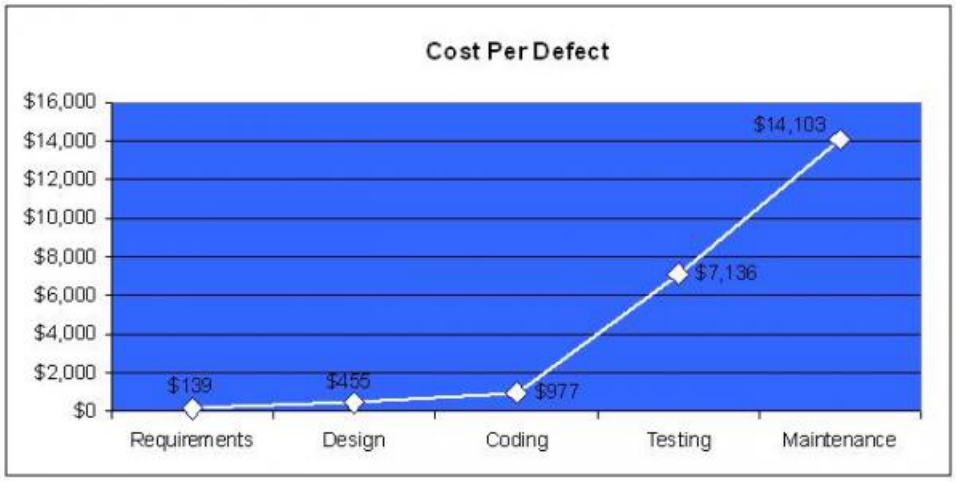
On many projects this back and forth between testers and developers consumes a significant portion of the overall project timeline or in many cases causes the project to be delivered late. The specific

amount of time it consumes is also very difficult to predict and manage. If you look at the overall development workflow from a lean[8] perspective, every time we have a defect we have work moving backwards in the value stream[9]. This is waste and we should try to find a way to eliminate it.

The true cost of a defect

We know defects cause a lot of rework but is it possible to know the real cost of a defect? A lot of folks have spent a considerable amount of time studying this very topic. Here is a little of what they found.

- Capers Jones, in his book[10] Software Assessments, Benchmarks, and Best Practices[11], stated that the cost to fix a defect is only $977 if it is found and resolved during development but the cost goes up to $7,136 if the defect is found and resolved during a testing phase that follows development.



Software Assessments, Benchmarks, and Best Practices

- Barry Boehm has published several studies over nearly three decades that demonstrate how "the cost of removing a software defect grows exponentially for each downstream phase of the development lifecycle in which it remains undiscovered[12]".
- Numerous studies[13] have confirmed Boehm's findings.

[8] http://en.wikipedia.org/wiki/Lean_manufacturing

[9] http://en.wikipedia.org/wiki/Value_stream_mapping

[10] http://www.amazon.com/Assessments-Benchmarks-Addison-Wesley-Information-Technology/dp/0201485427?ie=UTF8&s=books&qid=1209056706&sr=1-1

[11] CapersJones, SoftwareAssessments, Benchmarks, andBestPractices-Addison-Wesley, 2000

[12] BarryBoehm, SoftwareEngineeringEconomics-Prentice-Hall, 1981

[13] RobertGrady, ApplicationsofSoftwareMeasurementConference, 1999

- A major research project conducted by the United States Department of Commerce, National Institute of Standards and Technology showed that in a typical software development project "fully 80% of software development dollars are spent correcting software defects[14]".

The list could go on and on but I think you are getting the point here. Spending time fixing defects cost us money - a lot of money. But are there other costs associated with defects? Here are a few possible costs that come to mind. These items are hard to place a monetary value on but they do cost the team and company.

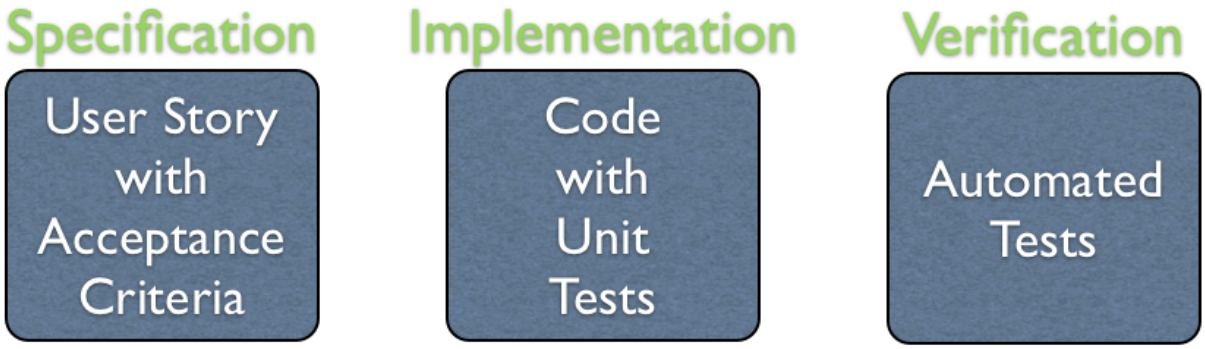
- Unhappy Customers. If our customers are unhappy because of low quality they might decide to use one of our competators products. They also might decide that they do not want to take an update of our software because they fear it will contain defects and they do not want to disrupt their business. This, in turn, causes us to support many versions of our software. This is ture cost and has an effect on the bottom line.
- Pressure placed on the team. Often management thinks that the way to increase the quality or drive down the number of defects in a system is to make the team work harder and often longer. They ask or sometimes mandate that the team work weekends or stay late. The problem is that this pressure on the team often has the reverse effect - causing more defects due to stress or lack of sleep.
- Demoralized and demotivated team. It is not fun working on a low quality software development project. People like to take pride in their work and it is hard to do this when the software you are producing has a lot of defects. Over time, teams with continuous low quality have the potential to suffer the loss of valuable employees and team members.

Differences and Duplication

At a high level there are three major deliverables that come out of software development initiatives. They each map directly to one of the traditional phases of traditional development methodologies. They are:

- Analysts and Product Owners work to define the requirements of the system to be constructed. If the team is an Agile team the requirements are broken up into much smaller deliverables called stories or features. These stories should be accompanied with acceptance criteria.
- Developers take the requirements provided and construct the code that is compiled into the requested software.
- Testers take the requirements and build test cases and test plans. From that they build automated test suites that validate the requirements are implemented properly in the code.

[14] National Institute of Standards & Technology, US Dept of Commerce, The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002



Agile Deliverables

The testers work hard to make sure they have a test to verify each requirement. For example, if there is a requirement that the last name field on a web page is required then the tester will create a test that attempts to submit the form leaving that field blank and verifies that the data is not saved and the appropriate error message is displayed. There really is a very close relationship between the requirements and the tests. There should be at least one test for every requirement. One might say that they are simply expressing the same thing in entirely different ways.

The challenge is that requirements are rarely defined one time and never change. On teams that are truly trying to deliver what their customers want, the requirements will change before, during, and after development. It is not uncommon for a discussion between a product owner and a developer to cause a ever so slight change to the way something is implemented. Often, these discussions do not include the tester.

If they are extremely lucky the test cases match the requirements exactly. This is usually not the case. Frequently, testers are much more thorough than product owners and business analysts. More often than not the testers think of edge cases that were not identified in the written or communicated requirements. Since these edge cases were not communicated to developers they usually were not handled by the application. As a result the test fails and the defect goes to development. Often the developers complain that this was not in the requirements and the unhealthy finger pointing begins. The developers get angry because the testers are testing things that were never in the specification that was delivered to them. The testers get angry because they can't understand how it is possible for a developer to deliver code that takes them only a few minutes to find a defect.

If everything works well the requirements and the test plans should contain the same information expressed in different formats and from different views. Since we have two separate groups creating and maintaining the same information in documentation of different forms it is enevitable that they will get out of sync. What if we were able to combine these into one document that could be used as both requirements and test documentation? Could we eliminate both the differences and duplication at the same time?

Testing Phase is Often Cut Short

Has anyone ever worked on a software project that was behind schedule? As long as we continue to make plans and set delivery dates based on pure speculation and guesswork projects that are behind schedule will be a fact of life.

When a software development effort is approaching the end of the development phase and we discover that the team is behind schedule we have a few options to address the problem. The first is to reduce scope so the software can still be delivered on schedule. In a waterfall project this is typically not practical because the way we scheduled the project had nothing to do with leaving lower priority items toward the end so we could possibly remove them from scope. From my experience this option is rarely taken by teams working in a traditional development approach. The second option we have is to ask the team to work overtime and/or shorten the time we had planned to test the software. The testers are told, "Don't worry. The quality of the software is very high and you should not need all of that time to test the software". More often than not, the second approach is taken and the team still misses the delivery date.

When working on an agile team you strive to deliver working tested software in short iterations; usually a week or two. As the team approaches the end of the iteration or sprint you often see developers putting in an extra effort to complete stories. What can a tester do if a story is delivered on the last day of the iteration? What about stories being delivered within a few minutes of the iterations end? The team will not get credit for those stories unless they are fully tested.

Manual Test Scripts

In my profession I spend a lot of time traveling. There are times when I find myself completing my work week on a Thursday or Friday only to travel across the country to spend the weekend with my family. There are several ways I could make that trip. I could begin walking and I would eventually arrive at my destination. I could ride a horse and buggy and would most likely arrive sooner. I could climb into the car and drive there as well. I could also catch a flight and get there at around 600 mph, for most of the trip. All of these modes of transportation would more than likely end in the same result but one of these approaches is superior.

Manual functional testing is a lot like this. Following a script entering data into a screen, clicking buttons and verifying output is not fun. Much of this manual work could be automated saving a lot of manpower and time. It is very repetitive and mind numbing. And it does not scale as the application grows larger and more complex. When we talk about manual regression testing we are trying to perform (and pay for) the exact same manual task again and again. Over time that becomes impossible so we end up making decisions about what not to test. Inevitabily, a defect will show up in one of these areas we chose not to test.

This is precisely the type of work that should be delegated to machines. Repetative tasks that must be executed exactly the same time again and again are very well suited for automation. Automating these tests would also allow a team to perform more testing in a shorter amount of time and thereby shorten the feedback loops. This could free up testers to focus on the types of testing that have

a higher return on investment - structured exploratory testing just to name one. So why is most software testing still manual? I have encountered three reasons.

1. The first reason is that management often believes that the testers are not capable of learning the skills necessary to implement a successful test automation initiative. As a result they often are not willing to make the type of investment necessary to develop those skills.
2. The second reason is that testers are often afraid of automation. They mistakenly believe that they cannot learn a programming language or they may have been involved in an effort that failed for one reason or another. Sometimes they might be afraid of change or afraid that automation might lead to the elimination of their jobs.
3. The third reason that most software testing is still manual is that a fairly high percentage of test automation efforts eventually fail and the team reverts back to manual testing. I do not have hard numbers for this but I frequently ask attendees at conferences in which I speak and they tell me about these type of experiences again and again. The next section goes into this in more depth.

Automation Efforts Fail

I frequently talk to testers who provide details about automation efforts they have been involved with that fail. The usual outcome is that the team goes back to testing the software manually. If automation is such a good thing then why do these efforts fail?

I believe tool manufacturers are partly to blame for these test automation failures. They sell us tools that promise to solve all of our problems. Just turn on the recorder, interact with the software under test, and when you are finished you will end up with the perfect tests. These tools do make it easy for testers to record scripts, run them and collect the results. But all of the record/playback type of tools I have seen suffer from the same problem. They tightly couple the activities of the tests with the software being tested. As the test suite grows and the application changes the team needs to spend an ever growing amount of their time keeping the existing scripts running. This ever present maintenance will occupy a larger and larger amount of their time and eventually cause the team to abandon the effort.

I have also seen much automation code written that has the same problem as the record playback approach. The tests directly access the system under test and the code doing that is duplicated in every script. Changes to the application ripple through the code causing the tester to make changes in many places. This form of automation is not sustainable. It's a classic False Economy[15].

Yet another reason many efforts fail is that the teams do not pay attention to or even have a strategy for test data management. Tests become coupled to data and coupled to each other and soon many of the tests fail for unexpected reasons. The team must troubleshoot each test failure to determine if it truly is a defect or if somehow the data has yet again been changed underneath the test.

[15] http://en.wikipedia.org/wiki/False_economy

Teams need a Hardening Phase

When using an Agile methodology, software development teams try to deliver working, tested software every week or two. This software is supposed to be of such quality that the software could be released to production if the Product Owner decides to do so.

Many teams that are developing using an Agile methodology are not able to deliver software that is production ready at the end of their timebox. As a result, when they do decide to go to production they often plan to have a multi-week "hardening" phase. This is a time in which the software is thoroughly tested and defects are addressed. Ask yourself, what would it take to have this level of testing and quality during the original development effort?

What's the Solution?

The picture looks pretty bleak. There seem to be a lot of problems everywhere. The good news is that a lot of very smart people have been thinking about these problems for the past few years and have come up with many solutions. One of those solutions that I would like to explore here is called Acceptance Test Driven Development[16].

Acceptance Test Driven Development

What if we could combine the software specifications with the functional tests into an easy to read artifact? What if we were able to take this artifact and automate it prior to building the software? What if the developers had the ability to run these automated specifications against the software while they were writing the code? What would this do to our process? Let's explore the idea and see how Cucumber fits with this vision.

Acceptance Tests Before Code

In this new vision I could see a product owner, tester, and developer collaborating to build up a single specification for a small software deliverable just prior to the development effort. The product owner knows what they want to get out of the system. The tester brings their knowledge of how the system works and their ability to think of the boundary conditions. The developer brings the knowledge of what's possible in the code. Together they would capture this specification in a form that could easily be automated to validate the correct software was delivered. What would this do for us?

First of all we introduce the tester much earlier in the project than we usually do. We leverage their ability to think of things to break the system and this becomes an asset to the developer instead of

[16] http://testobsessed.com/blog/2008/12/08/acceptance-test-driven-development-atdd-an-overview/

a curse. We also get the tester and the developer collaborating on a story from the beginning. The trio work to gain a shared and complete understanding of what they are about to build and what it must do in order to be considered complete. (Yes, this helps us define "done".)

But the collaboration does not end there. The next step is to automate the specification and write the production code. The developer and tester continue their collaboration throughout this phase. Our experience has convinced us that the automation can typically be put in place in a fraction of the time it takes to write the producton code. And yet the tester automating the specification needs critical information that only the developer can provide. This information is related to how the tests will interact with the system under test. For example, if we are building a web application the tests will need to use the element "ids" in order to perform the interaction. Through discussion the developer learns how critical it is to add these "ids" and they agree on the names. This helps us build software that is easier to test.

There are other discussions going on during this development. The developer and tester are continuously discussing the requirements to ensure they are both on the same page. Also, as the tester completes the automation of some of the Acceptance Tests they inform the developer so they can begin running them against the system as they develop. As the developer completes some portions of the system they inform the tester so they can begin some basic Exploratory Testing[17] on the application. Yes, we do promote exploratory testing on software that is not complete. The goal here is to find any issue or defect with the code as close to the time it was introduced as possible. The overarching goal is to collaborate to prevent any defects that might find their way into the software and deliver a very high quality product.

How does this help?

The defect that would have made it through development only to be discovered during a later testing session could now be found while the developer is still writing code. We can stop playing ping pong.

Next, it would be much harder to cut the testing phase short since a significant portion of it occurs prior to and during development. In fact, we could go a few steps further and state that the developer is not finished with the development effort until all of the automated tests pass. This would change the role of the developer and tester. No longer do we have the developer writing the code and the tester trying to find defects. Instead, we have the developer and tester working together to prevent defects.

We would no longer have to worry about inconsistencies between different peoples interpretation of the "real requirements". Instead, the product owner will need to read and understand the document the team will work from. The product owner must ultimately state that when all of the tests pass the story is functionally complete. Everybody has a chance to read and have input in the final spec. It is a collaborative effort in which the team defines and then delivers a known piece of functionality.

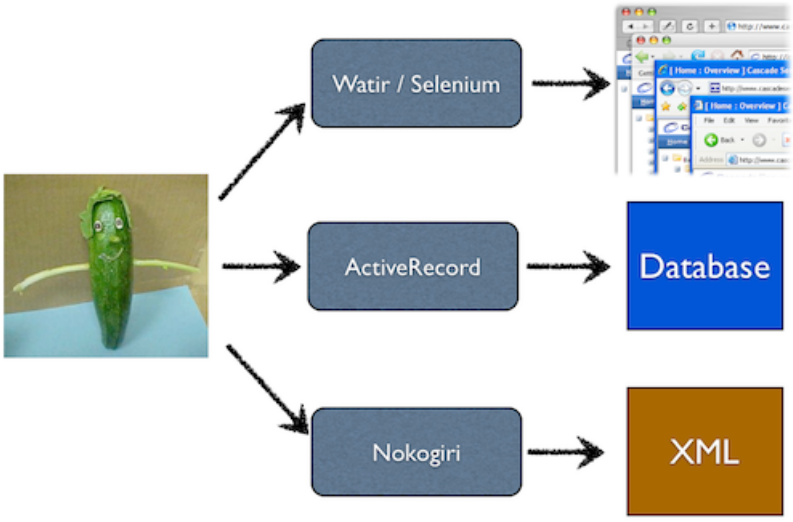
The specification tests could be run continuously to provide a regression around the application. Everything seems great. The only problem we haven't address is the brittle code causing automation failures. In the chapters that follow you will learn how to address this problem.

[17] http://en.wikipedia.org/wiki/Exploratory_testing

What is Cucumber?

Cucumber[18] is a tool that reads specifications written in a plain english format named Gherkin[19] and provides a bridge to code that implements that specification. It is a tool that can be used to implement the automation portion of Acceptance Test Driven Development.

It is also important to understand what Cucumber cannot do. The picture below has a couple of examples of things we might need to interact with when testing an application. We have an example of a web browser, a database, and an xml file. The truth is that Cucumber has no idea how to interact with any of these things. Cucumber uses these third-party library called *gems* in order to communicate and/or interact those things. We'll be learning a lot more about gems throughout the remainder of this book.



Cucumber using gems for interaction

Cucumber essentially fills the role of bridging the plain text specification with the coding layer that makes the tests run. Our tests will in turn use other third-party libraries to interact with the system we are testing.

How do we do ATDD?

For the past eight years I have worked with numerous teams helping them adopt Agile practices. Over the past three years I have used ATDD and Cucumber with nearly every team. Also, my fellow agile coaches at LeanDog[20] have been doing the same. As we compare notes on things that have worked and things that have not turned out so well we have noticed several common themes. Here they are:

[18] http://cukes.info/
[19] https://github.com/cucumber/cucumber/wiki/Gherkin
[20] http://www.leandog.com

- Collaboration is the primary goal
- The words matter
- Automation is necessary (and easy)

This is probably not the list you were expecting. You probably thought this list would detail out some specifics related to automation. We started off thinking this was all about automation and as a result we made a lot of mistakes along the way. Although automation is a by-product of the ATDD, there are other outcomes that are far more important.

Collaboration is the primary goal

The single most important benefit we have seen from ATDD is the collaboration that happens as a result of following this practice. All teams say they collaborate well, but in order to do ATDD the teams have to take collaboration to a whole new level. In order to see how this collaboration can work let's talk about a workflow.

Workflow

I have seen several different ATDD workflows be successful. What I am going to describe here is what I believe to be the best workflow. This is not the only one that will work and often you might need to make adjustments due to constraints within your team.

1. Sometime shortly before the beginning of an iteration or sprint, the product owner writes the initial specification and builds out the examples for a story[21] using Gherkin. The product owner might get the tester or business analysts involved during this phase to ensure the specification is thorough and as complete as possible.
2. When a story is moved from the "waiting for somebody to work on me" swimlane to the "let's get busy" swimlane on your story card wall the product owner, tester, and developer that plan to work on the story have a very brief informal meeting. The purpose of this meeting is for the three attendees to read through the document the product owner has created and complete the specification. The product owner needs the others in the meeting to ensure the specification is doable and complete.

   The outcome of this meeting should be agreement with the product owner that once all of the examples work as specified the story is functionally complete.
3. The tester begins automating the examples and the developer begins work on the application. There is the need for a lot of collaboration during this period. The tester will need to know how best to interact with the application and that can be answered easily by the developer. Also, the developer will need to know what he can do to make the application as testable as possible. The tester can help him here. There might be other conversations about what should

[21] http://en.wikipedia.org/wiki/User_story

be a unit vs. a functional test, etc. If there is a need for clarification of some element of the specification, the product owner should be available to answer questions.

The tester will typically finish in a fraction of the time it takes the developer to complete their coding effort. Once the acceptance tests are automated the developer should begin regularily running the tests against the application to see how he is progressing and to ensure he doesn't miss anything. The developers goal is to make all acceptance tests pass. Also, the tester will begin performing exploratory testing against the application as soon as the first pieces are complete. The goal of the developer and tester during this collaboration phase is to prevent defects and deliver a high quality test suite and product.

4. Once all of the tests pass and the tester is complete with the exploratory testing the product owner can take a look at the finished product. Also, the automated acceptance tests should be added to the test suite execution so they run on a regular basis.

The words matter

Getting the words right in the specification and examples is critical to making them valuable. It is important that our documentation not be scripty or include unnecessary details. Instead, it should be concise, clear, and in the language of the business[22]. A lot has been written about this topic and I will mention it through the remainder of this book.

Automation is necessary (and easy)

Finally we are getting to the focus of this book. Getting the automation right is an essential element for this practice to succeed. Our test code must be clean, well factored, and easy to maintain. If they are not, your automation effort has a high likely of failure, or at a minimum, the cost to maintain your tests will be high. This is the topic of this book.

[22] http://dannorth.net/2011/01/31/whose-domain-is-it-anyway/

2. Getting ready to go

In this chapter we will help you install the software and provide some background on some of the tools you will need to perform the exercises in the first few chapters. In later chapters we will walk you through the installation of additional, more specialized software for testing mobile applications and other targeted platforms.

How the tools are installed and configured depends on what operating system you are using. In this chapter we will cover the detailed steps necessary to install and validate all tools on the Windows, Mac OS X and Linux platforms.

On the Windows platform

The first software we need to install is called Ruby[23]. It is a programming language[24] that can be used when writing cucumber acceptance tests.

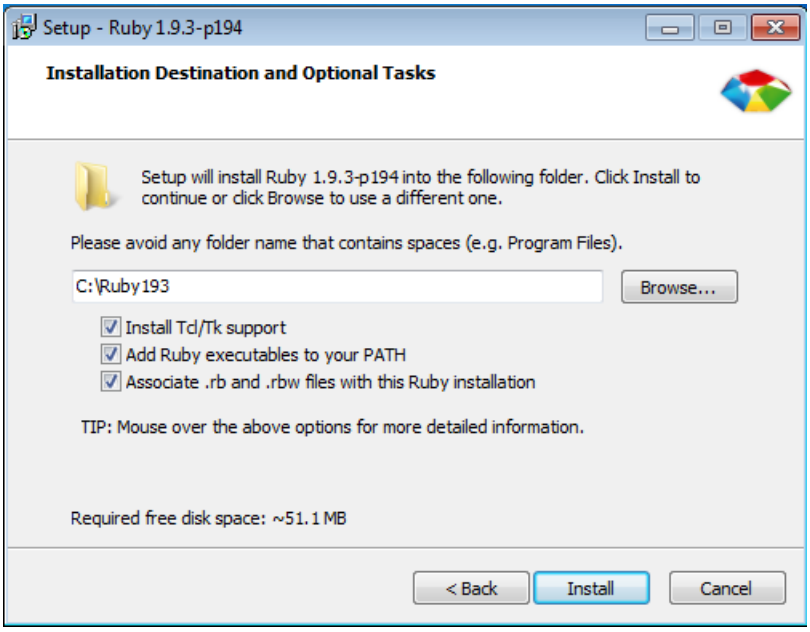
Installing Ruby

In this books all examples work with Ruby 1.9.3. If you are installing Ruby for the first time I suggest you get this version. If you already have Ruby installed and it is an older version, I recommend you upgrade.

The Windows platform has an installer that you can download from http://rubyinstaller.org/downloads/[25]. Just click the link for the latest version and download it to your local computer. While you're at it, download the latest RubyInstaller Development Kit[26]. You'll need it shortly.

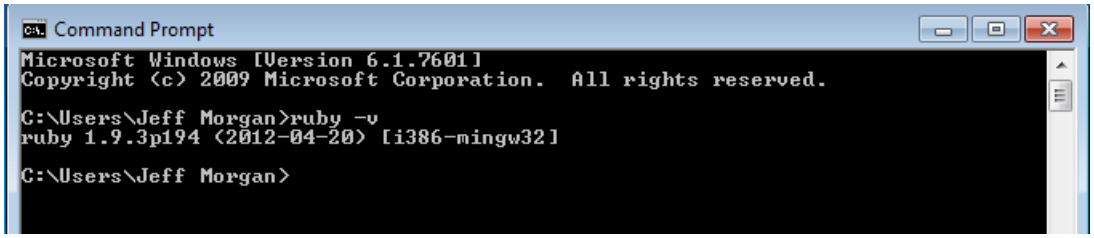
Run the Ruby installer. Please ensure you add the Ruby executable to your path and associate the files appropriately by selecting the two checkboxes as indicated below.

[23] http://www.ruby-lang.org/en/
[24] http://en.wikipedia.org/wiki/Programming_language
[25] http://rubyinstaller.org/downloads/
[26] http://rubyinstaller.org/add-ons/devkit/



Ruby Installation Options

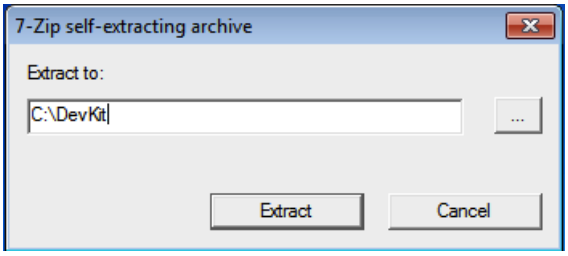
For the remainder of the installation just accept default settings. When the installation completes please open a *command prompt* and type `ruby -v`. You should see output that looks very similar to this:



Verify Installation

If you do not see this sort of output you will need to manually add Ruby to your path.

The next step is to install the Development Kit. You should have downloaded it at the same time you downloaded the Ruby installer. If you run the downloaded file it will prompt you for a directory to *Extract to:*. Simply enter *C:\DevKit* in the dialog and click *OK*.



DevKit Installation

After the extration completes return to the *command prompt* we opened earlier and change to the *C:\DevKit* directory. When you are there you should execute the following two commands:

```
ruby dk.rb init
ruby dk.rb install
```

Ruby is now installed on your computer and ready for the next step.

On the Mac OS X platform

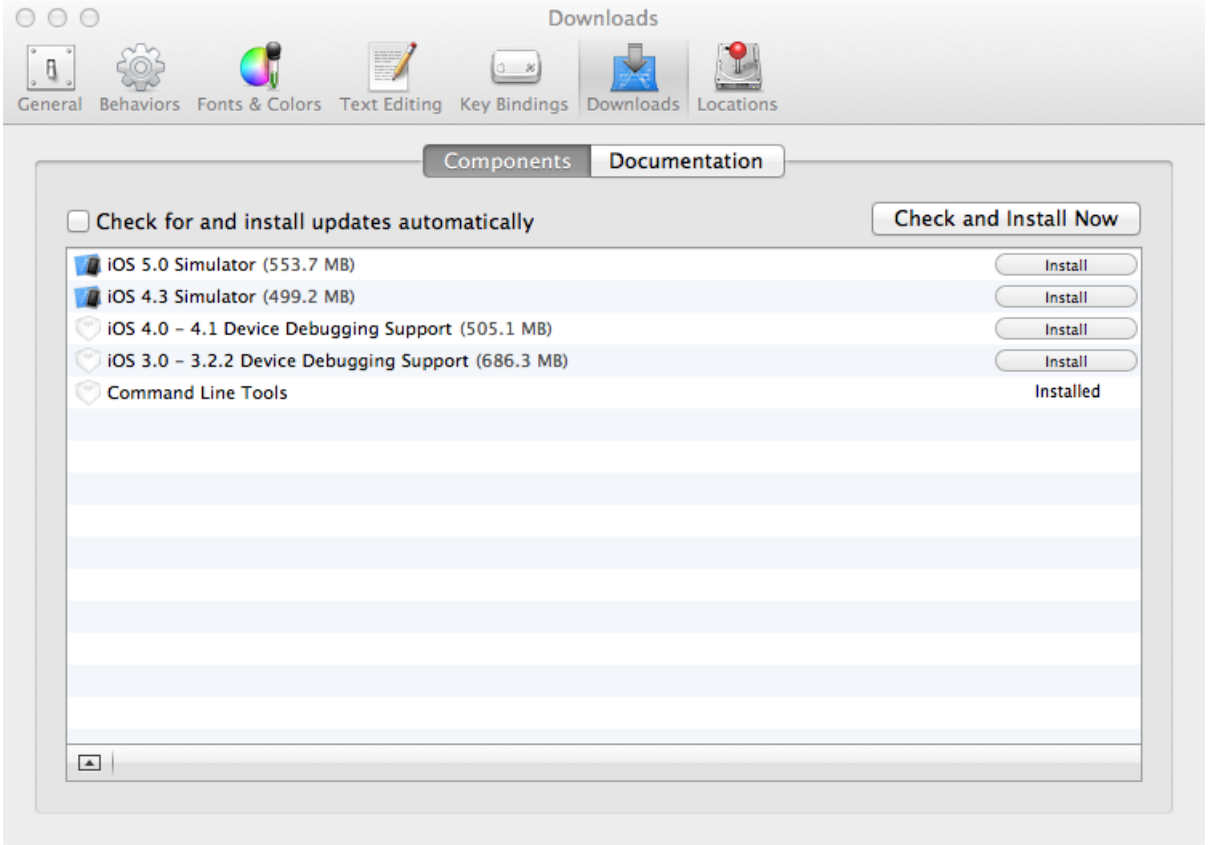
Although OS X comes with Ruby pre-installed it is a very old version. In the following sections we will update Ruby and install all of the additional software and tools we need at this time.

Xcode

Before we can install Ruby we need to install a compiler and a few command-line tools. The first tool we will install is called Xcode[27]. It is the Apple development environment for building Mac and iPhone/iPad applications. The installation contains the compiler and other build tools we will need in order to compile/install Ruby and some associated software. Be warned, the download from the AppStore is very large and once the download is completed the software is not installed on your system. The download simply installs an Install Xcode application that you will have to run to install the actual Xcode tool.

After you complete the installation of Xcode you will need to use it to install the command-line tools that are an option part of Xcode. Go ahead and start Xcode and then select the *Preferences* menu option from the Xcode menu. In the dialog that is displayed select the *Downloads* icon at the top and make sure the *Components* section is selected. From there you can chose to install the command-line tools.

[27] https://developer.apple.com/xcode/



Command-line Tools Installation

rvm

Rvm[28] is the Ruby Version Manager. It helps us install and manage multiple versions of Ruby and multiple sets of gems. In order to install rvm you can simply paste the following line into a terminal window and execute it:

```
1 curl -L https://get.rvm.io | bash -s stable
```

Once the installation of rvm is complete please exit out of your terminal session and start a new one. To ensure rvm was installed properly you can simply type `rvm list known` in the terminal. It will list all of the Ruby versions that can be installed. Finally, we will use rvm to install two packages that will be needed during the installation of Ruby. Please execute the following two commands in a terminal window:

[28] https://rvm.io/

```
rvm pkg install openssl
rvm pkg install iconv
```

We are finally ready to install Ruby.

Installing Ruby

We will be using rvm to install Ruby. Simply paste the following line into a terminal window and execute it:

```
CC=/usr/bin/gcc-4.2 rvm install 1.9.3 --with-iconv-dir=$rvm_path/usr --with-opens\
sh-dir=$rvm_path/usr
```

The process will take a while but when it is finished Ruby will be installed. The following command will select the version of Ruby you just installed and make it your default Ruby.

```
rvm use 1.9.3 --default
```

On the Linux platform

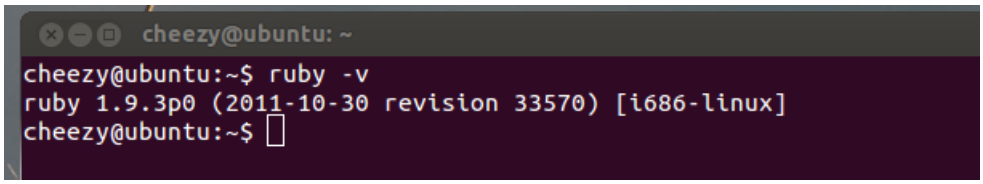
Linux is a very popular platform for Ruby development. The following steps will install Ruby on Ubuntu 12.04.

Installing Ruby

The Ruby 1.9.3 packages are labeled ruby-1.9.1 because that is their ABI version. In order to install Ruby 1.9.3 and make it active you simply need to execute the following commands on the terminal.

```
sudo apt-get update
sudo apt-get install ruby1.9.1 ruby1.9.1-dev rubygems1.9.1 irb1.9.1 ri1.9.1 rdoc1\
.9.1 build-essential libopenssl-ruby1.9.1 libssl-dev zlib1g-dev
```

When these commands are finished you should be able to check the version and see something like the following:



ruby -v on Linux

Ruby 1.9.3 should now be installed on your Linux computer.

For all Operating Systems

The following steps should be executed on all platforms.

Installing Ruby Gems

There are a few Ruby gems that we wish to install at this time. In a *command prompt* or *terminal* please enter the following:

```
1 gem install rake bundler yard watir-webdriver
```

If you are using Linux you will have to place `sudo` at the front of the preceeding line. This will install three gems that will be used as we process through the following chapters. As we introduce new gems throughout our exercises you will be prompted to install them.

Browser tools

Throughout the book we will be using the Firefox[29] browser. If you do not have it installed please do so now. Once the browser is installed we will need to install an additional plugin. Start Firefox and select the *Tools* ⇒ *Add ons* menu option. This will bring up a page on which you can perform a search. Enter *Firebug* in the search field and press enter. After the search completes click the *Install* button to install the *Firebug* plugin. After the installation completes you will need to restart *Firefox*.

If you wish to run some of the web application tests using Chrome[30] you will need to install the ChromeDriver[31] for your platform. Simply download the executable and place it somewhere in your path.

If you plan to use Internet Explorer please help yourself out and install a recent edition of the browser. Also you will need to install the IEDriverServer[32] by simply downloading the executable and placing it in your path. Finally, please install the Internet Explorer Developer Toolbar[33] if your version of the browser does not already have this plugin built in.

Editor or IDE?

If you already have a prefered editor or development environment you might see if it has support for developing Ruby. Many in the Ruby world use powerful editors like Emacs[34] or Vim[35]. The learning

[29] http://www.mozilla.org/en-US/firefox/new/
[30] https://www.google.com/intl/en/chrome/
[31] http://code.google.com/p/chromedriver/downloads/list
[32] http://code.google.com/p/selenium/downloads/list
[33] http://www.microsoft.com/en-us/download/details.aspx?id=18359
[34] http://www.gnu.org/software/emacs/
[35] http://www.vim.org/

curve on these tools are very high but the tools are also very powerful. If you are brand new to Ruby then probably now is not the time to tackle these tools and I would recommend you download and use RubyMine[36]. It will be referenced throughout the book and many of the screenshots are taken from this tool. Just accept all of the default options during the installation. After the installation completes go ahead and run the tool. Just select the option to evaluate the tool for 30 days in the dialog that is displayed the first time you run the software.

[36] http://www.jetbrains.com/ruby/

3. Dipping our toes in the Watir

Our first script

As we saw in the first chapter, *Cucumber*[37] uses other Ruby gems to interact with the application we wish to test. One way to control a browser and thereby test a web application is to use a Ruby gem named *watir-webdriver*[38]. This chapter will introduce us to *watir-webdriver* and provide enough experience with this gem to allow us to test basic web applications.

Before we start writing scripts we need to create a directory where we will keep them. If you are using *RubyMine*[39] begin by creating a new project. You do this by selecting *New Project* from the *File* menu. Name the project `learn_watir`. If you are not using RubyMine just create a directory named `learn_watir`.

Our first example script will be very simple. It will test that we can open a browser and go to the Apple website. Start by creating a new file named `first_script.rb` in the `learn_watir` directory.

Ruby

It is a Ruby standard to create files using all lower case letters with an underscore separating the words.

If you are using *RubyMine* you can right-click on the `learn_watir` project name and select *New - File* and name the file `first_script.rb`. Once you have created the script file you may add the following lines to it:

```
1 require 'rubygems'
2 require 'watir-webdriver'
3 browser = Watir::Browser.new :firefox
4 browser.goto 'http://www.apple.com'
```

The first two lines of the script inform Ruby of the libraries that need to be loaded in order to run the script. The first line loads the rubygems library which provides the ability to load gems. The second line specifically loads the watir-webdriver gem.

[37] http://cukes.info
[38] https://github.com/watir/watir-webdriver
[39] http://www.jetbrains.com/ruby/

Ruby

In Ruby, gems[40] are nothing more than a third party library that you can use. The Ruby gems system is actually a lot more than that. It is a package manager for gems that understands dependencies. When you install a gem using the `gem install` command, it connects to a server on the Internet and requests information about that gem. The server informs the tool about all of the dependencies necessary in order to install the gem and make it usable. The next step is for the Ruby gem tool to install all of the dependencies that are not already installed on the local computer. The end result is that the gem and all of its dependencies are installed and it is ready to use.

The third line is where things get interesting. On this line we are creating a new *Watir* `Browser` object for Firefox[41] and give it the name `browser`. This is a common pattern we will see throughout our scripts. When we create the new `browser` object it opens the actual browser application on our computer, and assigns it to the Watir Browser object, so that we can have control of that browser application.

Namespaces

You may have noticed that on line three of the script we are creating our `browser` object by calling `Watir::Browser.new`. Just what exactly is the `Watir::`? That is what we call a namespace. It is so we keep our classes organized and do not have name collisions. For example, there might be other *gems* that want to have a class named `Browser`. How can we tell them apart? Namespaces help us with this. When we make a call to `Watir::Browser.new` what we are saying is "look inside the Watir namespace for a class named Browser and make a new one for me".

The final line of the script is where we inform the actual browser, through our Browser object, to surf to the Apple website. The way we do this is by sending a message to the `browser` object. That message is `goto`. As you might imagine, the remainder of the line is passed along with the message informing the browser which URL to go to.

This script will use the Firefox browser. If you want to see this work with another browser, just move on to the next script and you will get your satisfaction. If you are using *RubyMine* you can right click on the file and select the Run "first_script" option near the bottom of the menu. If you are using the command-line then you can execute the following command:

[40] http://en.wikipedia.org/wiki/RubyGems
[41] http://www.mozilla.org/en-US/firefox/new/

```
ruby first_script.rb
```

You should see the Firefox browser launch and go to the Apple website. We have just created our first script.

Let's create a slight modification to this script. The following script will run the same test in Internet Explorer. We simply have to replace `:firefox` with `:ie`

```
require 'rubygems'
require 'watir-webdriver'
browser = Watir::Browser.new :ie
browser.goto 'http://www.apple.com'
```

We can make one additional slight modification and run the same script on *Chrome*[42]. As you might have guessed we can change `:firefox` to `:chrome`. In order to use the chrome driver you will need to download the ChromeDriver binary which you can find here[43]. Just place it in your path and you are ready to run this script.

```
require 'rubygems'
require 'watir-webdriver'
browser = Watir::Browser.new :chrome
browser.goto 'http://www.apple.com'
```

You will notice that with only one small modification we enabled our script to run in a completely different browser. This is part of the power of the watir-webdriver. When you write a script that works with Firefox it will also work with Internet Explorer, and it is easy to get that script to run in several other browsers.

Our test application

In this chapter we will be using a simple puppy adoption application for demonstration purposes (although, let's face it, who doesn't love adopting puppies). You can find the application at http://puppies.herokuapp.com. When you visit this url you should see the following list of puppies on the home page for the application:

[42] https://www.google.com/intl/en/chrome/browser/
[43] http://code.google.com/p/chromium/downloads/list

Landing page for puppy adoption application

Go ahead and take the time to explore the puppy application. Adopt a fake puppy and check out.

Getting started with puppies

Now it is time to write our first script for the puppy application. This script is going to be incredibly similar to the first script we wrote. If you have the previous script open go ahead and shut it and try to do this exercise from memory. If you are unsure of what to do please refer to the *Watir-Webdriver Quick Reference* appendix. The following list details the steps you script must complete.

1. Open the browser
2. Go to the puppy adoption site
3. Wait 5 seconds
4. Close the browser

Waiting for five seconds is accomplished by the following call:

```
1 sleep 5
```

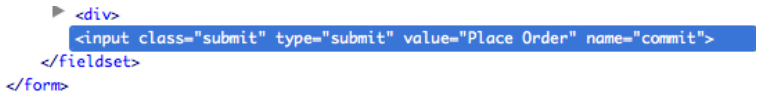
Once you have the script ready go ahead and run it. This script doesn't do very much at this time but I think we'll be changing that soon. But first you need to learn how to work with HTML elements on web pages.

Finding and interacting with HTML elements on a page

As you very likely know already, HTML pages consist of HTML elements. We'll cover several ways to interact with HTML pages, but first let's learn a brute-force and simple mechanism for finding and interacting with the smallest of page components - HTML elements. Open your browser and go to the puppy application. Adopt a puppy and continue to the checkout page (by clicking on the "Complete the Adoption" button). Once you are on the checkout page start the developer tools (Firebug if you are using Firefox) for the browser you are using. In Firefox you will need to install the Firebug addon[44]. If you are using a version of Internet Explorer that does not already have the development toolbar installed you can download it from http://www.microsoft.com/en-us/download/details.aspx?id=18359. If you are using FireFox or Internet Explorer and the proper tools are installed you can simply press the F12 button to start the developer tools. Once the tool is running you should press the "select element" button.

Select element button in Firebug

Now move your cursor around the checkout page. You should see a blue box that surrounds the HTML elements as you hover over them. Select the "Place Order" button. You will be able to see details about the element. On Firefox using FireBug you should see the following:



Place Order button selected

Watir-webdriver has a clean, consistent way of locating HTML elements. For our purposes we will always use the following three-step pattern. Imagine we are on the checkout page of the puppy application. In order to press the "Place Order" button we would type the following:

```
1 browser.button(:value => 'Place Order').click
```

Let's break down what's going on here. First we must send a message to the `browser` object telling it what type of element we wish to interact with. *Watir-webdriver* contains methods for most of the items you will find on a typical web page. In this case we are calling the `button` method.

Now that we have indicated what *sort* of element we want to interact with (`button`) we need to specify how to find it on the page. We do this by identifying a characteristic of the element we are trying to use. *watir-webdriver* supports finding elements by many characteristics - attribute values, location in the Document Object Model[45], and others. In this case we want to find the button that

[44] https://addons.mozilla.org/en-us/firefox/addon/firebug/
[45] http://en.wikipedia.org/wiki/Document_Object_Model

has a *value*[46] of "Place Order". We got that information from the developer tool when we selected the button. For a button, the *value* is the text that is displayed on the button. If there is more than one element matching the characteristic, *watir-webdriver* will return the first one on the page by default. To select the second button on the page with a label of "Search" we could use `button(:value => 'Search', :index => 1)`. The reason we specified a 1 instead of a 2 is that all indexes are 0-based[47] in Ruby. This means that the first element can be found at position 0, the second at position 1, and so on.

Finally we need to do something to the element that we have located. In this case we are sending the `click` message to the button.

Adopting a puppy

Let's go ahead and modify the previous script so that it completes the adoption of a puppy. This means you will view the details of a puppy, select the Adopt Me button, select the Complete the Adoption button and then fill in the information on the checkout page. Again, use the browser developer tools to identify the elements you want to interact with and the *Watir-Webdriver Quick Reference* at the back of the book to learn how. Try to complete the script without looking at the finished script below.

The finished script

Here is an example of the finished script using Firefox.

```
1  require 'rubygems'
2  require 'watir-webdriver'
3  browser = Watir::Browser.new :firefox
4
5  browser.goto 'http://puppies.herokuapp.com'
6  browser.button(:value => 'View Details').click
7  browser.button(:value => 'Adopt Me!').click
8  browser.button(:value => 'Complete the Adoption').click
9  browser.text_field(:id => 'order_name').set('Cheezy')
10 browser.text_field(:id => 'order_address').set('123 Main St.')
11 browser.text_field(:id => 'order_email').set('cheezy@foo.com')
12 browser.select_list(:id => 'order_pay_type').select('Check')
13 browser.button(:value => 'Place Order').click
```

The first three lines set us up to use Firefox. This should look very familiar to you as we have seen it several times already. You already know how to change the third line to use the browser of your choice.

[46] http://www.w3schools.com/tags/att_input_value.asp
[47] http://en.wikipedia.org/wiki/Zero-based_numbering

After that we go to the puppy site and click the "View Details" button. Any time we are locating an item by *value* or *text* it is case-sensitive. In other words we would not be able to find the button if we used `:value => 'view details'`. Since we did not specify any additional parameters *watir-webdriver* will select the first button it finds on the page with the value specified. If we want to select a specific puppy element we could indicate that. We will see how to do this later. The next two lines click the additional buttons to navigate through the site. Finally we land on the checkout page.

The remainder of the script fills in and submits the order form. Selecting an element in a `select_list` is also case-sensitive. We must match the entry exactly.

Run the script. You should see the message "Thank you for adopting a puppy!".

Complete the script

Our script does a nice job of selecting a puppy, adding the puppy to the cart and completing a checkout. It doesn't verify it completed successfully. From a rigorous testing standpoint, our script is incomplete if it doesn't verify that the task we automated actually worked. Let's add that to our script. If you have experience writing scripts in another language you might be tempted to write something like the following:

```
1 if not browser.text.include? 'Thank you for adopting a puppy!'
2   fail
3 end
```

In this example we are calling the text method[48] on the `browser` object which returns a `String` with all of the contents of the page. Next we are calling the `include?` method on the returned `String` passing it the value "Thank you for adopting a puppy!". This method call will return `true` if the `String` occurs anywhere on the page and `false` if it doesn't. The entire statement is stating "if the text is not included on the page, then fail".

Ruby

In Ruby it is a standard practice to append a question mark to method names that return true or false.

It is so common to state `if not` in our code that Ruby has provided a simpler way to express this idea. We could rewrite the code above to this:

[48] http://www.rubyist.net/~slagell/ruby/methods.html

```
unless browser.text.include? 'Thank you for adopting a puppy!'
  fail
end
```

`unless` means the same as `if not`. But even this does not read as well as we might like. Ruby provides us with a more concise way to say the same thing. Here is the same code written the Ruby way.

```
fail unless browser.text.include? 'Thank you for adopting a puppy!'
```

Let's run the application once again.

It works. Let's change the message to read "Thank you for adopting a coffee cup" and run it again. This time it fails but the failure message is not very clear. We can clear this up by passing a text message to the fail method. Let's replace our fail line with this:

```
fail 'Browser text did not match expected value' unless browser.text.include? 'Th\
ank you for adopting a puppy!'
```

Now we will get a much nicer message when our script fails. By the way, it's an essential automated testing practice to ensure that our scripts that run green (pass) can run red (fail), and fail in a way we expect it to. Sometimes a test cannot be made to fail properly, and we need to fix the test.

In thirteen lines of code our script adopts a puppy and verifies a confirmation message. Not bad. But it will get better!

Adopting two puppies

If you love dogs as much as I do you may be feeling like one puppy is not enough. Why adopt one puppy when you can adopt two? Let's write a new script that adopts two puppies. In fact, we want to purchase two different puppies. This means that our current method of selecting a puppy - `browser.button(:value => 'View Details')` - will not work on a page with multiple puppy elements since it always selects the first puppy. Fear not, *watir-webdriver* provides a solution. Many of the *watir-webdriver* methods allow you to provide multiple parameters for identification. The `button` method has this ability and therefore if you wish to click the second "View Details" button on the page you can use `browser.button(:value => 'View Details', :index => 1)`. We use `1` because, as we mentioned above, all indexes are zero-based. `0` would select the first and `2` would select the third.

Let's put this new script in a file called `second_script.rb`.

You now know everything you need to complete this new script that adopts two puppies. Go ahead and give it a try. This new script is nearly identical to the previous script. But is this a good thing?

Removing duplication (D.R.Y.)

An increasingly popular and important principle in good programming is the D.R.Y. principle, which is short for Don't Repeat Yourself[49]. It means if we find ourselves doing the same thing multiple times we need to find a way to do it only once. We want to make sure our scripts are always D.R.Y. in order to keep them as clean, and easy to maintain as possible.

Duplication is one of our biggest enemies when writing scripts. Duplication means that we have to change things in multiple places when, for example, we are reacting to one change in behavior, and should really only have to change one thing in the code. This leads to scripts that are difficult to maintain. One of the largest risks to any automation effort is high costs to maintain existing scripts.

Let's look at where we left off with the last script to see if we can discover duplication that perhaps we can consolidate into one place in the code.

```
1  require 'rubygems'
2  require 'watir-webdriver'
3  browser = Watir::Browser.new :firefox
4
5  browser.goto 'http://puppies.herokuapp.com'
6  browser.button(:value => 'View Details', :index => 0).click
7  browser.button(:value => 'Adopt Me!').click
8  browser.button(:value => 'Adopt Another Puppy').click
9  browser.button(:value => 'View Details', :index => 1).click
10 browser.button(:value => 'Adopt Me!').click
11 browser.button(:value => 'Complete the Adoption').click
12 browser.text_field(:id => 'order_name').set('Cheezy')
13 browser.text_field(:id => 'order_address').set('123 Main St.')
14 browser.text_field(:id => 'order_email').set('cheezy@foo.com')
15 browser.select_list(:id => 'order_pay_type').select('Check')
16 browser.button(:value => 'Place Order').click
17
18 fail unless browser.text.include? 'Thank you for adopting a puppy!'
```

At first glance you might not see duplication in our script. What about the two calls to view the details of a puppy? Other than the index number they are identical. But what can we do to remove this duplication?

In Ruby, as in many other Object Oriented programming languages, we have named reusable modules of code called methods[50]. We have already discussed calling methods but it is also possible to create your own. Indeed, as we'll see later, a large part of keeping any Object Oriented automated

[49] http://c2.com/cgi/wiki?DontRepeatYourself
[50] http://www.rubyist.net/~slagell/ruby/methods.html

testing code (or any code) D.R.Y. involves creating our own classes and methods, as if we were creating our own little testing language. Method definitions in Ruby have the following form, where "def" is short for "define":

```
def methodname

end
```

Let's create a method for selecting puppies and thereby eliminate some of the duplication in our script. In order to handle the different index numbers we will pass the index number as a parameter into our method. We will also need to pass our `browser` object (This is one way to enable the method to find and manipulate the page elements.). The method could look like this:

```
def adopt_puppy_number(browser, num)
  browser.button(:value => 'View Details', :index => num - 1).click
  browser.button(:value => 'Adopt Me!').click
end

browser.goto 'http://puppies.herokuapp.com'
adopt_puppy_number(browser, 1)
browser.button(:value => 'Adopt Another Puppy').click
adopt_puppy_number(browser, 2)
```

This seems a little better but it feels strange having to pass the `browser` object. Since the `browser` object is a local variable (see note below) we are forced to pass it to the method. Is there anything we can do to simplify this further? Absolutely. We can change the *local* variable to an *instance* variable by placing the @ symbol in front of it (see the note on local and instance variables below). If we change the local variable `browser` to the instance variable `@browser` the method can access it directly. Here is what this would look like:

```
require 'rubygems'
require 'watir-webdriver'
@browser = Watir::Browser.new :firefox

def adopt_puppy_number(num)
  @browser.button(:value => 'View Details', :index => num - 1).click
  @browser.button(:value => 'Adopt Me!').click
end

@browser.goto 'http://puppies.herokuapp.com'
adopt_puppy_number 1
@browser.button(:value => 'Adopt Another Puppy').click
adopt_puppy_number 2
```

This seems more natural. Also notice that we did not put parentheses around the number in the call to `adopt_puppy_number`. Ruby allows us to optionally not use parentheses, and sometimes omitting them adds to the readability. We decided to not use them here.

Ruby

In Ruby, as in many Object Oriented languages, variables[51] can be available only locally to individual methods (local variables[52]), or to all methods in a class (instance variables[53]). Our script is a Ruby class. Local variables have "method scope," whereas instance variables have "class scope." The main difference between the two is how long they live and where you can access them. A local variable is only available within the context in which it was created and ceases to exist once you exit that context. For example, the `browser` local variable in the previous example was not created in the `adopt_puppy_number` method. Therefore, the `browser` variable could not be accessed in the method unless we passed it in. On the other hand, instance variables are available anywhere in the class to which they belong. It made sense for us in our example to make the browser variable accessible to every method in our class, because any method in this class would likely need it. And it is messy and noisy to pass it around to any method that needs it, especially if several methods in a class *do* need it. We will learn more about classes in the next chapter but for now it is important to understand that our entire script, which is a Ruby class, has access to any instance variable we create, and instance variables can help keep our script and its methods clean and D.R.Y.

Reusable parts

In the last section we learned about making methods to eliminate duplication. Another reason we might create methods is to make our scripts more expressive and readable. For example, it is far clearer to read `continue_adopting_puppies` than to read `@browser.button(:value => "Adopt Another Puppy").click`. In this way we can create a set of reusable methods that add clarity and readability to our script. Note to manager: this readability can make a gigantic difference in keeping the cost of script maintenance low. Any code that is hard to understand is hard and expensive to maintain. Let's add this puppy-adopting method.

[51] http://www.rubyist.net/~slagell/ruby/variables.html
[52] http://www.rubyist.net/~slagell/ruby/localvars.html
[53] http://www.rubyist.net/~slagell/ruby/instancevars.html

```
def continue_adopting_puppies
  @browser.button(:value => 'Adopt Another Puppy').click
end

adopt_puppy_number 1
continue_adopting_puppies
adopt_puppy_number 2
```

Now that is much better. Let's keep going and see if we can make more methods to add clarity to the script. At the same time we are also creating a set of higher level methods that we could use in other scripts. We are, in fact, creating our own little Domain Specific Language[54] for testing a puppy adoption site. We'll cover this in more detail later, but again, always keep in mind when writing code that you are creating a language, as much as you are using one (in this case, Ruby). Go ahead and make the changes yourself to see what you come up with. What methods can you create to make the code easier to understand?

My updates

After this change your script might look something like this:

```
require 'rubygems'
require 'watir-webdriver'

def goto_the_puppy_adoption_site
  @browser = Watir::Browser.new :firefox
  @browser.goto 'http://puppies.herokuapp.com'
end

def adopt_puppy_number(num)
  @browser.button(:value => 'View Details', :index=>num - 1).click
  @browser.button(:value => 'Adopt Me!').click
end

def continue_adopting_puppies
  @browser.button(:value => 'Adopt Another Puppy').click
end

def checkout_with(name, address, email, pay_type)
  @browser.button(:value => 'Complete the Adoption').click
  @browser.text_field(:id => 'order_name').set(name)
```

[54] http://en.wikipedia.org/wiki/Domain-specific_language

```
21    @browser.text_field(:id => 'order_address').set(address)
22    @browser.text_field(:id => 'order_email').set(email)
23    @browser.select_list(:id => 'order_pay_type').select(pay_type)
24    @browser.button(:value => 'Place Order').click
25  end
26
27  def verify_page_contains(text)
28    fail unless @browser.text.include? text
29  end
30
31  def close_the_browser
32    @browser.close
33  end
34
35  goto_the_puppy_adoption_site
36  adopt_puppy_number 1
37  continue_adopting_puppies
38  adopt_puppy_number 2
39  checkout_with('Cheezy', '123 Main St', 'cheezy@foo.com', 'Check')
40  verify_page_contains 'Thank you for adopting a puppy!'
41  close_the_browser
```

Now granted, this script is much longer than our previous example. (At least, it's longer now. We'll divide it up into multiple scripts later.) But we have created expressive, reusable methods! Look at those last seven lines. What a difference! Much easier to understand than our previous example.

What vs How

If I am reading a test of this kind for the first time, I first want to know WHAT is being done. I can always go to the methods to see HOW it is being done. Our method names do a great job of expressing WHAT we are doing while artfully concealing HOW that is being done. We have the beginnings of a lovely, expressive DSL[55] for testing a puppy adoption site. A puppy-adopting-site testing-language!

The methods that Watir provides us are our "HTML testing" DSL. This is not WHAT we are testing, but HOW we are testing it. Notice that this Watir DSL, which is all about walking around in the HTML and manipulating and verifying element conditions, has been hidden away inside the methods of our puppy-adoption-site testing-language. This is critical. Our two Domain Specific Languages are separated from one another, like the Chapter names, Section names, and Subsection names of any well-organized book.

[55] http://en.wikipedia.org/wiki/Domain-specific_language

The only problem we have at this time is that the reusable methods are only available to the one script in which they are contained. In the next section we will focus on how to make the methods available to any script.

Sharing methods with multiple scripts

In the previous section we wrote a set of reusable methods that we can use in our scripts. We started creating our puppy-adoption-site testing-language. The problem is that in their current form they are not available to new scripts. Our puppy-adoption-site testing-language does not have broad enough scope[56]. Ruby has a solution for this and it is called a `Module`. We can define a `Module` like this:

```
1 module AdoptionHelper
2
3 end
```

Go ahead and create a new file named `adoption_helper.rb` in the same directory as your other scripts and create a `Module` with the definition above. Next you need to move all of the methods in our test script to the new `Module` we just created. Once the methods have been moved to the `Module` we can reuse them in any script. Here's how. After moving the methods we simply `require` the `Module` filename and then `include` the `Module` in the test script. Our updated script will look like this:

```
1  require 'rubygems'
2  require 'watir-webdriver'
3  require_relative 'adoption_helper'
4
5  include AdoptionHelper
6
7  goto_the_puppy_adoption_site
8  adopt_puppy_number 1
9  continue_adopting_puppies
10 adopt_puppy_number 2
11 checkout_with('Cheezy', '123 Main', 'cheezy@foo.com', 'Check')
12 verify_page_contains 'Thank you for adopting a puppy!'
13 close_the_browser
```

Wow. Now our testing script is short, sweet, and all about WHAT we are testing. Not only have we hidden the details of HOW we do this work, we have made that reusable code accessible to any other test module we need for testing the puppy adoption site. Very cool. The two key lines of code that

[56] http://www.techotopia.com/index.php/Ruby_Variable_Scope

make those reusable methods available to this script are lines three and five. On line three we require our `Module` but there are a couple of things that need to be explained. In Ruby there is something called the `LOAD_PATH` which defines where Ruby should look for required files. the `LOAD_PATH` defines the things we want to be accessible globally to all of our test scripts. Starting with Ruby 1.9.2, the current directory is not included by default. If you are using Ruby 1.9.2 or later you can simply use the `require_relative` call to include a file in the same directory as the file it is required from.

Ruby ⇒ Requiring a file with an older version of Ruby

If you are using a version of Ruby prior to 1.9.2 you should just be able to `require` the file since the `LOAD_PATH` already includes the current directory. It is a little more challenging if you wish to write a require statement that will work with both old and new versions of Ruby. In order to work with old and new Rubies you could add a line like this -> `require File.dirname(__FILE__) + '/adoption_helper'`. The `__FILE__` constant means the current file so in our case it means the script from which it was contained. This first part of this call is therefore asking for the directory of the current file. Once it has that it adds the module filename to the end.

We then `include` the `Module` on line five. Including the `Module` adds it's methods to our current environment or script so we can make the calls accordingly. If we had not included the `Module` we would have to make long-winded, so-called "fully-qualified" calls like this -> `AdoptionHelper.goto_the_puppy_adoption_site`. It's much simpler to just say `goto_the_puppy_adoption_site`.

Armed with this wonderful knowledge and a `Module` of reusable methods I would now like you to go back and refactor the first script that adopted one puppy and modify it to use our `AdoptionHelper` Module. Much simpler, isn't it?

Simple Watir Scripts

Pat yourself on the back. You now have the necessary knowledge and skills to write simple test scripts using *Watir*. In some cases, this simple approach might be all you need. But most people find that in the majority of cases this is simply not enough to build up a test suite for a large application. In the next two chapters we will be introducing some additional concepts and gems that will help you write more robust test suites.

4. Cucumber & Puppies

Our first Cucumber project

We are finally ready to add Cucumber to our testing toolbox. We'll be using the *testgen*[57] gem to create our project structure but first we need to install the gem. Hang on for a couple of pages. We will quickly introduce several concepts without fully defining them yet. Bear with us! Execute the following from a command window:

```
1 gem install testgen
2 gem install bundler
```

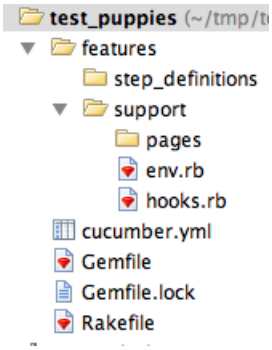
The *testgen* gem can be used to create empty test projects that use cucumber. We'll use it to get started. To generate our project please execute the following:

```
1 testgen project test_puppies --pageobject-driver=watir
```

This command will create our entire project directory structure. The final step is to change into the newly created `test_puppies` directory and execute:

```
1 bundle install
```

This command will install all of the gems needed for our project at this time. We'll talk more about bundler later in the book.



Directory structure of newly created project

[57] https://github.com/cheezy/testgen

The *testgen* gem with the provided command-line parameters creates the directories and files displayed above. If you are using *RubyMine* go ahead and open the directory. Otherwise, open the directory with a tool that shows the directories and files.

The first thing you will notice about our cucumber project is that there is a single directory at the root of the project named `features`. This directory is where all of your cucumber feature files[58] will reside. In this directory you will find two additional directories. They are `step_definitions` and `support`. The `step_definitions` directory is empty at this time. The `support` directory has two files. Let's take a quick look at them.

The first file is named `env.rb`. This file requires a few gems that we will be using throughout this chapter and then makes a call to a `World` method. We'll talk about that method a little later.

```
1  require 'rspec-expectations'
2  require 'page-object'
3
4  World(PageObject::PageFactory)
```

The other file in the support directory is `hooks.rb`.

```
1  require 'watir-webdriver'
2
3  Before do
4    @browser = Watir::Browser.new :firefox
5  end
6
7  After do
8    @browser.close
9  end
```

Here you can see that we have a `Before` block and an `After` block. The `Before` block is run before each test and is creating the `browser` object. We've seen this code before. The `After` block runs after each test and here it is closing the browser.

There are other files in our project in addition to the files under the `features` directory. We'll be covering them in later sections of the book.

Writing a simple cuke

Cucumber feature scripts are all about very high-level, feature-level "WHAT" DSLs. They describe, in natural English language (to all project stakeholders, including some who may never see the

[58] http://cuke4ninja.com/sec_cucumber_jargon.html#N106D9

code), WHAT the feature is, and also, WHY we want it. This is a characteristic of this kind of very high-level DLS: we are describing WHAT we are doing and at the same time WHY we are doing it. Project managers will simply adore you for providing them executable requirements of this kind! Trust me on this.

Let's write our first cucumber script that describes the WHAT and WHY of a cheese-making feature. While we're at it we'll discuss *Gherkin*[59], a language for describing application behavior (What can I say? Some people who love puppies also love cheese. And pickles. And beer. But that's a different story.). Create a file in the `features` directory named `making_cheese.feature`. We will start with our story description.

```
1     Feature: Making Cheese
2
3       As a cheese maker
4       I want to make cheese
5       So I can share my cheesiness
```

This seems to be a pretty succinct description of the WHAT and WHY of our cheese-making feature. Again, we are in effect providing documentation to the readers that explains the purpose of this feature we are building. Note: This is truly free-form text except for the first word - *Feature*. Each feature file must have one *Feature* description. The colon after the Feature keyword is important - cucumber will complain without it. All of the documentation you add will be contained in the reports that are produced when we run the cucumber command.

The remainder of the file is filled with *Scenarios*. A *Scenario* is nothing more than an example of HOW the user will interact with the system. Think of these as examples that demonstrate the functionality we intend to build. In other words, HOW our feature will work. This is a lower-level of requirements language. Our Feature definition above was, again, about WHAT the feature is and WHY we want it. Now these Scenario definitions are about HOW the feature should behave, from a user experience[60] (UX) standpoint. (Oh, you think Project managers will love you now? The UX designers will love you more. Much more.) The combination of all of the scenarios inside a Feature definition define all our feature's behavioral details. Let's look at one.

```
1     Scenario: Using the cheese machine
2       Given I have no cheese
3       When I press the make cheese button
4       Then I should have 1 piece of cheese
```

There are four keywords in this scenario. The first is *Scenario*. As we said before, a *Scenario* describes HOW the user will interact with the system. Cucumber supports several types of *Scenarios*. The

[59] https://github.com/cucumber/cucumber/wiki/Gherkin
[60] http://en.wikipedia.org/wiki/User_experience

example above is the simplest form but we will look at another type later in this chapter. The description for the scenario should describe what is unique about this particular *Scenario* and the keyword must end with a colon.

After the *Scenario* we have what cucumber calls steps. These steps begin with one of five keywords - *Given*, *When*, *Then*, *And*, and *But*. This format is known as *Gherkin*. *Gherkin* is a Simple Readable Domain Specific Language that describes a software's behavior without defining how the behavior is implemented. It is quickly becoming an industry standard and is implemented by numerous tools.

Given steps are used to define context and perform setup necessary for the *Scenario*. *When* steps define the action that is taken as a part of the *Scenario*. *Then* is where you verify the expected results. *Given*, *When*, and *Then* can have additional steps that begin with *And* or *But*, for adding more qualifying language to the context or expected results. This will become clearer as we write more *Scenarios*. All of this **Gherkin** machinery helps define exactly HOW a feature works. Very cool.

It's time to run the script and see what happens. If you are using *RubyMine* right-click on the filename and select the run menu option. If you are running from the command line type `cucumber features\making_cheese.feature`. You should see output that look similar to the following.

```
1     Feature: Making Cheese
2       As a cheese maker
3       I want to make cheese
4       So I can share my cheesiness
5
6       Scenario: Using the cheese machine
7         Given I have no cheese
8         When I press the make cheese button
9         Then I should have 1 piece of cheese
10
11    1 scenario (1 undefined)
12    3 steps (3 undefined)
13    0m0.232s
14
15    You can implement step definitions for undefined steps with these snippets:
16
17    Given /^I have no cheese$/ do
18      pending # express the regexp above with the code you wish …
19    end
20
21    When /^I press the make cheese button$/ do
22      pending # express the regexp above with the code you wish …
23    end
24
```

```
25      Then /^I should have (\d+) piece of cheese$/ do |arg1|
26        pending # express the regexp above with the code you wish …
27      end
```

Cucumber is informing us that there are no step definitions for the scenario. It is also providing a starting point for you to implement those steps. Aha! Step Definitions. These are what make Cucumber scripts executable, automated tests. This is where high-level feature DSL is converted to lower-level Ruby code, so we can actually run it, and watch it turn green, proving that not only are we Building Something Right, we are Building the Right Thing! Awesome! Let's create a new file in the `step_definitions` directory named `making_cheese_steps.rb`. Copy the step definitions provided by cucumber to that file. When you run the cucumber command again you should see something slightly different.

```
1       Feature: Making Cheese
2         As a cheese maker
3         I want to make cheese
4         So I can share my cheesiness
5
6         Scenario: Using the cheese machine
7           Given I have no cheese
8             TODO  (Cucumber::Pending)
9             ./features/step_definitions/making_cheese_steps.rb:3:in ...
10          When I press the make cheese button
11          Then I should have 1 piece of cheese
12
13      1 scenario (1 pending)
14      3 steps (2 skipped, 1 pending)
```

Here you can see that it found the steps but indicated that the first step was *pending* (Here, pending means "not yet fully implemented in code"). As a result it skipped the remaining steps. When cucumber runs into a step that is pending it does not continue running the scenario. We can easily resolve this. Let's write some place-holder code to make the steps run.

```
Given /^I have no cheese$/ do
  puts "I am so sad.  I have no cheese :("
end

When /^I press the make cheese button$/ do
  puts "There is hope.  I hope this machine works"
end

Then /^I should have (\d+) piece of cheese$/ do |num_pieces|
  puts "Rejoice!  We have #{num_pieces} pieces of cheese."
end
```

Notice that I have renamed the `arg1` parameter in the third step definition to `num_pieces`. This is to add clarity. Let's run the scenario again. This time you will notice that the scenario passes and the messages are printed to the screen.

You might be asking yourself why this scenario is passing. We haven't actually done anything yet so why should it work? Well cucumber is following the same pattern we saw when we wrote the Watir scripts in the last chapter. It is assuming success until either it encounters a code error or an assertion fails. So, yes, we have a Cucumber test that doesn't actually prove that we make cheese. Which of course is inherently bad, since we have established that we really like cheese.

But remember that we also like puppies, and in that problem domain, we have a puppy adoption site. We've already written *Watir* tests to prove that we can adopt a puppy. We may not be able to adopt a cheese, but we can definitely adopt a puppy. And we can use Cucumber tests to prove that indeed this occurs, and to do so at the puppy-adoption feature level.

We can combine the WHAT, WHY, and HOW of puppy adoption in our next Cucumber tests. Joy! We will see how this all comes together in the next section.

Adopting a puppy with cucumber

Now it is time to write a cucumber feature that does something more interesting than the feature we created in the last section. In this section we will write a feature that adopts a puppy. Create a new file named `adopting_puppies.feature` in the `features` directory. Our feature file should begin like this:

```
    Feature: Adopting puppies

      As a puppy lover
      I want to adopt puppies
      So they can chew my furniture
```

This is a good start. Now we need to begin writing the first *Scenario*. Let's start with this:

```
Scenario: Adopting one puppy
  Given I am on the puppy adoption site
  When I click the View Details button
  And I click the Adopt Me button
  And I click the Complete the Adoption button
  And I enter "Cheezy" in the name field
  And I enter "123 Main Street" in the address field
  And I enter "cheezy@example.com" in the email field
  And I select "Credit card" from the pay with dropdown
  And I click the Place Order button
  Then I should see "Thank you for adopting a puppy!"
```

Now go ahead and generate the step definitions by running the newly created feature file. Place the step definitions in a file named `adopting_puppy_steps.rb` in the `step_definitions` directory and rename the variables to something more meaningful. After the step definitions are generated I would like you to try to complete the script to make the *Scenario* run successfully. Here is the completed first step:

```
Given /^I am on the puppy adoption site$/ do
  @browser.goto "http://puppies.herokuapp.com"
end
```

Try to complete all of the remaining steps without referring to the work we did in the last chapter. Repetition will help us learn the *Watir* syntax and make us better automated testers. When you are finished go ahead and run the *Feature*. If you have a failure, correct it and try again until the script runs all of the way through.

RSpec Matchers

In the last chapter we verified the success message with the following code:

```
fail unless browser.text.include? "Thank you for adopting a puppy!"
```

At this time I would like to explore an alternative to this approach. RSpec[61] is a collection of Ruby gems used for unit testing Ruby applications. RSpec has a very good framework for defining expectations for how a system should work. It is very common for cucumber tests to use the RSpec expectations framework. We'll be doing that here. In our *env.rb* file we are already requiring `rspec-expectations` which makes the expectation framework available to our step definitions. This gem will help us rewrite this step to make it cleaner and make our error message better.

[61] https://github.com/rspec/

In the appendix *RSpec Matchers* you can find a complete listing of all of the matchers available for you to use out-of-the-box but we will cover one of them here. RSpec is an amazing gem that leverages the dynamic capabilities of the Ruby language. Let's walk through what it does to help us understand how it works but to also give us a brief glimpse into what Ruby is capable of.

The first thing that RSpec does is add a `should` and `should_not` method to all objects. This allows us to specify our expectations in a near-English form. For example we can say that some object `should` or `should_not` do something. In our case we want to say that the `browser` should include some text. We would do this by calling the `include?` method on the text returned from the `browser` object.

RSpec has a built-in `include` matcher that works on an `Array`, a `Hash`, or a `String`. In each case it calls the `include?` method on the actual value. This allows us to drop the question mark in the method call. RSpec will call the `include?` method on the `String` returned by `text` passing our expected text. If the call returns true the expectation is met. Otherwise it fails. The end result is a very expressive scenario expectation.

```
1     Then /^I should see "([^\"]*)"$/ do |expected|
2       @browser.text.should include expected
3     end
```

We will introduce more RSpec matchers as we move through the examples.

Your finished step definition file should look like this:

```
1     Given /^I am on the puppy adoption site$/ do
2       @browser.goto "http://puppies.herokuapp.com"
3     end
4
5     When /^I click the View Details button$/ do
6       @browser.button(:value => "View Details").click
7     end
8
9     When /^I click the Adopt Me button$/ do
10      @browser.button(:value => "Adopt Me!").click
11    end
12
13    When /^I click the Complete the Adoption button$/ do
14      @browser.button(:value => "Complete the Adoption").click
15    end
16
17    When /^I enter "([^\"]*)" in the name field$/ do |name|
18      @browser.text_field(:id => "order_name").set(name)
19    end
20
```

```
21    When /^I enter "([^\"]*)" in the address field$/ do |address|
22      @browser.text_field(:id => "order_address").set(address)
23    end
24
25    When /^I enter "([^\"]*)" in the email field$/ do |email|
26      @browser.text_field(:id => "order_email").set(email)
27    end
28
29    When /^I select "([^\"]*)" from the pay with dropdown$/ do |pay_type|
30      @browser.select_list(:id => "order_pay_type").select(pay_type)
31    end
32
33    When /^I click the Place Order button$/ do
34      @browser.button(:value => "Place Order").click
35    end
36
37    Then /^I should see "([^\"]*)"$/ do |expected|
38      @browser.text.should include expected
39    end
```

For the time being, don't worry about the gobbletygook that looks like this: "([^"]*)"$. This strange and ugly, but useful code, will prove its worth to you shortly. For now, take our word for it that this voodoo is completely worth the effort.

Adopting two puppies

Let's write a second *Scenario* in the directly below the first *Scenario* and call it "Adopting two puppies".. As you might guess, we want this new *Scenario* to adopt two puppies and complete the order. Your *Scenario* should be nearly identical to the previous *Scenario*.

One problem we need to address is specifying that we will click different View Details buttons. In the first scenario we just stated that we clicked the View Details button but for this scenario we will need to say that we click the first View Details button and the second View Details button.

After you are finished writing the second *Scenario* I suggest you change the first *Scenario* to state that you click the first View Details button and then go ahead and update the step definitions file to reflect the new syntax. You will also need to update the *Watir* call to add the `:index` parameter in order to click the correct button. The new *Scenario* should look like this:

```
  Scenario: Adopting two puppies
    Given I am on the puppy adoption site
    When I click the first View Details button
    And I click the Adopt Me button
    And I click the Adopt Another Puppy button
    And I click the second View Details button
    And I click the Adopt Me button
    And I click the Complete the Adoption button
    And I enter "Cheezy" in the name field
    And I enter "123 Main Street" in the address field
    And I enter "cheezy@example.com" in the email field
    And I select "Credit card" from the pay with dropdown
    And I click the Place Order button
    Then I should see "Thank you for adopting a puppy!"
```

Go ahead and generate the step definition and implement the methods. As you see, we were able to reuse much of the previous step definitions in this new *Scenario*. The two steps that we did have to create were the ones that clicked the first and second view details buttons. We already had a step in the first *Scenario* that clicked the first view details button. When we see this sort of duplication it is a good idea to remove it. Please change the first scenario to use the step When I click the first View Details button and then remove the step definition for the generic method so it no longer exists in your step definition file.

Scenario Outlines

Cucumber has another type of *Scenario* called a *Scenario Outline* that relies on the *Example* keyword. It allows us to separate out the small tables of example data used in the *Scenario* from the description of the steps in the *Scenario* and thereby run it with multiple sets of data. This is very useful for situations when you have a single path through the application that you wish to run with different sets of example data. These different examples might test different subtle behaviors in our code. So we are reusing the same test code, but substituting different rows of example data for each test execution. It is yet another trick to help us keep the code D.R.Y. (that is, without unnecessary duplication).

Scenario Outlines take this form:

```
  Scenario Outline: Using the cheese machine
    Given I have no Cheese
    When I press the make "<type>" cheese button
    Then I should see the "<message>" message

  Examples:
    | type      | message               |
    | Swiss     | I love Swiss cheese   |
    | Blue      | I love Blue cheese    |
    | Cheddar   | I love Cheddar cheese |
```

As you can see, *Scenario Outlines* have two parts. The first is the outline which provides the steps for the test and the second is the data used in the steps. These two parts are bound together by variables that are replaced when the *Scenario* runs. The variables are defined in the steps with angle brackets like `<name>`. This is matched with a heading in the Examples section and the value for each row is inserted into the step.

The *Scenario* above will run three times. The first time the *type* variable will be replaced with the word Swiss and the *message* variable will be replace with I love Swiss cheese. The second time the *Scenario* is run it will use the data from the second row and so on.

For your next assignment I want you to rewrite the first *Scenario* in our adopt puppies file to make it a *Scenario Outline* and provide three *Examples* so it will run with three different complete sets of data. The *Examples* portion of your script will look something like:

```
  Examples:
    | name   | address     | email              | pay_type      |
    | Cheezy | 123 Main St | cheezy@example.com| Credit card   |
    | Joseph | 555 South St| joe@guru.com       | Check         |
    | Jared  | 234 Leandog | doc@dev.com        | Purchase order|
```

Notice, by the way, that once a Product Owner is accustomed to the sets of inputs to a Scenario, this Examples table becomes a succinct way for them to see that several different scenarios are being tested. It is a handy shortand for data-intensive testing situations. At the same time, running a lot of data through the system is not always a good idea. We should be very pragmatic about the tests we add. We want each new test to help us learn something new about the system we are testing. Exercising the same code in the exact same way over and over again does not expand our understanding nor does it help us discover new defects.

Background

Our *Scenarios* have some duplication that we want to address. Both of them begin with the same Step:

```
  Given I am on the puppy adoption site
```

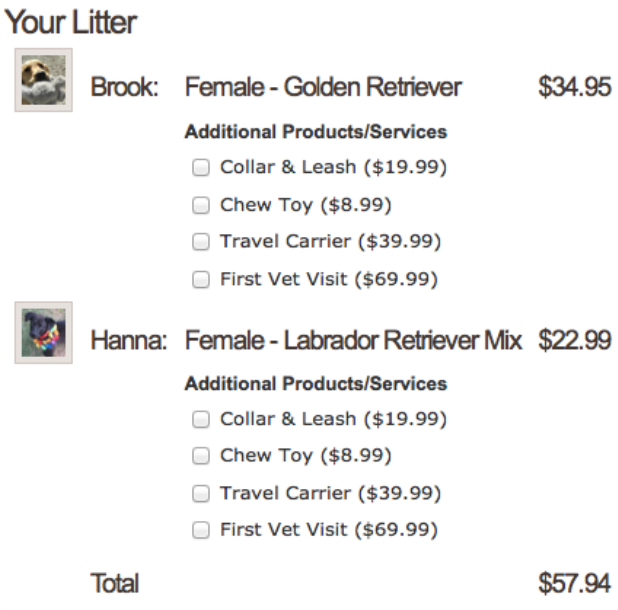
Since we have learned that duplication is evil we should find a way to eliminate this travesty. Luckily cucumber has a solution. It is call *Background*. Think of it as a step or set of steps that are executed prior to each *Scenario*. Using a *Background* the duplication in our *Scenarios* is eliminated and we end up with:

```
  Background:
    Given I am on the puppy adoption site

  Scenario: Outline: Adopting one puppy
    When I click the first View Details button
    ...

  Scenario: Adopting two puppies
    When I click the first View Details button
```

Now that feels better, doesn't it.

Verify the shopping cart

We are off to a good start with cucumber but so far our tests haven't really tested much. We are about to change that. In this section we will write some *Scenarios* that will test our shopping cart page. Let's look at an example shopping cart.



Your Litter Page

Looking at this shopping cart what *Scenarios* do you think we need to specify this page and verify correctness?

There are numerous *Scenarios* necessary to specify this page but for this section I would like you to write two.

1. Verify the shopping cart with one puppy.
2. Verify the shopping cart with two puppies.

In each *Scenario* I would like you to validate the puppy's name, the subtotal for the adoption, and the shopping cart total.

The shopping cart data is stored in an html table. The table does not have an `id`, `name`, or any other identifier. What do you think you can use to locate the table? If you said `index` you are correct.

You can access the fields in a table by using rows and columns. For example, to access the first row and second column you would do the following:

```
1  @browser.table(:index => 0)[0][1]
```

As you can see, the indexes we use in the square brackets are zero based. The shopping cart "Total" can be accessed directly using the `td` method.

Be aware! This is by far the most complex exercise we have done so far and I expect it to take a fair amount of time to complete.

Try to complete this exercise on your own before looking at the completed scripts below.

The completed scripts

Here are the *Scenarios* I came up with for this exercise:

```
1    Scenario: Validate cart with one puppy
2      When I click the first View Details button
3      And I click the Adopt Me button
4      Then I should see "Brook" as the name for line item 1
5      And I should see "$34.95" as the subtotal for line item 1
6      And I should see "$34.95" as the cart total
7
8    Scenario: Validate cart with two puppies
9      When I click the first View Details button
10     And I click the Adopt Me button
11     And I click the Adopt Another Puppy button
12     And I click the second View Details button
```

```
    And I click the Adopt Me button
    Then I should see "Brook" as the name for line item 1
    And I should see "$34.95" as the subtotal for line item 1
    And I should see "Hanna" as the name for line item 2
    And I should see "$22.99" as the subtotal for line item 2
    And I should see "$57.94" as the cart total
```

These Scenarios caused me to write three new step definitions. Here they are:

```
Then /^I should see "([^"]*)" as the name for line item (\d+)$/ do |name, line_it\
em|
  row = (line_item.to_i - 1) * 6
  @browser.table(:index => 0)[row][1].text.should include name
end

When /^I should see "([^"]*)" as the subtotal for line item (\d+)$/ do |subtotal,\
 line_item|
  row = (line_item.to_i - 1) * 6
  @browser.table(:index => 0)[row][3].text.should == subtotal
end

When /^I should see "([^"]*)" as the cart total$/ do |total|
  @browser.td(:class => 'total_cell').text.should == total
end
```

There is only one table on the page and it has no identifier so the easiest way to locate it is using the `index` parameter. Please be aware that this approach is brittle and will fail if the page is later modified and another table is added before this one.

Part of the lesson here is that it is a very poor web application development practice to create html elements that do not have an id or name attribute. This is what makes the tests brittle - unnecessarily brittle. This is when a conversation between a tester and programmer is valuable - even pair programming between them. It is frequently a trivial amount of work to add id's and name's to elements without them. Much less work, in fact, than trying to maintain test code that compensates for lack of them!

Tables are accessed by rows and columns so a call to retrieve the text for the third column in the second row would look like this:

```
    @browser.table(:index => 0)[1][2].text
```

Once we find the correct cell in the table we need to compare it's text with the value supplied in the step. To do this we will use a couple of different RSpec matchers. Since the name field includes

a colon in the text we use the `include` matcher. This matcher validates that the searched for value is *included* in the provided `String`. For the other steps we want to have an exact match so we use the == matcher.

When I looked at the table I realized there were six rows per puppy. In order to use the `line_item` parameter I needed to convert it to a number and perform a calculation to determine which row in the table to access. Cucumber is passing it as a string and yet our method is expecting it to be a number. We can convert it from a string to a number by using the `to_i` method.

We have duplicated this code to perform the calculation in the first two step definitions. We have also duplicated the code to find the table and select the appropriate row in both steps. We know that duplication is bad but what can we do?

The final thing to note is that we were able to go directly to the total field because it has a `class` defined. This is far simpler than digging through a table without any useful name, id, or class attribute to find the correct cell. As we mentioned, having these element attributes is crucial.

Removing the duplication

There is a lot of duplication in the three step definitions we just completed. There are many ways for us to eliminate this duplication but when confronted with several options I typically choose the simplest solution I can think of and then move to something more complicated when it is warrented. This philosophy comes from the Agile methodology called "eXtreme Programming (XP)," from which came the programming axiom "Do the simplest thing that could possibly work"[62].

The simplest way to remove the duplication that I can think of here is to introduce a couple of methods. To keep things extremely simple we'll just add the methods directly in our step definitions file.

Let's begin by eliminating the duplicated calculation for the table row. We can create a method and make a call to it in our first step definition.

```
1  def row_for(line_item)
2    (line_item - 1) * 6
3  end
4
5  Then /^I should see "([^"]*)" as the name for line item (\d+)$/ do |name, line_it\
6  em|
7    row = row_for(line_item.to_i)
8    @browser.table(:index => 0)[row][1].text.should include name
9  end
```

After we write this code we need to re-run the *Scenario* to ensure we didn't break anything. Once we get a passing test we can make the same change to the second step.

[62] http://c2.com/xp/DoTheSimplestThingThatCouldPossiblyWork.html

The next thing I wish to eliminate is the code that selects the table and correct row. We can do that by introducing another method.

```
def row_for(line_item)
  (line_item - 1) * 6
end

def cart_line_item(line_item)
  @browser.table(:index => 0)[row_for(line_item)]
end

Then /^I should see "([^"]*)" as the name for line item (\d+)$/ do |name, line_it\
em|
  cart_line_item(line_item.to_i)[1].text.should include name
end

When /^I should see "([^"]*)" as the subtotal for line item (\d+)$/ do |subtotal,\
 line_item|
  cart_line_item(line_item.to_i)[3].text.should == subtotal
end

When /^I should see "([^"]*)" as the cart total$/ do |total|
  @browser.td(:class => 'total_cell').text.should == total
end
```

Things change

If there is anything we learn quickly in our industry it is that things change - frequently for very good reason. The software we work on is often in a constant state of modification, improvement, extension, or other change. In fact, most of us are employed in order to bring about change to existing software. One of our challenges is to make sure that we minimize the work necessary - for ourselves and for others as our software changes.

In the last section we accessed the rows and columns of a table in order to validate values. What would happen if the order of those columns changed? What would happen if the number of rows per puppy changed due to the addition of some additional products? What would happen if this entire page was modified to no longer use tables?

Any form of change would cause us to make significant changes in a few places in our tests. Imagine what would happen if we had hundreds or even thousands of tests and a significant change was made to the site. We could have hundreds of tests fail.

It is ok if hundreds of tests fail due to a change in the application but it is not ok if we have to go to multiple places to fix the code. We want to be D.R.Y. so we only have to make the appropriate

change in one place. Mastering these techniques to keep our code D.R.Y. and well-tested are critically important, since most systems are both badly written and badly tested, and slowly become almost impossible to change over time. This is the trend that we are trying to grow out of, as an industry. What can we do to achieve this goal? We'll see that shortly but first of all we will need to introduce a new concept and coding structure.

Classes & Objects

In chapter three we introduced `Modules`. We learned that they are a place to store methods that we wish to use in multiple locations. In this section we will introduce `Classes`. They are similar to `Modules` in a few ways but are used in a totally different way. The main purpose of a `Class` is to represent a category of similar objects. To that end they contain both the information about the object as well as methods that execute the behavior of the object. Let's look at a concrete example.

An example of a `Class` is Dog. A dog has certain information that describes it including breed, color, age, and name just to name a few. Each of these would be represented as instance variables on the Dog class. It also has behaviors that we can observe such as eat, drink, and sleep. These behaviors would be represented as methods. There are two additional concepts, from Object Oriented Programming[63], that I would like to cover here and they are *Inheritance* and *Encapsulation*.

Inheritance[64] gives us the ability to express relationships between `Classes`; specifically the *is-a* relationship. For example, a Dog *is a* Mammal so our Dog class could inherit from a Mammal class. It also allows our code to stay D.R.Y. For example, both the Dog and Horse class are mammals so they could both inherit from the Mammal class. The code that represents the behavior of a Mammal would exist in only one place and yet both the Dog and Horse inherit that capability.

Encapsulation[65] is all about hiding details in a class and allowing them to change independent of who might be using that class. For example, we have used many methods on a `Watir::Browser` class throughout the examples so far and have no idea what code is inside the methods. Over time the authors of the *Watir* gem could decide to make changes to the method internals and it would have no effect on our code.

Inheritance and Encapsulation are very important techniques for keeping code D.R.Y. In fact, if they did not help us eliminate duplication, they would not have nearly as much use or importance.

A class is defined in much the same way as a module.

```
1  class Dog
2
3  end
```

Inside the class you can put methods and instance variables. How would we define that our Dog class inherits from Mammal?

[63] http://en.wikipedia.org/wiki/Object-oriented_programming
[64] http://en.wikipedia.org/wiki/Inheritance_(computer_science)
[65] http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)

```
class Dog < Mammal

end
```

That's all there is to classes, but what are these object things we keep hearing about? An object is simply a specific instance of a `Class` with the appropriate information provided.

Chair Class and Objects

In the top-left part of the image above you can see that we have a class that represents a chair. This class might define some details about the chair like what material it is made of and how many legs it has. We can create an object of our chair class and fill in the necessary information. The pictures on the right represent three (very!) distinct objects of our chair class.

A page object for our shopping cart

The step definitions we wrote for verifying the shopping cart have a problem. They are extremely brittle. They are still brittle, even after we eliminated a few kinds of duplication! What I mean by that is that many steps will break if the page is changed. Brittle tests are a major cause of test automation failures. Tests that fail every time the application changes and that need significant effort to fix cause us to spend more and more of our time keeping existing tests running. I have seen instances where nearly a third of the testers' time is spent keeping existing tests running. This takes away from writing new tests and activities outside of automation like exploratory testing.

But what can we do?

Let's introduce a new concept. This concept is a *Page Object* that is used to hide all of the details about our shopping cart page. A *Page Object* is a class that contains all of the code to access a web page. It should know the details of this page and should protect us from change to the responsibilities of this page. In other words, if we are successful in this effort we should not have to go to any other place to make changes if the production web application page changes. This will likely make a lot

more sense after you see an example. Let's create the basic structure of a Page object class first and then we will provide the details later. Start by creating a new file in the `features/support/pages` directory named `shopping_cart_page.rb` and adding the following code:

```
1     class ShoppingCartPage
2
3       def initialize(browser)
4         @browser = browser
5       end
6
7     end
```

To begin with we have added an `initialize` method. In our page we will need to have access to the `@browser` variable and since it lives outside of our class we need to pass it in. The `initialize` method is called when we create a new object of this class via the `new` method and all parameters are pass through. For example, the following code will in turn call our initialize method.

```
1     @cart = ShoppingCartPage.new(@browser)
```

Now that we have the ability to create our page object we need to add some of the behaviors that the actual production Shopping Cart Page has, so this Page Object can be asked to test them for us. First of all we need to decide what to add. One technique I like to use to help me decide what to add is to just write the test method I wish I had on that Page Object. In order to do this let's start in our step definitions. For the step to verify the name field I'll comment out the code there now and write the new code.

```
1     Then /^I should see "([^"]*)" as the name for line item (\d+)$/ do |name, lin\
2 e_item|
3       # cart_line_item(line_item.to_i)[1].text.should include name
4       @cart.name_for_line_item(line_item.to_i).should include name
5     end
```

In writing this new line of code I have made several design decisions. First of all I have decided that I would have an instance variable named `@cart` which would hold an instance of my `ShoppingCartPage`. Next I decided that there will be a method named `name_for_line_item` and it will take a single parameter which is a number. Finally, I decided this method would return the text of the name column for the appropriate puppy.

While I'm at it I'll go ahead and do the same for the two remaining methods.

```
When /^I should see "([^"]*)" as the subtotal for line item (\d+)$/ do |subtotal,\
 line_item|
  # cart_line_item(line_item.to_i)[3].text.should == subtotal
  @cart.subtotal_for_line_item(line_item.to_i).should == subtotal
end

When /^I should see "([^"]*)" as the cart total$/ do |total|
  # @browser.td(:class => 'total_cell').text.should == total
  @cart.cart_total.should == total
end
```

What do you think?

Why do you think I converted the `line_item` variable to a number in my step definition instead of inside the method? The fact that `line_item` is a `String` and needs to be converted to a number in order to be used is due to how Cucumber passes the argument. In our class, we should always insist our values be passed in the form we need to use them and delegate any conversions to the caller.

Let's add these methods to our `ShoppingCartPage` class. First of all I'll copy the two helper methods we created in our step definition file to the class and then write the new methods. Here's what I came up with:

```
    class ShoppingCartPage

      def initialize(browser)
        @browser = browser
      end

      def name_for_line_item(line_item)
        cart_line_item(line_item)[1].text
      end

      def subtotal_for_line_item(line_item)
        cart_line_item(line_item)[3].text
      end

      def cart_total
        @browser.td(:class => 'total_cell').text
```

```
17        end
18
19        def row_for(line_item)
20          (line_item - 1) * 6
21        end
22
23        def cart_line_item(line_item)
24          @browser.table(:index => 0)[row_for(line_item)]
25        end
26      end
```

This is a much more full-fledged ShoppingCartPage. It's methods give us a handy DSL for testing behaviors on the actual Shopping Cart page in the puppy adoption web application. I think we're ready to put it to use. The only remaining thing to do is create the `@cart` instance variable. We arrive on the shopping cart page after we click the "Adopt Me!" button so that seems like the best place to create our object. The updated step will look like this:

```
1       When /^I click the Adopt Me button$/ do
2         @browser.button(:value => "Adopt Me!").click
3         @cart = ShoppingCartPage.new(@browser)
4       end
```

Go ahead and run the script. Everything should work just fine. We've taken a lot of the complexity related to verifying this page and encapsulated it in the new page object. Now that we know everything works we'll go back and remove the commented-out code and the two methods we defined in the step definitions. Things are off to a good start but there are still a few things to do.

Magic Numbers

Magic Numbers is a term used to describe numbers that appear in code that do not express their reason for existence or intent. An example of that would be the 1 in the following statement.

```
1       cart_line_item(line_item)[1].text
```

Looking at it now after we completed the code we know that it represents the second column (the Name column) of the table but if we come back to this code in two or three months it might not be as clear. A worse potential problem is that we might end up repeating this 1 elsewhere to represent the same column - or to represent some other concept entirely. This is both confusing and brittle.

I think we can make our code express our intent more clearly and simplify the maintenance of it by eliminating this magic. Let's introduce the idea of constants. A `Constant` is a type of variable in Ruby that is supposed to remain constant. In other words, once it has a value it is not supposed to change. Constants are always capitalized in Ruby with the words separated by underscores. Add this code to the top of our `ShoppingCartPage`:

```
class ShoppingCartPage
  NAME_COLUMN = 1
```

Creating this constant allows us to our code like this:

```
cart_line_item(line_item)[NAME_COLUMN].text
```

Go ahead and replace the two remaining magic numbers in our class. When you are finished you should have these defined:

```
class ShoppingCartPage
  NAME_COLUMN = 1
  SUBTOTAL_COLUMN = 3
  LINES_PER_PUPPY = 6
```

If the designer decides to change the order of the columns in the table we only have to change the numbers that are assigned to the constants representing the columns. If somebody decides we need to offer one additional product from the cart we only need to update the `LINES_PER_PUPPY` to a `7`. The remainder of the class remains untouched. So, like many of the other techniques we've described, Constants help us keep the code D.R.Y.

Ruby

It is a Ruby standard to name Constants with all capital letters with an underscore separating the words.

Keeping things private

We added five methods to our `ShoppingCartPage` class. Three of these methods are called from our step definitions but the remaining two were designed to only be called from other methods of the class. Ruby has a `private` keyword which provides a way for us to make sure that such "internal" methods stay internal. By placing our `row_for` and `cart_line_item` method below a call to `private` we can ensure they stay private and cannot be called from outside of the class:

```
    private

    def row_for(line_item)
      (line_item - 1) * 6
    end

    def cart_line_item(line_item)
      @browser.table(:index => 0)[row_for(line_item)]
    end
  end
```

This ensures that this method will not be called by anyone except another method in this class.

What about our buttons?

There are two buttons on this page that we have used in other steps. Since we are trying to encapsulate this entire page in our class we will need to add methods to handle them as well. What should we call them? I have often seen people name them after the label of the button. I prefer to name them after the function they are performing and that we want to test. Again, we prefer to name things after WHAT they are doing, as opposed to HOW they get it done.

```
  def proceed_to_checkout
    @browser.button(:value => 'Complete the Adoption').click
  end

  def continue_shopping
    @browser.button(:value => 'Adopt Another Puppy').click
  end
```

And the corresponding step definitions:

```
When /^I click the Complete the Adoption button$/ do
  @cart.proceed_to_checkout
end

When /^I click the Adopt Another Puppy button$/ do
  @cart.continue_shopping
end
```

This seems like a lot of additional code. Soon we'll see what we can do to reduce the amount of code we have to write in order to achieve the same level of encapsulation and greater flexibility.

Setting Values on an Object

Ruby allows us to add an equal sign to the end of method names. When we call the methods we can leave a space before the equal sign to make it look like we are performing a simple assignment. Let's take a look at an example.

```
1     def address=(an_address)
2       @browser.text_field(:id => "order_address").set(an_address)
3     end
4
5     # Can be called like this
6     @page.address = "123 Main Street"
```

This makes the code easier to read and write. Let's see how we might use this in a Page Object.

Checking out

Armed with the ability to easily set values it is time to create a Page Object for the Checkout page. On this page we have a few text fields, a select list and a button. Let's create a new file named `checkout_page.rb` in the `pages` directory give it a try.

```
1     class CheckoutPage
2
3       def initialize(browser)
4         @browser = browser
5       end
6
7       def name=(name)
8         @browser.text_field(:id => "order_name").set(name)
9       end
10
11      def address=(address)
12        @browser.text_field(:id => "order_address").set(address)
13      end
14
15      def email=(email)
16        @browser.text_field(:id => "order_email").set(email)
17      end
18
19      def pay_type=(pay_type)
```

```
20          @browser.select_list(:id => "order_pay_type").select(pay_type)
21        end
22
23        def place_order
24          @browser.button(:value => "Place Order").click
25        end
26      end
```

Our goal is to make the step definitions as simple and intuitive as possible. We know that we go to the checkout page when we click the "Complete the Adoption" button. Let's take a look at that step and the others related to this page:

```
1       When /^I click the Complete the Adoption button$/ do
2         @cart.proceed_to_checkout
3         @checkout = CheckoutPage.new(@browser)
4       end
5
6       When /^I enter "([^"]*)" in the name field$/ do |name|
7         @checkout.name = name
8       end
9
10      When /^I enter "([^"]*)" in the address field$/ do |address|
11        @checkout.address = address
12      end
13
14      When /^I enter "([^"]*)" in the email field$/ do |email|
15        @checkout.email = email
16      end
17
18      When /^I select "([^"]*)" from the pay with dropdown$/ do |pay_type|
19        @checkout.pay_type = pay_type
20      end
21
22      When /^I click the Place Order button$/ do
23        @checkout.place_order
24      end
```

This seems simple and intuitive but I think we can make it easier. Let see how in the next section.

Introducing the PageObject gem

So far we have taken our two most complicated pages and moved all of the page access code into page object classes. We haven't really reduced the amount of code we had to write. If anything we

have increased the amount of code. Also, the changes we made will still cause us to search through the corresponding page object class to make changes when the actual web page changes. To help us with this I would like to introduce another Ruby gem.

The *PageObject*[66] gem was designed to make it easier to define page objects and easier to make changes as your site changes. One nice feature of the gem is that it works with both *Watir* and *Selenium*[67]. The gem is very large and has a lot of capabilities and will therefore take a while to learn. We'll start by introducing some of the most basic capabilities here and build on that understanding as we progress.

The way the *PageObject* gem works is you call some methods describing what elements are on our page and then it generates methods for you. These methods are very similar to the *Watir* methods we used before with the addition of a name. The *PageObject* gem will use those names during the method generation. Let's look at an example.

If we have a login screen with a username and password text field as well as a login button we might create a class like this:

```
1    class LoginPage
2      include PageObject
3
4      text_field(:username, :id => "user_id")
5      text_field(:password, :id => "user_password")
6      button(:login, :value => "Login")
7
8    end
```

In this example we called a simple method for each element we wished to interact with. Two immediate things to notice are we include the `PageObject` module on the second line and we do not need to create an `initialize` method as the gem already adds one for us.

Let's examine the first call in detail to see what it is doing. First of all we are passing two arguments. The first parameter is the name we want to give the text field and the second parameter is the locators needed by *PageObject* to find the element on the page. When we call this method *PageObject* will write four additional methods for us.

[66] https://github.com/cheezy/page-object
[67] http://seleniumhq.org

```
    def username
      @browser.text_field(:id => "user_id").value
    end

    def username=(username)
      @browser.text_field(:id => "user_id").set(username)
    end

    def username_element
      @browser.text_field(:id => "user_id")
    end

    def username?
      @browser.text_field(:id => "user_id").exists?
    end
```

The first method returns the value contained within the text field. The second method sets a value in the text field. The third method returns the text field element. The fourth method informs us if the text field exists. That is a lot of code to be generated from one simple line. That is part of the beauty and power of *PageObject*.

Let's make this a little more real. Below you see the `CheckoutPage` class rewritten using *PageObject*.

```
    class CheckoutPage
      include PageObject

      text_field(:name, :id => "order_name")
      text_field(:address, :id => "order_address")
      text_field(:email, :id => "order_email")
      select_list(:pay_type, :id => "order_pay_type")
      button(:place_order, :value => "Place Order")

    end
```

This is much shorter and easier to maintain. As we said earlier, we always want to know where to go and what to change as the page evolves. Using *PageObject* helps define where we make changes. For example, if we assumed that something on the screen mockup was a button but it was later implemented as a link styled to look like a button. To adapt to this change we would only have to change the call to `button` to a call to `link`. Everywhere else in the test suite we are simply refering to 'place_order' so it would not need to change. If we discovered that a developer used a different id than what we had in the script we would simple change it at the top of the class. This makes it simpler and that is what we are striving for.

Since the new `CheckoutPage` was such a success let's see what happens when we introduce *PageObject* to our `ShoppingCartPage` object.

```
1   class ShoppingCartPage
2     include PageObject
3
4     NAME_COLUMN = 1
5     SUBTOTAL_COLUMN = 3
6     LINES_PER_PUPPY = 6
7
8     button(:proceed_to_checkout, :value => "Complete the Adoption")
9     button(:continue_shopping, :value => "Adopt Another Puppy")
10    table(:cart, :index => 0)
11    cell(:cart_total, :class => "total_cell")
12
13    def name_for_line_item(line_item)
14      table_value(line_item, NAME_COLUMN)
15    end
16
17    def subtotal_for_line_item(line_item)
18      table_value(line_item, SUBTOTAL_COLUMN)
19    end
20
21    private
22
23    def table_value(lineitem, column)
24      row = (lineitem.to_i - 1) * LINES_PER_PUPPY
25      cart_element[row][column].text
26    end
27  end
```

Notice that I combined the two private methods into a single method and am using the generated table method. Overall this version of the page object is a dramatic simplification of the code.

Arrays & Hashes

Before we continue our exploration of the *PageObject* gem I thought it would be a good idea to spend a little time with a couple of important Ruby types. They are `Array` and `Hash`. The purpose of these collections is to hold a group of objects. The types of objects they can hold doesn't matter. Let's look at a couple of examples on how they can be used.

Function	Array	Hash
Create empty	[]	{}
Assign a value	collection[0] = 'value'	collection['key'] = 'value'
Retrieve a value	retrieved = collection[0]	retrieved = collection['key']

At this point the only difference between the two types are the way you reference the items they hold. With an Array you use an index and with a Hash you use a key. Let's look at a little more code for each type to learn a little more.

Array

Let's look at some example Array code and then explain some interesting aspects.

```
a = ["a", "e", "i", "o", "u"]
a[0]                 # First element is "a"
a[-1]                # Last element is "u"
a[-2]                # Second to last element is "o"
a[1,2]               # ["e", "i"] start at pos 2 for 2 elements
a[2..-1]             # ["i", "o", "u"] start at pos 3 until end

a.each {|x| print x} # prints "aeiou"
a.each_index {|x,i| puts "#{x} is at position #{i}"}
```

Arrays store their objects and provide access via a zero based index. You can access the first item in the Array with the index 0. You can start accessing elements from the end of the Array by using negative numbers. You can retrieve the last item in an Array using a -1 index. You can also retrieve a group of objects from the Array by using the substring notation of [1,2] or the range notation of [1..3]. Finally, you can iterate through all of the objects in an Array by using one of the each methods.

Hash

A Hash is a collection that stores a set of key/value pairs, like the names and phone numbers in a phone book. Each name is a key, a way to look up the value, and each phone number is a value you want to look up. Let's look at some code that uses a Hash.

```
h = {:one => 1, :two => 2, :three => 3}
h = {one: 1, two: 2, three: 3}    # Ruby version 1.9 or higher
h[:one]                           # 1

h.each_key {|k| print k}
h.each_value {|v| print v}
h.each {|k,v| puts "#{k} has a value of #{v}"}
```

A Hash is simply a container to hold key and value pairs. It is a very common practice to use symbols as the key. In fact, it is so common that starting with Ruby 1.9 there is a shorthand way of creating a Hash with symbol keys as demonstrated on the second line above. There are several forms of each methods that can be used to iterate through the collection.

Enumerable

Enumerable[68] is a Ruby module that provides the ability to perform numerous operations on a collection. You really should spend some time looking at the capabilities provided by this module. You will use them frequently. They provide many shortcuts for searching and manipulating collections, which has the effect of simplifying the code you write. Here are a couple of examples:

```
1    [5,3,9,7,1].sort                        # [1,3,5,7,9]
2    [1,3,5,7,9].find { |x| x > 5 }          # 7
3    [1,3,5,7,9].find_index { |x| x > 5 }    # 3
4    [1,3,5,7,9].find_all { |x| x > 5 }      # [7,9]
5    [1,3,5,7,9].reject { |x| x > 5 }        # [1,3,5]
```

The `find` series of methods will pass each entry to a block and will return the first value in which the block returns true. The one exception to this is the `find_all` method which will return all entries with which the block returns true. The `reject` method does the opposite; it will return all of the entries in which the block returns false.

In the *PageObject* gem the `Table`, `TableRow`, `OrderedList` and `UnorderedList` classes all include the `Enumerable` module. Let's look at how we might use this with the `Table` class:

```
1    table(:report, :id => 'report')
2
3    # Report has city in first column and population is second
4    def population_for(city)
5      row = report_element.find {|r| r[0] == city}
6      row[1]
7    end
```

If this example we declare a table and give it the name report. In the method we access the table element and call the `Enumerable find` method to locate the correct row. When we have the row we simply return the second column.

Convert all pages to use page objects

There are only two web page we have not created page objects for. They are the initial landing page for the site and the puppy details page. Let's go ahead and complete the refactoring to page objects by beginning with the details page.

The details page is the simplest page on the site with only a single button. Here's the entire code for this page.

[68] http://ruby-doc.org/core-1.9.3/Enumerable.html

```
class DetailsPage
  include PageObject

  button(:add_to_cart, :value => 'Adopt Me!')

end
```

In order to put it to use we have to determine where to create our object. We know that we go to this page when we click either the first or second "View Details" button on the first page so we will need to update the code in two step definitions in order to use it. Here are the three step definitions that were modified.

```
When /^I click the first View Details button$/ do
  @browser.button(:value => 'View Details', :index => 0).click
  @details = DetailsPage.new(@browser)
end

When /^I click the second View Details button$/ do
  @browser.button(:value => 'View Details', :index => 1).click
  @details = DetailsPage.new(@browser)
end

When /^I click the Adopt Me button$/ do
  @details.add_to_cart
  @cart = ShoppingCartPage.new(@browser)
end
```

That was very simple and now we are ready for the landing page, or instead let's call it `HomePage`.

The home page of the site is where a user clicks a "View Details" button. But which button? The first one or the second one? The truth is we do not know what button we want to click ahead of time so we will need to make our decision as the script is running. What is the best way to do this?

We could define two buttons using page object and then click the appropriate one in our step definitions. The page would look like this:

```
class HomePage
  include PageObject

  button(:first_puppy, :value => 'View Details', :index => 0)
  button(:second_puppy, :value => 'View Details', :index => 1)
end
```

Although this would work what would happen if we wanted to click the third of fourth button? There might be a simpler way. Let's fall back to our technique of writing the code we wish we had in the step definitions and see where that leads us.

```
Given /^I am on the puppy adoption site$/ do
  @browser.goto "http://puppies.herokuapp.com"
  @home = HomePage.new(@browser)
end

When /^I click the first View Details button$/ do
  @home.select_puppy_number 1
  @details = DetailsPage.new(@browser)
end

When /^I click the second View Details button$/ do
  @home.select_puppy_number 2
  @details = DetailsPage.new(@browser)
end
```

After writing the code we wish we had it looks like we need to add a new method to the `HomePage` class. This method will need to dynamically select the appropriate "View Details" button and click it. But how can we do this?

We're in luck. *PageObject* has the ability to not only define elements ahead of time but to also find them when the script is running. For each method we can use to declare an element is an equivalent method that ends with `_element` for locating them when the script is running. Instead of using a `button` method we will just use a `button_element` method. This method does not have the name parameter since it will not be generating any methods for us. Let's take a look at the updated `HomePage` class.

```
class HomePage
  include PageObject

  def select_puppy_number(num)
    button_element(:value => 'View Details', :index => num - 1).click
  end
end
```

With this addition we are finally converted over to using the PageObject gem. There still is a little more cleanup to perform and we'll cover that in the next few sections.

Going to the Factory

One thing that still is very messy is the creation of all of the instance variables to represent the page objects. It can become very difficult to know where to create an instance variable especially if you can arrive at a step that needs that instance variable from a variaty of paths. Imagine that you are working on a large project - one that has thousands of step definitions. How hard will it be to keep track of all of the page object instance variables?

Lucikly the *PageObject* gem has a solution for this problem. In fact, we have already made this solution available to our project. If you look at the `env.rb` file in the `features/support` directory you will see the following:

```
1    require 'rspec-expectations'
2    require 'page-object'
3
4    World(PageObject::PageFactory)
```

We have seen the require statement before but what is this `World` thing?

Extending the Cucumber World

When Cucumber starts it performs the following tasks:

- Creates a class named World
- Loads all of the files in the support directory
- Adds all of the step definitions to the World class
- Runs the Scenarios by executing the steps as methods on World

One of the ways you can extend Cucumber is by adding new methods to the `World` class yourself. You do this by passing `Modules` to the `World` method. This will make these methods available to all of your step definitions. The second line in the code above is adding the two methods from the `PageObject::PageFactory` module to `World`.

Back to the Factory

`PageFactory` is a module that is a part of the *PageObject* gem that adds two methods for navigating to or selecting pages to work with. Let's look at a couple of examples.

Let's say we have a login page and wish to log into the system:

```
class LoginPage
  include PageObject

  page_url("http://mysite.com/login")  # required for visit
  text_field(:username, :id => "user_id")
  text_field(:password, :id => "user_password")
  button(:login, :value => "Login")

  def login_with(username, password)
    self.username = username
    self.password = password
    login
  end
end
```

In my step definitions I could do the following:

```
visit(LoginPage) do |page|
  page.username = 'cheezy'
  page.password = 'secret'
  page.login
end
```

This code will cause the browser to go to the login page, fill in the text fields, and submit the page. How does it know what page to go to? The `page_url` call in the page object informs `PageObject` of where the page is located and is required in order to use `visit`. Here is another way to perform the same action:

```
visit(LoginPage).login_with 'cheezy', 'secret'
```

In this case we are taking advantage of the fact that the `visit` method returns the page and we are calling the `login_with` method on that return value. What can we do if we are already on a page? We can use the `on` method:

```
on(SomePage).do_something
```

Both forms of this method are available as well.

The final item I'll mention here is that the `PageFactory` sets an instance variable `@current_page` each time you use a new page. This instance variable can be used in situations where you might have the same activity that needs to be performed on multiple pages like checking the text on a page.

Let's go back to our step definitions and update them to use the `PageFactory` methods. The first step will use the `visit` method and the remainder will use `on`. Don't forget to add the `page_url` call to your `HomePage`. You should also remove the creation of the page objects and assignment to instance variables. For example - this step:

```
When /^I click the first View Details button$/ do
  @home.select_puppy_number 1
  @details = DetailsPage.new(@browser)
end
```

should change to this:

```
When /^I click the first View Details button$/ do
  on(HomePage).select_puppy_number 1
end
```

Transforming a line item

Extending the `World` class is one way to extend the functionality of Cucumber. In this section we will look at another. In a few of our step definitions we have specified that we wanted to use a specific "line item". In those steps we had to always take the `String` Cucumber passed and convert it to a number using the `to_i` method. If we want Cucumber to automatically convert it for us we can use a `Transformation`. Let's look at that now.

The first thing we need to do is create a new file in the `support` directory named `transformations.rb`. The file name doesn't matter; I just like to collect all of my transformations in the same file. On a typically large project you might see up to five or six transformations. In this file we will create our first transformation:

```
Transform /^line item (\d+)$/ do |line_string|
  line_string.to_i
end
```

This structure should look familiar. It is very similar to step definitions except it begins with the word `Transform` instead of `Given`, `When`, or `Then`. To break it down completely, what we are saying with this code is find a capture group of a step definition (portion of a step definition surrounded by parentheses) that contains "line item" plus a number and perform the following action on the value of the number. In order for this *Transformation* to find the appropriate step definitions we will need to move the opening parentheses to surround the entire match.

```
    Then /^I should see "([^"]*)" as the name for (line item \d+)$/ do |name, lin\
e_item|
```

Now we no longer need to convert the line_item parameter to a number since the Transformation takes care of this for us.

This technique is very useful when you have values that need to be converted and you wish to move the logic from multiple steps to one Transformation. For example, imagine you need to perform a lot of date handling. It would be nice to say things like today, tomorrow, or yesterday and have a *Transformation* object create a date object.

One More Thing

There is one final cleanup I'd like to do at this time. Let's look at a portion of one of our Scenarios:

```
    When I click the first View Details button
    And I click the Adopt Me button
    Then I should see "Brook" as the name for line item 1
```

The correlation between "the first View Details button" and "Brook" is not clear. I'd much rather say:

```
    When I click the View Details button for "Brook"
    And I click the Adopt Me button
    Then I should see "Brook" as the name for line item 1
```

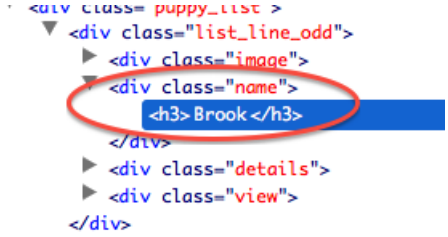
Let's take a look at what it would take to make this happen.

The Simplest Thing

The original extreme programming team came up with a phrase "Do the simplest thing that could possibly work". The idea is to implement the simplest code possible to get the job done and only evolve to a more complex solution if and when it becomes necessary.

There are several complex ways to solve the problem of finding the correct button to click using the name. We could use *xpath* to locate the button. We could somehow navigate up and down the DOM using the parent/child relationship of the elements on the page. I suggest we start with a much simpler solution and migrate to one of these more complex solutions when we determine that our simple solution is no longer sufficient. Using our page object class we have the ability to change the internal implementation without the consumers of the classing knowing. This is called encapsulation.

As I was looking at the structure of the Home Page I notice that the names were nested inside of a `div` that had a `class` of 'name'.



Puppy's name in FireBug

The simplest solution I can think of is to get all of the names into an `Array` and all of the buttons into another `Array`. When a name is passed into the method we could get the index of that name in the name `Array` and use that index to find the desired button. In order to do this we will have to use the plural version of the declarations - `divs` and `buttons`. Let's take a look at the updated `HomePage` class:

```
1   class HomePage
2     include PageObject
3
4     page_url "http://puppies.herokuapp.com"
5
6     divs(:name, :class => 'name')
7     buttons(:view_detail, :value => 'View Details')
8
9     def select_puppy(name)
10      index = index_for(name)
11      view_detail_elements[index].click
12    end
13
14    private
15
16    def index_for(name)
17        name_elements.find_index { |the_div| the_div.text == name }
18    end
19  end
```

On lines 6 and 7 we are using the plural version of the declaration. In each case it will create a method named "[name]_elements" where [name] is replaced by the name we gave in our declaration.

On line 17 we are using the `find_index` method we mentioned back in the section that discussed `Enumerable`. We are passing a block to the method call and it will return the index of the first block that returns true. In our case we are matching the `text` of the div passed in as the parameter with the name of the puppy. This will give us the index we need.

In the `select_puppy` method that begins on line 9 we are calling the private `index_for` method to get the index and then using it to select the proper button. The final step is to call the `click` method on that button.

With the final changes in place our step definitions now look like this:

```
1   Given /^I am on the puppy adoption site$/ do
2     visit(HomePage)
3   end
4
5   When /^I click the View Details button for "([^"]*)"$/ do |name|
6     on(HomePage).select_puppy name
7   end
8
9   When /^I click the Adopt Me button$/ do
10    on(DetailsPage).add_to_cart
11  end
12
13  When /^I click the Complete the Adoption button$/ do
14    on(ShoppingCartPage).proceed_to_checkout
15  end
16
17  When /^I click the Adopt Another Puppy button$/ do
18    on(ShoppingCartPage).continue_shopping
19  end
20
21  When /^I enter "([^"]*)" in the name field$/ do |name|
22    on(CheckoutPage).name = name
23  end
24
25  When /^I enter "([^"]*)" in the address field$/ do |address|
26    on(CheckoutPage).address = address
27  end
28
29  When /^I enter "([^"]*)" in the email field$/ do |email|
30    on(CheckoutPage).email = email
31  end
32
33  When /^I select "([^"]*)" from the pay with dropdown$/ do |pay_type|
34    on(CheckoutPage).pay_type = pay_type
35  end
36
37  When /^I click the Place Order button$/ do
```

```
38        on(CheckoutPage).place_order
39      end
40
41      Then /^I should see "([^"]*)"$/ do |expected|
42        @current_page.text.should include expected
43      end
44
45      Then /^I should see "([^"]*)" as the name for (line item \d+)$/ do |name, lin\
46  e_item|
47        on(ShoppingCartPage).name_for_line_item(line_item).should include name
48      end
49
50      When /^I should see "([^"]*)" as the subtotal for (line item \d+)$/ do |subto\
51  tal, line_item|
52        on(ShoppingCartPage).subtotal_for_line_item(line_item).should == subtotal
53      end
54
55      When /^I should see "([^"]*)" as the cart total$/ do |total|
56        on(ShoppingCartPage).cart_total.should == total
57      end
```

5. More Puppies

High level tests

Many of the *Scenarios* we have written so far suffer from a problem. They are too verbose. What I mean by this is they specify every single keystroke even when it is not important for the actual test. They are covering way too much HOW (implementation details) in order to test WHAT (behavior) our product stakeholders want in their features. They are literally bogging down in noisy details. Let's take a look at the original *Scenario* where we adopted a puppy.

```
1   Scenario: Adopting one puppy
2     When I click the View Details button for "Brook"
3     And I click the Adopt Me button
4     And I click the Complete the Adoption button
5     And I enter "Cheezy" in the name field
6     And I enter "123 Main Street" in the address field
7     And I enter "cheezy@example.com" in the email field
8     And I select "Credit card" from the pay with dropdown
9     And I click the Place Order button
10    Then I should see "Thank you for adopting a puppy!"
```

What specifically are we verifying? Lots and lots of HOW, for very little WHAT, that's what! The only thing we are verifying is that the "Thank you" message is displayed. Does it really matter what name I enter in the name field? What about the puppy I select for adoption?

The truth is that none of the data I enter or select on the screens has any impact on the outcome for this particular *Scenario*. That is also the case with most *Scenarios*. Although we often have to enter a lot of information in order to complete a *Scenario* the majority of the information is not directly relevant for the thing we are testing. The amount of feature-level behavior we wish to verify is typically very small.

A major part of writing good *Scenarios* is ensuring they are written well and easy to understand. Imagine if we have to traverse through multiple pages to get to the page we plan to test and each page requires us to enter a lot of data. If we specify all of the typing and clicks in our *Scenario* there would be a lot of brittle, noisy HOW-related details. It would be very easy to see how one could miss the simple behavior we wish to describe in all of those details.

Below is the same *Scenario* as above except all of the unnecessary details are hidden:

```
  Scenario: Thank you message should be displayed
    When I complete the adoption of a puppy
    Then I should see "Thank you for adopting a puppy!"
```

Hey! That's just saying exactly WHAT I want the feature to do, and barest minimum of HOW I want to verify it! Cool. This *Scenario* is much easier to understand and we can clearly see what is being specified. We still need to perform all of the activities necessary to adopt a puppy even though we did not specify them in the Scenario. The next few sections will provide some techniques you can use to enable you to specify your *Scenarios* at a right level of granularity and still have all of the detailed steps take place.

Inline tables

In chapter four we saw how we can build up a table of *Examples* to provide the data for a *Scenario Outline*. It is also possible to have a table of data that is a part of an individual step. Let's see how one could place an order using a table of data. We'll make a modified version of the *Scenario* from the last section.

```
  Scenario: Adopting a puppy using a table
    When I click the View Details button for "Brook"
    And I click the Adopt Me button
    And I click the Complete the Adoption button
    And I complete the adoption with:
    | name   | address         | email              | pay_type |
    | Cheezy | 123 Main Street | cheezy@example.com | Check    |
    Then I should see "Thank you for adopting a puppy!"
```

In this case we've collapsed five steps into one. Go ahead and write this *Scenario* and generate the one new step definition. In this case we will be passing all of our test data to the page in one step. When you generate your step definition you will notice that a table variable is passed into your step. For our purposes we can just call a `hashes` method on `table` which will return an `Array` of `Hashes`. Each entry in the `Array` represents one row of data with the key being the table header and the value being the data in the table. Since we only have one row of data we will just call the `first` method on this `Array` to get the single `Hash` containing our data.

We have two options for the implementation. The first option is to use the methods generated by PageObject like this:

```
When /^I complete the adoption with:$/ do |table|
  data = table.hashes.first
  on(CheckoutPage) do |page|
    page.name = data['name']
    page.address = data['address']
    page.email = data['email']
    page.pay_type = data['pay_type']
    page.place_order
  end
end
```

Although this step definition works fine most of the code seems to be a little out of place in the step definition. The knowledge of how to take a group of data, complete the form, and submit it would seem to be better place in the page object itself. Let's take a look how that might work.

Here is a new method on `CheckoutPage`:

```
def checkout(data)
  self.name = data['name']
  self.address = data['address']
  self.email = data['email']
  self.pay_type = data['pay_type']
  place_order
end
```

The first thing you might notice is that we are using the keyword `self` in our new method. The reason for this is that equals methods (one that end with an equal sign) are required to have a receiver so they are not mistaken for local variable assignment. `self` represents the current object and therefore when we call `self.name =` we are really saying call the `name=` method on this same object.

With this method on our page object our step definition can be simplified to this:

```
When /^I complete the adoption with:$/ do |table|
  on(CheckoutPage).checkout(table.hashes.first)
end
```

Do not overuse tables

Tables are nice and do help us consolidate multiple steps into one but they also make things a little harder to read. I have gone to work with teams where they have used inline tables in nearly every *Scenario*. Their *Scenarios* were hard to read and the stakeholders had already decided that they did not want to be involved. Use you best judgement on when you believe it adds to the overall Scenario and make sure you do not overuse them to the point where it takes away from what you are trying to say.

Default data

It is fairly common for your *Scenarios* to require a lot of data in order to complete a successful run. In the last section we discussed how to use tables to create higher level tests. In that *Scenario* it really didn't matter if we used the name "Cheezy" or "Mickey Mouse"; the outcome would be the same. In fact, none of the data we provide made any difference on the outcome. As long as we select a puppy and complete the checkout form we see the thank you message.

We should ask ourselves why do we need to provide all of this data if it is not important to what we are testing. Does providing this data really add clarity or is it distracting? Does specifying every click necessary to traverse through the system add to our understanding? Does it make the specification more complete?

If we were able to rephrase the *Scenario* from the last section to state the exact essence of what we were trying to specify it would be this:

```
1   Scenario: Thank you message should be displayed
2     When I complete the adoption of a puppy
3     Then I should see "Thank you for adopting a puppy!"
```

This is much cleaner and easier to understand. It specifies the exact thing we are trying to describe - a thank you message should be displayed when the adoption is completed.

In order to implement the *Scenario* above we still need to provide the information necessary to fill in and submit the pages of our application. But where will it come from? This is where Default Data comes in.

Default Data is a widely used pattern in the testing community. The pattern is implemented by providing a set of data that can be used to generically traverse through the application but at the same time allowing you to override any data specific to your context. For example, take the last *Scenario*. In that *Scenario* we should just be able to use the Default Data to complete all of the pages and the validate the message. It required us to use a "Credit card" then we should be able to specify it override the default value and use "Credit card". Let's see how we could implement this in our CheckoutPage.

```
1     class CheckoutPage
2       include PageObject
3
4       DEFAULT_DATA = {
5         'name' => 'cheezy',
6         'address' => '123 Main Street',
7         'email' => 'cheezy@example.com',
8         'pay_type' => 'Purchase order'
9       }
```

```
10
11        text_field(:name, :id => "order_name")
12        text_field(:address, :id => "order_address")
13        text_field(:email, :id => "order_email")
14        select_list(:pay_type, :id => "order_pay_type")
15        button(:place_order, :value => "Place Order")
16
17        def checkout(data = {})
18          data = DEFAULT_DATA.merge(data)
19          self.name = data['name']
20          self.address = data['address']
21          self.email = data['email']
22          self.pay_type = data['pay_type']
23          place_order
24        end
25      end
```

There are several new things here to discuss. The first is the `Hash` near the top of the class. It simply provides the default data that will be used by the page. The way this happens is that the first line of the `checkout` method merges the data passed to the method with the default data of the page. The merge simply tries to match up a key and if a match is found it will update the corresponding value.

The other new thing here is the modified parameter passed to our `checkout` method - `data = {}`. This is Ruby's way of specifying a default parameter to a method. If we provide a parameter when we call this method then it is passed on through. If we call the method without passing a parameter the default will be used which in this case is an empty Hash. The effect of using the empty `Hash` is to just use all of the default data.

If you are using PageObject 0.5.3 or higher there is an even simpler way to populate your page with a `Hash` of data. A new method has been added named `populate_page_with` which takes a `Hash`. It will match up the keys from the `Hash` with the names you provided for the elements when you declared them on the page. All values must be Strings except for `Checkboxes` and `Radio Buttons` must be true or false. Let's look at our `checkout` method if we use this instead.

```
1       def checkout(data = {})
2         populate_page_with DEFAULT_DATA.merge(data)
3         place_order
4       end
```

The nice thing about this `populate_page_with` method is that if there is an entry in the `Hash` that does not have a corresponding control that is present and visible on the page it will simply ignore it. This works great for situations in which you have different elements on the page based upon context. Your default data can have the total superset of data and this method can determine how to match up the appropriate data values with the element that exist in the current context.

Let's write two new variations of our *Scenario* from the last section to see the ways we can use this updated method.

```
  Scenario: Adopting a puppy using partial default data
    When I click the View Details button for "Brook"
    And I click the Adopt Me button
    And I click the Complete the Adoption button
    And I complete the adoption using a Credit card
    Then I should see "Thank you for adopting a puppy!"

  Scenario: Adopting a puppy using all default data
    When I click the View Details button for "Brook"
    And I click the Adopt Me button
    And I click the Complete the Adoption button
    And I complete the adoption
    Then I should see "Thank you for adopting a puppy!"
```

Here's the step definitions:

```
    When /^I complete the adoption using a Credit card$/ do
      on(CheckoutPage).checkout('pay_type' => 'Credit card')
    end

    When /^I complete the adoption$/ do
      on(CheckoutPage).checkout
    end
```

I think we have achieved the goal of setting up and specifying useful, sensible default data. We can now individually set the values on the page using the methods generated by PageObject. We can use the default data by calling the `checkout` method passing no parameters. We can use partial or no default data by passing in a `Hash` that contains the data we wish to use.

The only additional page we have in our system where we have to provide data is the `HomePage`. Providing default data for this page is as simple as providing a default parameter for the adopt method.

```
    def select_puppy(name = 'Brook')
      index = index_for(name)
      button_element(:value => 'View Details', :index => index).click
    end
```

With this in place we can now get back and add the *Scenario* we introduced at the beginning of this section and add the missing step definition.

```
When /^I complete the adoption of a puppy$/ do
  on(HomePage).select_puppy
  on(DetailsPage).add_to_cart
  on(ShoppingCartPage).proceed_to_checkout
  on(CheckoutPage).checkout
end
```

Randomizing the data

Let's take this one step further. The name, address, and email fields can really be anything as long as they pass the validations. In fact, it might be a little more interesting if the data is truly random. Let's introduce a gem to help us do just that. Open your `Gemfile` and add the following:

```
gem 'faker'
```

Next run the `bundle` command in the `test_puppies` directory. Once this is complete open the `env.rb` file and add `require 'faker'`. We are set to use the *faker* gem. Replace your `DEFAULT_DATA` with the following:

```
DEFAULT_DATA = {
  'name' => Faker::Name.name,
  'address' => Faker::Address.street_address,
  'email' => Faker::Internet.email,
  'pay_type' => 'Credit card'
}
```

Now each time the class is used it will provide some different random data for the default data. How cool is this! *Faker* is a gem that provides random data that you can use in your tests. In order to understand the full power of this gem I would suggest looking at the documentation[69].

Bundler and Gemfile

Bundler[70] is a ruby gem that we can use to help manage our project dependencies. It was added to our project when we ran `testgen` to generate the initial shell. The way it knows what gems our project relies on is by looking at a file named *Gemfile*[71] located in the root of your project. When you execute the `bundle` command it will build out a list of dependencies, install them locally, and create a new file named *Gemfile.lock* that shows how the dependencies were resolved. Going forward, when you need to add a new gem to your project you will simply add a new entry to the *Gemfile* and run `bundle install`.

[69] http://rubydoc.info/github/stympy/faker/master/frames
[70] http://bundler.io
[71] http://bundler.io/v1.3/gemfile.html

Moving our default data

Having the Default Data inside the page object can be somewhat limiting - especially if you wish to provide different sets of default data at runtime. Fortunately there is a very simple way to move the data out of the class. In order to do this we need to introduce a new gem - *data_magic*. Add this entry to your *Gemfile* and run `bundle install`..

```
1     gem 'data_magic'
```

The next thing we need to do is configure *data_magic* telling it where to find the files that contain the default data. We'll do this by adding the following code to your `features/env.rb` file.

```
1     require 'data_magic'
```

These entries require the *data_magic* gem making it available to us and then informs *data_magic* where it can find the files it needs. Go ahead and create a new top-level directory in our project named 'config' and then create yet another directory under it named `data`. The `config/data` directory is the default directory used by *data_magic* should be at the same level as the `features` directory. Next, create a new file named `default.yml` and add the following to it:

```
1     checkout_page:
2       name: Cheezy
3       address: 123 Main Street
4       email: cheezy@example.com
5       pay_type: Check
```

This file format is called *yaml*[72]. It is a very simple way for defining structured data. *data_magic* looks for all data associated with a top level key and deliveres it to your page objects as a *Hash*. In this case our top level key is `checkout_page`. Please note that all of the lines under the `checkout_page:` key are indented. This is important because it is how yaml identifies element nesting. Let's go ahead and update the `CheckoutPage` to use *data_magic*.

[72] http://en.wikipedia.org/wiki/YAML

```
class CheckoutPage
  include PageObject
  include DataMagic

  text_field(:name, :id => "order_name")
  text_field(:address, :id => "order_address")
  text_field(:email, :id => "order_email")
  select_list(:pay_type, :id => "order_pay_type")
  button(:place_order, :value => "Place Order")

  def checkout(data = {})
    populate_page_with data_for(:checkout_page, data)
    place_order
  end
end
```

We made three changes to our `CheckoutPage` class. First of all we included the `DataMagic` module. Next, we completely removed the `DEFAULT_DATA` hash from our page. Finally we called the method `data_for` in our `checkout` method. The line reads `populate_page_with data_for :checkout_page`. That reads nicely. The first parameter to `data_for` is the key of the data to retrieve. The second parameter is an optional `Hash` that is merged with the data from *data_magic*.

By default *data_magic* will look for a file named `default.yml`. You can inform the gem to use another file by simply calling the `load` method like this:

```
DataMagic.load('another_file.yml')
```

So now we have completely externalized our default data and we have the ability at runtime to select different sets of default data depending on the requirements of the test. But can we randomize the data? Well, the answer is yes. *data_magic* has a large number of built-in methods (most of which are just wrappers around *faker*) that randomize our data. Let's work on the previous example:

```
checkout_page:
  name: ~full_name
  address: ~street_address
  email: ~email_address
  pay_type: ~randomize ['Check', 'Purchase order', 'Credit card']
```

As you can see we are calling a built-in method by simply beginning our data with the ~ symbol. Here is a table with the current set of built-in methods in the *data_magic* gem.

Method	Method	Method

full_name	state_abbr	yesterday(format='%D')
first_name	zip_code	month
last_name	country	month_abbr
name_prefix	company_name	day_of_week
name_suffix	catch_phrase	day_of_week_abbr
title	domain_name	3.days_from_today
email_address	url	3.days_ago
phone_number	user_name	characters(char_count=255)
cell_phone	randomize(1..1000)	words(number=5)
street_address	randomize(['foo', 'bar'])	sentence(min_workd_count=4)
secondary_address	mask('AAaa##')	sentences(sentence_count=3)
city	today(format='%D')	paragraphs(paragraph_count=3)
state	tomorrow(format='%D')	

In addition to using built-in methods there are two ways you can extend DataMagic. You can add your own keywords. The way you do this is by creating a `module` that contains public methods and pass it as a parameter to the `add_translator` method like this:

```
# define a module with methods that return the data we want
module Account
  def account_number
    # return the value to be displayed
    'blah'
  end
end

# In the env.rb file you can register this module
DataMagic.add_translator(Account)
```

After registering this new module you can now add `~account_number` to your *yml* files.

In addition to registering your own modules, you can also specify any Ruby code in the *yml* file. For example, if you wanted to have a field that always contained today's date you could simply do this:

```
  today: ~Date.today
```

Regular Expressions

Let's take a quick break from cucumber and talk about a related topic. We have been using regular expressions since we first touched cucumber but haven't pointed them out or explained their purpose. Let's start by understanding what they are and then we'll focus on how to use them. Wikipedia say "In computing, a regular expression provides a concise and flexible means

for "matching" (specifying and recognizing) strings of text, such as particular characters, words, or patterns of characters"[73]. Simply put it is a way of matching strings.

Let's look at three simple regular expressions. The first regular expression is any character. In other words, /abc/ will match "abc". /ZZZZ/ will match "ZZZZ". The next expression is `/^`. This means that the matching element starts with whatever follows this symbol. The next is `$/`. This means that the matching element ends with whatever proceeds this symdol. In other words, `/^This is what I want to match$/` will match "This is what I want to match". Now this should start to look familiar to you. Let's take a look at the last step definition we completed:

```
1     When /^I complete the adoption of a puppy$/ do
2       ...
3     end
```

What is really happening here is we are calling a method named `When` passing a regular expression and a block. The regular expression is `/^I complete the adoption of a puppy$/`. This is how cucumber matches the feature file steps to the step definitions file and knows what to execute.

Cucumber has something called Capture Groups. This is simply anything that is surrounded by parentheses. Whatever matches the capture group is placed into a variable passed to the block. Let's look at an example. In the previous chapter we had the following step:

```
1     When /^I enter "([^"]*)" in the address field$/ do |address|
```

The capture group is `([^"]*)` and the value found there is placed in the `address` parameter. We forced this to happen by placing the double quotes around the value in the feature file and cucumber took this as a clue to create the capture group. The truth is that there is really no need for the double quotes. If you remove them from both the feature file and step definition you will see that it works exactly the same.

The last capture group and regular expression `([^"]*)` is somewhat complex. Let's spend a little time and see what we can do to simplify it. Here are a few simple things to learn to help you write simple regular expressions for cucumber.

- The `.` character will match any character. For example `..` matches "at" and "on" but it doesn't match "off" since it is three characters.
- The `*` character means zero or more of the previous element so `ab*` matches "ab", "abb" and "a". `a.*` matches "a", "ab", "abb", "ac", etc.
- The `+` character means one or more of the previous element so `ab+` matches "ab", "abb" but does not match "a". `a.+` matches "ab", "abb", "ac" but does not match just "a".

[73] http://en.wikipedia.org/wiki/Regular_expression

- Character classes are any set of characters contained within `[]`. For example, `[0123456789]` matches any number. `[0-9]` is shorthand for this example. Another common character class is `[A-Za-z]` to represent any alpha character.
- There are shorthand expressions for character classes. Here are a few: `\d` is equal to `[0-9]`, `\w` is equal to `[A-Za-z0-9_]`, `\s` is equal to `[ \t\r\n\v\f]`.

With this knowledge we could have written

```
1     When /^I enter "([^"]*)" in the address field$/ do |address|
```

as

```
1     When /^I enter "(.+)" in the address field$/ do |address|
```

and we now know what was happening in this step definition

```
1     Then /^I should see "([^"]*)" as the name for line item (\d+)$/ do |name, lin\
2 e_item|
```

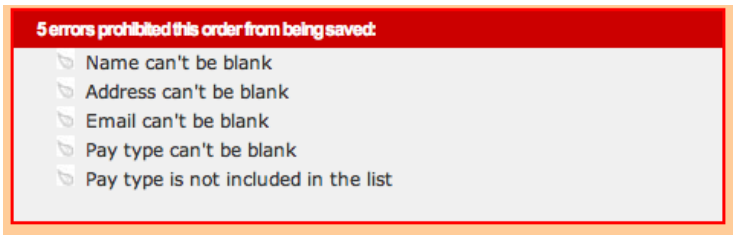
and know we could have written it as

```
1     Then /^I should see "(.+)" as the name for line item (\d+)$/ do |name, line_i\
2 tem|
```

Regular expressions can be very complicated and I have seen many people get lost in their power and inherent confusion. I find that a few simple rules will provide everything you need when it comes to using regular expressions in cucumber.

Edits

You may have noticed that error messages will display if you try to place an order on the last page leaving the form empty.



Errors on page

Let's write a simple *Scenario* to specify that the name field is a required field:

```
  Scenario: Name is a required field
    When I checkout leaving the name field blank
    Then I should see "Name can't be blank"
```

Based on what we have learned the step definitions should be very easy to write. We simply use the methods on the page objects that we created in previous steps.

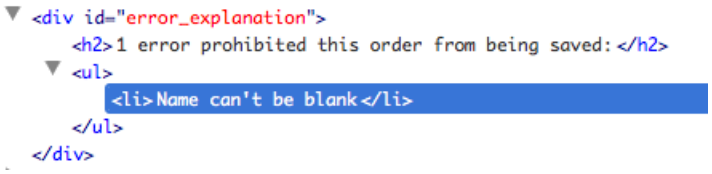
```
    When /^I checkout leaving the name field blank/ do
      on(HomePage).select_puppy
      on(DetailsPage).add_to_cart
      on(ShoppingCartPage).proceed_to_checkout
      on(CheckoutPage).checkout('name' => '')
    end
```

In this step definition we are simply telling our `CheckoutPage` to use all default data except for the name field for which we are passing an empty String.

The step I would like to focus on now is the second step.

```
    Then I should see "Name can't be blank"
```

This step will look for the provided text anywhere on the page. Often this is fine but in some cases we need to be a little more specific. In this case we are expecting the message to be an error message. On this page an error is displayed in an unordered list nested within a div.



Error Explanation Unordered List

The div has an id of `error_explanation`. Let's see what it would take to have our step definition look for the error message in the place where error messages reside on this page.

The challenge we have is that the unordered list does not have any way to identify it. We can say find the unordered list within a div with an id of error_explanation. Let's take a look at how we can do that.

```
    div(:error_div, :id => 'error_explanation')
    unordered_list(:error_messages) do |page|
      page.error_div_element.unordered_list_element
    end
```

The first line is fairly straight forward. We are simply identifying a div using the id. The next line is where it gets interesting. In this line we are not providing our basic identifiers. Instead we are passing a block which receives a local variable which is an instance of our page. On this page variable we are calling the method to retrieve the error div element defined on the proceeding line and then calling a method to find an unordered list within that div.

Let's talk about how this works. Each of the methods you can call on the class generates methods. One of the methods always generated is `<name>_element` where name is the name value you pass to the method. So our call to `div(:error_div, :id => 'blah')` will generate an `error_div_element` method. This method will return an object that represents the actual element on the page. Each object that represents an element also has an entire set of `<type>_element` methods which will find a new element nested within it. So our call to `page.error_div_element.unordered_list_element` finds the div element that we named `error_div` and then looks for an unordered list element nested within it. Each of the `<type>_element` methods has a default parameter of `:index => 0`. You can override this default parameter by supplying any identifier that is valid to find that type of element.

The final thing to do is update the *Scenario* and step definition to use this new capability. Let's update the *Scenario* to this:

```
  Scenario: Name is a required field
    When I checkout leaving the name field blank
    Then I should see the error message "Name can't be blank"
```

Notice how I change the last step to say that I should see the "error message". The updated step definition that uses our new ability looks like this:

```
    Then /^I should see the error message "([^"]*)"$/ do |msg|
      on(CheckoutPage).error_messages.should include msg
    end
```

All pages can have error messages

It is a common practice to build web site pages from reusable panels. Many sites have a common header, or footer, or perhaps a common navigation bar that appears on many of the pages within the site. For our example, it would be very common to find out that the way errors are displayed is in a panel reused throughout the entire site. In the puppy adoption application this is the case. Every place where we display errors we use the same unordered list nested within a div.

It would not be wise to have the code we just added to our `CheckoutPage` copied into all of the other pages where we need to verify error messages. But what can we do? What is the best way to have this code exist in only one place but use it in multiple pages? We will turn to our good friend `Module`.

I would like you to create a new file in the `pages` directory named `error_panel.rb` and create an empty `Module` named `ErrorPanel`. You will need to include `PageObject` and then move the error handling code from the `CheckoutPage` to this new module.

```
module ErrorPanel
  include PageObject

  div(:error_div, :id => 'error_explanation')
  unordered_list(:error_messages) do |page|
    page.error_div_element.unordered_list_element
  end
end
```

Next we need to require and include the module in our `CheckoutPage` class.

```
require_relative 'error_panel'

class CheckoutPage
  include PageObject
  include ErrorPanel

  ...
end
```

In each page where we want to validate error messages we just need to require the `error_panel` file and then `include` the `ErrorPanel` module. The final thing to do is go back and make the step definition generic so it will work on all pages.

```
Then /^I should see the error message "([^"]*)"$/ do |msg|
  @current_page.error_messages.should include msg
end
```

Custom Matchers

We have used a couple of the RSpec Matchers throughout our step definitions. Specifically we have used the `should include` and `should ==` matchers. It is very easy to write you own matchers. Let's spend a few minutes to write a matcher for the *Scenario* we completed in the last section.

We ended up with the following step definition:

```
Then /^I should see the error message "([^"]*)"$/ do |msg|
  @current_page.error_messages.should include msg
end
```

After writing the new RSpec Matcher we want to be able to say:

```
Then /^I should see the error message "([^"]*)"$/ do |msg|
  @current_page.should have_error_message msg
end
```

Let's start by declaring our matcher. Create a new file in the `support` directory named `matchers.rb` and add the following code:

```
RSpec::Matchers.define :have_error_message do |expected|
  match do |actual|
  end
end
```

The `define` method takes two parameters - a symbol to use as the name for the matcher and a block. The expected value that is passed into the block is the value that is to the right of the matcher when it is used. In the example above this would be the `msg` variable. Inside the block we make a call to the match method which also takes a block. The actual value passed to this block is the value that is to the left of the `should` call. In the example above it would be the page object. I often like to change those variables so it is clearer what they are.

The next step is to implement the match block. The block should return true if it matches and false otherwise. We are not able to use matchers within matchers so here so this is what I came up with:

```
RSpec::Matchers.define :have_error_message do |message|
  match do |page|
    page.error_messages.include? message
  end
end
```

There is one other optional thing we can do. We can provide custom messages that will be displayed if the matcher determines failure. Here is the full listing with the messages.

```
RSpec::Matchers.define :have_error_message do |message|
  match do |page|
    page.error_messages.include? message
  end

  failure_message_for_should do |page|
    "Expected '#{page.error_messages}' to include '#{message}'"
  end

  failure_message_for_should_not do |page|
    "Expected '#{page.error_messages}' to not include '#{message}'"
  end
end
```

The final thing to do is update our step definition.

```
Then /^I should see the error message "([^"]*)"$/ do |msg|
  @current_page.should have_error_message msg
end
```

With this completed let's run our script.

It passes. Let's see what a failure would look like. Change the step definition to:

```
Then I should see the error message "Name can't be a banana"
```

Run the *Scenario* and see the error message. Now change the step back so we can continue with the next sections.

Sending a Message

Time for another Ruby break. When you make a call in Ruby what is actually happening is a message is sent to the object (or class) with the name of the method and any parameters you passed. This is so much a part of the way Ruby works that there is a `send` method that is a part of the standard library. For example these two lines of code do exactly the same thing:

```
page.login
page.send 'login'
```

You might be asking yourself how will this help me? What if we have a series of radio buttons that represent the days of the week and we wish to select one based on a string that contains that day. Selecting a radio button is accomplished by calling the "select" method. If we have a page that defines:

```
    radio_button(:sunday, :id => 'sun')
    radio_button(:monday, :id => 'mon')
    radio_button(:tuesday, :id => 'tue')
    radio_button(:wednesday, :id => 'wed')
    radio_button(:thursday, :id => 'thr')
    radio_button(:friday, :id => 'fri')
    radio_button(:saturday, :id => 'sat')
```

and we define a method on the class like this one:

```
    def select_radio_for(day_of_week)
      self.send "select_#{day_of_week.downcase}"
    end
```

we can call the method like this:

```
    select_radio_for "Tuesday"
```

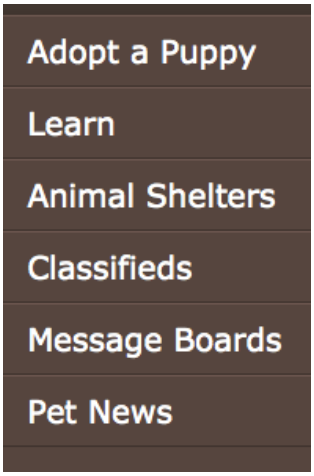
What is happening here is we are intercepting the day of the week and converting it to lowercase using `downcase` while we use it to build up the method name. This is then passed to the `send` method on `self`. In the end, the appropriate radio button is selected. I think you might want to commit this to memory. You might see it again sometime in the future!

Reusable page panels

Many websites have a portion of a page duplicated on other pages. For example, it is quite common to see a common header and footer section on a site show up on most of the pages. It is also somewhat common to have a sidebar or menu show up on many pages.

Since we do not want to introduce duplication we need to find a way to script these common page elements once and use them in many pages. In Chapter 3 we introduced the concept of modules as holders of methods we wish to use across multiple scripts. We can use the same idea here.

Our puppy application has a set of menu items located on the left side of every page.



Menu from the Puppy Application

Let's go ahead and create a new `Module` in our `pages` directory named `side_menu_panel.rb` and place the code for these elements.

```
1   module SideMenuPanel
2     include PageObject
3
4     link(:adopt_puppy, :text => "Adopt a Puppy")
5     link(:learn, :text => "Learn")
6     link(:animal_shelters, :text => "Animal Shelters")
7     link(:classifieds, :text => "Classifieds")
8     link(:message_boards, :text => "Message Boards")
9     link(:pet_news, :text => "Pet News")
10
11  end
```

Now that we have this page partial let's put it to work. In order to add it to our home page we simply need to require it and then include it in the class.

```
1   require_relative 'side_menu_panel'
2
3   class HomePage
4     include PageObject
5     include SideMenuPanel
6
7     ...
8   end
```

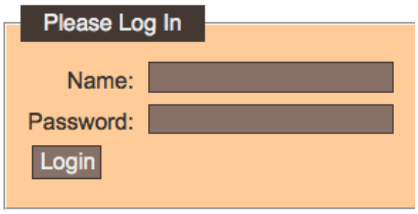
We can repeat this with our other pages and have access to these links from any place on our site.

Blind Automation

There is another side to the puppy application that we haven't explored. In this section we will write a scenario to specify a feature that exists on this side of the application. In the first chapter of this book we discussed the process of writing tests prior to the application code. We'll simulate that here by writing the scenario to our best ability and then I'll reveal how to get to the remainder of the application so we can complete and validate the functionality.

So what exactly does this "other side" of the application do? Well it is the administration functions. There are numerous pages and a lot of functionality. To describe what we will be doing in this section I will use screenshots and narrative.

- First of all you will go to a url that is the landing page for the admin site. If you are not already logged into the system you will be redirected to a login page.
- You need to login to the admin side. The login screen is fairly simple. For the record, there already is a user in the system. The name is "jared" and the password is "rainbow".

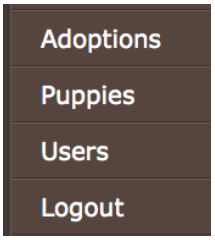


Login Screen

- After you login you will be redirected back to the landing page. It is very simple - just contains a welcome message.

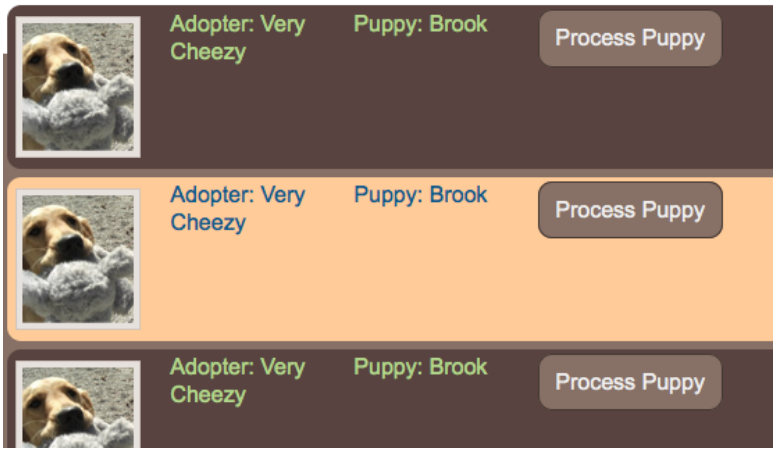
Landing Page

- You will notice that when you are logged in to the admin application you will see some additional links on the side navigation that we created in the last section.



New menu items

- In this scenario you will select the Adoptions link to take you to the page we are interested in.
- On the adoptions page you will find a list of all of the adoptions you have processed in the application. The order of the entries is with the most recent adoption displayed first. Each entry displays the adopter's name with the puppy they adopted.



Adoptions page

- When you press the Process Puppy button a message is displayed on the screen and the entry for the button you just clicked is removed. In the scenario I just want you to process the first adoption.

Please thank Very Cheezy for the order!

Message that is displayed on page

- Your scenario should specify that this message is to be displayed when an adoption is processed. Please note that the Adopter's name is a part of the message so you will need to ensure you capture it somehow.

I suggest you write the scenario and as much of the code for the step definitions and page objects as possible before proceeding to the next section and reading the finished scenario. There are a few challenges here but I am confident you are up for the test.

The other side

By now you should have completed the *Scenario*, step definition and all of the page objects from the previous section. All that's left is to physically go to the pages and sync up the page objects so they match the actual elements on the page. You can do that by opening your browser and going to http://puppies.herokuapp.com/admin. Please complete the scenario and verify it works before proceeding to the next section.

The results please

Are you ready to see the results from the previous section? Here we will introduce two scenarios that take different approaches to implementing the specification. Let's jump straight into the first solution.

```
1   Scenario: Verify message when adoption is processed
2     When I process a pending adoption
3     Then I should see the thank you message
4     # Message is "Please thank <adopter's name> for the order!"
```

In this case we really cannot specify the exact message because we are just processing the first adoption on the page. Note that I placed a comment after the scenario to describe what I expect the message to be. This is designed to be a clue to the developer so he will not have to run the scenario in order to know what the message should be.

The second scenario takes a different approach. Let's have a look.

```
1   Scenario: Verify message when adoption is processed
2     Given I have a pending adoption for "Tom Jones"
3     When I process that adoption
4     Then I should see "Please thank Tom Jones for the order!"
```

Here we specify the name for the pending adoption. This implies that we will create a new adoption with the provided name.

Which approach is better? The first scenario assumes that there is an adoption in the database. What if there is no adoption in our database? Having a test fail because some data you expected to exist in the database is either not there or has been changed is a very frustrating event. We should try to avoid this at all costs. We'll have a lot to say about this in the next chapter. Based on this realization I would go with the second scenario.

PageObjects

Let's look at the PageObjects for this scenario. We'll start with the login page.

```
class LoginPage
  include PageObject

  text_field(:username, :id => 'name')
  text_field(:password, :id => 'password')
  button(:login, :value => 'Login')

  def login_to_system(username='admin', password='password')
    self.username = username
    self.password = password
    login
  end
end
```

There is nothing unusual or exciting about this page. The only interesting item is the way I provided default data for the page. I use default values for the parameters for the `login_to_system` method.

I added a few new lines of code to my `SideMenuPanel` module. These lines of code represent the new link items that appear when the user completes a login.

```
link(:adoptions, :text => 'Adoptions')
link(:puppies, :text => 'Puppies')
link(:users, :text => 'Users')
link(:logout, :text => 'Logout')
```

With this simple change I was easily able to complete the landing page.

```
require_relative 'side_menu_panel'

class LandingPage
  include PageObject
  include SideMenuPanel

  page_url 'http://puppies.herokuapp.com/admin'
end
```

The final page to write is the `ProcessPuppyPage`.

```
class ProcessPuppyPage
  include PageObject

  def process_first_puppy
    button_element(:value => 'Process Puppy').click
  end
end
```

Step definitions

The PageObjects were very basic. There was nothing complicated or even interesting so there must be a fair amount of magic hidden in the step definitions. Let see.

```
Given /^I have a pending adoption for "([^"]*)"$/ do |name|
  on(HomePage).select_puppy
  on(DetailsPage).add_to_cart
  on(ShoppingCartPage).proceed_to_checkout
  on(CheckoutPage).checkout('name' => name)
end

When /^I process that adoption$/ do
  visit(LandingPage)
  on(LoginPage).login_to_system
  on(LandingPage).adoptions
  on(ProcessPuppyPage).process_first_puppy
end
```

No magic here either! The truth is you already knew how to put this all together. I'm very proud of you. We will revisit this scenario again in the next chapter to see how we can make it better.

Removing the duplicate navigation

Some of you may have noticed; we have a fair amount of duplication in our step definitions and we need to do something about it right away. Let's take a look at what I'm talking about. Here are three steps we have created so far:

```
When /^I complete the adoption of a puppy$/ do
  on(HomePage).select_puppy
  on(DetailsPage).add_to_cart
  on(ShoppingCartPage).proceed_to_checkout
  on(CheckoutPage).checkout
end

When /^I checkout leaving the name field blank/ do
  on(HomePage).select_puppy
  on(DetailsPage).add_to_cart
  on(ShoppingCartPage).proceed_to_checkout
  on(CheckoutPage).checkout('name' => '')
end

Given /^I have a pending adoption for "([^"]*)"$/ do |name|
  on(HomePage).select_puppy
  on(DetailsPage).add_to_cart
  on(ShoppingCartPage).proceed_to_checkout
  on(CheckoutPage).checkout('name' => name)
end
```

Do you see any duplication here? In each case we are navigating through a few pages only to perform some action on the last page. Wouldn't it be nice if the *PageObject* gem could help us with this?

We're in luck. The `PageFactory` module that provides the `on` and `visit` methods also adds a couple of methods that help with navigation. The catch is that we have to do a little setup up front. In order to help us with this setup I want to add another gem dependency. Open your `Gemfile` and add the following line:

```
gem 'require_all'
```

Next you will need to run `bundle` to update your project. Once this is complete we are ready to add the navigation configuration. Open the `env.rb` file located in your support directory and add the following:

```
require 'require_all'

require_rel 'pages'

PageObject::PageFactory.routes = {
  :default => [[HomePage, :select_puppy],
               [DetailsPage, :add_to_cart],
               [ShoppingCartPage, :proceed_to_checkout],
               [CheckoutPage, :checkout]],
}
```

Let's go through this one line at a time. The first line is simply requiring the gem that we just added to our project. After this we are putting this gem to work. We need to require each of our Page Objects prior to using them in the next code segment. The *require_all* gem will require all of the files located in the directory you provide. In addition to providing a `require_all` method, this gem also has a `require_rel` method which will require all of the files in a directory relative to the current directory. This is far simpler than requiring each of the files individually. This is what we are doing on the second line of code.

The remainder of the code is setting up a route through the system. A route is an `Array` where each element in the `Array` is yet another `Array` that contains a PageObject class and a method to call. You can also pass parameters to the specific page call in the route by adding additional parameters like `[HomePage, :select_puppy, 'Brook']`. The method should take no parameters and complete everything necessary to proceed to the next page. You must define a `:default` route but you can provide as many routes as you deem necessary. There are three methods from `PageFactory` that you can use. Let's refactor the steps above to explore how to use these methods.

```
When /^I complete the adoption of a puppy$/ do
  navigate_all
end

When /^I checkout leaving the name field blank/ do
  navigate_to(CheckoutPage).checkout('name' => '')
end

Given /^I have a pending adoption for "([^"]*)"$/ do |name|
  navigate_to(CheckoutPage).checkout('name' => name)
end
```

We are using two of the three methods provided from a gem named *page_navigation*. In all cases it will choose the route with the key `:default` unless specified otherwise. The `navigate_all` method will complete an entire route from beginning to end calling the specified methods on instances of the classes. The `navigate_to` method begins at the beginning of a route and calls the methods on

instances of the classes until it arrives at the desired location. The method then returns the page so you can call a method on the page object. You may also use the block form of the method in the same way that `on` works. There is another method that is available that we didn't use in this exercise but lets take a look at it so we will know how to use it in the future.

```
When /^I navigate to some page but do something along the way/ do
  navigate_to(SomePageInTheMiddleOfTheRoute).do_something
  continue_navigation_to(FinalPageInTheRoute).complete_this_step
end
```

In this contrived step I navigated to some page in the middle of a route and performed some action. Then I picked up where I left off and continued the navigation to another page on the route using the `continue_navigation_to` method. This method starts at the page pointed to by `@current_page` and continues until it reaches the target page.

As I mentioned above, you can define as many routes as you like. If you want to use a route other than the `:default` you simply state that in your call like this:

```
navigate_all(:using => :the_other_route)
# or
navigate_to(SomePage, :using => :the_other_route)
# or
continue_navigation_to(TheOtherPage, :using => :the_other_route)
```

There is one more thing to note about routes. We can associate a *DataMagic* file to automatically be used by a *route* by associating them using the *route_data* variable like this:

```
PageObject::PageFactory.route_data = {
  :the_other_route => :some_data,
  :yet_another_route => :more_data
}
```

In each case, when you use the specified route the framework will automatically use the specified *DataMagic* data file.

That's all there is to it!

6. Using the database

Have you ever come into work in the morning and proceeded to check last night's test run only to see that several tests failed. Upon further investigation you discover the tests failed because "somebody was messing with the data in your database". Have you had two testers attempt to run the tests in the same environment at the same time and have it result in failures due to one suite changing data the other was expecting? As we try to run our test suites in parallel we run into this more frequently. These are all too common problems in the testing community.

This chapter is about using the database in our tests. There are really two topics covered here. The first is making our tests independent of the data that is in our test database - a very good test data management strategy. The second is verifying records in our database get updated properly. Along the way we will learn about several new ruby gems.

Installing the puppy application on your computer

The puppy adoption application uses a lightweight database named sqlite3. It is a file system based database which means that the data is written to files on the hard drive of the computer. Since you cannot access these files in the puppy application on the Internet we will need to install the application locally on your computer. This is a very simple activity. You will need to simply download a zip file from that can be found at http://www.cheezyworld.com/cucumber-cheese and unzip it into the same directory where you created the `test_puppies` directory. When you are finished you should have the puppies and test_puppies directories in the same parent directory.

```
1  <my directory>\test_puppies
2  <my directory>\puppies
```

After you have unzipped the file open a command window and change to the `puppies` directory. There you should execute the following command:

```
1  bundle install
```

After the command completes you are ready to start the puppy application. Execute the following command:

```
1  rails s
```

You should see some text output that looks similar to this:

```
=> Booting WEBrick
=> Rails 3.2.1 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-07-24 10:02:27] INFO  WEBrick 1.3.1
[2012-07-24 10:02:27] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.4.0]
[2012-07-24 10:02:27] INFO  WEBrick::HTTPServer#start: pid=89153 port=3000
```

Starting Rails

After the application starts you should be able to view the puppy application. You can do this by opening a browser and going to http://localhost:3000.

If you want to run your tests against your locally running version of the puppy adoption application you will have to open the HomePage and LandingPage page objects and update the `page_url` value. This is not a very good solution. What would happen when we deploy the puppy adoption application to yet another environment? We would have to change the files again. This is a common problem we face when testing software. It is all too common to have to run the application in multiple environments (test, system-test, pre-production, etc.) and to have information about the environment be different each time. We could have items like the base url (which we have in this example) as well as database connection information and even the proper user and password to use. In chapter eleven we will explore what can be done to manage the configuration for different test environments. For now, just change the `page_url`.

Adding the necessary gems

We will be exploring a few gems in this chapter so I think it is a good idea to go ahead and add them to our project now. Add these four lines to the `Gemfile` in your `test_puppies` directory.

```
1 gem 'activerecord', '3.2.1'
2 gem 'factory_girl'
3 gem 'database_cleaner'
4 gem 'sqlite3'
```

Once you have added these entries you simply need to execute `bundle install` in the directory. The `sqlite3` gem is needed to connect to the database. We'll see how this works in the next section. The remaining gems will be explored throughout the remainder of this chapter.

Getting ready for ActiveRecord

Let's start by creating a new file in the `features/support` directory named `database.rb`. As you might expect, this file will contain all configuration related to using a database in our tests. Please begin by adding the following to the file:

```
require 'active_record'
require 'database_cleaner'
require 'factory_girl'

ActiveRecord::Base.establish_connection(
          :adapter => 'sqlite3',
          :database => '../puppies/db/development.sqlite3')
```

In this new file we are requiring a couple of the gems we will be using and establishing a connection to the puppies database using *ActiveRecord*. We will use this connection to perform queries and validate data. Please note that the last line contains a path to the puppies database file. Notice that it goes back one directory and then into the puppies directory. This path assumes the puppies directory is at the same level as the test_puppies. If they are not you will have to adjust the directory path to the puppies application. Before we start writing tests let's spend a little time learning about *ActiveRecord*.

Convention over configuration

ActiveRecord allows us to map database tables and rows to simple classes. This makes it very easy and natural to use database data in our tests.

ActiveRecord is a member of the *Ruby on Rails* collection of gems. All gems in this collection subscribe to an approach often called *Convention Over Configuration*. What this means is that by default it assumes your code is following what the Rails developers think are a set of best practices (the convention). If your code follows this convention then you have to do very little in order to get *ActiveRecord* to work for you. If your code or environment does not follow these conventions then you have a little extra work to perform (the configuration). Let's take a look at an example of how this works. Let's build the initial *ActiveRecord* class so we can get started. Here it is:

```
class User < ActiveRecord::Base
end
```

We are using something new here for the first time. The < symbol means that our User class inherits all of the functionality from ActiveRecord::Base. As a result our User class has a lot of capability and assumes a certain convention.

Tablename

Imagine we have a table in the database that represents Users of our system. We want to create a class named `User` that maps to this table. By convention, *ActiveRecord* would expect the table to be named `users`. What else would you expect something to be called that holds a bunch of `User` objects? *ActiveRecord* has a default behavior where it takes the name of the class and pluralizes it

for the table name. It is very smart in the way it works. For example, it knows that the pluralized version of *Child* is *Children* and the pluralized version of *Puppy* is *Puppies*.

Unfortunately, most existing databases do not have such nicely named tables. How can you create simple easy to understand class names and have it map to a table with a nasty names? We can simply call the `table_name=` method like this:

```
1    class User < ActiveRecord::Base
2      self.table_name = 'usr'
3    end
```

This new line informs *ActiveRecord* that the tablename our User object will map to is named *usr*. So even though we can get the benefits of convention, we can still use configuration when necessary.

Primary key

Let's continue in our imaginary world. *ActiveRecord* assumes that there is a primary key column on the table named *id*. Again, if that column exists in the table and it is the primary key you have to do nothing. For our example let's pretend the primary key column is named *usr_id*. What do we have to do to make our class aware of this breach of convention?

```
1    class User < ActiveRecord::Base
2      self.table_name = 'usr'
3      self.primary_key = 'usr_id'
4    end
```

How simple was that? We have told our class what table to use and which column contains the primary key. What else is left to do?

Column names

ActiveRecord automatically maps columns to attributes so we can access the data in rows. For example, if there is a column in our *usr* table named *fname* you can access it directly using `fname` and `fname=` methods. In many cases the database column names in databases are much harder to understand. What we really want to do is make our names as expressive as possible.

Let's assume our *usr* table has three columns named `fnm`, `lnm`, and `unm`. We can use the `alias_attribute` method to make our accessors more readable.

```
class User < ActiveRecord::Base
  self.table_name = 'usr'
  self.primary_key = 'usr_id'
  alias_attribute :first_name, :fnm
  alias_attribute :last_name, :lnm
  alias_attribute :username, :unm
end
```

We have just mapped the unclear column names to new names that help us better understand their meaning. Of course, for tables that do not yet exist, we could simply name the columns expressively in the first place and that is vastly preferable!

Table relationships

The next item to discuss in this section is how to represent relationships between tables. *ActiveRecord* has three methods to assist in this; `has_one`, `has_many`, and `belongs_to`. For the sake of our example let's say that our *usr* table has a one-to-many relationship with a table that holds addresses. We would represent that in our classes like this.

```
class User < ActiveRecord::Base
  has_many :addresses
end

class Address < ActiveRecord::Base
  belongs_to :user
end
```

Notice that since the *User* object has many addresses we use the pluralized `:addresses` while the *Address* only belongs to one user so we use the singular `:user`. After adding the relationships we can now access the *Addresses* belonging to a *User* by calling `user.addresses`. We can also ask an *Address* for its' *User* by calling `address.user`.

Finding objects

ActiveRecord makes it incredibly easy to find the record we are looking for. You simply ask for it. I think an example is in order here. Let's assume that our *User* has a column (and thereby an instance variable) named *last_name*. If we want to find the user with the last name "Carry" we simply ask *ActiveRecord* to find it for us.

```
User.find_by_last_name('Carry')
```

ActiveRecord is performing a little magic here for us. The first thing that happens when you call this method is *ActiveRecord* checks to see if there is a method named *find_by_last_name*. Guess what, there isn't. Next *ActiveRecord* notices that the method call begins with *find_by* so it guesses that you are more than likely attempting to perform a query. It will then check to see if there is a column name for what you are trying to find. Guess what, there is a column named *last_name*. Next *ActiveRecord* will build the query to find the record. Since there is a good chance that you will ask for this query again, *ActiveRecord* creates the method *find_by_last_name*, adds the code to it, and then executes the method returning the results. The next time you ask for this query it will exist so it will not need to go through all of this method creation magic.

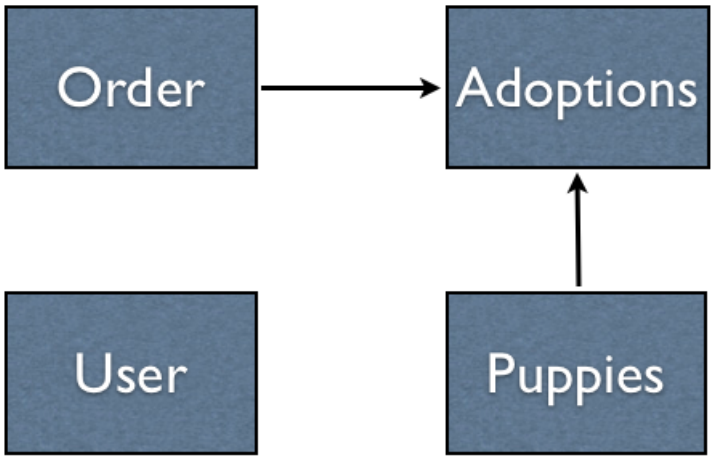
But what should you do if you want to find all of the people with the last name "Carry" living in the state of "Ohio"? You simply ask for it.

```
1     User.find_by_last_name_and_state('Carry', 'Ohio')
```

Here is the bad news. The aliases do not work in the find methods. If the column for last name was *lnm* then you would have to call `find_by_lnm`. (Again, how much better it is to have tables with nice clear column names in the first place.)

The puppy table structure

Let's spend a minute and take a look at the table structure for the puppy adoption application. There are four tables and they form a relationship like this:



Puppy table structure

The puppies available for adoption reside in the *Puppies* table. When a visitor of the site places an order the application creates an entry in the *Orders* table with a entry in the *Adoptions* table for each puppy being adopted.

You are in luck with this table structure because it is in full compliance with the *ActiveRecord* conventions. With that in mind our first activity of this chapter is to create a set of classes that map to the tables above. Start by creating a new directory in the *features/support* directory called *database*. Create a separate file for each class and build the relationships as described in the previous section.

```
# In the file user.rb
class User < ActiveRecord::Base
end

# In the file puppy.rb
class Puppy < ActiveRecord::Base
  has_many :adoptions
end

# In the file order.rb
class Order < ActiveRecord::Base
  has_many :adoptions
end

# In the file adoption.rb
class Adoption < ActiveRecord::Base
  belongs_to :order
  belongs_to :puppy
end
```

Inheritance

In the simple example *ActiveRecord* classes above we are introducing a new Ruby concept. The < symbol means that our class inherits from the class to the right of it. But what exactly does it mean to inherit from another class?

You can think of an *inheritance relationship* as an *is a relationship*. In order to drive this home let's look at an example. A dog is a type of Mammal. If we were building classes to represent this we might say that a Dog *is a* Mammal so our Dog class would inherit from the Mammal class. You might also travel further up the hierarchy and see that a Mammal *is an* Animal so our Mammal class should inherit from the Animal class. Why do we build such relationships? So we do not have to duplicate code. The behavior and attributes that are unique to a generic Animal should exist only in that class. The behavior and attributes that are unique to a Mammal should be in the Mammal class and the same is true about a Dog.

Now I must say that I do not believe that an Adoption *is an* ActiveRecord::Base. This relationship seems wrong to me but the authors of the gem made it so it works this way. In Ruby it is a standard

practice to use Modules and mixins when you have cross cutting behavior for multiple classes. This is how the *PageObject* gem works.

Our first database cukes

It's time for us to take our newfound understanding of how *ActiveRecord* works and apply it to our tests. In order to gain a good initial foundation on the various approaches to working with the database we will create Scenarios for each of the CRUD (Create, Read, Update, and Delete) operations. Let's create a new Feature file and add the following scenario to cover the Create portion of CRUD.

```
Feature: Using the database in our tests

  Scenario: Creating a new order in the database
    Given I know how many orders I have
    When I create a new order
    Then I should have 1 additional order
```

Since there are several new concepts here I'll go ahead and jump straight into the step definitions.

```
Given /^I know how many orders I have$/ do
  @number_orders = Order.count
end

When /^I create a new order$/ do
  order = Order.new
  order.name = "Cheezy"
  order.address = "123 Main"
  order.email = "cheezy@example.com"
  order.pay_type = “Credit card”
  order.save
end

Then /^I should have (\d+) additional order$/ do |additional_orders|
  Order.count.should == @number_orders + additional_orders.to_i
end
```

In the Given and Then steps I use the `count` method to determine how many items are in the database and save it to an instance variable `@number_orders`. `count` is one of many methods that was added to our class when we inherited from *ActiveRecord::Base*. The When step creates a new *Order* object,

populates it with data, and saves it in the database. In the Then step we are simply checking that we have 1 more record than we had when we counted them earlier.

I think the last Scenario satisifies the C (create) portion of CRUD. Let's move on to the R and write a scenario that reads a record from the database.

```
  Scenario: Read an order object from the database
    Given I have an order for "George Washington"
    When I read that order
    Then the order should have the name "George Washington"
```

and now on to the step definitions:

```
Given /^I have an order for "([^"]*)"$/ do |name|
  order = Order.new
  order.name = name
  order.address = "123 Main"
  order.email = "cheezy@example.com"
  order.pay_type = “Credit card”
  order.save
  @original_name = name
end

When /^I read that order$/ do
  @order = Order.find_by_name(@original_name)
end

Then /^the order should have the name "([^"]*)"$/ do |name|
  @order.name.should == name
end
```

The Given should look familiar. It is similar to a step in the previous example except we are using the name provided for the creation of the order and we are saving that name in an instance variable named `@original_name`. The When is where we are performing the query to find the record. The method `find_by_name` will locate the records in the database where the name has the provided value and return them as *Order* objects. We are saving that to an instance variable named `@order`. In the Then step we are comparing the name of the `@order` with what is provided in the feature file.

Let's write a new Scenario that updates a record

```
  Scenario: Updating an order object
    Given I have an order for "Goofy"
    When I update the name to "Minnie"
    Then I should have a record for "Minnie"
    And I should not have a record for "Goofy"
```

This Scenario will cause us to write three new step definitions.

```
When /^I update the name to "([^"]*)"$/ do |name|
  order = Order.find_by_name(@original_name)
  order.name = name
  order.save
end

Then /^I should not have a record for "([^"]*)"$/ do |name|
  order = Order.find_by_name(name)
  order.should be_nil
end

Then /^I should have a record for "([^"]*)"$/ do |name|
  order = Order.find_by_name(name)
  order.should_not be_nil
end
```

The When step should be no surprise. We are simply reading a record from the database, updating the name field, and then saving it to the database. The first Then step attempts to read a record from the database using the provided name and then checks to ensure the return value is nil. In Ruby, nil means that it does not exist. The second Then step reads the record using the provided name and then ensures that it is not nil.

We are three fourths of the way through our CRUD exercise. All that is remaining is the delete Scenario. Let's write that now.

```
  Scenario: Delete an order object
    Given I have an order for "Daisey Duck"
    When I delete that order
    Then I should not have a record for "Daisey Duck"
```

As you can see, we are again reusing the step defined in the Read Scenario to create a new order and we are reusing a step from our Update Scenario to state that we expect the record to be deleted. The remaining step looks like this:

```
When /^I delete that order$/ do
  order = Order.find_by_name(@original_name)
  order.delete
end
```

In this step we simply read the record and then call the `delete` method.

We have completed all of the CRUD exercises. As you can see, using *ActiveRecord* is very simple. Now I think it is time to discover how we can apply the Default Data concept to our test data.

Default data

In chapter 5 we learned about using default data with our page objects. We learned that default data allowed us to specify as much or little data as was necessary for our specific Scenario. This section is about applying the same pattern to our database access.

First of all we need to introduce a new gem - *factory_girl*[74]. *factory_girl* allows us to create "Factories" for database tables. A Factory is nothing more than a set of default data. Let's see what a Factory would look like for our Order class. Create a new file in the *datebase* directory named *adoption_factory.rb* and add the following code.

```
FactoryGirl.define do
  factory :order do
    name 'Cheezy'
    address '123 Main'
    email 'cheezy@me.com'
    pay_type 'Check'
  end
end
```

We have just added a set of default data that can be used for our Order class. There are two items I would like to point out here. The first one is that I am setting values for each of the fields on the object. The second item is that *FactoryGirl* knows to map this to the `Order` class because the factory was named `:order`. If I named the factory `:the_other_order` then I would have to modify the line to read `factory :the_other_order, :class => Order` so *FactoryGirl* knows to map the factory to the `Order` class. If you want to use it in your Cucumber step definitions you will have to add it to World and then call the appropriate methods. Let's see how to do this.

```
World(FactoryGirl::Syntax::Methods)
```

This added the methods to our Cucumber World. Now we are free to create objects using the following methods:

[74] http://github.com/thoughtbot/factory_girl

```
# Save instance
order = create(:order)

# Unsaved instance
order = build(:order)

# Override default values
order = create(:order, :name => 'Joe')
```

Now it is time for us to go back and update our CRUD cukes to utilize the shiny new factory. Here are the affected step definitions:

```
When /^I create a new order$/ do
  order = Order.new
  order.name = "Cheezy"
  order.address = "123 Main"
  order.email = "cheezy@example.com"
  order.pay_type = “Credit card”
  order.save
end
```

is updated to:

```
When /^I create a new order$/ do
  create(:order)
end
```

and

```
Given /^I have an order for "([^"]*)"$/ do |name|
  order = Order.new
  order.name = name
  order.address = "123 Main"
  order.email = "cheezy@example.com"
  order.pay_type = “Credit card”
  order.save
  @original_name = name
end
```

is updated to:

```
Given /^I have an order for "([^"]*)"$/ do |name|
  create(:order, :name => name)
  @original_name = name
end
```

Very nice! The only thing remaining is to create factories for a few of our other *ActiveRecord* classes. This task might be easier than you think. First of all, we will not create a default value for User since there is already a record in the database we can use. Also, we will not be inserting puppies and will therefore not need a factory for it. We will need a factory for Adoptions and it will help us understand a little about how mappings work in *factory_girl*. Let's look at the code first and then we'll talk about how it works.

```
require_relative 'puppy'

FactoryGirl.define do
  factory :order do
    name 'Cheezy'
    address '123 Main'
    email 'cheezy@me.com'
    pay_type 'Check'
  end

  factory :adoption do
    association :order
    puppy Puppy.find_by_name('Hanna')
  end
end
```

The `:adoption` factory introduces a few new concepts. The first concept is an `association`. As you recall, the `Adoption` class `belongs_to :order`. The way we represent this in *FactoryGirl* is by building as association. The parameter to the `association` call is the name of another factory. The second line of this factory is building an association as well but instead of calling another factory it is performing a query against the database to retrieve the object for the association.

We are making progress in our quest to develop a robust approach to test data management and using the database in our cucumber tests but we still have one very significant problem. If you were to try to run the puppy adoption web application at this time you would run into problems. You see, we have been adding `Order` objects in the database that do not have the proper relationships with the `Adoption` and `Puppy` classes. What we have failed to do is keep the database clean. Don't worry, we will cover how to do that in the next section.

Keeping the Database Clean

What we are striving for in our test data management strategy is to have each test be responsible for setting up the data it needs to complete, execute the test, perform the validations, and then cleanup the database. We will use a gem called *database_cleaner*[75] to keep the database clean.

The way *database_cleaner* works is you select a set of tables, apply a cleanup strategy, and then use hooks to perform the actual cleaning function. Let's add the following to the bottom of our `database.rb` file:

```
1 DatabaseCleaner.strategy = :truncation, {:except => %w[puppies users]}
2
3 Before do
4   DatabaseCleaner.start
5 end
6
7 After do
8   DatabaseCleaner.clean
9 end
```

The first line of code sets up our strategy and selects the tables to apply the strategy to. In our case we are using the `:truncation` strategy. This will remove all data from the tables. We are also selecting all tables `:except` the puppies and users tables. The `%w[]` Ruby notation might be something new to you. All that is happening here is that Ruby is creating an array of Strings out of the array. This is exactly the same as if I would have written `['puppies', 'users']`.

The remaining lines of code demonstrate how to run the appropriate *database_cleaner* calls in the hooks so it actually does the work. Go ahead and run all of your tests. They should work fine and you should have the confidence that your tests are not leaving around old stale data that could cause later test runs to erroneously fail.

Putting our factory to work for us

In the last chapter we wrote a Scenario that created a pending adoption and then use the admin side of the puppy application to process the pending adoption. The first step in the Scenario was this:

[75] https://github.com/bmabey/database_cleaner

```
Given /^I have a pending adoption for "([^"]*)"$/ do |name|
  on_page(HomePage).select_puppy
  on_page(DetailsPage).add_to_cart
  on_page(ShoppingCartPage).proceed_to_checkout
  on_page(CheckoutPage).checkout('name' => name)
end
```

In this step definition we navigate through the puppy adoption application completing the entire process of adopting a puppy. How much simpler and faster would the test be if we just inserted the pending adoption into the database. Let's rewrite this step using our factories.

```
Given /^I have a pending adoption for "([^"]*)"$/ do |name|
  order = build(:order, :name => name)
  create(:adoption, :order => order)
end
```

In the first line we are building an unsaved version of our `:order` object using the name that is passed into the step definition. On the second line we are passing that `Order` to the factory that is creating the `Adoption`. As you recall, that factory is also building the relationship to the `Puppy`.

Go ahead and run this Scenario again. It should complete successfully and in much less time than it took to run the previous version that navigated through the pages in the application to adopt a puppy. This is a simple example of setting up the data our test needs, running the test, and then letting *database_cleaner* clean up the data after the test runs.

Verifying the delivery date

The order table has a timestamp column named delivered_on that gets updated whenever a puppy is processed. In this section we are going to write a Scenario that verifies the delivered_on value gets set with the proper value. Let's start with the Scenario.

```
Scenario: Delivered on date should be set when a puppy is processed
  Given I have a pending adoption for "Dog Lover"
  When I process that adoption
  Then the adoption delivered on date should be set to the current time
```

In order to complete the Scenario we will have to write only one step definition. Let's see what this might look like:

```
Then /^the adoption delivered on date should be set to the current time$/ do
  adoption = Adoption.first
  # How should we verify the delivered_on value?
end
```

So `Adoption.first` will return the first (and in this case only) Adoption object from the database. The second line is simply a comment at this time. How should we verify that the `delivered_on` timestamp is set to a valid value? If we simply compare it against `Time.now` it will not match. You see, by the time our step definition runs the application has already completed its behavior and updated the timestamp value so it will contain a value that is one or more seconds in the past.

What we want to do is make sure the `delivered_on` value is within a few seconds of the current time. We could write the step definition like this:

```
Then /^the adoption delivered on date should be set to the current time$/ do
  now = Time.now
  adoption = Adoption.first
  adoption.delivered_on.should be <= now
  adoption.delivered_on.should be > now - 3
end
```

This will work but it is not quite as expressive as I would like the code to be. Let's write a RSpec Matcher to make this easier to understand. Please add the following to the end of the `matchers.rb` file in the `support` directory.

```
RSpec::Matchers.define :be_on_or_near_the_time do |expected|
  match do |actual|
    started = expected - 3
    finished = expected + 3
    actual > started and actual < finished
  end

  failure_message_for_should do |actual|
    "Expected '#{actual}' to be within 3 seconds of '#{expected}'"
  end

  failure_message_for_should_not do |actual|
    "Expected '#{actual}' to not be within 3 seconds of '#{expected}'"
  end
end
```

We are checking to see if the actual time is within 3 seconds of the expected time. You might want to adjust the time to make it work for your situation. With this matcher in place we can rewrite the step.

```
1 Then /^the adoption delivered on date should be set to the current time$/ do
2   adoption = Adoption.first
3   adoption.delivered_on.should be_on_or_near_the_time Time.now
4 end
```

Should we test the design?

There are many in the testing community that would not accept the test we created earlier in this chapter in the section *Putting our factory to work for us*. The fact that we created the adoption without going through the application would make them nervous. Their view is that you should only test "through the application". There is a lot of validity in their view and for me it is all about understanding the tradeoffs.

If we were to only test through the application then that test would have run for a significantly longer time. We were able to create an adoption in the database in less than a second. If it had to run through the user interface it would have taken many seconds. The tradeoff here was speed.

Another issue we would have experienced in the last test was that the delivered_on date is not visible anywhere in the application. We would have to either expose it someplace in the application so we could verify it or read the database directly. We went the simple route and read the database.

Gems for creating data in the database

There are two additional gems that I find very useful when I need to generate data for my tests. They are described in the next two sections.

Faker

Faker[76] is a gem that you can use to generate random data. I often find that I use them in combination with *factory_girl* in order to randomize the data used with my factories. Here is how I would rewrite the factory from the section Default data

[76] http://rubydoc.info/github/stympy/faker/master/frames

```
FactoryGirl.define do
  factory :order do
    name Faker::Name.name
    address Faker::Address.street_address
    email Faker::Internet.email
    pay_type 'Check'
  end
end
```

In this case I am using Faker to provide fake data for the name, address, and email fields. Each time the factory is used, Faker will generate a new and random value for each of those fields.

Populator

Populator[77] generates large amounts of data in the database using *ActiveRecord* classes. Ryan Bates, the author of this gem, has created a very nice screencast[78] that demonstrates how it can be used. Here is a simple example of how we can use it with *Faker* to generate five thousand unique `Order` objects in the database:

```
Order.populate 5000 do |order|
  order.name = Faker::Name.name
  order.address = Faker::Address.street_address
  order.email = Faker::Internet.email
  order.pay_type = ['Check', 'Credit card', 'Purchase order']
end
```

This call will create 5000 orders in the database. In the last line I am passing an array of values. When *Populator* is passed an array it will randomly select one of the values.

[77] https://github.com/ryanb/populator
[78] http://railscasts.com/episodes/126-populating-a-database

7. eXaMpLes of XML and Services

XML[79] is everythwere in the enterprise. You will find it in the messages we send to business partners and in the messages we receive back. They are the message payload of many web services[80]. You will find it in the configuration of applications and messaging systems and you will find it driving many applciation build systems. Testers will potentially run into XML files all of the time. The first part of this chapter is designed to help you understand how to understand, read, validate, and create your own XML files.

The second part of this chapter covers an important topic. In environments were applications are constructed using a service oriented architecture it is fairly common for a team to be dependant on a service that is being constructed by a different team. It is not uncommon for the first team to be finished with their application code and learn that it could be a couple of weeks before the services team has the service ready for them to use. What should the first team do in this situation? Should they wait a few weeks to verify their code is functionally correct? The answer is "no" of course. The second half of this chapter will discuss building surrogate or mock[81] services that you can use to tests your application when the services it relies on is not available.

What is XML?

XML[82] stands for EXtensible Markup Language. It was designed to transport and store data in a way that could be consumed by many endpoints in differing languages and platforms. The tags[83] are not predefined like they are in HTML. Let's take a look at a simple XML document.

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <subject>Reminder</subject>
  <body>Don't forget me this weekend!</body>
</note>
```

As you can see from this example, tags are created by using the less-than and greater-than signs with a name for the tag. The ending tag uses a forward slash to close the tag. In the example above

[79] http://en.wikipedia.org/wiki/XML
[80] http://en.wikipedia.org/wiki/Web_service
[81] http://en.wikipedia.org/wiki/Mock_object
[82] http://www.w3schools.com/xml/xml_whatis.asp
[83] http://www.w3schools.com/xml/xml_syntax.asp

you see several tags (*to*, *from*, etc.) nested inside a top level *note* tag. Nesting is a common technique in XML to create parent / child relationships.

XML is a complex language and I suggest you spend more time reading about it. The links I have provided above are good places to start.

Reading and Validating XML

One of the gems we will be using in this section is Nokogiri[84]. It is very fast at searching and finding things inside XML files. Before we can find something inside an XML file we must first read it. `Nokogiri` has ways to read from disk as well as reading from a remote site using HTTP. Here's how to read from a file

```
1 the_file = File.open('shows.xml')
2 xml_doc = Nokogiri::XML(the_file)
3 the_file.close
```

In this short example we are simply opening a file located on the hard drive named *shows.xml* and passing that file to `Nokogiri`. `Nokigiri` reads and parses the XML from the file. After it has completed reading the file we are free to close it. We can also make an HTTP request for a XML document like this example shows.

```
1 require 'open-uri'
2 xml_doc = Nokogiri::XML(open('http://somesite.com/sample.xml'))
```

In this simple example we are using the open-uri[85] Module to request the file and then passing the contents to `Nokogiri`. open_uri is an incredibly simple module to use. You simply require the module if your ruby file and it adds an `open` method that you can use to request remote resources. After reading and parsing the XML file we need to be able to perform queries against the contents to get values or verify if something exists. `Nokogiri` supports two querying languages. They are xpath[86] and css[87]. Let's look at some examples of both.

Searching XML Documents

In order to search through an xml file we need an example to work with. You can download our example xml file from http://www.cheezyworld.com/wp-content/uploads/2012/11/shows.xml_.zip. Copy the contained xml file named *shows.xml* to the top level directory of your project. The files contents should contain the following:

[84] http://nokogiri.org

[85] http://www.ruby-doc.org/stdlib-1.9.3/libdoc/open-uri/rdoc/OpenURI.html

[86] http://en.wikipedia.org/wiki/XPath

[87] http://en.wikipedia.org/wiki/Cascading_Style_Sheets

```
<root>
  <sitcoms>
    <sitcom>
      <name>Married with Children</name>
      <characters>
        <character>Al Bundy</character>
        <character>Bud Bundy</character>
        <character>Marcy Darcy</character>
      </characters>
    </sitcom>
    <sitcom>
      <name>Perfect Strangers</name>
      <characters>
        <character>Larry Appleton</character>
        <character>Balki Bartokomous</character>
      </characters>
    </sitcom>
  </sitcoms>
  <dramas>
    <drama>
      <name>The A-Team</name>
      <characters>
        <character>John "Hannibal" Smith</character>
        <character>Templeton "Face" Peck</character>
        <character>"B.A." Baracus</character>
        <character>"Howling Mad" Murdock</character>
      </characters>
    </drama>
  </dramas>
</root>
```

Now it's time to talk about searching through this document. We'll start by using xpath to find all of the *character* elements in the file. We would do this by calling the `xpath` method from `Nokogiri`.

```
characters = xml_doc.xpath('//character')
```

This call will return an object of type `NodeSet`[88] that acts very much like an `Array` that contains the following:

[88] http://nokogiri.org/Nokogiri/XML/NodeSet.html

```
<character>Al Bundy</character>
<character>Bud Bundy</character>
<character>Marcy Darcy</character>
<character>Larry Appleton</character>
<character>Balki Bartokomous</character>
<character>John "Hannibal" Smith</character>
<character>Templeton "Face" Peck</character>
<character>"B.A." Baracus</character>
<character>"Howling Mad" Murdock</character>
```

If we wanted to get the first element we would simply execute the following code:

```
characters[0].to_s  # <character>Al Bundy</character>
```

If we just wanted to get the value without the tags we could execute the following code:

```
characters[0].content  # Al Bundy
```

But what if we wanted to get only the *character* elements for shows that were labeled *drama*? The code for that would look like this:

```
characters = xml_doc.xpath('//dramas//character')
# <character>John "Hannibal" Smith</character>
# <character>Templeton "Face" Peck</character>
# <character>"B.A." Baracus</character>
# <character>"Howling Mad" Murdock</character>
```

The good news is that the css version of the calls is very simular. I bet you can guess what this call does.

```
characters = xml_doc.css('sitcom character')
```

If you guessed it will return a `NodeSet` containing all of the *character* elements nested inside the *sitcom* elements then you are correct.

If you know you will only get one value `Nokogiri` provides shortcut methods you can use. They are `at_xpath` and `at_css`. Here's an example of how to use these methods:

```
characters = xml_doc.at_css('dramas name')  # <name>The A-Team</name>
```

This is the end of my brief introduction to `Nokogiri`. Please visit the site for the gem as it contains very good documentation and numerous examples.

A Cuke that use XML

It's time for us to get started with a few simple examples of using Cucumber to read and validate XML. We'll start with a very basic example. I would like you to write a scenario that validates that we have two sitcoms and one drama in the xml file *shows.xml*. But the first thing we will need to do is add the appropriate gems to our project. Open your *Gemfile* and add the following:

```
gem 'nokogiri'
gem 'builder'
```

We are adding *builder* now because we will need it later. Once you have added these entries you should run `bundle install` to add them to your project. Finally, you should add the appropriate `require` statements to *env.rb*.

```
require 'nokogiri'
require 'builder'
```

Now we are ready to begin writing cukes that use XML. Create a new feature file and add the following:

```
Feature: using xml in my cukes

  Scenario: find the number of sitcoms and dramas
    When I open my shows xml
    Then I should see 2 sitcoms
    And I should see 1 drama
```

Now it is time for you to generate and implement the step definitions. Please try to do so without looking at the solution below. I know you can do it. Hint, the `NodeSet` behaves very much like an `Array`.

The solution

I hope you had fun with the last exercise. It was small and fun and got us started using XML in our cukes. Here's the solution I came up with.

```
1  When /^I open my shows xml$/ do
2    file = File.open("shows.xml")
3    @xml = Nokogiri::XML(file)
4    file.close
5  end
6
7  Then /^I should see (\d+) sitcoms$/ do |num_sitcoms|
8    @xml.xpath('//sitcom').length.should == num_sitcoms.to_i
9  end
10
11 When /^I should see (\d+) drama$/ do |num_dramas|
12   @xml.xpath('//drama').length.should == num_dramas.to_i
13 end
```

This code is very simple. On line 2 I am simply creating a file object that opens our *shows.xml* file. The next line passes that file object to `Nokogiri` and the return value is stored in an instance variable named `@xml` which is used in the remaining steps.

The second and third steps simply use the `@xml` instance variable and call the `xpath` method to find the corresponding sitcoms or dramas. These calls return a `NodeSet`. Since the `NodeSet` acts like an array I can call `length` on it to determine how many elements it contains. Next I simply compare this value with the value passed in to the step.

Building with Builder

It is time to shift our focus away from searching through xml documents and focus on building our own xml documents. To do this we will introduce a new gem named `builder`[89] written by the renowned ruby developer Jim Weirich[90]. In addition to the wonderful gem *builder*, Jim is also the developer for the gem named `rake`[91]. We will be discussing *rake* in detail in Chapter 11 but for now we'll just focus on *builder*.

Builder is incredibly simple to use. You simply call methods that represent the elements you wish to add to your xml file and it will generate the xml. You use [*blocks](http://rubylearning.com/satishtalim/ruby_-blocks.html)* to nest elements within other elements. I think an example is in order here.

[89] https://github.com/jimweirich/builder
[90] https://github.com/jimweirich
[91] http://rake.rubyforge.org

```
require 'builder'

builder = Builder::XmlMarkup.new
xml = builder.person do |p|
  p.name 'Cheezy'
  p.phone '555-1234'
end
```

The previous code generates the following xml:

```
<person>
  <name>Cheezy</name>
  <phone>555-1234</phone>
</person>
```

There's not much else to explain because the gem is so simple to use. Are you ready to put it to use?

What's your number?

For our next scenario I am going to have you use both *Nokogiri* and *builder*. I want you to start by creating a contact list using *builder* with at least five entries. The list will take this form:

```
<contacts>
  <contact>
    <name>Cheezy</name>
    <phone>555-1234</phone>
  </contact>
  <contact>
    <name>Sneezy</name>
    <phone>999-5555</phone>
  </contact>
</contacts>
```

After you create the contact list I want you to read the xml document and validate the phone number of one of the entries. You will use *Nokogiri* for the validation.

This time I want you to try to write the Scenario as well as the step definitions. Give it a try.

Contact List

I hope you had fun with this exercise. Here's my solution starting the the scenario:

```
Scenario: find a phone number from a collection
  Given I have a phone book:
    | name   | phone    |
    | Cheezy | 525-5309 |
    | Sneezy | 123-4567 |
    | Wheezy | 908-9999 |
    | Sleazy | 666-6666 |
    | Freezy | 333-3333 |
  When I look up the phone number for "Sneezy"
  Then I should see the phone number "123-4567"
```

This caused me to add the following three step definitions:

```
Given /^I have a phone book:$/ do |table|
  builder = Builder::XmlMarkup.new
  @xml = builder.contacts do |contacts|
    table.hashes.each do |row|
      contacts.contact do |contact|
        contact.name row['name']
        contact.phone row['phone']
      end
    end
  end
end

When /^I look up the phone number for "([^"]*)"$/ do |name|
  doc = Nokogiri::XML(@xml)
  contacts = doc.xpath('//contact')
  @node = contacts.find { |row| row.content.include? name}
end

Then /^I should see the phone number "([^"]*)"$/ do |phone_number|
  @node.at_xpath('.//phone').content.should == phone_number
end
```

There's a lot going on here so let's break it down. On line 2 we are creating our `Builder::XmlMarkup`[92] object. We start constructing our xml document on line 3 by creating our outer *contacts* element. The next line loops through each of the entries passed in from the scenario and the remainder of the code for this step definition creates the inner *contact* element with *name* and *phone* elements. Also, note that we stored the `Builder::XmlMarkup` object in an instance variable named `@xml`. This is used in the next step.

[92] http://builder.rubyforge.org/classes/Builder/XmlMarkup.html

In the next step we pass the `@xml` instance variable directly to `Nokogiri::XML`. On line 15 we use xpath to get a `NodeSet` containing all of the *contact* elements. In the final line of this step we use the `find`[93] method of `Enumerable`[94] to locate the correct node containing the name we are looking for. We assign this return value to an instance variable named `@node`.

In the final step we use the `at_xpath` method to locate the *phone* element nested within the *contact*. Notice we started the xpath statement with a . (dot) character. This is the xpath notation stating to find the next element nested within the other. This can be interpreted as 'Find the phone element nested within the contact represented by `@node`'. We compare the return value with the expected phone number passed from the Scenario.

This is all I have for the XML section of this book but I do recommend you spend some time with *Nokogiri* and *builder*. They are very powerful gems and should provide what you need to work with xml in your tests.

Web Services

It is common these days for development teams to build reusable software components and package them as Web Services[95]. This means that the software is packaged and deployed in such a way as to make it available over the web or some other network for applications or other services to use. In this section we are going to look at testing web services from both sides. First we will tackle the activity of testing web services and later we will look at what it takes to mock[96] them out when our application consumes them.

Testing Web Services

The term Web Service describes a type of application component. There are many implementations of Web Services and the gems you would use depend on what type you are testing. The most widely used Web Services architecture at this time is called SOAP[97] and that is what we will focus on for this book.

How do we test / specify a web service? We call the service passing the necessary parameters and ensure we get the correct response. If we are doing ATDD, we need to actually specify all of the return values we expect to see. It is typically not important to dictate the exact message structure in your Scenario (it is an implementation detail) but you will need to understand it for your step definitions as they will be parsing the message in order to valid the results.. We also need to specify how the error handling should work so it is important to include Scenarios that describe all appropriate behavior.

[93] http://ruby-doc.org/core-1.9.3/Enumerable.html#method-i-find
[94] http://ruby-doc.org/core-1.9.3/Enumerable.html
[95] http://en.wikipedia.org/wiki/Web_service
[96] http://en.wikipedia.org/wiki/Mock_object
[97] http://en.wikipedia.org/wiki/SOAP

Let's start by examining the service we plan to test. If you would navigate to the webservicex site[98] in your browser you can read about the *USA Zip code Information Webservice*. If you scroll down to the bottom of the page you can see the different operations available on this service. I think our first test will be to confirm that these operations are on the service. This information is enough for us to get started.

As always, we will start by creating a feature file. Create a new file in your current project with the following content.

```
Feature: Validating the USA Zip Code Information Web Service

  Scenario: Verify it contains the expected operations
    When I ask the service for the supported operations
    Then "get_info_by_area_code" should be supported
    And "get_info_by_city" should be supported
    And "get_info_by_state" should be supported
    And "get_info_by_zip" should be supported
```

The thing to note here is that we are asking about operations using a ruby-like name instead of the name provided by the service. The gem we are going to use, soap-object[99], converts the names of the operations to snakecase. `GetInfoByAreaCode` becomes `get_info_by_area_code`.

The next step will be to create a PageObject like wrapper around the call to the SOAP Web Service. But before we can do that we will need to add a new gem to our bundle. Please add `gem 'soap-object'` to your `Gemfile`. Next we need to run `bundle install` in order to add the gem.

soap-object has a Factory[100] module much like the `PageObject::PageFactory` module that exists for *page-object*. In order to use it we must add the module to `World` so open up your `env.rb` file and add the following:

```
require 'soap-object'

World(SoapObject::Factory)
```

And now we can write our new class. Create a directory under your `support` directory (or in the `lib` directory if you have structured your project in that structure) called `services` and create a new file named `zip_code_information_service.rb` in that directory. Place the following contents in the newly created file.

[98] http://www.webservicex.net/ws/WSDetails.aspx?WSID=42&CATID=7
[99] https://github.com/cheezy/soap-object
[100] http://en.wikipedia.org/wiki/Factory_(software_concept)

```
class ZipCodeInformationService
  include SoapObject

  wsdl 'http://www.webservicex.net/uszip.asmx?WSDL'

end
```

That is all we need initially. We got the url for the WSDL[101] from the page describing the service. There are many additional settings we could define in our class but let's not focus on those right now. We'll cover a few of them later. Now I think we should create the step definitions and wire everything up to complete our tests. Here are the step definitions.

```
When /^I ask the service for the supported operations$/ do
  @operations = using(ZipCodeInformationService).operations
end

Then /^"([^"]*)" should be supported$/ do |operation|
  @operations.should include operation.to_sym
end
```

There are two things to note here. First of all, notice that I am using the `using` method in the first step. This is the factory method that is added by the `Factory` module. It will create and cache the object the first time it is called and will use the cached object on all subsequent calls. The second thing to note is that we converted the operation variable to a Symbol[102] in the second step. The `operations` method returns an array of Symbols representing the supported operations on the service.

When we run this Scenario we see that it works. So we know that it has the operations we expected. Let's see if those operations actually work. In our example we will validate one of the service endpoints[103]. We'll focus our efforts on the `get_info_by_zip` endpoint. If you click the link on the page that describes that method you will see details about that call. The page describes that the request takes a single parameter - a `String` named `USZip`. If you enter `44114` in the text field and press the `Invoke` button you will see an example of the response. Let's write another Scenario that performs this action.

[101] http://en.wikipedia.org/wiki/Web_Services_Description_Language
[102] http://www.ruby-doc.org/core-2.0/Symbol.html
[103] http://en.wikipedia.org/wiki/Endpoint_interface

```
Scenario: Getting the zip code information by zip code
  When I ask for the zip code information for "44114"
  Then I should get the following information:
  | city      | state | area_code | time_zone |
  | Cleveland | OH    | 216       | E         |
```

In this *Scenario* we are specifying what results we would want to see in a return message when we supply the zip code "44114". Let's implement the step definitions to learn what changes we need to make to our service object.

```
When /^I ask for the zip code information for "([^"]*)"$/ do |zip_code|
  using(ZipCodeInformationService).get_info_by_zip('USZip' => zip_code)
end

Then /^I should get the following information:$/ do |table|
  expected = table.hashes.first
  using(ZipCodeInformationService) do |service|
    service.city.should == expected['city']
    service.state.should == expected['state']
    service.area_code.should == expected['area_code']
    service.time_zone.should == expected['time_zone']
  end
end
```

Notice that I am using the block version of the `using` method on line 7. I am doing this because I have several methods I wish to call on the service object.

In the first step above I am simply calling the service passing the argument. As you can see, the arguments take the form of a `Hash` where the key is the argument name and the value is the value we wish to pass for the argument to the service. This call returns the XML response from the service but I am ignoring it for the time being. I do not want to understand the structure of the messages in my step definitions. Instead, I want the service object to be responsible for parsing the messages and returning the results. The service object is providing the abstraction.

In the second step we are simply calling a series of methods on the service that I expect to return a corresponding value from the results of the call to the web service. We compare those results with our expected values.

Completing our service object

Now that we have completed the step definitions it is time to focus on the service object. Our step definitions have informed us of what methods we need to add. So how do we get the information out of the response from the actual service? soap-object added a method to our service named `body`

which returns the body of the response in a `Hash`. We'll use this to help us get the information we need. We'll wrap the call to `body` in a private method which will return the correct `Hash` we can use to find our values.

Let's add these methods.

```
1  class ZipCodeInformationService
2    include SoapObject
3
4    wsdl 'http://www.webservicex.net/uszip.asmx?WSDL'
5
6    def city
7      message[:city]
8    end
9
10   def state
11     message[:state]
12   end
13
14   def area_code
15     message[:area_code]
16   end
17
18   def time_zone
19     message[:time_zone]
20   end
21
22   private
23
24   def message
25     body
26   end
27
28 end
```

At this point we should talk to the developers to understand the exact structure of the response message. If the case of the service we are testing the messages are nested. In order to get the values we desire we need to remove the nesting. To do this we will simply update our `message` method to handle this for us.

```
def message
  body[:get_info_by_zip_response][:get_info_by_zip_result][:new_data_set][:table]
end
```

After this final change our Scenario works. We have called a web service and validated that we received the correct response.

Time for some refactoring

Even though the service object works just fine there is something about it that bothers me. All of the public methods simply delegate to the message method passing a `Symbol` that is the same as the method call. When I see this I see a form of duplication and we know that it is very important to remove duplication as soon as we discover it.

The simple solution I can think of is to create a new method that takes the symbol and returns the value. After adding that method and removing all of the duplicate methods my service object looks like this:

```
class ZipCodeInformationService
  include SoapObject

  wsdl 'http://www.webservicex.net/uszip.asmx?WSDL'

  def response_for(key)
    message[key]
  end

  private

  def message
    body
  end

end
```

Don't forget to update your step. It should look like this:

```
Then /^I should get the following information:$/ do |table|
  expected = table.hashes.first
  using(ZipCodeInformationService) do |service|
    service.response_for(:city).should == expected['city']
    service.response_for(:state).should == expected['state']
    service.response_for(:area_code).should == expected['area_code']
    service.response_for(:time_zone)_zone.should == expected['time_zone']
  end
end
```

Surrogate Services

If you work at a company where the teams build software using Service Oriented Architecture[104] you may run into difficult situations. Imagine a situation in which you build a component of your application that requires a service being constructed by another team and your portion is completed but the other team is not. To make matters worse let's pretend that the other team will not have their service ready for two weeks.

What should you do? Should you ask the other team to drop everything and work on your request so you can get it sooner? What if they are unable to get to it based on their priorities? Should you wait two weeks to test your code? The simple answer is "No!".

Imagine another situation in which you are using services provided by another company and you have no way to control what data is returned. If the data returned from a service is different each time we call it then it is impossible to have reliable tests.

We need to test all code as close to the time it was developed as possible. This often means we will need to create fake services to stand in for services that are not ready when our code is ready. What's more, we need to completely control the data that is returned by these services as this is the only way we can build reliable tests.

What we need is a pinch hitter

As you might have already guessed, there is a gem for this. Steve Jackson[105], another fellow Leandogger, created a gem named pinch_hitter[106] that does exactly what we need. In order to use the gem you will have to point your application to the host and port configured when you start the service with *pinch_hitter*. Hopefully this is as simple as changing a configuration setting for the test environment. The gem has three simple usages.

[104] http://en.wikipedia.org/wiki/Service-oriented_architecture

[105] https://github.com/stevenjackson

[106] https://github.com/stevenjackson/pinch_hitter

Returning a simple JSON message

What if you want to have a surrogate service return a simple JSON message? *pinch_hitter* allows you to save multiple JSON messages to be returned. Here is a simple example:

```
include PinchHitter

start_service host, port
connect host, port
store '/pricecheck', '{ "price": "$12.99" }'
```

When your aplication makes a call to the `/pricecheck` endpoint it will recieve the JSON response you registered.

Return larger XML and JSON messages

If you have captured XML or JSON messages you can place them in a directory and have them associated with a specific call.

```
include PinchHitter

start_service host, port
connect host, port
messages_directory = File.join(File.dirname(__FILE__), 'messages')
prime '/findStores', :tower_city_stores
```

In this case, *pinch_hitter* will look in the `messages` directory. The file type will be determined by the file extension.

You can use simple substitution for some of the values in the XML or JSON by passing in the value with the call to `prime`.

```
prime '/availableSeats', :available_seats, "GameDay" => "05-01-2014"
```

This call will find the element named *GameDay* in the file and replace its value with *05-01-2014*.

Writing code to return our message

The final way to control the return messages returned by *pinch_hitter* is to write some Ruby code. If you simply create a module with a method like `respond_to(message)` then you can associate it with an endpoint.

```
module CheeseSelector
  def respond_to(message)
    return 'Cheddar' if message.include? 'Wisconsin'
    return 'Brie' if message.include? 'France'
    return 'Swiss' if message.include? 'Switzerland'
  end
end
```

You associate the module with an endpoint like this:

```
include PinchHitter

start_service host, port
connect host, port
register_module '/cheesePlease', CheeseSelector
```

Using these three techniques, *pinch_hitter* is an incredibly powerful tool for building surrogate services and controlling the data pass to your application under test.

8. Moble me moble you

There are few topics as hot and exciting as the topic of Mobile Application development. With the profileration of smart mobile phones and tablet computers there is a rush to get applications into the marketplace. Many companies are attempting to get a presence in this space. And yet the majority of testers I talk to that are testing mobile applications are doing it manually. This manual testing implies that the testing is taking place after the application is developed.

At the company I work for, LeanDog, we have a development studio[107] that builds applications for customers. A significant percentage of the applications we build are mobile applications. When we developed our first application targeting the Android[108] platform, we were appalled at the state of the testing tools. Fellow LeanDoggers Levi Wilson[109], Joel Byler[110], and I set out to change this. We built a set of three gems that can be used with Cucumber to automate applications running in either the Android Emulator[111] or on an Android Device[112]. This chapter will walk you through setting up a new project and writing Cucumber *Scenarios* against an example Android application. In the final section we will discuss what it takes to do the same against an iOS application.

Installing and configuring the necessary software

Installing a Java Development Kit

The first thing we need to do in order to test an Android application is to install the software necessary in order to run the application in an emulator[113]. Android runs on Java so if you do not have a Java Development Kit (JDK[114]) installed you must do this first. If you do not have a JDK installed on your computer you can download a copy from the Java download page[115]. Once the file is downloaded you can simply run the installer and accept all of the default values. Please make sure you install a JDK and not a JRE[116].

[107] http://leandog.com/what-we-do/studio/
[108] http://www.android.com
[109] https://github.com/leviwilson
[110] https://github.com/joelbyler
[111] http://developer.android.com/tools/help/emulator.html
[112] http://www.android.com/devices/
[113] http://en.wikipedia.org/wiki/Emulator
[114] http://en.wikipedia.org/wiki/Java_Development_Kit
[115] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[116] http://en.wikipedia.org/wiki/Java_Runtime_Environment#Execution_environment

Installing the Android SDK

The next software you will need is the Android SDK[117]. Please go to the sdk page[118] and click the "Download the SDK" button to get the software you need. It is a rather large download so please allow yourself some time to for it to complete. Once you have downloaded the zip file, extract it to a place where you can find it later.

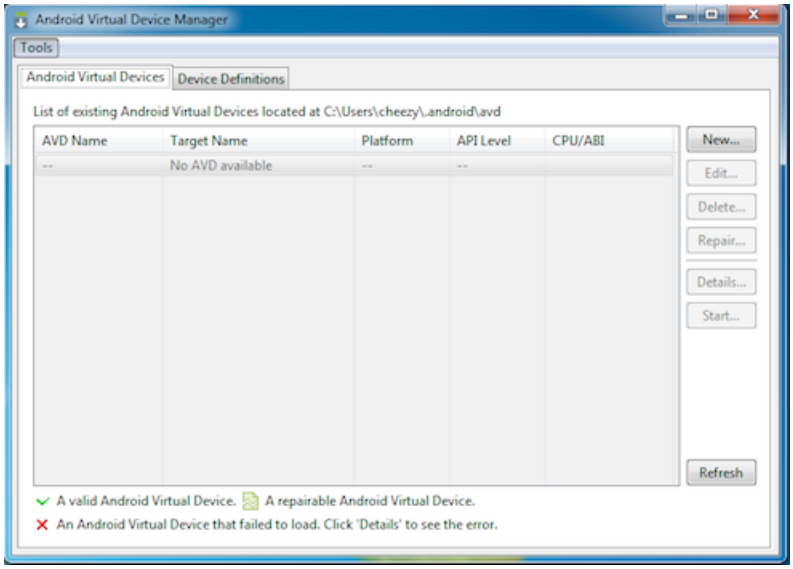
The next step is to make three changes to your environment. The first change is to add an environment variable[119] named `ANDROID_HOME`. When you extracted the zip file from the last paragraph it created a directory named *atd-bundle-<os_platform>-<date>* where *os_platform* is the name of the operating system you are running on. The value for `ANDROID_HOME` should be the path to the *sdk* directory in the directory created by unzipping the file. Next we need to add two directories to our path - the `tools` and `platform-tools` directories that are in the `ANDROID_HOME` directory.

If everything went well you should be able to execute `android -help` in a command line and see some output. You should also be able to execute `emulator -help` and see its' output. If you see the output then you may continue to the next section. Otherwise, please review this section and verify you have the path's pointing to the correct location and `ANDROID_HOME` setup properly.

Creating the emulator

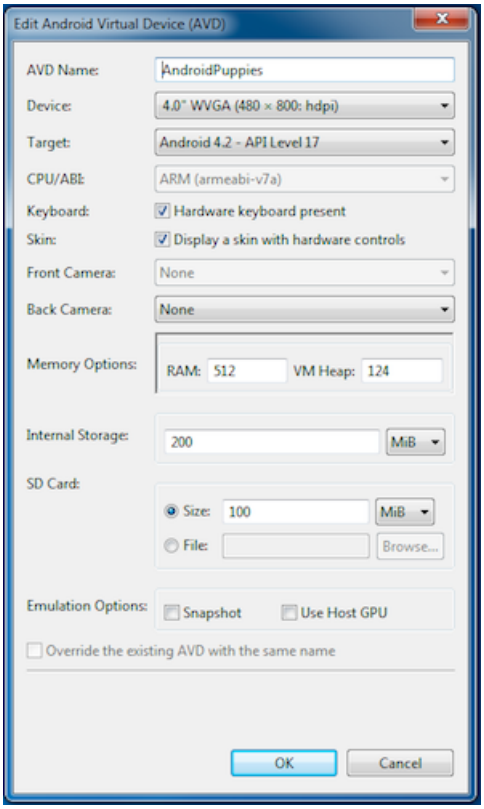
The next step we must complete in order to get us closer to being ready to test Android applications is to create an emulator. We'll use the AVD Manager[120] to perform this task. You can start the AVD Manager by executing `android avd` on the command line.

[117] http://developer.android.com/sdk/index.html
[118] http://developer.android.com/sdk/index.html
[119] http://en.wikipedia.org/wiki/Environment_variable
[120] http://developer.android.com/tools/devices/managing-avds.html



Android Virtual Device Manager

Click the *New...* button and fill in the dialog with the following information:



Values for Emulator

Once you complete the setup click the *OK* button to return to the previous screen.

Another option for creating an emulator is to select the Device Definitions tab on the AVD Manager and choose one of the predefined configurations. Select one of the defined devices and click the *Create AVD...* button.

You are now ready to run your emulator. Select the emulator from the list and click the *Start...* button and then click the *Launch* button to continue. Once the emulator starts you will want to turn off the screen lock. To do this you should go to the Applications area and then start the *Settings* application. Once you are in the *Settings* application you should scroll down and select the *Security* option from the list. The first option is *Screen lock*. Select this option and when presented with the possible values select *None*. You may now return to the main menu screen. You are ready to begin testing Android applications on this emulator!

Staying up to date

The mobile environment is fast paced. New versions of the Android SDK are coming out at a rapid pace. You will want to have the ability to install them when your team chooses to support it. You might also want to install older versions of the SDK in order to test your application on older simulated device. You will use the Android SDK Manager[121] to install and remove versions of the SDK as well as other packages. The way you will do this is by running the `android` command with no parameters from the command line.

The test subject

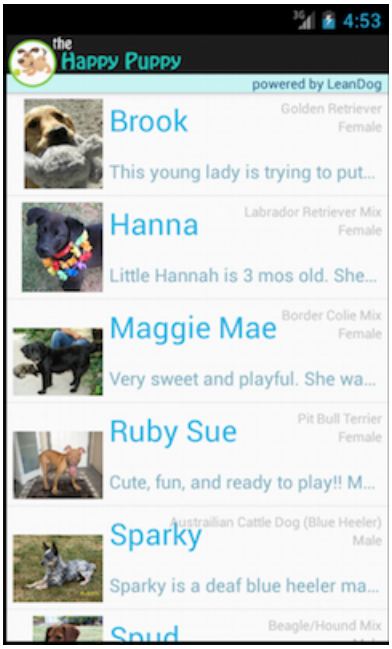
We are going to test an Android adaptation of the puppy site. Please download the Android puppy adoption application[122] now and unzip it to a know location. Inside the *android_puppies* directory you will find two files that we will use throughout this chapter. The first file we will need is the apk file[123]. Open a command line and change to the directory where you unzipped the file. If an emulator is not running go ahead and start it now. After the emulator is running execute `adb install puppy-app.apk` on the command line. This will install the application. After it is installed take a little time to explore the application. There is not much to the application; a landing screen that retrieves some information from a server on the Internet, a screen that lists all of the puppies that are available for adoption, and a screen that shows the details of a puppy. If you do not see information about the puppies then you probably are not able to communicate to the server processes that contain the data. Please ensure that you are connected to the Internet.

If the application loaded the puppy information correctly you should land on the Puppy List screen.

[121] http://developer.android.com/tools/help/sdk-manager.html

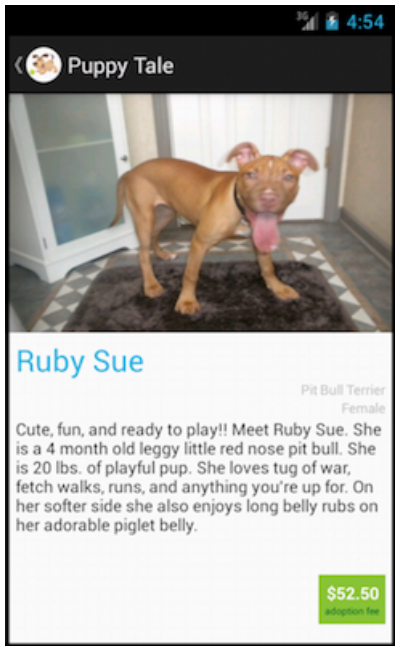
[122] http://www.cheezyworld.com/wp-content/uploads/2013/04/android_puppies.zip

[123] http://en.wikipedia.org/wiki/APK_(file_format)



Puppy List Screen

If you select a puppy by tapping on it in the list you will end up on the Puppy Detail screen.



Puppy Detail Screen

We'll begin by specifying and automating the specifications for the Puppy List Screen.

Start with a new project

Before we get started we will need to create a new project for our tests. Just like we did with the web application, we'll use *testgen* to build out the shell of the project for us. It is always a good idea to make sure you have the latest version of *testgen* because it is being updated and enhanced frequently. You can do this by executing `gem update testgen`. Now we can create our starter project by executing `testgen project android_puppies --with-gametel`. This will create a directory named *android_puppies* and place several files in that directory. Go ahead and open that directory if you are using *RubyMine*.

The next step is to copy the *puppy-app.apk* file we downloaded earlier to the *features/support* directory in the newly created project. We will also need to the keystore file from the directory where we unzipped the download. Once you locate the file, place it in the *features/support* directory as well. In a real project I would not place the apk and keystore in the project directory structure. Instead, I'd have the application checked out and point to the application *apk* that I would build locally. This way I could always checkout the updated code to ensure I am running against the latest version of the application.

Now we have everything we need to start writing tests.

Open the *env.rb* file found in the *features/support* directory. It should look like this:

```
require 'rspec-expectations'
require 'gametel'

World(Gametel::Navigation)

keystore = {
  :path => File.expand_path('~/.android/debug.keystore'),
  :alias => 'androiddebugkey',
  :password => 'android',
  :keystore_password => 'android'
}

Gametel.apk_path = PATH_TO_APK
Gametel.keystore = keystore

Before do
  @driver = Gametel.start(ACTIVITY_NAME_GOES_HERE)
end

After do
  Gametel.stop
end
```

This first thing happening in this file is that it is requiring a couple of gems. Next on line 4 it is registering a module named *Gametel::Navigation* with the Cucumber *World*[124]. This should be familiar to you as we did something very simular when we were testing web applications. We will need to make a couple of changes to the remainder of the file.

When our tests run against the Android application it will need to read some files that are packaged inside of the *apk* file and then repackage it prior to deploying into the emulator. In order to do this we will need access to the keystore used to sign the application during development. In our case, we downloaded the file and placed it into the *support* directory so we can update the code that creates the `keystore` to look like this:

```
1  keystore = {
2    :path => File.dirname(__FILE__) + '/debug.keystore'
3    :alias => 'androiddebugkey',
4    :password => 'android',
5    :keystore_password => 'android'
6  }
```

Immediately after the declaration of `keystore` on line 13 we can see that it is asking us to set the location of the *apk* file. We can do that by updating this line to this:

```
1  Gametel.apk_path = File.dirname(__FILE__) + '/puppy-app.apk'
```

The final change we need to make is to specify the correct Activity[125] to open when the application is started. We will talk about Activities in more detail in the next sections but for now you can simply update the code to this:

```
1  Before do
2    @driver = Gametel.start('PuppiesActivity')
3  end
```

We are finally ready to begin writing our specifications for the Android version of our Puppy Adoption Application.

All of the puppies

To get started we need to write a few scenarios that might help us describe the behavior of the Puppy List Screen.

[124] https://github.com/cucumber/cucumber/wiki/A-Whole-New-World
[125] http://developer.android.com/reference/android/app/Activity.html

```
Feature: Displaying information about the puppies available for adoption

  This screen really has no behavior.  It simply displayes the
  list of puppies available for adoption.

  Background:
    When I am looking at the available puppies

  Scenario: Knowing the name of the available puppies
    Then I can see that "Sparky" is available for adoption

  Scenario: Knowing the breed and gender of available puppies
    Then I can see the "Ruby Sue" is a "Pit Bull Terrier"
    And I can see that "Ruby Sue" is a "Female"

  Scenario: Being overwhelmed by the cuteness of the available puppies
    Then I will be moved when I look into "Maggie Mae"s eyes

  Scenario: Getting to know the available puppies
    Then I can see that "Brook"s description starts with "This young lady is tryi\
ng"
```

And of course the next step is to generate the step definitions. Let's do that and go ahead and write the code we wished we had. Here is what I came up with:

```
When /^I am looking at the available puppies$/ do
  on(PuppyListScreen).wait_for_text 'Hanna'
end

Then /^I can see that "([^"]*)" is available for adoption$/ do |name|
  on(PuppyListScreen).view_for(name).should have_text name
end

Then /^I can see the "([^"]*)" is a "([^"]*)"$/ do |name, breed|
  on(PuppyListScreen).view_for(name).should have_text breed
end

When /^I can see that "([^"]*)" is a "([^"]*)"$/ do |name, gender|
  on(PuppyListScreen).view_for(name).should have_text gender
end

Then /^I will be moved when I look into "([^"]*)"s eyes$/ do |name|
```

```
18    on(PuppyListScreen).view_for(name).should have_image
19  end
20
21  Then /^I can see that "([^"]*)"s description starts with "([^"]*)"$/ do |name, de\
22  scription|
23    on(PuppyListScreen).view_for(name).should have_text description
24  end
```

A few of the steps need a little clarification. The first step is using the `on` method that works the same way it does for the page-object gem. But notice that we are calling `wait_for_text` in that first step on line 2. This is because the first page actually has a progress bar (decorated to look cool of course) that waits until the data is received from the server call before rendering the list. We need to wait until all of the calls to the server are complete before proceeding. When the data arrives in the application the list is rendered and we can see the available puppies. In our step definition we are simply waiting until the word *Hanna* appears on the screen which will not happen until the puppies are loaded.

In the second step you can see that we are calling a method named `view_for` on line 6. I want this method to return a view - some information that is displayed - about a specific puppy. This is a method we will have to add to our *Screen Object*.

Our calls to the `on` method pass a class named `PuppyListScreen`. Since this class does not exist we will need to add it. Create a class in the `screens` directory and add this content:

```
1   class PuppyListScreen
2     include Gametel
3
4     activity "PuppiesActivity"
5
6     list_item(:sparky, :text => 'Sparky')
7     list_item(:ruby_sue, :text => 'Ruby Sue')
8     list_item(:maggie_mae, :text => 'Maggie Mae')
9     list_item(:brook, :text => 'Brook')
10
11    def view_for(name)
12      self.send "#{name.downcase.gsub(' ', '_')}_view"
13    end
14  end
```

On line 2 above you see that we include Gametel[126]. This is one of a series of ruby gems that I developed with Levi Wilson and Joel Byler to make it easy to test Android applications. This gem

[126] https://github.com/leandog/gametel

acts much like the *page-object* gem in that it provides an abstraction over the system under test making it easier to absorb changes as they occur.

On line 4 I am calling the `activity` method passing the value `"PuppiesActivity"`. An Activity[127] losely maps to a screen in an Android application. Calling the `activity` method adds an `active?` method which returns `true` of `false` depending on if the *Activity* is active. This method is called when we call the `on` method which causes the method to wait until the *Activity* is fully loaded and displayed.

So you might ask "If we are waiting for an *Activity* to be active then why do we have to wait for the text to appear in the first step definition?". The answer is that the *Activity* is active as soon as the progress bar is displayed even though it is still making a server call to get the data to be displayed. Waiting for the text "Hanna" to show up on the screen allows us to ensure this asynchronous activity is finished prior to the test proceeding.

On lines 6 through 9 I am simply declaring `list_item` items on the page and locating them by text contained within their view. In many cases you would not want to use text as a locator but in this case we completely control all of the test data in all of our test environments so these tests will not fail for the wrong reason.

The final thing to note in this class is the definition of the `view_for` method. In this method we want to retrieve information about the view for a `list_item` using the name of the puppy. Just like each Element declaration in *page-object* generates an `[name]_element` method, each View declaration in *Gametel* generates a `[name]_view` method. If we wanted to get the view for "Sparky" we could simply call the `sparky_view` method. Since we want our `view_for` method to be generic we are using the buit-in Ruby capability to send a message to a receiver and have it invoke the corresponding method. We described this in more details in Chapter 5 in a section named Sending a Message. In addition to sending the message we do a little of prep work on the name that is passed as an argument. First of all we need to make names like "Sparky" lowercase and that is why we are calling `downcase`. Next, names like "Ruby Sue" need to be converted to `ruby_sue` and that is why we are calling the global substitution method `gsub` so we can substitute a space character with an underscore character.

The class `Gametel::Views::ListItem` class, which is what is returned when we call the `[name]_view` method on a list item, has a method called `has_text?`. This is the method that is called in the majority of our steps like this:

```
1   on(PuppyListScreen).view_for(name).should have_text gender
```

In the step definition on line 18 we are calling the `have_image` method which in turn calls the `has_image?` method on the returned view. This method simply returns `true` if there is a place holder for an Image present within the view. We'll have more to say about images when we complete the *Scenarios* for the Puppy Detail Screen.

And with that, the first feature works. Make sure you have the emulator running prior to executing the Cucumber *Scenarios*. You can do this from the Virtual Device Manager window. Simply select

[127] http://developer.android.com/guide/components/activities.html

the emulator you wish to start and select the *Start...* button. You can also do this from the command line like this - `emulator -avd <avd_name>`. If using the command line please make sure you use the correct name.

Show me the details

Let's continue forward and specify the behavior for the Puppy Detail Screen. Once you tap a puppy in the Puppy List Screen you are presented with some details about the puppy of interest. Here are the *Scenarios* I came up with:

```
Feature: Displaying the details of a puppy

  Background:
    Given I am looking at the available puppies

  Scenario: Learning more about a puppy I am interested in
    When I want to learn more information about "Sparky"
    Then I will be able to see "Sparky"s details

  Scenario: Seeing the name and breed of my puppy
    When I want to learn more information about "Sparky"
    Then I know that he is a type of "Austrailian Cattle Dog (Blue Heeler)"
    And I know that he is a "Male"

  Scenario: Learning more about the life of my puppy
    When I want to learn more information about "Spud"
    Then I can see that my puppy "is playful and friendly and would make a great \
addition to your family"

  Scenario: Looking into the eyes of my puppy one last time
    When I want to learn more information about "Ruby Sue"
    Then I can look into the eyes of my puppy before I make my decision

  Scenario: Knowing what my puppy will set me back
    When I want to learn more information about "Tipsy"
    Then I know that the adoption fee is "$42.00"
```

I am reusing a step from my first Feature here in my *Background*. After that, each scenario simply selects a specific puppy and then specifies the correct values to be displayed. My new step definitions look like this:

```
When /^I want to learn more information about "([^"]*)"$/ do |name|
  on(PuppyListScreen).details_for name
end

Then /^I will be able to see "([^"]*)"s details$/ do |name|
  on(PuppyDetailScreen).name.should == name
end

Then /^I know that he is a type of "([^"]*)"$/ do |breed|
  on(PuppyDetailScreen).breed.should == breed
end

When /^I know that he is a "([^"]*)"$/ do |gender|
  on(PuppyDetailScreen).gender.should == gender
end

Then /^I can see that my puppy "([^"]*)"$/ do |description|
  on(PuppyDetailScreen).description.should include description
end

Then /^I can look into the eyes of my puppy before I make my decision$/ do
  on(PuppyDetailScreen).wait_for_headshot
end

Then /^I know that the adoption fee is "([^"]*)"$/ do |adoption_fee|
  on(PuppyDetailScreen).fee.should == adoption_fee
end
```

We have a lot of work ahead of us to make the steps a reality. The first thing you will notice is that we are calling a new method named `details_for` on the `PuppyListScreen` object on line 2. I think we'll get started there. In the previous section you learned that a call to `list_item(:sparky, ...)` generates a method named `sparky_view` which returns information about the `ListItem`. This call will also generate a method named `sparky` which simply clicks the `ListItem`. Let's see how this will work in our `PuppyListScreen` method.

If we simply wanted to click the `ListItem` for Sparky we would call the `sparky` method. Since we will need to also cause the provided name to be converted to lowercase and we will also need to substitute an underscore for the spaces our method would look something like this:

```
  def details_for(name)
    self.send name.downcase.gsub(' ', '_')
  end
```

The problem that we now have is that there is some duplication between the new `details_for` method and the `view_for` method we added in the previous section. We must eliminate that duplication. One way to do that is to create yet another method that takes the name and returns the value with the correct modifications. We will make this method `private` since it is only intended to be used by the `PuppyListScreen` class. After creating this method we will need to update the original methods to use this new one. Our updated methods now look like this:

```
1   def view_for(name)
2     self.send "#{method_for(name)}_view"
3   end
4
5   def details_for(name)
6     self.send method_for(name)
7   end
8
9   private
10
11  def method_for(name)
12    name.downcase.gsub(' ', '_')
13  end
```

The other thing we will need to do to our `PuppyListScreen` before moving on is to declare a couple of additional `ListItems` since we are using a couple of new names in the latest scenarios. The finished screen object looks like this:

```
1  class PuppyListScreen
2    include Gametel
3
4    activity "PuppiesActivity"
5
6    list_item(:sparky, :text => 'Sparky')
7    list_item(:ruby_sue, :text => 'Ruby Sue')
8    list_item(:maggie_mae, :text => 'Maggie Mae')
9    list_item(:brook, :text => 'Brook')
10   list_item(:spud, :text => 'Spud')
11   list_item(:tipsy, :text => 'Tipsy')
12
13   def view_for(name)
14     self.send "#{method_for(name)}_view"
15   end
16
17   def details_for(name)
```

```
18      self.send method_for(name)
19    end
20
21    private
22
23    def method_for(name)
24      name.downcase.gsub(' ', '_')
25    end
26  end
```

Now it is time to turn our attention to the new details screen. The first thing we can do is create the basic shell for the class. It should look like this:

```
1  class PuppyDetailScreen
2    include Gametel
3
4    activity "PuppyTaleActivity"
5  end
```

Notice that I am calling the `activity` method again. In this case the activity is named `"PuppyTaleActivity"`. How do I know what the activity name is? If I'm developing the application then it is very obvious. It is the Java class name of the Activity represented by this screen object. But what should I do if I am not the developer? What if my job is to simply write / automate the specification? I glossed over this in the previous section. The answer is simple. You should have a conversation with the developer creating this screen.

In order to complete the screen object we will need to declare each of the views that will appear on the screen. In each case we will be identifying them by `id`. Again, if you are not developing the screen then the simplest way to complete the screen object is to have a conversation with the developer and agree on what `ids` will be used when the developer writes the code to make your *Scenarios* pass. If the developer is not around you can find the `ids` in the `layouts` for the application. You will need access to the application source code in order to find these.

Here is our completed `PuppyDetailScreen` class:

```
1  class PuppyDetailScreen
2    include Gametel
3
4    activity "PuppyTaleActivity"
5
6    text(:name, :id => 'name')
7    text(:breed, :id => 'breed')
8    text(:gender, :id => 'gender')
9    text(:description, :id => 'description')
10   text(:fee, :id => 'adoption_fee')
11   image(:headshot, :id => 'headshot')
12 end
```

This is one final detail I wish to explore. In the step definitions on line 22 we see this code:

```
1    on(PuppyDetailScreen).wait_for_headshot
```

Why do we have to wait for something and where did this method come from?

The details screen has an ImageView[128] which holds an image of the puppy we are viewing. This view is simply a placeholder for an image. When the screen is loaded it makes a request to the server to retrieve the actual image. In this step I wish to ensure that not only is the placeholder present, but that the request for the image has been completed and the placeholder contains an actual image. One of the methods generated when we declare an image like we did on line 11 of the `PuppyDetailScreen` is a method named `wait_for_[name]` where `name` is the name you provided. This method will wait up to 10 seconds for the `ImageView` to contain an image. If it does, it simply continues. Otherwise, an error is raised.

Specifying Android applications with Cucumber

As you have seen from the previous sections, it is very easy to create executable specifications for Android applications. *Gametel* is an extremely feature rich gem that provides everything needed to automate those specifications. *Gametel* also works seamlesly with other gems that make testing easy - gems like *DataMagic*. In addition to running your tests in an emulator, this testing stack also supports running on physical Android devices[129] connected to the test server using USB[130]. It is a common practice to have one or more devices connected to your Continuous Integration server so your tests can run on real hardware continuously.

[128] http://developer.android.com/reference/android/widget/ImageView.html
[129] http://developer.android.com/tools/device.html
[130] http://en.wikipedia.org/wiki/Universal_Serial_Bus

But we’re building iOS applications!

If you are developing iOS applications then you will want to look at another gem by Levi Wilson named *furter*[131]. It works with a low level driver gem named *frank cucumber*[132] to allow you to build screen objects in the same manner you will with *Gametel*.

[131] https://github.com/leviwilson/furter
[132] http://testingwithfrank.com

9. A batch of windows

Wouldn’t it be nice if we could use Cucumber to test native Windows applications?

MORE TO COME SOON

10. Web 0.1 and 2.0

The web examples we have worked on so far have be fairly straight forward. The puppy application renders clean html and we simply located the elements we wished to interact with and performed the intended action. Wouldn't it be nice if all web applications were this easy to test? The truth is that many applications have additional challenges we must overcome in order to find and interact with the basic elements on the pages. This chapter introduces these challenges and discusses what we must do in order to overcome them.

The web is an interesting environment. At its most basic form it is simply a software residing on our computers (a web browser) making requests to a server for text and binary files. In the early days of the web it was used exclusively for sharing information and research data. Eventually businesses realized they could share information about thier products on this new medium. And so the world wide web began.

As more people came to the web businesses tried to find ways to capitalize on this new audiance. They learned the could do more than inform consumers about their products - they could actually sale their wares to people over the web. Users were also demanding a more interactive and rich enviornment.

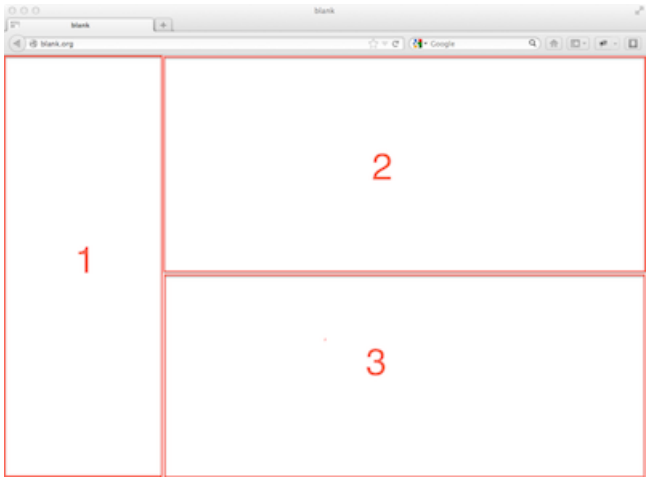
The first attempts at making the web more responsive and interactive introduced frames, iframes, and basic Javascript running the browser. For the purposes of this book we'll call this Web 0.1. More recently we have seen the rise of far more advanced Javascript libraries that make Ajax and UI Controls built with Javascript commonplace. We'll call this Web 2.0.

Web 0.1

What are frames and iframes? Why is it harder to tests web pages that have frames and iframes? Let's start this section with an exploration of frames and iframes.

Frames and iFrames

Let's begin our look at frames by thinking about an example. Our example is an email web application. Our main window might have three distinct sections like this:



Example Email Application

In this example, the pane labeled "1" might have a list of mail boxes or like *inbox*, *sent*, *trash*, etc. The pane labeled "2" would have a list of emails where it might display the *date*, *sender*, and the *email subject*. The pane labeled "3" would have the body of the email.

In the old days of the monolythic single page, updating one of the three panes would require a request to the server and the entire page (all three panes) would have to be rebuilt. This required a lot of processing on the server and the user experience was far from desirable. You would literally see the entire page refreshed.

Frames allow us to update only one of the panes and leave the other two intact. For example, if we wanted to updated pane 3 when somebody selects a different item from the list in page 2 we would simply make a server call to get the contents for the new email which would be returned as an HTML page for that pane.

Let's talk about how this actual works. In our example page we are using a frameset[133] and each page is a frame[134]. The thing that is important for you to know is that the browser treats each frame as a completely separate page with its own DOM[135]. This makes things far more complicated. If you were using Watir or Selenium directly, you would have to specifically navigate down into each frame in order to locate and interact with the elements on that page. You would have to treat it like it was three separate pages.

The *PageObject* gem has a built-in way of making this far easier. Let's write a *Scenario* to see how this works.

PageObject frame support

Let's begin by creating a new feature file named *web01.feature*. As you know, this file should be in the *features* directory. Go ahead and add the following content:

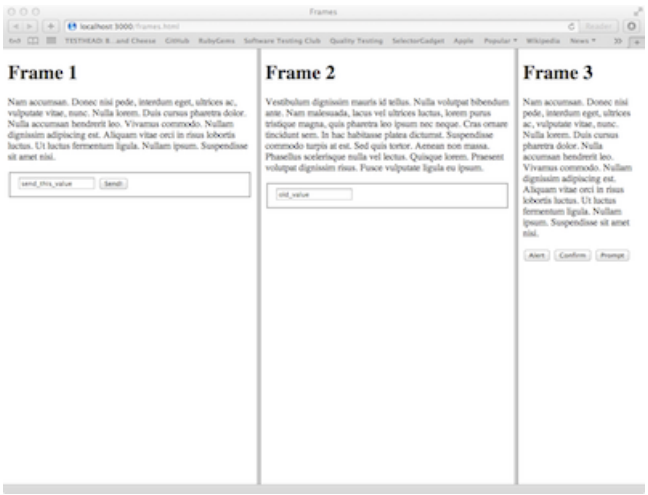
[133] http://www.w3schools.com/tags/tag_frameset.asp
[134] http://www.w3schools.com/tags/tag_frame.asp
[135] http://en.wikipedia.org/wiki/Document_Object_Model

```
Feature: Testing Web 0.1

  Scenario: Learning about Frames
    Given I am on the frames page
    When I send the text "frames are not my friend"
    Then the receiver should have "frames are not my friend"
```

Go ahead and generate the empty step definitions and you should have this (after renaming the parameters):

```
Given /^I am on the frames page$/ do
  pending
end

When /^I send the text "([^"]*)"$/ do |text|
  pending
end

Then /^the receiver should have "([^"]*)"$/ do |expected|
  pending
end
```

It's time to build our page object but first let's take a look at the page. If you have the puppy application running locally you can go to the url http://localhost:3000/frames.html or you can use http://puppies.herokuapp.com/frames.html to access the public site to see our target page.



Frames page

Our page has three frames. In the first frame we have a `text_field` and a `button`. When you enter text in the `text_field` and press the `button` the text is placed in the `text_field` of the second frame. That is all we want to test at this time so let's get started with our page object.

In the `pages` directory create a file named `frames_page.rb` and place the following contents:

```
1  class FramesPage
2    include PageObject
3
4    page_url "http://localhost:3000/frames.html"
5
6    in_frame(:id => 'frame_1') do |frame|
7      text_field(:sender, :id => 'senderElement', :frame => frame)
8      button(:send_to_receiver, :id => 'send', :frame => frame)
9    end
10
11   in_frame(:id => 'frame_2') do |frame|
12     text_field(:receiver, :name =>'recieverElement', :frame => frame)
13   end
14
15 end
```

The first part of this class should look very familiar. We are declaring our class, including the `PageObject` module, and then specifying our `page_url`. The remainder of the class introduces something new. You can see the standard calls to `text_field` and `button` but they are wrapped in a `in_frame` block and they pass an additional `:frame` locator. Let's explore this in more details.

Each time we want to declare some elements on a page that are inside a frame we need to wrap those declaraions with a call to `in_frame`. These calls can be nested as many times as needed. For example, if we have two nested frames we could declare something that looks like this:

```
1  in_frame(:id => 'outer_frame') do |frame|
2    in_frame(:id => 'inner_frame', frame) do |frame|
3      text_field(:sender, :id => 'senderElement', :frame => frame)
4      button(:send_to_receiver, :id => 'send', :frame => frame)
5    end
6  end
```

Notice how I passed the `frame` from the outer declaration to the inner declaration.

Not only do we have to declare the elements inside the `in_frame` block be we also have to pass the frame in as a locator. In the declaration of the `text_field` above we use

```
text_field(:sender_to_receiver, :id => 'senderElement', :frame => frame)
```

We are not only passing in the :id locator but we are also passing in the :frame that was a parameter to the block. After declaring elements inside the in_frame block we treat them exactly as we would any element not inside of a frame. In essence, we have neutralized the somewhat nasty effects frames can have on our tests. Let's move forward and complete this Scenario by completing the step definitions.

```
Given /^I am on the frames page$/ do
  visit_page(FramesPage)
end

When /^I send the text "([^"]*)"$/ do |text|
  on_page(FramesPage).send_message(text)
end

Then /^the receiver should have "([^"]*)"$/ do |expected|
  on_page(FramesPage).receiver.should == expected
end
```

The first step is very straight forward - we are simply telling the browser to go to the frames page as defined by our call to page_url. In the next step I need to do two things - I need to put the text in the text_field and I need to click the button. I decided to use a method that will do both so I will need to add this method to my page object shortly. In the final step I am simply getting the value from the text_field in the second frame and comparing it to our expected value. The final thing I need to do before running this test is add the missing method to my FramesPage page object.

```
  def send_message(message)
    self.sender = message
    send_to_receiver
  end
```

If you run the test now it will pass. Other than the declaration, we are not changing any code to handle the elements that are in the frames.

There is one more thing I wish to point out before we move on from our discussion on frames. Not only can use use the in_frame method in our class declarations, we can also use the call to find elements dynamically in methods. Here is an example of what that might look like.

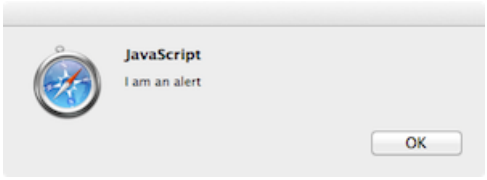
```
def adopt_puppy(name)
  in_frame(:id => 'frame_1') do |frame|
    button_element(:value => 'View Details', :index => lookup(name), :frame => fr\
ame).click
  end
end
```

Javascript Popups

Another challenge we sometimes face when testing application is handling Javascript popups[136]. I am happy to let you know that `PageObject` makes interacting with these beasts quite simple. There are three types of Javascript popups so let's get busy writing three Scenarios to demonstrate how it's done.

Alert Popups

The first type of popup we will work with is a simple alert[137] popup. It is simply a message box that has a single 'OK' button that dismisses the popup.



Alert Popup

Our frame page that we used in the last section has an alert popup in the third frame. This is the subject of our next Scenario. Add the following Scenario to the `web01.feature` file.

```
  Scenario: Testing with alert popups
    Given I am on the frames page
    When I popup the alert
    Then the text from the alert should read "I am an alert"
```

Next we need to generate our step definitions. In this case I think I'll go ahead and write the code to interact with the page object immediately after generating the steps. Here's what I came up with.

[136] http://www.w3schools.com/js/js_popup.asp
[137] http://www.w3schools.com/jsref/met_win_alert.asp

```
When /^I popup the alert$/ do
  on_page(FramesPage) do |page|
    @alert_text = page.alert_text
  end
end

Then /^the text from the alert should read "([^"]*)"$/ do |expected|
  @alert_text.should == expected
end
```

In these steps we are calling a new method named `alert_text` on our page object and saving the result in an instance variable named `@alert_text`. In the next step we are simply comparing the value in the instance variable against what we expected. There is nothing new here. The secret sauce is in our new method so let's get to it now.

```
  def alert_text
    in_frame(:id => 'frame_3') do |frame|
      @alert_text = alert(frame) do
        button_element(:id => 'alert_button', :frame => frame).click
      end
    end
    @alert_text
  end
```

There is a lot going on here so let's disect it and go over it piece by piece. The first thing happening in this method is the `in_frame` declaration. We are doing this because our alert popup is in a frame. We covered this in the last section so it should be familiar.

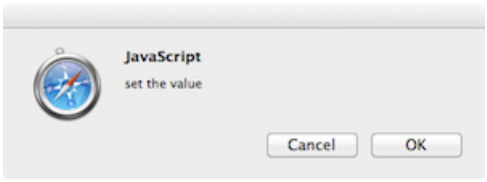
The next thing we are doing is declaring an `alert` block. In this case we are passing the `frame` parameter because we are inside the frame. If we were not in a frame we would simply call `alert` passing no parameters. This block will return the text that was displayed in the popup alert. In our case we are saving that text value in an instance variable named `@alert_text`.

Finally, inside the alert block we place the activity that causes the alert popup to occur. In our case it is the click of a button so we place the code there. The alert block will simply intercept the popup and click the 'OK' button for us. The final thing we do is return the `@alert_text` value returned from the alert block.

Run the test and see what happens. In page-object versions 0.7.2 and earlier you will not even see the alert popup. The gem is interecepting the call to popup the alert and just handling the call. In later versions you will see the popup briefly before it is dismissed by pressing the 'OK' button.

Confirm Popups

Confirm popups[138] are nearly identical to alert popups with one simple difference. Instead of the single 'OK' button, the confirm has two buttons - by default it has on 'OK' and 'Cancel' button.



Confirm Popup

Again, our frames page has a confirm popup so let's get busy writing a new Scenario.

```
1   Scenario: Testing with confirm popups
2     Given I am on the frames page
3     When I popup the confirm
4     Then the text from the confirm should read "set the value"
```

The steps are nearly identical to the ones for the alert example.

```
1  When /^I popup the confirm$/ do
2    on_page(FramesPage) do |page|
3      @confirm_text = page.confirm_text
4    end
5  end
6
7  Then /^the text from the confirm should read "([^"]*)"$/ do |expected|
8    @confirm_text.should == expected
9  end
```

Again, we are calling a new method on our page object that has the confirm popup magic. Let's take a look at that method to see how we handle confirms.

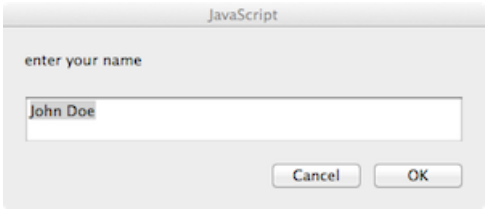
[138] http://www.w3schools.com/jsref/met_win_confirm.asp

```
def confirm_text
  in_frame(:id => 'frame_3') do |frame|
    @confirm_text = confirm(true, frame) do
      button_element(:id => 'confirm_button', :frame => frame).click
    end
  end
  @confirm_text
end
```

This method is extremely similar to the `alert_text` method. The only structural difference is that we are passing an additional parameter to the `confirm` method. The first parameter is either true or false and this determines what button is clicked. If you pass true the 'OK' or affirmative button is clicked and if you pass false the other button will be clicked.

Prompt Popups

Prompt popups[139] are more complicated that alert and confirm popups. The have a text prompt, a text field to capture a value, and a possible default value that populates the text field as well as an 'OK' and 'Cancel' button.



Prompt popup

Let's get straight into the code.

```
Scenario: Testing with prompt popups
  Given I am on the frames page
  When I popup the prompt and enter "Jared Gatorboy"
  Then the message from the prompt should read "enter your name"
  And the default value from the prompt should be "John Doe"
```

And the corresponding step definitions.

[139] http://www.w3schools.com/jsref/met_win_prompt.asp

```
When /^I popup the prompt and enter "([^"]*)"$/ do |value_to_enter|
  on_page(FramesPage) do |page|
    @prompt_response = page.prompt_value(value_to_enter)
  end
end

Then /^the message from the prompt should read "([^"]*)"$/ do |message|
  @prompt_response[:message].should == message
end

When /^the default value from the prompt should be "([^"]*)"$/ do |default|
  @prompt_response[:default_value].should == default
end
```

The first thing to notice here is that we are expecting a `Hash` to be returned from our page object method call. We are using that `Hash` with the keys `:message` and `:default_value` in the verification steps. Let's take a look at the new method on the `FramesPage` to see how this works.

```
  def prompt_value(value)
    in_frame(:id => 'frame_3') do |frame|
      @prompt_response = prompt(value, frame) do
        button_element(:id => 'prompt_button', :frame => frame).click
      end
    end
    @prompt_response
  end
```

Again, this method looks very similar to the previous popup methods. As you can see, we are passing the value to the block which is set in the text field. The value returned from the `prompt` block is returned to the caller. The main difference that is not obvious is that the call to `prompt` is returning the `Hash`. This `Hash` has two entries; the first is the message and has a key of `:message` and the second is the default value for the text field and it has the key `:default_value`. This has is used in the steps for the validation.

That is all there is to handling Javascript popups.

Popup Windows

In some cases where the basic Javascript popups were not enough web designers have turned to popping up new browser windows. In this section we will show you how to handle these situations and of course we'll do it by writing code. Let me introduce you to our Scenario.

```
Scenario: Testing with popup windows
  Given I am on the frames page
  When I popup a window
  Then I should be on a page with the text "Success"
  And I should be able to close the popup and return to the original window
```

We'll have to add three steps in order to complete this Scenario. Let's do that now.

```
When /^I popup a window$/ do
  on_page(FramesPage).popup_a_window
end

Then /^I should be on a page with the text "([^"]*)"$/ do |expected|
  @current_page.text.should include expected
end

When /^I should be able to close the popup and return to the original window$/ do
  on_page(FramesPage).return_from_popup
end
```

There are two methods that we will need to add to our `FramesPage` class but before we do that I want to add a declaration for the link that opens the new window. We'll do that by adding a new block at the top of the page under the two `in_frame` blocks we have already defined.

```
in_frame(:id => 'frame_3') do |frame|
  link(:popup_window, :text => 'Popup Window', :frame => frame)
end
```

This gives us the ability to click this link which will in turn open the popup window. Let's now take a look at the two new methods needed for the Scenario.

```
  def popup_a_window
    popup_window
    attach_to_window(:title => 'Success')
  end

  def return_from_popup
    attach_to_window(:title => 'Frames')
    attach_to_window(:title => 'Success') do
      button_element(:value => 'Close Window').click
    end
  end
```

There's actually a lot going on here so let's begin to disect what's happening. In the first method we are clicking the link that pops up the new window and then calling the `attach_to_window` method to change focus to the new window. In this example we are using the `:title` to identify the other window but we could easily have used the `:url`. When using the `:url` it is not necessary to include the full url. For example, in this case we could have used `:url => success.html`.

The second method is where things get a little confusing. Our goal in this method is to close the 'Success' window and return focus to the 'Frames' window. It would be easy to assume that I have the code backwards and what I really want to do is click the close button on the 'Success' window first and then attach to the 'Frames' window. Actually, that approach will not work. I have found out that closing a window when you are attached to it results in an error the majority of the times. Instead we need to first of all attach to the 'Frames' window. After that we can use a different version of the `attach_to_window` method - one that takes a block - to temporary attach to the 'Success' window, click the button, and then return control back to the 'Frames' window.

When we pass a block to the `attach_to_window` method it will switch control to the new window, execute the code in the block, and then return control back to the original window. After adding the methods to your page object the test should run successfully.

Web 2.0

As things progress in the world wide web, developers wanted to give the users a richer experience. They looked for ways to make web applciations seem more like the desktop applications that people were already familiar with. Along the way the created something called [Ajax](http://en.wikipedia.org/wiki/Ajax*(programming). Ajax allowed developers to make direct calls from _within* a page to a server, gather the results returned, and strategically update only the portions of the page that needed to change.

What happened next is an explosion of powerful Javascript libraries that build more advanced user interface elements and far more interactivity than the old static pages of yesterday.

The challenge we face when testing applications that use Ajax or highly dynamic pages is knowing when it is okay to access an element. Often, elements are either hidden or do not exist until some event on the page and accessing them before they are visible on the page will result in an error. The next two sections will demonstrate this problem and provide approaches you can take to make testing such sites much easier.

Playing the waiting game

When you have a page that has a lot of interactive Javascript you quickly learn that you need to add code to your tests that know when to wait for different things to happen on the page before continuing. The `PageObject` gem provides some nice methods to help us with this. Let's take a brief look at this ability prior to starting new Scenarios.

Waiting for something on a page

All pages have a method named `wait_until` which can be used like this short example. Let's say that we have declared a button with the following:

```
button(:continue, :value => 'Continue')
```

In our example this button is hidden until some Javascript event completes. In our test we want to wait until this element is visible and then we want to click it. We can write the following code:

```
def continue_when_ready
  wait_until do
    continue_element.visible?
  end
  continue
end
```

The `wait_until` call will invoke the block about every .5 seconds and will continue when the block returns true. In this case we are asking the continue element if it is visible. When it is visible the method will return and we will continue to the next line which clicks the continue button. If the element is not visible within a certain amount of time an error is thrown. At the page level the default timeout is 30 seconds but you can override this in your call. Also, you can provide a custom message if you so desire. Here's an example of how to do this.

```
def continue_when_ready
  wait_until(5, "Continue element not visible in 5 seconds") do
    continue_element.visible?
  end
  continue
end
```

In this example we are overriding the default 30 second timeout and providing a custom message that will be displayed if the element is not visible within 5 seconds.

Waiting for an element

Just like we have a method on the page that assists with waiting, `PageObject` provides five methods on `Element` that help with the same type of situations. The methods are:

```
when_present(timeout=5)
when_not_present(timeout=5)
when_visible(timeout=5)
when_not_visible(timeout=5)
wait_until(timeout=5, message=nil)
```

The first thing you should know is that the timeout is 5 seconds instead of 30. Secondly, only the `wait_until` method takes a message parameter. Finally, the first four methods will return the `Element` on which they were called. Let's look at how we might rewrite the example in the previous section to take advantage of one of these methods.

```
def continue_when_ready
  continue_element.when_visible
  continue
end
```

or we could take advantage of the fact that the `when_visible` method returns the element and shorten it to this:

```
def continue_when_ready
  continue_element.when_visible.click
end
```

The present methods (`when_present` and `when_not_present`) check for existence. They will return when the element either exists or ceases to exist.

Our waiting Scenario

Finally we are ready to put our newly found knowledge to work. For our example we will be writing a Scenario that uses on of Googles example pages for the GWT[140] framework. Let's begin by creating a brand new feature file named `web20.feature`.

The page we will be testing can be found at http://gwt.google.com/samples/DynaTable/DynaTable.html. Take a few seconds to play with the page and see how it behaves. We'll be selectng checkboxes and validating the values in the table.

Now that you are familiar with the page let's get started with our Scenario.

[140] https://developers.google.com/web-toolkit/

```
Feature: Testing Web 2.0

  Scenario: Dynamic Table Example
    Given I am on the google DynaTable Example page
    When I select the schedule for Monday
    Then I should see that Inman Mendez has a class at Mon 9:45-10:35
    And I should see that Omar Smith has a class at Mon 4:30-5:20
    When I select the schedule for Tuesday
    Then I should see that Eddie Edelstein has a class at Tues 12:15-1:05
    And I should see that Teddy Gibbs has a class at Tues 10:00-10:50
    And I should see that Teddy Gibbs has a class at Tues 3:15-4:05
```

Let's add the three new steps in a new file named `web20_steps.rb`. They should look like this:

```
Given /^I am on the google DynaTable Example page$/ do
  pending
end

When /^I select the schedule for (.+)$/ do |day_of_week|
  pending
end

Then /^I should see that (.+) has a class at (.+)$/ do |name, schedule|
  pending
end
```

We also can write the beginning of our page object for this page.

```
class DynaTablePage
  include PageObject

  page_url "http://gwt.google.com/samples/DynaTable/DynaTable.html"
  button(:none, :value => 'None')
  checkbox(:sunday, :id => 'gwt-uid-1')
  checkbox(:monday, :id => 'gwt-uid-2')
  checkbox(:tuesday, :id => 'gwt-uid-3')
  checkbox(:wednesday, :id => 'gwt-uid-4')
  checkbox(:thursday, :id => 'gwt-uid-5')
  checkbox(:friday, :id => 'gwt-uid-6')
  checkbox(:saturday, :id => 'gwt-uid-7')
  table(:dyna_table, :class => 'table')
end
```

Since I am not quite sure what methods I need on my page object I'll do a little design by intention in my step definitions. It is obvious what I need for the first two steps so let's go ahead and write those now.

```
Given /^I am on the google DynaTable Example page$/ do
  visit_page(DynaTablePage)
end

When /^I select the schedule for (.+)$/ do |day_of_week|
  on_page(DynaTablePage).select_schedule_for day_of_week
end
```

The first step is simply navigating to the page. Nothing exciting here. The second method, on the other hand, is passing the `String` from the Scenario to the page to determine what checkbox to select. We are calling the method like this `-> select_schedule_for 'Monday'`. There will be a couple of challenges so I think we should add this new method to our page object now. Here is what I came up with.

```
  def select_schedule_for(day_of_week)
    none
    self.send "check_#{day_of_week.downcase}"
  end
```

The first thing we are doing is pressing the 'None' button to clear out all previous selected checkboxes. We knwo that declaring a checkbox causes it to generate several methods.

```
  checkbox(:sunday, :id => 'gwt-uid-1')

  # generates
  def check_sunday
  def uncheck_sunday
  def sunday_checked?
  def sunday_element
  def sunday?
```

We can take advantage of this and call the "check_

Now we have to get back to that last step definition. This is where the waiting needs to take place. We need to ask ourselves if the waiting should occur in the page or in the steps.

First of all, I want to ask the page for the schedule for a specific person. Should I wait until that value appears and then return it? It seems more natural in this instance to have the waiting code in the page object so my step is very simple.

```
Then /^I should see that (.+) has a class at (.+)$/ do |name, schedule|
  on_page(DynaTablePage).class_schedule_for(name).should include schedule
end
```

Our page object needs a new method. Let's take a shot at writing it.

```
  def class_schedule_for(person)
    row = nil
    wait_until(2) do
      row = dyna_table_element.find { |row| row[0].text == person }
    end
    row[2].text
  end
```

The first thing we need to do is try to find the correct row and column in the table. We are doing this with the `find` method from `Enumerable`. Once we get the value from the column we assign it to an instance variable. We are return true if the row is found which causes us to leave the block and return the value. Luckily, `PageObject` has a simpler way for us to do this. When we access a row (or column) using the `[]` call we typically use a number. If we pass a string, it is assumed to be a header and it is treated as such. For example, passing a `String` for the row will look down the first column to try to find a match. Likewise, passing a `String` for the column will look at the column headers across the first row for a match. Let's rewrite our example to use the feature.

```
  def class_schedule_for(person)
    row = nil
    wait_until(2) do
      row = dyna_table_element[person]
    end
    row[2].text
  end
```

That's simpler but we can really combine the two rows into a single call and clean this up further by replacing the magic number with the column title.. We'll use the column heading "Schedule" to find the proper column for our query.

```
  def class_schedule_for(person)
    schedule = nil
    wait_until(2) do
      schedule = dyna_table_element[person]['Schedule'].text
    end
    schedule
  end
```

Overriding the default timeout values

If you find yourself constantly overriding the default timeouts for either the page or element level waiting methods you can globally change the defaults by adding the following to your `env.rb` file:

```
PageObject.default_page_wait = 10
PageObject.default_element_wait = 2
```

This call will set the global default timeout for page calls to 10 seconds and the global default timeout for element calls to 2 seconds. Again, this only applies to the methods described in this 'Web 2.0' section.

Waiting for Ajax

There is one more thing we need to cover in this chapter. I was working with a team that had a landing page that loaded very quickly and then proceeded to make numerous Ajax calls to populate each section of the page. In their test they wanted to wait until all sections were populated before the individual tests began. They tried to determine which section would be the last to load that started to wait for one of those elements to be present but frequently that section would not be last. What they really wanted was for the test to wait until all of the Ajax calls completed regardless of which one completed first or last. To solve this problem I added the following method to all pages.

```
def wait_for_ajax(timeout = 30, message = nil)
```

This method will wait until all current pending Ajax requests have completed. In order for it to work you need to let it know what Ajax Framework[141] you are using. Currently, `PageObject` only supports jQuery[142] and Prototype[143] with early support for YUI[144] framework. You inform `PageObject` which one you intend to use by declaring it in your `env.rb` file like this:

[141] http://en.wikipedia.org/wiki/Ajax_framework
[142] http://jquery.com
[143] http://www.prototypejs.org
[144] http://yuilibrary.com

```
PageObject.javascript_framework = :jquery
```

Adding your own Javascript Framework

The `wait_for_ajax` method and corresponding code is completely extensible. If you want to add your own Javascript Framework you simply have to follow these steps.

Create a new `Module` that has a single method that returns a `String` containing the Javascript necessary to get the current number of pending Ajax requests. Here's the `Module` for jQuery from the `PageObject` gem.

```
module JQuery
  #
  # return the number of pending ajax requests
  #
  def self.pending_requests
    'return jQuery.active'
  end
end
```

Once you have this `Module` you simply need to register it with the `PageObject` gem and then express your intent to use this new Framework. Here's an example that demonstrates how to do this with a fictitous framework and Module. This code would be placed in the `env.rb` file.

```
PageObject.add_framework(:silly, SillyModule)
PageObject.javascript_framework = :silly
```

Now everytime I call the `wait_for_ajax` method it will pass the Javascript returned from your `Module` to the browser and wait until it gets and answer of zero.

11. Running your tests

It might seem a little strange having an entire chapter dedicated to running our tests. After all, we've been running our tests repeatedly as we develop our examples.

Managing environment configuration

In chapter six we installed the puppy application locally. When we attempted to run our tests we discovered there were three things that needed to change with the new environment. They were the base url, username, and password. It is very common to have a need to run your tests in different test environments and it is also common to have to change some of the values we use in our tests as we move from environment to environment. Let's see how we can easily configure this and have our tests automatically pick up the correct values.

It will come as no suprise to you that there is a gem that can help us with this. The gem is called fig_newton[145]. This gem behaves very similar to data_magic[146] in that it uses yaml files[147] for the data and has a default directory in which it looks for the files. Let's configure our project to use this gem.

FigNewton

The first thing we need to do is make the gem available to our project. We'll do this by adding it to our Gemfile and then executing `bundle install`.

```
1     gem 'fig_newton'
```

Now that our project knows about the `fig_newton` gem we can begin to use it. We'll start by opening the `env.rb` file and add the following lines:

```
1     require 'fig_newton'
2
3     FigNewton.load('local.yml')
```

By default *fig_newton* will look for files in the `config/environments` directory. Let's create a simple file named `local.yml` in that directory and place the following contents into it.

[145] http://github.com/cheezy/fig_newton
[146] http://github.com/cheezy/data_magic
[147] http://en.wikipedia.org/wiki/YAML

```
    base_url: http://localhost:3000

    admin_username: steve
    admin_password: secret
```

Finally we need to update the two page objects to use the values from the yaml file. You can simply call a method on the `FigNewton` module that is the same as the key. For example, the `HomePage` class should be updated to this:

```
class HomePage
  include PageObject
  include SideMenuPanel

  page_url FigNewton.base_url

  # ...
end
```

and update the `LandingPage` class to this:

```
class LandingPage
  include PageObject
  include SideMenuPanel

  page_url "#{FigNewton.base_url}/admin"

end
```

The `login_to_system` method on the `LoginPage` class should be updated to this:

```
def login_to_system(username = FigNewton.admin_username, password = FigNewton.adm\
in_password)
  self.username = username
  self.password = password
  login
end
```

If you have a lot of configuration information, `fig_newton` allows you to nest items to make your yaml file more readable. Here is the above example modified to demonstrate nesting:

```
base_url: http://localhost:3000

admin:
  username: steve
  password: secret
```

In order to use the nesting you simply chain together the calls to get to the information you want. For example, to use the `username` field above you would simply make a call to `FigNewton.admin.username`.

You should be able to run the `adopting_puppies.feature` feature file and everything should work just fine as long as you have the puppy application running locally. But how do we add the configuration for the heroku site? Let's start by creating a new file in the `config/environments` directory named `heroku.yml` and add the following contents:

```
base_url: http://puppies.herokuapp.com

admin_username: admin
admin_password: password
```

You can now change the line in the `env.rb` file to load the file we just created:

```
FigNewton.load('heroku.yml')
```

Run the tests and they should work again but you could possibly see an error with the tests that are trying to access the database directly since we do not have access to the database of the heroku version of the puppy application. Nevertheless, we did get the correct configuration information. But it is not nice to have to go to our `env.rb` file and change the line to load the correct file each time we run in a different environment. Never fear. `fig_newton` has a solution for this.

When `fig_newton` is running it will use the filename you specify in your call to `FigNewton.load`. If you do not call this method it will next look for the value of an environment variable named `FIG_NEWTON_FILE`. If it finds this environment variable it will be used. If it did not find the environment variable it will look for a file that is named after the host name of the computer. For example, if the hostname is *systemtest*, *FigNewton* will look for a file named `systemtest.yml`. Finally, it will use the filename `default.yml` if you do not call `load` and the environment variable is not set. We can use the environment variable to our advantage to help us choose which file to use without making any changes to the `env.rb` file. Here's how.

One of the files that was generated in chapter four when we generated our cucumber project was a file named cucumber.yml[148]. This file is a yaml file that defines things called profiles. A profile is nothing more that a set of command-line parameters that are passed to cucumber when it executes. By default cucumber will use a profile named `default`. The file generated by testgen[149] has the following contents:

[148] https://github.com/cucumber/cucumber/wiki/cucumber.yml

[149] http://github.com/cheezy/testgen

```
default: --no-source --color --format pretty
```

To get started I want us to make two small changes. First of all, please delete this line from your `env.rb` file:

```
    FigNewton.load('heroku.yml')
```

and then update your `cucumber.yml` file to this:

```
default: FIG_NEWTON_FILE=local.yml --no-source --color --format pretty
```

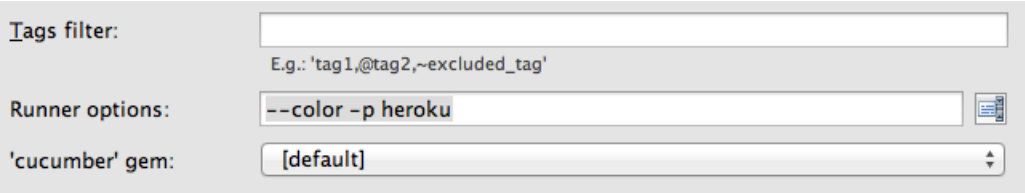
When we run our tests it should use the local configuration and everything should work fine. But how do we add the heroku configuration? We simply need to add a new profile to our `cucumber.yml` file and then direct cucumber to use the other profile when we want to point to that site. Here's the updated file:

```
default: FIG_NEWTON_FILE=local.yml --no-source --color --format pretty
heroku:  FIG_NEWTON_FILE=heroku.yml --no-source --color --format pretty
```

Now if we want to run our tests to point at the heroku site we need to tell cucumber to use the `heroku` profile. You can do this by using the `-p` command-line option like this:

```
bundle exec cucumber -p heroku features/adopting_puppies.feature
```

If you are using RubyMine you can simply modify the run configuration. Select the *Edit Configurations ...* option under the *Run* menu. In the dialog you should select the item `adopting_puppies` under the *Cucumber* heading on the left hand side. Finally, you should add `-p heroku` to the *Runner options:* field.



RubyMine Runner options

We'll learn how to leverage cucumber profiles more in the next section.

Creating a script to run our tests (Rakefiles)

Rake[150] is the gem most Ruby projects use to build their application and run the applications tests. When using *rake* you have to create a file at the root directory of your project named *Rakefile*. If you used the *testgen* gem to generate the shell for your project it created a simple *Rakefile* for you.

[150] http://rake.rubyforge.org

```
1 require 'rubygems'
2 require 'cucumber'
3 require 'cucumber/rake/task'
4
5 Cucumber::Rake::Task.new(:features) do |t|
6   t.profile = 'default'
7 end
8
9 task :default => :features
```

This file creates a new task on line 5 named *features* that uses the *default* profile. You can run this task by executing `rake features` in the top level of the project structure.

Rake also supports the notion of namespaces[151]. In *rake* a namespace[152] is simply a way to group similar tasks. Here's an example taken from the *Rakefile* for the *page-object* gem:

```
1  namespace :features do
2    Cucumber::Rake::Task.new(:watir_webdriver, "Run features with Watir") do |t|
3      t.profile = "watir"
4    end
5
6    Cucumber::Rake::Task.new(:selenium_webdriver, "Run features with Selenium") do \
7  |t|
8      t.profile = "selenium"
9    end
10
11   desc 'Run all features using watir and selenium'
12   task :all => [:watir_webdriver, :selenium_webdriver]
13 end
```

This partial *Rakefile* defines three tasks. The first one is features:watir_webdriver, the second is features:selenium_webdriver, and the final is features:all. They each control the way that the features are run and defer out to the *cucumber.yml* file to define the profiles / differences. The features:all task simply runs the other two tasks.

If you do not create profiles[153] in your *cucumber.yml* file you can pass command-line options to cucumber via a *Rake* task like this:

[151] http://en.wikipedia.org/wiki/Namespace_(computer_science)
[152] http://theadmin.org/articles/organizing-rake-using-namespaces/
[153] https://github.com/cucumber/cucumber/wiki/cucumber.yml

```
Cucumber::Rake::Task.new(:features) do |t|
  t.cucumber_opts = "features --format pretty"
end
```

The *cucumber_opts* get passed to the cucumber command-line when it runs if you use this task.

Another common strategy is to define different tasks to run different subsets of your feature files. These features usually have tags to identify which *rake* task will run them. That brings us to the next section on using tags.

Using tags to control test execution

Tags[154] are a great way to control which scenarios run when we execute cucumber. A tag is simply a label you place on a feature or scenario that begins with the @ character. Cucumber can use these tags to treat them differently at runtime. Let's explore three examples of how they can be used on our projects.

I'm not ready - don't run me

When we follow an ATDD workflow we will be writing acceptance tests prior to the developer writing the code that makes them pass. When our tests run we do not want incomplete tests to execute because they will fail. One way to handle this scenario is to add a tag to the features that are in process.

```
@not_ready
Feature:  Some feature that is define but not complete at this time
```

We place a `@not_ready` tag on the feature. We now need to instruct cucumber to not run all Features / Scenarios that have the `@not_ready` tag. We can do that by adding the appropriate entry in our *cucumber.yml* file.

```
default: --no-source --color --format pretty --tags ~@not_ready
```

The last part of this line - `--tags ~@not_ready` tells cucumber to run all of the features except the ones that have the `@not_ready` tag. When the tests and production code are complete, all tests for the new feature run cleanly, and all code has been checked into the source code repository you can simply remove the tag and check the changes in. The next time your test suite runs it will pick up this new addition.

[154] http://github.com/cucumber/cucumber/wiki/Tags

Dividing tests by speed

There are sometimes when you end up with tests that are painfully slow. In fact, they are so slow that you may not want to run them with your regular test suite since they will cause each run to take too long. We can use tags to help us here. In this case we could place a `@slow` tag on the scenarios that are slow and we could make the two entries in our *cucumber.yml* file:

```
slow: --no-source --color --format pretty --tags ~@not_ready --tags @slow
fast: --no-source --color --format pretty --tags ~@not_ready --tags ~@slow
```

With this setup it would be very easy to create three *rake* tasks that could run either the slow, the fast, or all tests. If you want to run the tests from the command line you have the choice of which profile to run. To run only the fast tests you could do this:

```
bundle exec cucumber -p fast
```

We are using `bundle exec` in order to run the tests in the bundler sandbox and the `-p` option tells cucumber to run the provided profile.

Dividing tests by risk

Another approach I have seen is to divide tests by risk. If you have an area of the application that is very high risk then you will want to run these very often. By high risk I mean that the functionality of this area of the application is so important to the overall application that the team would look bad if there was a defect. A second tier would be an area in which you have found a lot of defects in the past. In both of these cases you could place a special tag on the corresponding features or scenarios. This would allow you to run those tests more frequently than you run the entire suite of tests.

When you are building software in an iterative fashion, in general, the most likely acceptance tests to fail are the ones that describe the behavior we are working on now (in the current iteration) because these are subject to a lot of change and are therefore volatile. The second most likely tests to fail are the ones we just completed (the previous iteration) because we are likely building on top of the previous code. I have seen teams place an iteration number tag on each feature file and run the current iterations features with the developers build - a build that is triggered each time a developer check into source control. The entire test suite is kicked off on a timer that should run as frequently as possible. For example, if it takes two hours for your entire test suite to run, then you should schedule the entire suite to run every two and one half hours while the current iterations features will kick off with each checkin (hopefully more frequently).

Configuring Continuous Testing (CI)

Twenty years ago we used to build code in teams for several months with limited interaction with the other teams working on the same application. At the end of this period there was something we called the "dreadded integration week". This was the week where we tried to bring all of the code together to see how it works. This was the week where we found out all of the mistakes we made. This was a week of high risk. This was the week when many people called in sick.

There was so much risk in waiting a long time to integrate our code that the original eXtreme Programming[155] team decided that we should do it all of the time (several times a day). By doing it all of the time we significantly reduce the risk by minimizing the amount of new code that is integrated. The called this Continuous Integration (CI)[156]. This section talks about how to leverage Continuout Integration[157] (or at least the CI server process) to run our tests continuously. In other words, testers are no longer responsible for running the tests. A machine should run them all of the time and we should only have to look at them if we are automatically notified that a test has failed.

Setting up Continuous Integration

The first thing you need to have in place in order to implement this practice is a CI server. It is likely that your development team already has a CI server and you should just add your builds to it. If you are doing this for the first time you will quickly discover there are many CI servers available. I am not going to explain how to select one or explain how to set it up. I'll leave that to the CI vendors. I would however recommend you place Jenkins[158] and TeamCity[159] on your short list for evaluation. Jenkins is an open source project while TeamCity is a commercial product.

Once you have the server installed you will want to install plugins to make it behave the way you wish. Jenkins has a Rake Plugin[160] that can be used to easily execute a *rake* task defined in a *Rakefile*. While your at it you might want to go ahead and install the generic Ruby Plugin[161] which makes it easy to run Ruby scripts. TeamCity has a Rake Runner Plugin[162] to make it easy to run your *rake* tasks.

Deciding what test to run and when to run them

In the last few sections we have discussed managing configuration, using *rake* to run your tests, and using *tags* to run different tests at different times. The overall goal in setting up a continuous testing environment is to have your tests run as often as possible in order to get quick feedback. If you are

[155] http://en.wikipedia.org/wiki/Extreme_programming
[156] http://en.wikipedia.org/wiki/Continuous_integration
[157] http://martinfowler.com/articles/originalContinuousIntegration.html
[158] https://wiki.jenkins-ci.org/display/JENKINS/Home
[159] http://www.jetbrains.com/teamcity/
[160] https://wiki.jenkins-ci.org/display/JENKINS/Rake+plugin
[161] https://wiki.jenkins-ci.org/display/JENKINS/Ruby+Plugin
[162] http://confluence.jetbrains.net/display/TW/Rake+Runner

segmenting them out into multiple runs it is desirable to have your most critical tests run as often as possible.

I usually go with the example described in the Dividing tests by risk section - run the most recent tests as part of the developers build and run all tests as frequently as possible. Tests that will not fit into this pattern (for example tests that require some massive setup or that access an external system that is not always available) should not slow dowan all other tests. Just have thos tests run as often as you can while having all other tests run continuously.

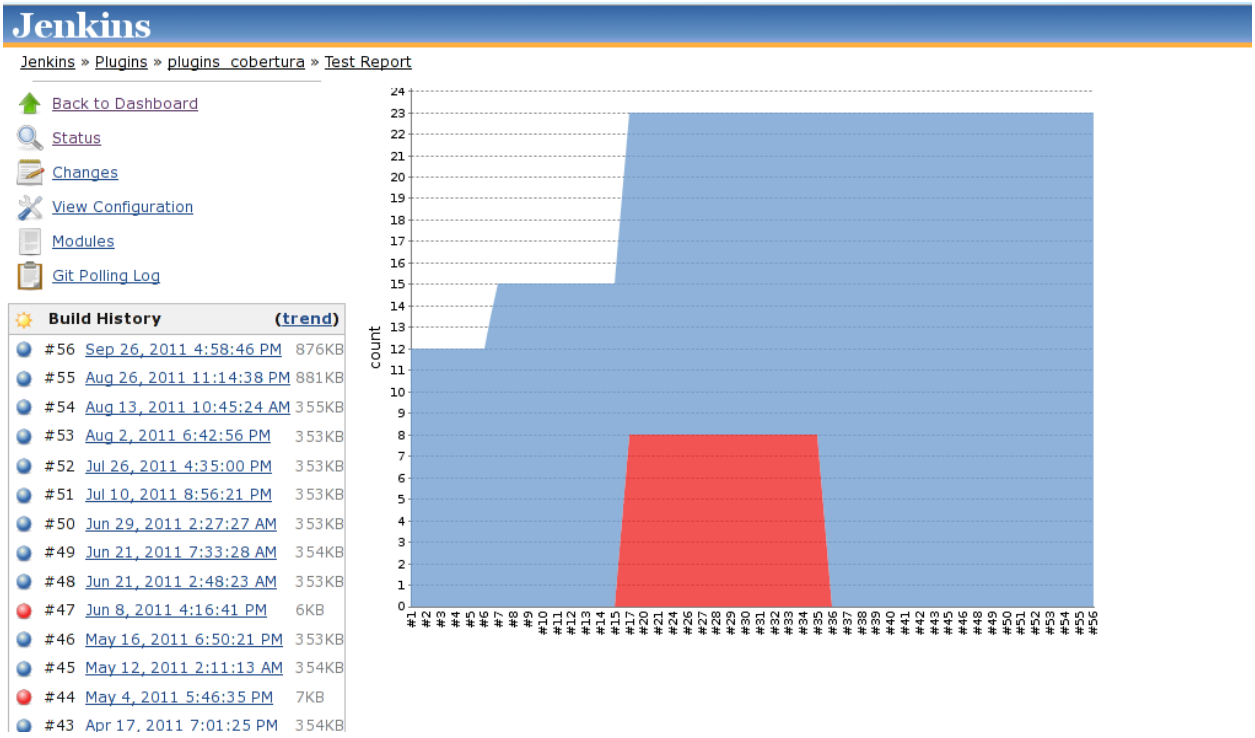
Reporting the results

When we have our Cucumber tests running on a server it would be a hassle to have to go read the command line output of the server process to see if all of the tests passed. We are in luck. Cucumber supports several output types. Let's look in detail about a couple of those and see how they make it easier to keep track of our test runs.

Cucumber's JUnit format

One of the cucumber report formats is the JUnit format. If you are using Jenkins[163] for your continuous integration then it is very simple to configure the server to display the JUnit results. You simply pass `--format junit --out build/testreport.xml` as command line parameters to cucumber and then enable "Publish JUnit test result report" as a post-build action in Jenkins pointing it to the output XML files. When this is setup you should see a report that looks something like this:

[163] https://wiki.jenkins-ci.org/display/JENKINS/Home



JUnit Report in Jenkins

The problem with the JUnit report is that there is not a lot of information available to you about the test details. This graph is nice but it is not enough.

Cucumber's Html format

Cucumber also has a built-in html report that is easy to configure. You can generate this report by passing `--format html --out testresults/report.html` as command line parameters to cucumber.

Cucumber Html Report

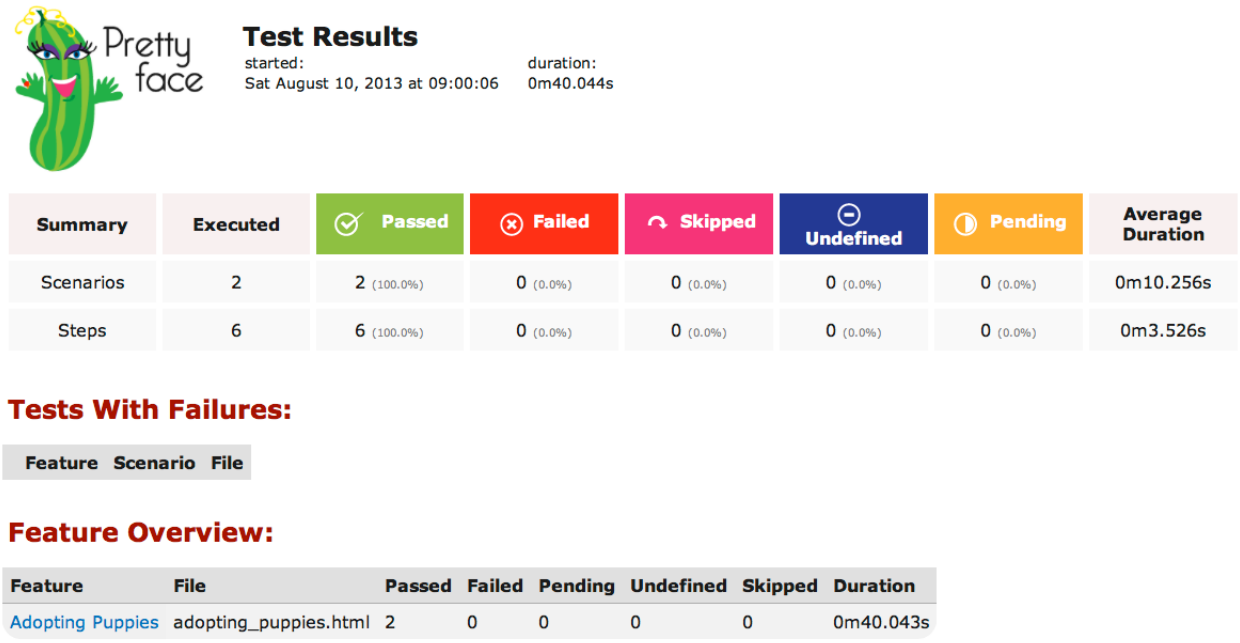
If we are going to produce a report it would be nice to make it easily accessible from the Jenkins build pages. There is another Jenkins plugin that makes that easy. It is the HTML Publisher Plugin[164]. Simply follow the instructions on the plugin page and you will see a link to your reports appear on the build pages.

Let me show you a Pretty Face

Although Cucumber's Html report is easy to setup and use there are a few drawbacks. The first is that it places the entire output for all Features in a single file. If you have a test suite that contains hundreds or thousands of *Scenarios* this file can become quite large. Also, there is no easy way to jump straight to the *Scenarios* that fail. Instead, you wil have to scroll through the file until you see something that looks like the error output. Finally, the report does not look very nice.

pretty_face is the name of a gem the produces an alternative HTML output for Cucumber. There are many differences between the standard Html report and the *pretty_face* report. First of all, *pretty_face* does not place the entire report in a single page. Instead, it creates a nice landing page that contains overview information about the test execution. This landing page also contains a section where it lists all *Scenarios* that had an error and includes a quick link so you can go directory to it to investigate. The final thing on the landing page is a list of each *Feature* with a link to the page containing the results of the execution of just that file.

[164] https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin



Test Results

started: Sat August 10, 2013 at 09:00:06

duration: 0m40.044s

Summary	Executed	Passed	Failed	Skipped	Undefined	Pending	Average Duration
Scenarios	2	2 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0m10.256s
Steps	6	6 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0m3.526s

Tests With Failures:

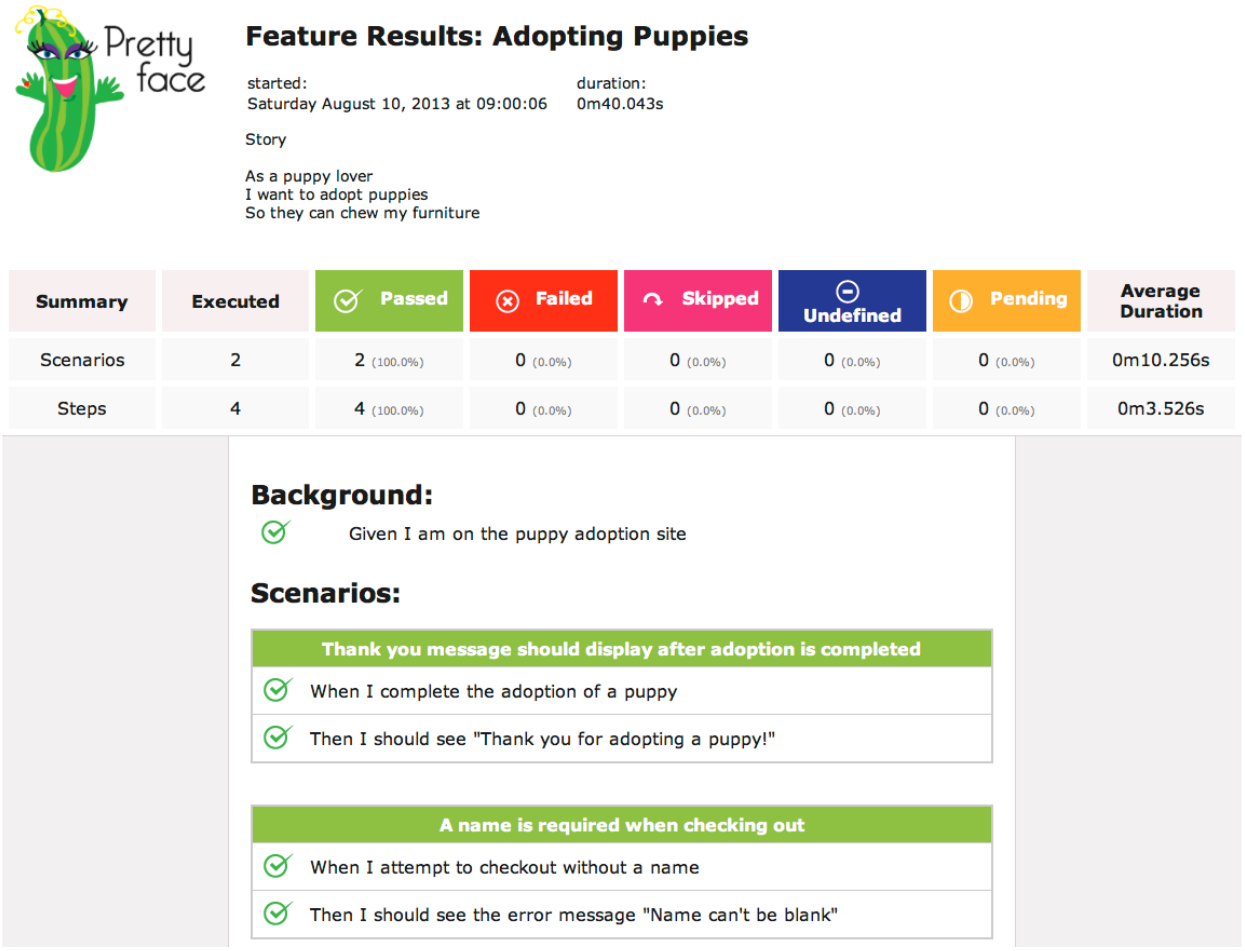
Feature	Scenario	File

Feature Overview:

Feature	File	Passed	Failed	Pending	Undefined	Skipped	Duration
Adopting Puppies	adopting_puppies.html	2	0	0	0	0	0m40.043s

pretty_face Landing Page

For each feature file in your test suite *pretty_file* will create an output file with the results from just that *Feature*.



pretty_face Feature Page

As you can see from the pictures above, the reports look much nicer than Cucumber's standard html report. But it doesn't end there. *pretty_face* allows you to customize the html output is several ways.

Customizing the pretty_face report

In order to add customizations to your *pretty_face* report you must first create a new directory inside your `features/support` directory named `pretty_face`. All customization files should be placed in this directory.

One of the customizations you can perform with *pretty_face* is to replace the lovely picture of the dog with another image. Some folks use this ability to display the company or team logo on the reports. To do this you simply place an image in the `pretty_face` directory. The image must be named `logo.<extension>` where the extension is either png, gif, jpg, or jpeg. This image will be displayed on all pages of the report. The image size should be close to 220 X 220 pixels.

Another customization you can perform is to replace the header on the main landing page. To perform this customization you must place a file in the `pretty_face` directory with the appropriate

html. The file must be named `_suite_header.erb`. Here is an sample demonstrating some of the content you can include:

```
<h1>The Code Monkeys</h1>
<h2>Test Results</h2>
```

In this case I am simply listing the team name and a title for the page. Again, this HTML will only be displayed on the main landing page for the report.

A final customization you can perform is to replace the header for each of the feature pages. This customization requires you to place a file in the `pretty_face` directory named `_feature_header.erb`. It should be simular to the previous HTML example.

Using the pretty_face report

In order to use *pretty_face* you need to do a little setup. First of all you need to add an entry to your Gemfile and run `bundle install`:

```
gem 'pretty_face'
```

The next thing we need to do is update our profiles so they use *pretty_face* to generate the reports. To do this you can open the `cucumber.yml` file and add the following:

```
--format PrettyFace::Formatter::Html --out results/index.html
```

The next time you run your cucumber tests you will see a pretty face.

The final thing I will mention about the *pretty_face* gem is that I know the authors of the gem (yes, I am one of them) and they have informed me that they plan to release a new version that will place a historical chart on the landing page. This chart will show pass/fail/pending etc from previous executions of your tests. I suspect you will see this released by the Summer.

Embedding a screenshot into your report

When a test fails you get a decent amount of information with either of the html reports above. Sometimes you might want to take a picture of the application at the time of the error and have it included in the report. This is very easy with both the standard Cucumber report and *pretty_face*. In both cases you can update your `After` hook to the following:

```
1 After do |scenario|
2   if scenario.failed?
3     filename = "error-#{@current_page.class}-#{Time.now}.png"
4     @current_page.save_screenshot(filename)
5     embed(filename, 'image/png')
6   end
7   @browser.close
8 end
```

Notice that `@current_page` is being used to get hold of a page object. On line 4 we are saving the image and on line 5 we are embedding it into the report only if the scenario fails.

Strategies to make our tests run faster

Let's face it - the scripts that make the application run end-to-end are slow. Wouldn't it be nice if we could speed up our tests? Wouldn't it be nice if our tests could run blazingly fast in order to provide feedback to our team in a matter of a couple of minutes?

The truth is that **there is little we can do to make our tests run faster**.

There are many in the industry that believe the proper way to make the tests run faster is to have them talk directly to the code bypassing the user interface. It is true that this will allow the tests to run faster but at the same time it introduces risks. The risks are that we will not be touching, and thereby testing all of the code. Another risk with this approach is that we will not be executing the code in the same way we would when running through the user interace. You will need to make the decision if this is right for you.

Faster feedback loops

In my opinion the real reason most people want the tests to run faster is so they can have faster feedback loops[165]. High performing teams[166] thrive on fast feedback loops. They use this feedback to validate quality, ensure they are moving in the correct direction, and to help them make the righ decisions as questions surface. Luckly for use there are many things we can do to shorten feedback loops in our tests.

Running our tests in parallel

One way to make our feedback loops shorter is to run our tests in parallel. What I mean by this is having more than one test run at a time.

[165] http://www.infoq.com/news/2011/03/agile-feedback-loops

[166] http://www.odysseyhps.com/high-performance-teams/common-problems

The first and simplest way to do this is to create multiple builds and run small number of critical tests more frequently to achieve faster feedback. Using the techniques identified in Use tags to control test execution is a very easy way to get started. We can have one build on our continuous server run critical tests very frequently to provide fast feedback while another build on the same server could run the remainder of the tests in a larger batch. Using this approach we get fast feedback on critical areas of the application and also get frequent feedback on the remainder.

Another similar technique is to use tags to identify functional areas of the application and have numerous builds that each run the tests for a functional area. In this pattern, if we change something in our application related to User Administration the tests related to this area will run as soon as the code is checked in. We get faster feedback since a small batch of tests run. I would suggest combining this approach with another build process that runs all of the tests for the application three of four times per day.

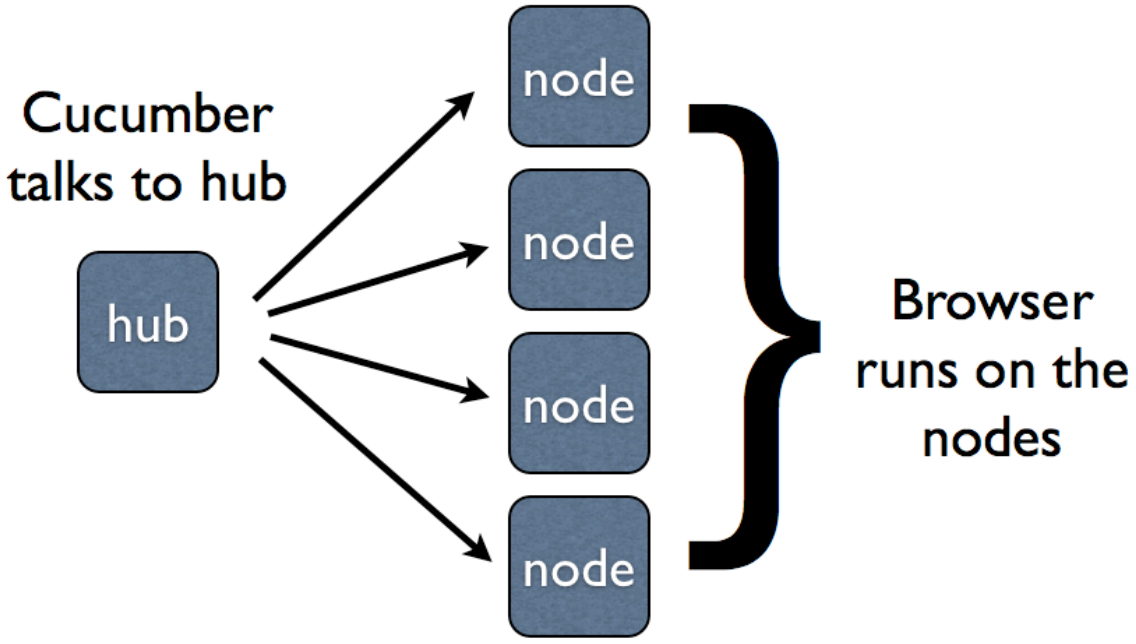
These simple techniques of dividing your tests using a tag and then setting up multiple builds on your continuous integration server to run the tests is a simple way to run them in parallel in order to tighten feedback loops. But there comes a time when we need to take further steps to help us achieve the tight feedback loops[167] we crave.

Setting up a Selenium Grid

If you are testing web applications using the methods described in this book then running your tests on a Selenium Grid[168] is another way to run your tests in parallel.

The way Selenium Grid works is you configure a single computer to act as your *hub*. You will also configure several computers to act as *nodes*. The next image shows what this might look like.

[167] http://37signals.com/svn/archives2/the_importance_of_instant_feedback.php
[168] http://code.google.com/p/selenium/wiki/Grid2



Selenium Grid 2

Let's define *hub* and *node* before moving on. The *hub* is the place that receives requests for browser interaction. Your tests will connect to the *hub* instead of connecting to the browser directly. When the *hub* receives a request for a browser connection it will find a *node* that is running that type of browser and establish the connection. The *node* in turn starts the browser and waits for further commands. Each time a request is sent to the *hub* it immediately forwards it to the proper *node*.

In order to run the grid you will have to have Java installed on each computer. If you have not already done this please refer to the section in the Mobile Testing chapter named Installing a Java Development Kit. Once you have it installed Java you simply download the latest selenium-server-standalone jarfile[169] and copy it to each of the computers in your Grid.

The next step would be to start the *hub*. Log into the *hub* computer and change to the directory that contains the selenium jar file and run the following command:

```
1 java -jar selenium-server-standalone-2.34.0.jar -role hub
```

The filename may be different depending on the version you downloaded.

Now that we have the *hub* running it is time to start each of the *nodes*. Again, you will need to log into the *node* computers one at a time and execute the following:

[169] http://code.google.com/p/selenium/downloads/list

```
java -jar selenium-server-standalone-2.34.0.jar -role node  -hub http://[hub_host\
name]:4444/grid/register
```

Again, the filename may be different and the *hub* url should be set to the name of the computer where the *hub* is running.

When starting up the *node* like this you accepting default configurations. By default, 11 browsers are started (5 Firefox, 5 Chrome, and 1 Internet Explorer). Also, by default the maximum number of concurrent tests is set to five. If you wish to override these default settings you will need to provide additional options at startup time. Here is the additional options you would need to provide when starting up the *node* if you only wanted to start three Firefox browsers and nothing else:

```
-browser browserName=firefox,version=23.0,maxInstances=3,platform=LINUX
```

Please see the Selenium Grid 2 documentation for a list of all options.

With the grid running we now need to modify our tests so they can connect to the *hub* which will in turn send the request out to run on one of the *node* computers. The only line of code you will need to change is the line where you create your instance of `Watir::Browser`. Change it to the following:

```
@browser = Watir::Browser.new(:remote, :url => 'http://[hub_hostname]:4444/wd/hub\
', :desired_capabilities => :firefox)
```

After you make this change go ahead and try to run a simple *Feature* to make sure everything is wired up properly.

Once you are sure you grid is properly configured we will need to make one final change in order to run multiple tests at the same time. I use a gem named parallel_test[170] to run more than one tests at a time.

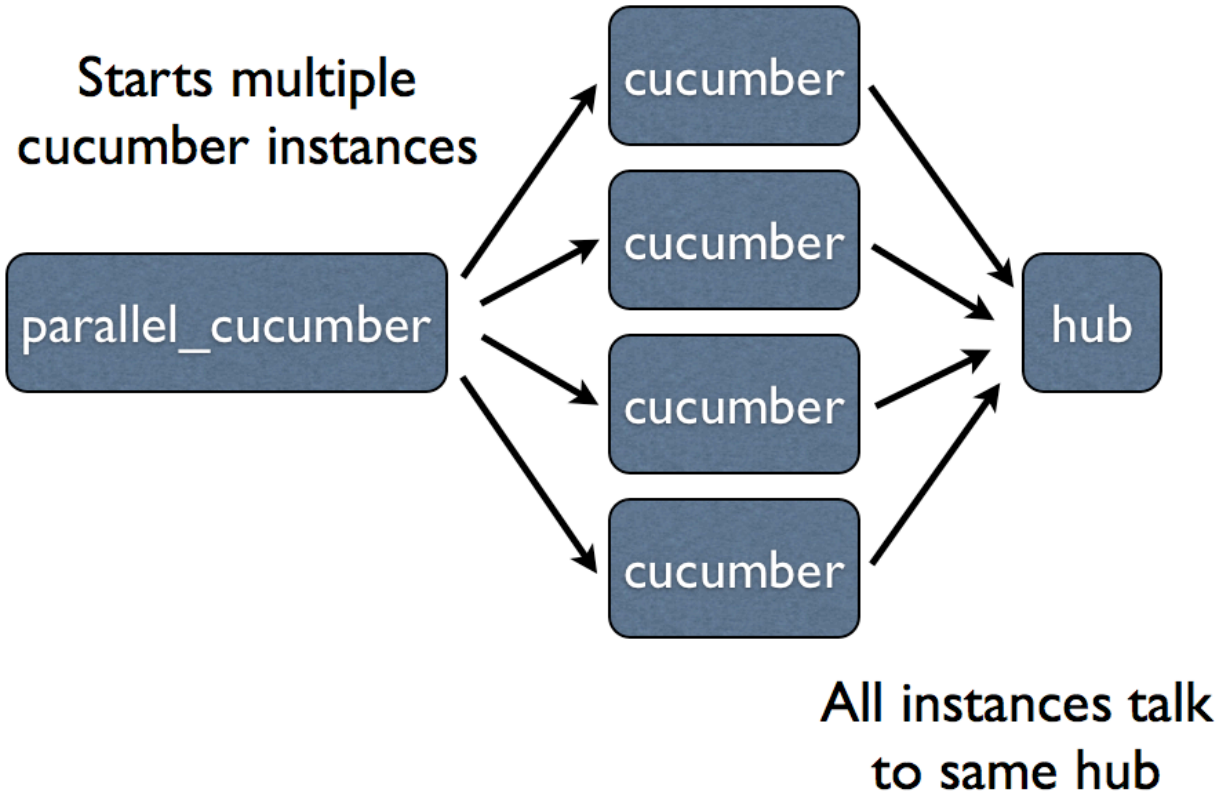
Note

At the time of this writing *parallel_tests* does not work on the Windows platform. This means that *parallel_tests* needs to run on a non-Windows computers but your *hub* and *node* can be on Windows computers.

Add the gem to your *Gemfile* and make sure you perform a `bundle update`. You now have a new command available to you - `parallel_cucumber`. There are many options available but the most common one is a `-n <num>` option where you tell `parallel_cucumber` how many processes to run

[170] https://github.com/grosser/parallel_tests

in parallel. One additional item to note is that you can create a profile in your *cucumber.yml* file named `parallel:` which will be automatically picked up by `parallel_cucumber`.



parallel_cucumber used with Selenium Grid 2

The configuration I usually have is one in which I run the *hub* on the same computer as my continuous integration server and *nodes* residing on several other computers. I then have a rake task that runs `parallel_cucumber` which executes my tests that connect to the *hub*.

12. Build your own Ruby gem

The first question we should ask ourselves is "Why would we want to build our own Ruby gem?". I think there are two reasons why we would want to do this. The first is that we have created some code that we found very useful and feel that others would find it useful as well. The second is that we have some functionality that we are using in a project and there are other projects that need to same capability. Clearly, in both cases we do not want to duplicate the code between the projects. A gem would seem to be the best choice.

Creating a Ruby Gem Project

The first thing we need to do is introduce our project. *page-object* has an extension point that allows us to define our own `Element` types and register them with the gem. At the time of this writing, there are three gems that take advantage of this extension point. They are jqueryui_widgets[171], gwt_-widgets[172], and gxt_widgets[173]. Each of these gems provides a nice high-level wrapper around a set of JavaScript[174] UI Controls. There currently is not a gem to provide the same wrapper for the very popular dojo toolkit[175]. The UI Library for dojo is called dijit[176] and it offers a nice set of ui controls to build robust applications. Our new gem will provide the wrapper so applications using dijit can easily be tested with *page-object*.

Believe it or not, we will be using bundler[177] to generate our initial project structure. Begin by executing:

```
1 bundle gem dojo_widgets
```

This will generate the initial project structure. We'll begin by updating a few of the generated files. If you open a file named *README.md* will will notice a few sections that have the word *TODO:*. This is how bundler notifies you that you should update something. The *README.md* file contains the markdown[178] that will be converted into HTML and displayed on the main project page in github[179]. After updating the *TODO* sections my README file looks like this:

[171] https://rubygems.org/gems/jqueryui_widgets
[172] https://rubygems.org/gems/gwt_widgets
[173] https://rubygems.org/gems/gxt-widgets
[174] http://www.w3schools.com/js/
[175] http://dojotoolkit.org
[176] http://dojotoolkit.org/reference-guide/1.9/dijit/
[177] http://bundler.io/v1.3/bundle_gem.html
[178] http://en.wikipedia.org/wiki/Markdown
[179] http://github.com

```
# DojoWidgets

This is a wrapper around some of the Dijit controls.  The purpose is to make appl\
ications that use this UI control set easier to test using the [page-object](http\
://github.com/cheezy/page-object) gem.

## Installation

Add this line to your application's Gemfile:

    gem 'dojo_widgets'

And then execute:

    $ bundle

Or install it yourself as:

    $ gem install dojo_widgets

## Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create new Pull Request
```

There are two additional files we need to look at here. We'll start with the simplest. Under the */lib/dojo_widgets* directory you will find a file named *version.rb*. This file contains the version number that us used for releases. Here is the entire file:

```
module DojoWidgets
  VERSION = "0.0.1"
end
```

I want my first release of the gem to be version `0.1` so I'll go ahead and update the file:

```
module DojoWidgets
  VERSION = "0.1"
end
```

The next file we want to look at is the heart of a Ruby gem. It is our gemspec[180] file. Here is the file that was generated.

```
# -*- encoding: utf-8 -*-
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'dojo_widgets/version'

Gem::Specification.new do |gem|
  gem.name          = "dojo_widgets"
  gem.version       = DojoWidgets::VERSION
  gem.authors       = ["Jeffrey S. Morgan"]
  gem.email         = ["jeff.morgan@leandog.com"]
  gem.description   = %q{TODO: Write a gem description}
  gem.summary       = %q{TODO: Write a gem summary}
  gem.homepage      = ""

  gem.files         = `git ls-files`.split($/)
  gem.executables   = gem.files.grep(%r{^bin/}).map{ |f| File.basename(f) }
  gem.test_files    = gem.files.grep(%r{^(test|spec|features)/})
  gem.require_paths = ["lib"]
end
```

Again, notice the lines that contain a *TODO:* indicator. We need to provide a summary and description. While we're at it we'll add a homepage and license. The final item to add is our dependencies. After making these updates our gemspec file looks like this:

```
# -*- encoding: utf-8 -*-
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'dojo_widgets/version'

Gem::Specification.new do |gem|
  gem.name          = "dojo_widgets"
  gem.version       = DojoWidgets::VERSION
  gem.license       = 'MIT'
```

[180] http://docs.rubygems.org/read/chapter/20

```
10    gem.authors       = ["Jeffrey S. Morgan"]
11    gem.email         = ["jeff.morgan@leandog.com"]
12    gem.description   = %q{Wrapper around dijit controls for use with page-object g\
13  em}
14    gem.summary       = %q{Wrapper around dijit controls for use with page-object g\
15  em}
16    gem.homepage      = "https://github.com/cheezy/dojo_widgets"
17
18    gem.files         = `git ls-files`.split($/)
19    gem.executables   = gem.files.grep(%r{^bin/}).map{ |f| File.basename(f) }
20    gem.test_files    = gem.files.grep(%r{^(test|spec|features)/})
21    gem.require_paths = ["lib"]
22
23    gem.add_dependency 'page-object', '>= 0.9.1'
24
25    gem.add_development_dependency 'cucumber', '>= 1.3.2'
26    gem.add_development_dependency 'rspec', '>= 2.13'
27  end
```

Notice that I've added the *page-object* gem as a dependency. This will cause an installation of the *page_object* gem whenever the *dojo_widgets* gem is installed (if it is not already installed of course). I've also added *cucumber* and *rspec* as development dependencies. We do plan to write tests and people that wish to contribute to the development of the gem will need both of these gems.

With all of this initial work out of the way we should go ahead and commit the code to our local git[181] repository, create a new github project[182], and push the code to that project repository.

Creating our first test

We will be using *Cucumber* to write functional tests that describe the behavior of our gem. We'll start by creating the directory structure needed to run cucumber. Create the following directories:

- features
- features/support
- features/support/pages
- features/step_definitions

Once these directories are created we will need to create a new file in the *features/support* directory named *env.rb*. Add these lines to the file:

[181] http://en.wikipedia.org/wiki/Git_(software)

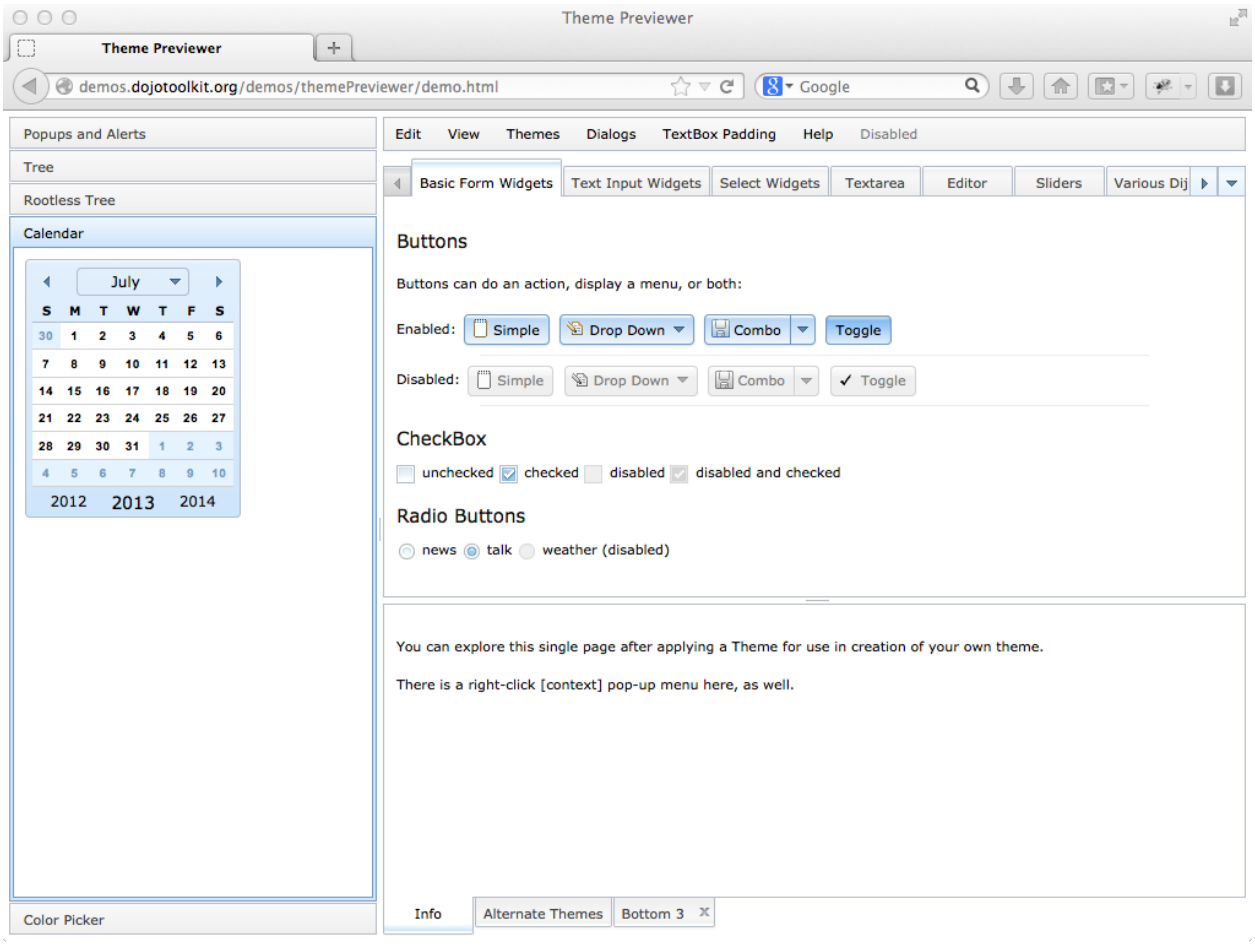
[182] https://github.com/cheezy/dojo_widgets

```
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', '..', 'lib'))

require 'rspec/expectations'
require 'dojo_widgets'
require 'page-object'
require 'watir-webdriver'

World(PageObject::PageFactory)

Before do
  @browser = Watir::Browser.new :firefox
end

After do
  @browser.close
end
```

The first line of this file adds our *lib* directory to the LOAD_PATH which makes it possible for us to simply require files found in that directory structure. The next four lines of code require three gems and the actual top-level `Module` for our gem. The next line of code registers `PageFactory` with the *Cucumber* `World` so we can call its' methods. The final two blocks create our `@browser` object. In this case I am hard-coding it to use a `Watir::Browser` object. In the final product I will change this so I can run my tests with either *watir_webdriver* or *selenium_webdriver*. All of the code in this file should look familiar so I am going to move along.

We are now ready to create our *Feature* file and populate it with *Scenarios*. Before we can do this we need to decide what will be included in the initial release of the gem. The dojo project has a demo page[183] where we can see the ui controls available. We'll have our tests use this page as our test page.

[183] http://demos.dojotoolkit.org/demos/themePreviewer/demo.html



Dojo demo page

There are so many controls available it might be hard to decide where to start. The first thing I see is that there are top-level containers that hold the individual elements. On the left of the screen is an accordion-like container and there is a tab container filling up the right portion of the screen. I think we'll start by creating a simple widget that is the Accordion container.

As always, we will start by writing a *Scenario*. Create a new file in the *features* directory named *accordion.feature* and add these contents:

```
1  Feature: The Accordion Widget
2
3    Scenario: Selecting a panel from the Accordion
4      Given I am on the dojo demo page
5      When I select the "Popups and Alerts" Accordion panel
6      Then the Accordion panel for "Popups and Alerts" should be visible
```

This is the simplest *Scenario* I could think of and it will drive out the creation of the `Accordion` class. After generating and writing the code for the step definitions it looks like this:

```
1  Given(/^I am on the dojo demo page$/) do
2    visit DojoDemoPage
3  end
4
5  When(/^I select the "(.*?)" Accordion panel$/) do |label|
6    on(DojoDemoPage).the_accordion.select_panel label
7  end
8
9  Then(/^the Accordion panel for "(.*?)" should be visible$/) do |label|
10   on(DojoDemoPage).the_accordion.panel_for(label).should be_visible
11 end
```

Add this code to a file in your *step_definitions* directory. As our steps reveal, we need a page object to represent this demo page. This page object has a method named `the_accordion`. Let's create that now by placing the following code in a file named `dojo_demo_page.rb` in the *features/support/pages* directory.

```
1  class DojoDemoPage
2    include PageObject
3
4    page_url 'http://demos.dojotoolkit.org/demos/themePreviewer/demo.html'
5
6    dojo_accordion(:the_accordion, :id => 'leftAccordion')
7
8    def initialize_page
9      wait_until(10) do
10       not div_element(:id => 'loader').visible?
11     end
12   end
13
14   def the_accordion
15     the_accordion_element
16   end
17 end
```

The `:id` value in the call on line six above was taken from the example page and corresponds to the `div` that is the main `Element` for the *Accordion*. I was able to find this using FireBug. Another item I should point out is that I have added an `initialize_page` method to our page object. I noticed when I loaded this demo page that there is an *div* that is displayed while the code for the page is being loaded. This *div* is hidden when the page loads. I wanted to ensure that I did not proceed with the test until the page was loaded so I added the `wait_until` block so the execution waits for up to ten seconds for the loaded *div* to be hidden. With our page object in place we are finally ready to run our test and eventually make it pass. To run our tests you can execute the following line:

```
bundle exec cucumber
```

Immediately we are presented an error:

```
undefined method `dojo_accordion' for DojoDemoPage:Class (NoMethodError)
/Users/cheezy/dev/dojo_widgets/features/support/pages/dojo_demo_page.rb:6:in `<cl\
ass:DojoDemoPage>'
/Users/cheezy/dev/dojo_widgets/features/support/pages/dojo_demo_page.rb:1:in `<to\
p (required)>'
```

The error is telling us that the `dojo_accordion` method does not exist on the `DojoDemoPage` class. We are in a good position now. We have a test that is failing and in order to make it pass we need to add the behavior of the Accordion widget.

Implementing the gem

A requirement for all gems is that when a top level `module` that exists in the *lib* directory and when it is loaded the gem should work. In our case that means that we should only have to require the `dojo_widgets.rb` file in our lib directory and it should, in turn, require everything else. The `DojoWidgets` module included in the file is our entrypoint for the gem.

In order to make the `dojo_accordion` method available we need to register it with *page_object*. We will add the necessary code to our `DojoWidgets` module.

```
require "page-object"
require "dojo_widgets/version"
require "dojo_widgets/accordion"

module DojoWidgets

  PageObject.register_widget(:dojo_accordion, DojoWidgets::Accordion, 'div')

end
```

On the first line we are requiring the `page-object` file as we are using it throughout this gem. On line three we are including `dojo_widgets/accordion`. It does not exist yet but it will shortly. Finally, on line seven we are registering a new widget with *page-object*. This is the call that adds the `dojo_accordion` method - the first parameter to the call. The other parameters are to inform *page-object* of the class that represents the *Widget* and the element type to look for when locating the `Element` on the page. The next thing I want to do is create an empty *Widget*. Create a file named *accordion.rb* in the *lib/dojo_widgets* directory and add the following:

```
module DojoWidgets
  class Accordion < PageObject::Elements::Div
  end
end
```

Notice that our `Accordion` class inherits from the `Div` class from *page-object*. All *Widgets* must inherit from `Element` or one of its' subclasses. When we run the test again we get the following output:

```
  Scenario: Selecting a panel from the Accordion
    Given I am on the dojo demo page
*** DEPRECATION WARNING
*** You are calling a method named select_panel at /Users/cheezy/dev/dojo_widgets\
/features/step_definitions/dojo_demo_steps.rb:6:in `block in <top (required)>'.
*** This method does not exist in page-object so it is being passed to the driver.
*** This feature will be removed in the near future.
*** Please change your code to call the correct page-object method.
*** If you are using functionality that does not exist in page-object please requ\
est it be added.
    When I select the "Popups and Alerts" Accordion panel
      undefined method `select_panel' for #<Watir::Div:0x007fc148adc968> (NoMetho\
dError)
      ./features/step_definitions/dojo_demo_steps.rb:6:in `/^I select the "(.*?)"\
 Accordion panel$/'
      features/accordion.feature:5:in `When I select the "Popups and Alerts" Acco\
rdion panel'
    Then the Accordion panel for "Popups and Alerts" should be visible

Failing Scenarios:
cucumber features/accordion.feature:3

1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m6.937s
```

This error is complaining because we tried to call a `select_panel` method on our `Accordion` object and it did not exist. When it attempted to pass it on to the driver (watir_webdriver in our case) it did not find the method either. Let's add that method now.

After looking at the structure of the Accordion on the page I determined that the way to select a panel is to find a *span* where the text is equal to the label and that also has a class of *dijitAccordionText*. Once I find that span I can simply click on it.

```
module DojoWidgets
  class Accordion < PageObject::Elements::Div

    def select_panel(label)
      span_element(:class => 'dijitAccordionText', :text => label).click
    end

  end
end
```

After running the test again I see that this step works but we are failing on the next step. Again, it is complaining that a method is missing - the `panel_for` method. This method is a little more compliceated. The first thing I need to do is get an `Array` of all of the panels in the *Accordion*. Next I need to walk through them and find the one that has the label we are looking for. Once I have pinpointed that panel, I simply have to find the correct nested `div` that is the panel contents.

```
module DojoWidgets
  class Accordion < PageObject::Elements::Div

    def select_panel(label)
      span_element(:class => 'dijitAccordionText', :text => label).click
    end

    def panel_for(label)
      panels = div_elements(:class => 'dijitAccordionInnerContainer')
      the_panel = panels.find do |panel|
        panel.span_element(:class => 'dijitAccordionText', :text => label)
      end
      the_panel.div_element(:class => 'dijitAccordionChildWrapper')
    end

  end
end
```

And with that final change our *Scenario* is passing. Since the purpose of this chapter is to introduce you to the process of creating Ruby gems we will not take the time to complete all of the gem's functionality in the book. There is still a lot to do with this gem but in the spirit of release early, release often[184] we will move forward with what we have now and make a release of the gem. Don't forget to commit the changes and push them to github.

[184] http://en.wikipedia.org/wiki/Release_early,_release_often

Release the gem

I always include a file named *ChangeLog* in all of my gems. It contains information about what is included with each new release of the gem. Create a file named *ChangeLog* in the root directory of the project and add the following contents:

```
=== Version 0.1 / 2013-7-20
* Included basic implementation for the Accordion widget
```

The next thing we need to do is make sure all of our changes are committed into the git repository and pushed to github.

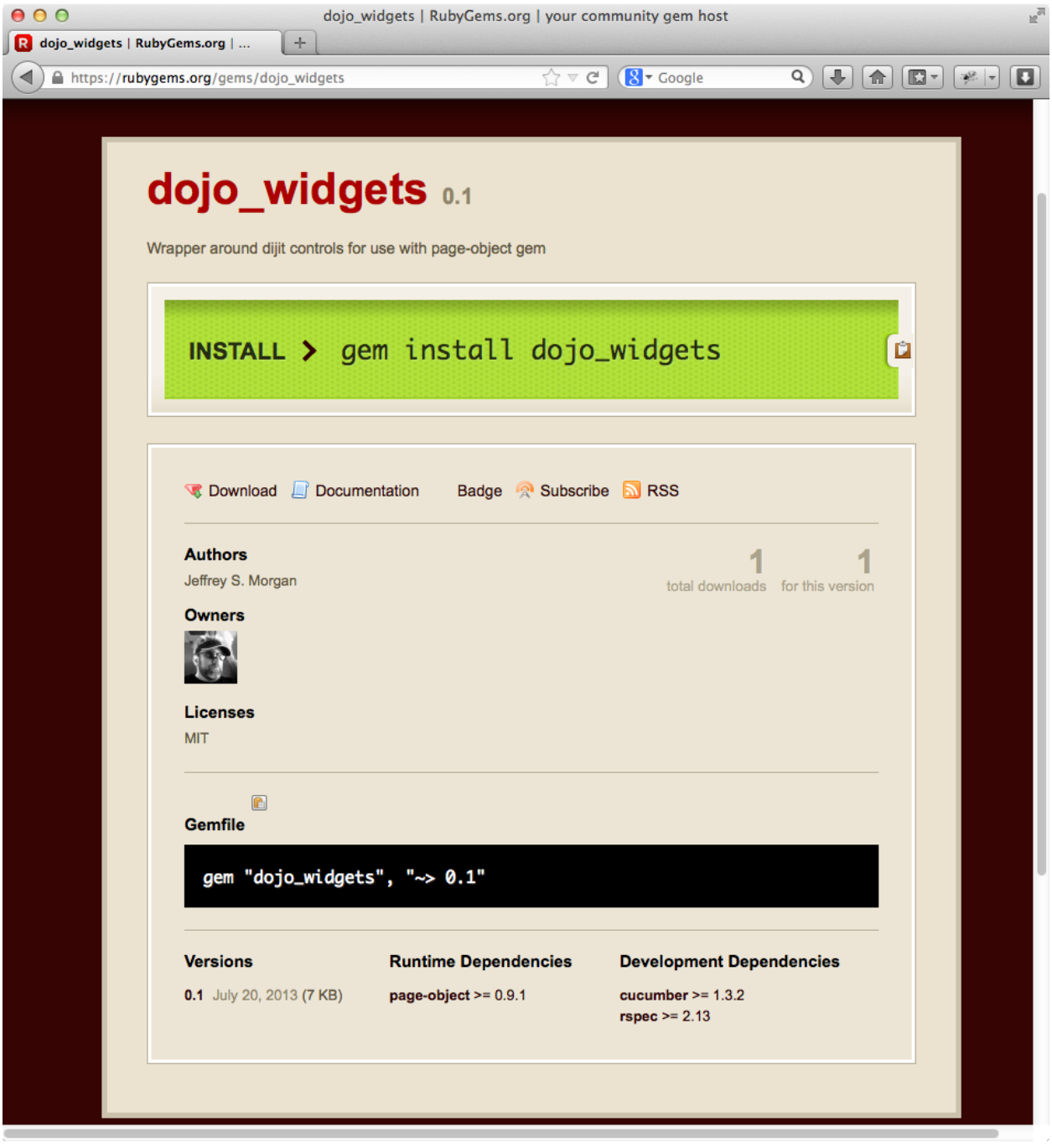
The final thing to do is to publish our gem. In order to do this you will need to create an account on rubygems.org. Once this account is created we need to tag the sourcecode and push our gem up to the rubygems.org site. Bundler added rake tasks to make this easy for us. We can get a list of all of the rake tasks by executing `rake -T`. Here is the output:

```
rake build    # Build dojo_widgets-0.1.gem into the pkg directory
rake install  # Build and install dojo_widgets-0.1.gem into system gems
rake release  # Create tag v0.1 and build and push dojo_widgets-0.1.gem to Rubyge\
ms
```

The first task, `rake build`, will simply build a gem file locally. `rake install` will build and install a new version of the gem locally. The final task, `rake release`, will tag the github repository with the appropriate tag, build the gem, and publish it to *rubygems.org*. Let's run the `release` target. Here's the output:

```
dojo_widgets 0.1 built to pkg/dojo_widgets-0.1.gem.
Tagged v0.1.
Pushed git commits and tags.
Pushed dojo_widgets 0.1 to rubygems.org.
```

If we go to *rubygems.org* and search for the new gem we will see the gem's page:



dojo_widgets page on rubygems.org

In this chapter we have created a new gem from scratch and published it to rubygems so others can install the gem

Appendix A - Watir-Webdriver Quick Reference

A.1 Getting Started

Load the Watir gem

```
require 'watir-webdriver'
```

Open a browser

```
browser = Watir::Browser.new :firefox
```

Open a browser at a specific URL

```
browser = Watir::Browser.start "http://www.google.com", :firefox
```

Go to a specific URL

```
browser.goto "http://www.google.com"
```

Close the browser

```
browser.close
```

A.2 Check Browser Contents

Return the HTML of a page or any element

```
browser.html
```

Return the text of the page or any element

```
browser.text
```

Return the title of the page

```
browser.title
```

Get the text from the status bar

```
browser.status
```

Return true if the specified text is on the page

```
browser.text.include? 'Llama'
```

Return the contents of a table as an array

```
browser.table(:id => 'recent_records').to_a
```

A.3 Access / Manipulate an Element

Text box or text area

```
browser.text_area(:id => 'username').set('Mickey Mouse')
```

Button

```
browser.button(:value => 'View Details').click
```

Drop down list

```
browser.select_list(:name => 'month').select('November')
browser.select_list(:name => 'month').clear
```

Checkbox

```
browser.checkbox(:name => 'active').set
browser.checkbox(:name => 'active').clear
```

Radio button

```
browser.radio(:name => 'paid_in_full').set
```

Form

```
browser.form(:action => 'submit').submit
```

Link

```
browser.link(:text => 'Checkout').click
```

Table cell in a table (2nd row 1st column)

```
browser.table(:name => 'recent_records')[1][0]
```

Table cell that can be accessed directly

```
browser.td(:class => 'total_cell')
```

Multiple parameters

```
browser.button(:value => 'View Details', :index => 1).click
```

Appendix B - PageObject Quick Reference

B.1 Getting Started

Load the PageObject gem

```
require 'page-object'
```

Create a page-object

```
page = MyWebPage.new(@browser)
```

Create a page-object and navigate to its' url

```
page_url = "http://mysite.com"  # declared in your page
page = MyWebPage.new(@browser, true)
```

Go to a specific URL

```
page.navigate_to "http://www.google.com"
```

Page creation hook (will be called when page is created)

```
def initialize_page
```

B.2 Page Level Functionality

Return the html of the page or any element

```
page.html
```

Return the text of a page

```
page.text
```

Return the title of a page

```
page.title
```

Wait for a page level until -> until block returns true

```
page.wait_until(timeout, message, &block)
```

Attach to another window

```
page.attach_to_window(:title => 'other title')
```

Page Navigation

```
page.forward, page.back
```

B.3 Frames and iFrames

The PageObject gem provides a way to declare items are within frames or iframes.

```
class RegistrationPage
  include PageObject

  in_frame(:id => 'left-frame') do |frame|
    text_field(:address, :id => 'address_id', :frame => frame)
  end
end
```

This example shows how you might declare a text field that resides inside of a frame. You can also nest frames within frames.

```
class RegistrationPage
  include PageObject

  in_frame(:id => 'left-frame') do |frame|
    in_frame({:id => 'left-top-frame'}, frame) do |frame|
      text_field(:address, :id => 'address_id', :frame => frame)
    end
  end
end
```

Here you see a text field that is inside a frame with an id of left-top-frame which is itself inside a frame with an id of left-frame.

Notice how in both examples we had to pass the frame that is passed into the block as a parameter to the elements that are inside the block. Also notice how we had to place {} around the in_frame identifier when it is nested.

After the declaration of the frames you can use the element like any other one declared on the page.

B.4 Javascript Popups

The PageObject gem handles Javascript popups by intercepting the calls to the popup and cause it to not happen. At the same time the gem provides all of the information to the user that they would wish to receive if they had access to the popup.

Alert

The PageObject module has a method named alert. Here is how you can use it.

```
message = @page.alert do
  @page.button_that_causes_the_alert
end
message.should == "Text from the alert popup"
```

This call demonstrates that we are passing a block to the alert method. That block is the code that causes the alert to occur. The alert will not popup but the message that was included in the popup is returned by the block.

Confirm

The PageObject module also has a confirm method that handles confirm popups. It works the same way as the alert method except it requires a boolean parameter which is the value returned to the browser when the confirm popup is called. Here's an example.

```
message = @page.confirm(true) do
  @page.button_that_causes_the_confirm
end
message.should == "Text from the confirm popup"
```

Prompt

The PageObject module has a prompt method that follows the same pattern. There are two differences. The first is that it accepts a parameter that is the value returned from the prompt. The second is that it returns a Hash with two keys - :message contains the message from the confirm popup and :default_value contains the default value if one is provided.

```
result = @page.prompt("Cheese") do
  @page.button_that_causes_the_prompt
end
result[:message].should == "What do you like?"
```

B.5 Handling Ajax

The secret to working with Ajax is having the ability to wait until different events occur. The PageObject gem supports waiting at two different levels; the page level and the element level.

Page level waiting

On the PageObject module (and therefore on your page objects) there is a method that assists with waiting.

```
def wait_until(timeout = 30, message = nil, &block)
```

This method will wait until the passed in block returns true. If the block does not return true within the specified timeout seconds then an error is thrown with a default message or the provided message. Let's take a look at how we might use this.

```
@page.wait_until do
  @page.text.include? "Success"
end
```

In the example above we are using the default timeout and message. This code will wait until the page includes the text "Success". Let's look at another example.

```
@page.wait_until(5, "Call not returned withing 5 seconds") do
  @page.text.include? "Value returned from Ajax call"
end
```

In this example we are setting the timeout to 5 seconds and providing a custom message.

Element level waiting

On the Element class we have several methods to assist with waiting.

```
def when_present(timeout = 5)
def when_visible(timeout = 5)
def when_not_visible(timeout = 5)
def wait_until(timeout = 5, message = nil, &block)
```

The usage of the first three methods are fairly self explanatory. They will simply wait until the element is present, visible, or not visible and then continue. If the wait exceeds the timeout value then an error occurs. Let's look at an example.

```
@page.continue_element.when_visible do
  @page.continue
end
```

In this code we are waiting until a link defined by a call to link(:continue, :id ⇒ 'cont') is visible and then we are clicking it.

There are times when you want to wait for something other than the element being present or visible. The last wait call wait_until is for these times. It will wait until the block returns true.

B.6 Supported HTML Elements

Elements are the items that appear on the html page. Using PageObject you can find and interact with a large number of elements. This section lists a few of the most used elements, their generated methods, the basic locators that can be used to identify them, and what methods exist on the classes that represent the element on the page.

Element

The Element class is the base of all other classes that represent something that is visual on the page. It has a lot of methods you can use in your script and all of these methods are inherited by all other elements listed below. Here are those methods:

```
click             # click the element
double_click      # double click the element
enabled?          # return true if the element is enabled
disabled?         # return true if the element is not enabled
style(property)   # get the value of a given CSS property
visible?          # return true if the element is visible
exists?           # return true if the element exists
flash             # flash the element by temporarily changing
                  # the background color
text              # return the text for the element
html              # return the html for the element
value             # return the value of this element
==                # compare this element to another to determine
                  # if they are equal
tag_name          # get the tag name of this element
attribute(name)   # get the value of the given attribute
fire_event(evt)   # fire the provided event on the element
hover             # hover over the element
parent            # return the parent element
focus             # set the focus to this element
when_present      # waits until the element is present
when_not_present  # waits until the element is not present
when_visible      # waits until the element is visible
when_not_visible  # waits until the element is not visible
wait_until        # waits until the provided block returns true
send_keys(*keys)  # send keystrokes to this element
clear             # clear the contents of the element
id                # return the id of the element
scroll_into_view  # scroll until the element is viewable
```

Button

You declare a Button using the following method call:

```
button(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # clicks the Button
your_name_element     # returns the Button element
your_name?            # checks the Button's existence
```

You can locate a Button by providing one or more of the following values:

- :alt ⇒ Watir and Selenium (input type = image only)
- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :src ⇒ Watir and Selenium (input type = image only)
- :text ⇒ Watir and Selenium
- :value ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

CheckBox

You declare a CheckBox using the following method call:

```
checkbox(:your_name, :id => 'an_id')
```

Five methods are generated with this call. They are:

```
check_your_name           # checks the CheckBox
uncheck_your_name         # unchecks the CheckBox
name_element_checked?     # Returns if the CheckBox is checked
your_name_element         # returns the CheckBox element
your_name?                # checks the CheckBox's existence
```

You can locate a CheckBox by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :value ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The CheckBox class adds the additional methods:

```
check                # check the checkbox
uncheck              # uncheck the checkbox
checked?             # return true if it is checked
```

Div

You declare a Div using the following method call:

```
div(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name            # returns the text within the Div
your_name_element    # returns the Div element
your_name?           # checks the Div's existence
```

You can locate a Div by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :title ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

FileField

You declare a FileField using the following method call:

```
file_field(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name=           # sets the value in the FileField
your_name_element    # returns the FileField element
your_name?           # checks the FileField's existence
```

You can locate a FileField by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :title ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The FileField class adds the following methids:

```
value=            # set the value for the FileField
```

Form

You declare a Form using the following method call:

```
form(:your_name, :id => 'an_id')
```

Two method is generated with this call. It is:

```
your_name_element     # returns the Form element
your_name?            # checks the Form's existence
```

You can locate a Form by providing one or more of the following values:

- :action ⇒ Watir and Selenium
- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The Form class adds the following method:

```
submit            # submit the form
```

H1, H2, H3, H4, H5, H6

You declare a Heading using one of the following method calls:

```
h1(:your_name, :id => 'an_id')
h2(:your_name, :id => 'an_id')
h3(:your_name, :id => 'an_id')
h4(:your_name, :id => 'an_id')
h5(:your_name, :id => 'an_id')
h6(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the text within the Heading tag
your_name_element     # returns the Heading element
your_name?            # checks the Heading's existence
```

You can locate a Heading by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Waitr and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

HiddenField

You declare a HiddenField using the following method call:

```
hidden_field(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the value in the HiddenField
your_name_element     # returns the HiddenField element
your_name?            # checks the HiddenField's existence
```

You can locate a HiddenField by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :value ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

Image

You declare an Image using the following method call:

```
image(:your_name, :id => 'an_id')
```

Two method is generated with this call. It is:

```
your_name_element     # returns the Image element
your_name?            # checks the Image's existence
```

You can locate an Image by providing one or more of the following values:

- :alt ⇒ Watir and Selenium
- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :src ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The Image class adds the following methods:

```
width            # return the width of the image
height           # reutrn the height of the image
```

Label

You declare a label using the following method call:

```
label(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the text of the Label
your_name_element     # returns the Label element
your_name?            # checks the Label's existence
```

You can locate a Label by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

Link

You declare a Link using the following method call:

```
link(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # clicks the Link
your_name_element     # returns the Link element
your_name?            # checks the Link's existence
```

You can locate a Link by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :href ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :link ⇒ Watir and Selenium
- :link_text ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :title ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

ListItem

You declare a ListItem using the following method call:

```
list_item(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the text in the ListItem
your_name_element     # returns the ListItem element
your_name?            # checks the ListItem's existence
```

You can locate a ListItem by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

OrderedList

You declare an OrderedList using the following method call:

```
ordered_list(:your_name, :id => 'an_id')
```

Two method is generated with this call. It is:

```
your_name             # returns the text in the OrderedList
your_name_element     # returns the OrderedList element
your_name?            # checks the OrderedList's existence
```

You can locate a OrderedList by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The OrderedList class adds the following methods:

```
each                # iterate over the nested ListIem elements
[]                  # return the ListItem for the provided index
items               # return the number of nested ListItem elements
```

Paragraph

You declare a Paragraph using the following method call:

```
paragraph(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the text in the Paragraph
your_name_element     # returns the Paragraph element
your_name?            # checks the Paragraph's existence
```

You can locate a Paragraph by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

RadioButton

You declare a RadioButton using the following method call:

```
radio_button(:your_name, :id => 'an_id')
```

Five methods are generated with this call. They are:

```
select_your_name      # selects the RadioButton
clear_your_name       # clears the RadioButton
your_name_selected?   # returns if the RadioButton is selected
your_name_element     # returns the RadioButton element
your_name?            # checks the RadioButton's existence
```

You can locate a RadioButton by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :label ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :value ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The RadioButton class adds the following methods:

```
select              # select the RadioButton
clear               # clear the RadioButton
selected?           # return true if it is selected
```

SelectList

You declare a SelectList using the following method call:

```
select_list(:your_name, :id => 'an_id')
```

Four methods are generated with this call. They are:

```
your_name             # returns value selected in the SelectList
your_name=            # sets selected value in the SelectList
you_name_options      # returns the nested Options
your_name_element     # returns the SelectList element
your_name?            # checks the SelectList's existence
```

You can locate a SelectList by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :label ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir only
- :value ⇒ Watir only
- :xpath ⇒ Watir and Selenium

The SelectList class adds the following methods:

```
[]                  # return the Option for the index provided
select(value)       # select a value from the list
select_value(value) # select the option(s) whose value attribute
                    # matches the provided String
options             # return an Array of the nested Options
selected_options    # return an Array of Strings representing the
                    # selected options
selected_values     # return an Array of Strings with the values
                    # of the selected options
include?(value)     # return true if the select list has the value
selected?(value)    # return true if the value is selected
```

Span

You declare a Span using the following method call:

```
span(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name           # returns the text within the Span
your_name_element   # returns the Span element
your_name?          # checks the Span's existence
```

You can locate a Span by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :title ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

Table

You declare a table using the following method call:

```
table(:your_name, :id => 'an_id')
```

Two method is generated with this call. It is:

```
your_name             # returns the text contained inside the Table
your_name_element     # returns the Table element
your_name?            # checks the Table's existence
```

You can locate a Table by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The Table class adds the following methods:

```
each            # iterate over the nested TableRow elements
first_row       # return the first row
last_row        # return the last row
[]              # return the TableRow matching the index
rows            # return the number of rows
```

TableCell

You declare a TableCell using the following method call:

```
cell(:your_name, :id => 'an_id')
```

Three methods are generated with this call. They are:

```
your_name             # returns the text within the TableCell
your_name_element     # returns the TableCell element
your_name?            # checks the TableCell's existence
```

You can locate a TableCell by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Watir and Selenium
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The TableCell class adds the following method:

```
enabled?              # return true if the cell is enabled
```

TextArea

You declare a TextArea using the following method call:

```
text_area(:your_name, :id => 'an_id')
```

Four methods are generated with this call. They are:

```
your_name             # returns the value in the TextArea
your_name=            # sets the value in the TextArea
your_name_element     # returns the TextArea element
your_name?            # checks the TextArea's existence
```

You can locate a TextArea by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :label ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The TextArea class adds the following methods:

```
value=          # set the value of the element
clear           # clear the element
```

TextField

You declare a TextField using the following method call:

```
text_field(:your_name, :id => 'an_id')
```

Four methods are generated with this call. They are:

```
your_name             # returns the value in the TextField
your_name=            # sets the value in the TextField
your_name_element     # returns the TextField element
your_name?            # checks the TextField's existence
```

You can locate a TextField by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :label ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :text ⇒ Watir and Selenium
- :title ⇒ Watir and Selenium
- :value ⇒ Watir only
- :xpath ⇒ Watir and Selenium

The TextField class adds the following methods:

```
append(text)          # append the text to the end of the current value
value=                # set the value of the TextField
```

UnorderedList

You declare an UnorderedList using the following method call:

```
unordered_list(:your_name, :id => 'an_id')
```

Two method is generated with this call. It is:

```
your_name             # returns the text inside the UnorderedList
your_name_element     # returns the UnorderedList element
your_name?            # checks the UnorderedList's existence
```

You can locate an UnorderedList by providing one or more of the following values:

- :class ⇒ Watir and Selenium
- :css ⇒ Selenium only
- :id ⇒ Watir and Selenium
- :index ⇒ Watir and Selenium
- :name ⇒ Watir and Selenium
- :xpath ⇒ Watir and Selenium

The UnorderedList class adds the following methods:

```
each            # iterate over the nested ListItem elements
[]              # return the corresponding ListItem element
items           # return the number of nested ListItem elements
```

Appendix C - RSpec Matchers

This Appendix lists several of the matchers that are provided by the RSpec gem. This list was comprised as of version 2.7.0 of the RSpec gem. All examples support both should and should_not.

Equivalence

```
actual.should eq(expected)  # passes if actual == expected
actual.should == expected   # passes if actual == expected
actual.should eql(expected) # passes if actual.eql?(expected)
```

Identity

```
actual.should be(expected)   # passes if actual.equal?(expected)
actual.should equal(expected)# passes if actual.equal?(expected)
```

Comparisons

```
actual.should be > expected
actual.should be >= expected
actual.should be <= expected
actual.should be < expected
actual.should be_within(delta).of(expected)  # float comparison
```

Regular expressions

```
actual.should =~ /expression/
actual.should match(/expression/)
```

Type/classes

```
actual.should be_an_instance_of(expected)
actual.should be_a_kind_of(expected)
```

Truthiness

```
actual.should be_true   # passes if actual is not nil or false
actual.should be_false  # passes if actual is nil or false
actual.should be_nil    # passes if actual is nil
```

Expecting errors

```
expect { ... }.to raise_error
expect { ... }.to raise_error(ErrorClass)
expect { ... }.to raise_error("message")
expect { ... }.to raise_error(ErrorClass, "message")
```

Expecting throws

```
expect { ... }.to throw_symbol
expect { ... }.to throw_symbol(:symbol)
expect { ... }.to throw_symbol(:symbol, 'value')
```

Predicate matchers

```
actual.should be_xxx          # passes if actual.xxx?
actual.should have_xxx(:arg) # passes if actual.has_xxx?(:arg)
```

Ranges (Ruby >= 1.9 only)

```
(1..10).should cover(3)
```

Collection membership

```
actual.should include(expected)

# Examples
[1,2,3].should include(1)
[1,2,3].should include(1, 2)
{:a => 'b'}.should include(:a => 'b')
"This string".should include("is str")
```

Appendix D - ActiveRecord Quick Reference

D.1 Mapping

Mapping to a table - maps to pluralization of class name

```
class User < ActiveRecord::Base  # maps to table named users
```

override the default behavior by calling

```
table_name=
```

Mapping the primary key - maps to a column named `id`

override the default behavior by calling

```
primary_key=
```

Relationships: how are they defined

```
has_many    # defines a one to many relationship
has_one     # defines a one to one relationship
belongs_to  # defines a many to one relationship
```

Accessors: making them clean Column names are mapped to accessors Can override by calling the `alias_attribute` method

D.2 Creating

Simple:

```
user = User.new
user.name = 'Cheezy'
```

Hash:

```
user = User.new(:name => 'Cheezy', :occupation => 'Dancer')
```

Block:

```
user = User.new do |u|
  u.name = 'Cheezy'
end
```

D.3 Finding

Find by single attribute

```
User.find_by_email('a@example.com')
```

Find by several attributes

```
User.find_by_email_and_password('a@example.com', 'secret')
```

Find or Initialize (If record exists, returns it. Else return new record without saving it first)

```
User.find_or_initialize_by_email('a@example.com')
```

Find or Create (Creates new record unless it exists)

```
User.find_or_create_by_email('a@example.com')
```

Conditions: the WHERE part of the SQL query

```
:conditions => { "email = ? AND password = ?", email, pass }
:conditions => { :email => email, :password => pass }
```

Find all that match criteria

```
User.find_all_by_last_name('Smith')
```

Order: specify SQL order by criteria

```
Post.find(:all, :order => 'colmun desc')
```

Group: an attribute name by which the results should be grouped

```
Post.find(:all, :group => 'member_id')
```

Limit: limits the number of rows returned

```
Post.find(:all, :limit => 10)
```

Offset: an integer determining the offset from where the rows should be fetched

```
Post.find(:all, :offset => 10)
```

D.4 Operating on ActiveRecord objects

Getting the number of rows in a table

```
User.count
```

Saving an object into the database

```
my_user.save
```

Deleting an object from the database

```
my_user.delete
```

Deleting all objects in a table

```
User.delete_all
```