DESIGNING AND TESTING SOFTWARE AND USER-INTERFACES

The reason that Object Oriented Programming (OOP) has been so successful is that it demonstrably makes software easier to understand, to develop and to maintain. Proper OOP involves creating an application as an interactive set of components – objects – where each has a clear and simple responsibility within the application, and communicates with the other objects in the application to get information from them or perform a service for them. Such an assembly of objects is often called an Object Model, indicating that the object assembly models some aspect of the real-world to perform tasks for us. Object Model Design is a set of techniques that is used to create simplified pictures of the application, rather like an assembly diagram for a piece of machinery.

User-interfaces are a key part of most applications, and are best when properly designed as an object model. HTML provides most of the User-Interface machinery for JS applications; using a JS object model and JS events, HTML forms can be easily integrated into an application's structure.

General principles of software analysis and design

For a long time, it has been accepted that developing **large** software systems is best done using well-established engineering principles. In the early days of computing, computer programs tended to be written by experts in whatever field the software was needed for – architects wrote architectural design tools, accountants wrote accounting packages etc. In that respect, computer programming was like a craft: a small-scale discipline done by enthusiasts.

Major computing problems that emerged in the late 1960's coincided with an increase in the scale of programs that were needed: IBM's OS-360 was a very large (for the time) operating system, financial packages written by Data General for a consortium of Swiss banks was the first major piece of international software – both of these projects failed. IBM went on to re-do OS-360 from scratch, the banks sued Data General out of existence (establishing a general principle – don't mess with the Swiss where money is concerned). Other big projects of the time fared similarly. The general situation was known as the "Software Crisis", and was the spark that started the entire discipline of "Software Engineering".

Having decided that sound engineering principles were needed, there then followed a 40-50 year argument about what form these principles should take. Software was not really like building a bridge (often used at the time as an analogy to indicate what could happen if slap-dash development was done), where you could easily measure the stresses in the finished product, and programming was certainly not like building.

The consensus was that software should be built using a life-cycle approach, similar to major engineering projects. That is, organising the process into stages:

Stage	Bridge Building Stages	Software Development Stages
1	Establish that the bridge is needed and perform a study to identify the most cost effective type to build for the situation	Analyse the task(s) that the software is to accomplish and produce a list of functional requirements

2	Produce a basic design for the bridge, possibly building a model for testing	Create a design of the software in terms of the functional blocks needed and how these will interact
3	Perform detailed analysis of the bridge components, selecting materials and designing the individual elements to meet their needs	Create detailed designs of the core software components (database, user-interface elements, functional modules)
4	Fabricate major components (girders, cables, joints, surfacing materials etc.) and test samples of these to make sure they are up to the job	Write code for each of the major components and test each in isolation to make sure that they perform as expected
5	Deliver the components to the site for assembly (the site should have been prepared beforehand) and assemble	Integrate the major components and test at each integration stage to make sure the components interoperate properly (the host operating system may need to be configured – memory allowed, disk space needed, network connections configured)
6	Test the assembled bridge by running increasing amounts of traffic up to its design limit over it – demonstrate that these tests have worked to the body (council or government) that commissioned it	Test the final software and make sure it meets the acceptance criteria of the customers

Table 7.1: Comparing the engineering stages of building a bridge and a substantial piece of software

The basic steps laid out in table 7.1 are not too dissimilar between the building a bridge and the building software scenarios: Requirements Analysis (1), System Design (2), Detailed Design (3), Implementation (4), Integration (5) and Testing (6). In each case, the steps overlap a bit (e.g. the components built in step 4 are tested individually before being assembled).

This model became the basis of the software Waterfall Model – the name suggests that the results of one step flow into the next. This then became the archetypal software development process for the next 20-30 years. It is still used to an extent now, but advances in development techniques and technology over the years have refined it quite a bit.

- Requirements Analysis has always been about creating a definitive specification of what the system has to do. In a bridge, the requirements are likely to remain the same throughout the period of its design and construction and even the bridge's working life. In software, the requirements are almost bound to change as the design emerges. This is due to the flexibility of the medium (you can do a lot quickly in software) and the customer's realization as designs develop that their original picture of what they wanted might be flawed, or that new objectives might be possible. A general principle in software is that "requirements will change", and the process must be able to cope with this. Waterfall is bad at handling requirements changes later in the life cycle
- Software design techniques have changed much over the years. Very early designs were based on formalized textual descriptions, although these were awkward to make and often difficult to follow. Graphical forms took over in the 1980s – data-flow diagrams, state charts etc. These tended to be very tightly focussed and therefore lots of different types were needed. Experts who knew ALL of the possible types were few and expensive. Current thought is that an overall design for a software system can be depicted with very few details,

and these can be expanded on better in the next stage

- Detailed software design methods have also evolved over the years. For a long while Pseudo-Code or Program Design Language, a mix of plain English and coding structures like loops and if statements, were used. Recently it has been recognised that this is largely wasted time – producing something that looks like program code but can not actually be executed just duplicates work. Worse, if a detailed design were to change (which we now know is inevitable), it would have to be changed both in its design form and in the implementation (program code). It is now accepted that real, working program code is better, both because the programmers who write it will be more aware of the execution environment than a disconnected designer ever would and because it can be tested immediately and flaws eradicated quickly
- Since we've already replaced the detailed design stage with coding, it is more important that code that has been written is tested in isolation from other code as quickly as possible. For years, software testing was a task given to either testing specialists (if you were a major, well funded software company) or the new-start in the office (if you were a normal company). This approach is seriously flawed, since testing is either very expensive and un-responsive to change (in the big company) or ineffective (for the rest of us). Current practice is that tests should be devised by the person who writes the code, probably in collaboration with the rest of the team for rigour. However, best practice is to write a test *before* writing the software that is to pass it. In fact, this approach can be taken to an extreme, but logical, conclusion by using the tests to *specify* the detailed design of the software – Test Driven Development (TDD) is one of the more effective innovations in software development over the past few years, where what a program is to do is defined by the tests it must pass
- Integration testing is still needed in software, but quite a lot of the effort of this is mitigated by the use of up to date tools for software management. Testing how components interact can still be written as specifications, so the major task in integration testing is to gain access to the different components as they are completed; this is a job that is best done using software tools such as Version Control Systems (VCS) and software libraries. This is the most unchanged area in software development, although emerging standards (such as better specification for software modules, new modular design models etc.) are being shown effective in simplifying it
- An acceptance test is still the final stage. The main change to it is that current practice is to involve representatives of the end-user during the whole development process, so an acceptance test is now more of a formality than it once was.

If we accept this version of a software development process, and complete each of the steps one after the other, we are using the (basic) waterfall process. By messing with the sequence of the stages (typically, allowing them to overlap so that design starts before analysis is complete for example), then we are using a modified version of waterfall, of which there are many.

More recent developments in software processes have led to waterfall becoming less popular (bear in mind it was never too popular with the developers who had to use it in the first place). Most working software developers resented the amount of time handling the paperwork of a software process, and were keen to get to the coding stage as fast as possible. Most managers realised that the stages prior to coding were necessary to a degree, simply because without them programmers tended to code without putting too much thought into it (would you use a bridge designed by the construction welders during their tea breaks?).

Currently popular process models have recognised a few important principles:

- The people who write the code are probably well qualified to design the system as well, if only they could be forced to put enough thought and rigour into it
- Because requirements change almost continually, much of the paperwork involved in the analysis, design and detailed-design stages could be wasted effort – what is needed is a lightweight way of accomplishing the same ends; preferably one that the programmers can use comfortably
- Earlier software process models were based on generating documentation (software analysis produces specification documents, design produces diagrams, detailed design produces state charts and pseudo-code). The output of one stage fed into the next stage as input. The best form of documentation is program code itself, provided some clarification is added as part of it. Software specifications, in the form of tests, are best when they are executable statements or functions that tell you what is expected of the software and whether what you have built works
- Doubt can be removed from a software process if the people who will actually use it get to interact with the development team frequently. Having a customer representative as a fully-fledged member of the team has been demonstrated to greatly increase the quality of the system developed
- Testing of software should be done as early as possible. The longer time there is between creating a statement of what software should do and testing the software to see whether it does it, the more difficult things become. Ideally, testing should be continuous as the software is built, since it is very common to break a piece of software by making a seemingly unimportant change to it

These principles became the basis of the Manifesto for Agile Software Development, devised in 2001 and now widely accepted as **the law** in many software companies. See http://agilemanifesto.org/ for the manifesto in full. **Agile Software Development** became the working definition for a process that was accepted by developers, responsive to change and lightweight in operation. As the first software development process to be devised for the development of software in the Internet era, it is particularly important in the way it includes development principles that support the use of dynamic languages, parallelism, distributed systems and multi-language systems that were not covered so well in earlier models.

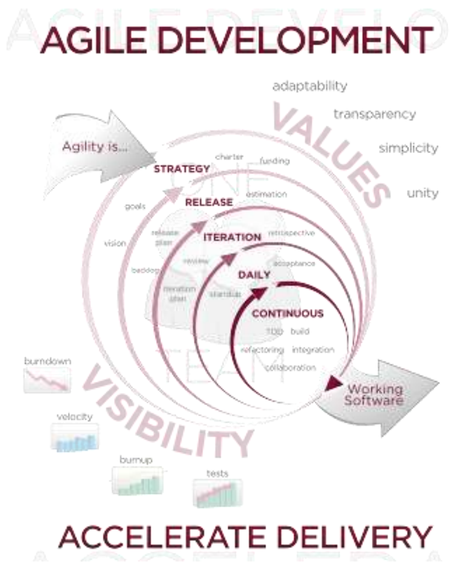


Figure 8.1: The agile principle (source Wikipedia, 2012)

The aims of agile development methods are to speed up the delivery of software while remaining responsive to change. The methods applied to do this involve getting the development team more involved in the whole process (not just a single stage, like 'programming' or 'testing'), automating the process of building and testing software as much as possible, involving end users in the design and testing of software, and, as far as possible, working directly with working software at the earliest possible stage of development.

As you might recognise, this process is more about building "something" as quickly as possible and then refining it until it becomes what the end-user wants. While that would be a haphazard way to build a bridge, the flexibility and malleability of software makes it a perfectly sensible approach for creating programs. It also removes much of the fear that developers feel when confronted with a huge set of requirements and they start to wonder whether they are capable of building this thing. Start small and grow into the system that is needed.

Software Development Principles and Writing Programs

You would not be blamed for pointing out here that this is all very well when you're building a big system, but how does it help you to build your end-of-semester assessment project? Two answers here:

- First, it might be a good idea to understand what will be being assessed at the end of the module – obviously, you will write a program and part of the assessment will simply be "does it work?" However, that's a small part of the assessment, and very much less important than the question "is this student or group able to solve a problem in a way that suggests they could solve other, very different, problems?" That's the killer part of the assessment – we need to be able to assume that when you go on to other modules that involve some programming, the programming part is a done deal because in these modules there will probably be other things to assess
- Secondly, being able to use a programming language so that your code can execute without errors is only part of programming. You have to be able to know what to build, and ideally you need to know how to build it efficiently. That is a difficult thing to teach through the medium of explaining the syntax of a programming language. Having a few rules about how to specify what your program code needs to do is as important, probably more important than whether you can use the Javascript language. Every program has objectives, and often (always for large programs) finding out what the objectives are is more important than how to write code to meet them.

Taking a little time out to understand why you need to state in detail what the software must do before you try to write it is important. Being able to start developing a system by first writing down how it will behave, then creating tests to determine whether what you build does behave this way, and ***only then*** developing a system or component that can take the test and pass it is the goal (and also the main thing we will be assessing at the end of the module).

Behaviour-driven Development (BDD)

This is a very up to date approach to software development, based on the principles of test-driven development, but centred on the objectives that software requirements analysis and software design have always had – to be able to state clearly and unambiguously what software should do. In this respect, it goes further than TDD, which is used to define software by writing tests, where the tests are written by and for developers.

BDD simplifies the TDD principle by trying to phrase every software requirement in a standard way; one that emphasizes not only what the software will do in a way that a developer will understand, but that also sticks with terminology that is meaningful to the end-user, who is unlikely to be a software specialist. If, along the way, it helps new developers to understand how a software specification should look, that's a bonus that you should appreciate.

Situations and Outcomes

BDD is a way of expressing what a system should do in specific circumstances. Given a particular situation (e.g. a number is greater than zero and we try to take its square root) we can expect a specific outcome (e.g. the square root result times itself equals the original number). BDD allows us to stipulate this in plain English in simple sentences that always have the same form:

```
When <something happens in the system>
Then <this outcome should be expected>
```

Simple enough, but an example would be useful.

```
①   Specification: Bank Account
    When a new bank account is created
    And the customer opens it with a £X deposit
    Then the bank account should exist
    And the account balance should be £X

②   When a deposit of £X is made to a bank account
    Then the account balance should increase by £X

③   When a withdrawal of £X is attempted from a bank account
    And the bank account has £X or more in it
    Then the bank balance should reduce by £X
    And the cash should be paid out

④   When a withdrawal of £X is attempted from a bank account
    And the bank account has less that £X in it
    Then the bank should not allow the withdrawal
    And the account balance should not change
```

Listing 8.1: A BDD specification for a bank account

The first thing to notice about this is that it is almost trivial. If the suite of 4 specifications were more complex, then it wouldn't be as easy for the end-user to understand. It also defines rules about a bank account that anyone is likely to take for granted. It often turns out that these are the parts of a software system that cause problems.

The second thing to notice is that the format always results in indicating something you can do to test whether it worked or not. This is what makes it work as a bridge between the end-user's understanding and the developer's implementation. We can easily write software tests to show that whatever software we write as an implementation of a bank account either works or doesn't. If the tests weren't easy to write, we could end up with errors in the tests, which would be worse than errors in the actual bank account implementation because the tests could fool us into thinking a faulty system was working properly.

Finally, it is worth noticing that the specifications (and eventually the tests) are defined in terms of objects. We are describing a thing that is a bank account. In doing so we are actually providing a clear statement of the things a bank account ought to be able to do (its methods – deposit and withdraw) and the things a bank account ought to have (its properties – account balance).

From behaviour specification to tests

There are several Test-driven-Development tools available for Javascript code – jsUnit is the most well known one. Most of them are really easy to install and use. However, a very lightweight system called Jasmine is easier to use than almost all of them, and has the benefits of requiring no installation, and using a Behaviour Driven Development approach. Let's see how the bank account tests specified in listing 8.1 could be implemented in Jasmine. To make things clearer, we'll start with just the first test:

```
describe( "BankAccount Test Suite", function(){

      var account,
            initialDeposit = 100.0;

      beforeEach(function(){
         account = new BankAccount(initialDeposit);
      });

      it("contains an amount equal to the initial deposit amount", function(){
         expect(account).toBeDefined();
         expect(account.balance).toEqual(initialDsposit);
      });
   }
);
```

Listing 8.2: a functional Jasmine test.

The code in listing 8.2 might look a little unfamiliar, but it is fairly east to grasp. To create a ***test suite*** (a group of related tests), we use the ***describe()*** function. This takes two parameters – a description of the suite as a plain string (here – "BankAccount Test Suite") and a function to execute that contains a set of tests. I could have defined the actual test function as a normal named function and then simply put its name as the second parameter, but in this case there is no need for that, so the function has been entered directly into the call to ***describe()*** - I know it can get a bit complicated with all the brackets, but you can quickly get used to that.

Inside the function passed to ***describe()***, there is a variable declaration part – ***account*** and ***initialDeposit*** are declared here – followed by two function calls. The first call is to ***beforeEach()***, which again takes a function declaration as a parameter – in this case, the function says what to do before any test is run. Here, we create a new ***BankAccount*** by calling its constructor and passing in the known ***initialDeposit*** value.

The second is the actual test. The ***it()*** function (its name becomes clearer when you see the actual test results) is called passing in a string describing a test and (you've guessed it) another function definition – this time, one that executes some actual test cases.

From here, it gets easier to follow – the ***expect()*** function takes a single parameter and returns a result. The parameter is typically a variable but could be another function call (with brackets after it, meaning that the function would actually be called). The returned result is a test result object, and we could write it to be passed into another variable and then evaluated like this:

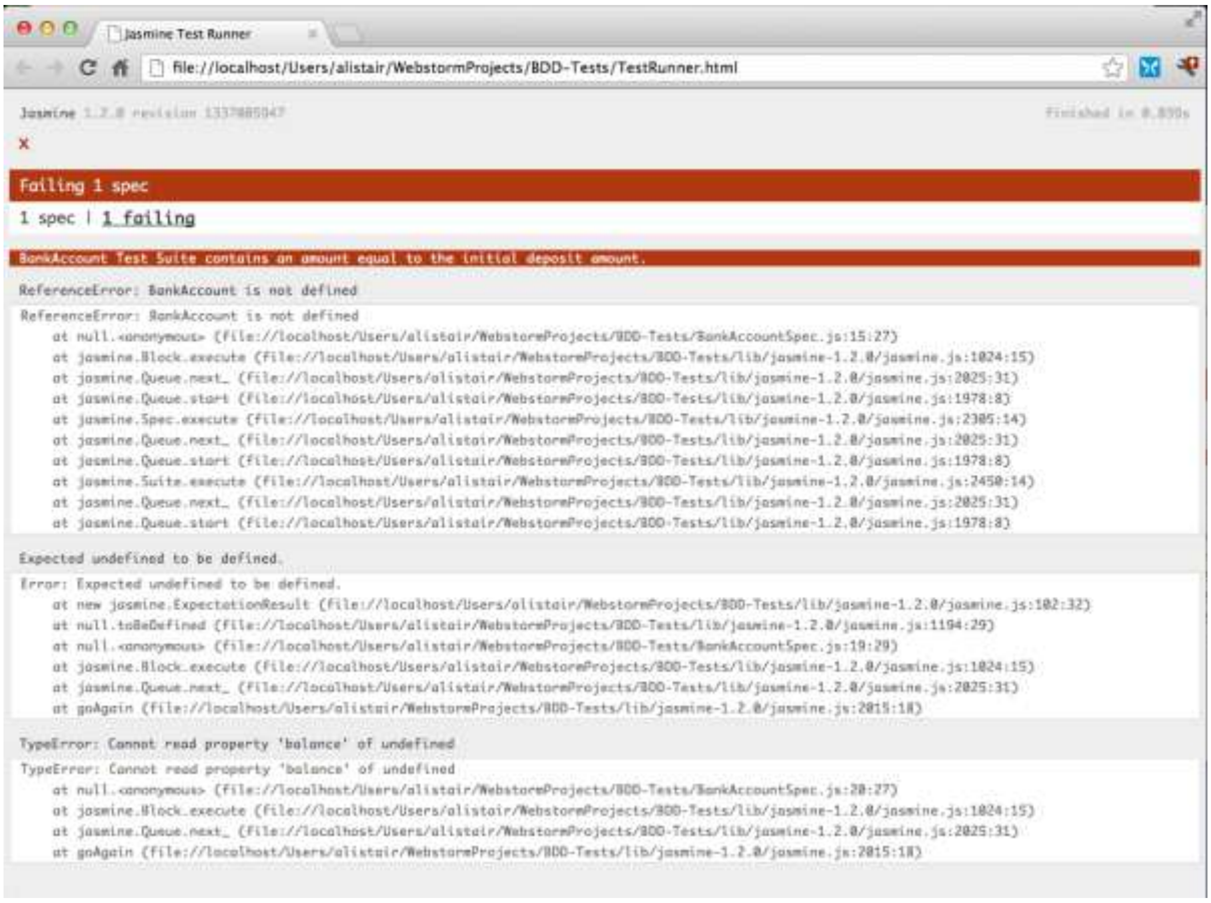
```
var testResult = expect(account);
testResult.toBeDefined();
```

However, that would actually mess up how the code would flow as you read it, whereas…

```
expect(account).toBeDefined();
```

…makes complete sense.

There are two tests defined for the i***t()*** call in listing 8.2. We want a BankAccount object to exist, and we want it to have a balance the same as the value passed to the constructor. Now that the tests are defined, all that remains is to run them. Jasmine comes complete with a simple HTML file called TestRunner.html. We simply add a normal <script> reference to the file, and provided the test runner is in the same folder as the test file, we can just open it.


Jasmine Test Runner
file://localhost/Users/alistair/WebstormProjects/BDD-Tests/TestRunner.html

```
Jasmine 1.2.0 revision 1337005947                                   Finished in 0.035s
x
Failing 1 spec
1 spec | 1 failing
BankAccount Test Suite contains an amount equal to the initial deposit amount.
ReferenceError: BankAccount is not defined
ReferenceError: BankAccount is not defined
    at null.<anonymous> (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/BankAccountSpec.js:15:27)
    at jasmine.Block.execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1024:15)
    at jasmine.Queue.next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
    at jasmine.Queue.start (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1978:8)
    at jasmine.Spec.execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2305:14)
    at jasmine.Queue.next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
    at jasmine.Queue.start (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1978:8)
    at jasmine.Suite.execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2450:14)
    at jasmine.Queue.next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
    at jasmine.Queue.start (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1978:8)
Expected undefined to be defined.
Error: Expected undefined to be defined.
    at new jasmine.ExpectationResult (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:102:32)
    at null.toBeDefined (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1194:29)
    at null.<anonymous> (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/BankAccountSpec.js:19:29)
    at jasmine.Block.execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1024:15)
    at jasmine.Queue.next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
    at goAgain (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2015:18)
TypeError: Cannot read property 'balance' of undefined
TypeError: Cannot read property 'balance' of undefined
    at null.<anonymous> (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/BankAccountSpec.js:20:27)
    at jasmine.Block.execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1024:15)
    at jasmine.Queue.next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
    at goAgain (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2015:18)
```

Figure 8.2: The full Jasmine report having run the tests in listing 8.2

All failing Jasmine tests are reported in the format shown in figure 8.2 – they provide full diagnostic information so that you can find the test that failed and the line of code that caused that to happen. Here, we only need the report headline, right under the name of the test suite – "ReferenceError:

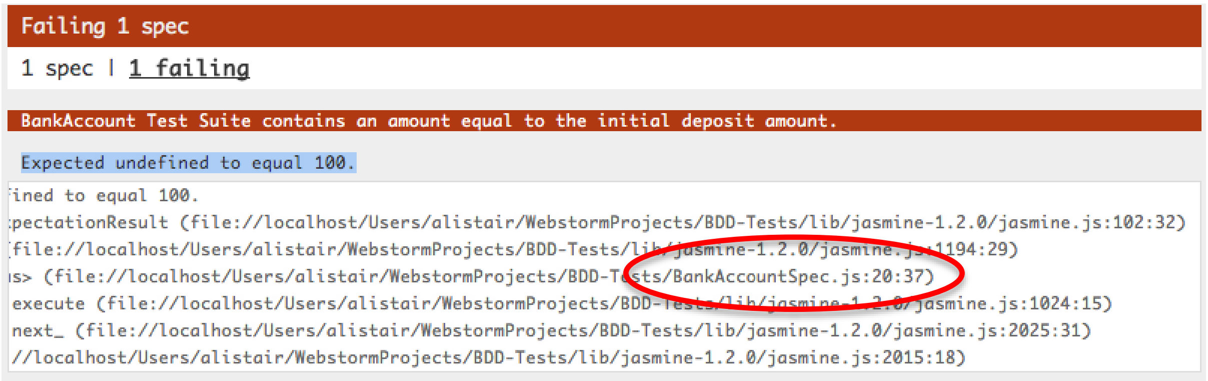
BankAccount is not defined". Yep – we've not actually provided a BankAccount constructor function yet, so the tests have no object to work with. Even so, Jasmine has explained why, and while this might be over-the-top for this example, in a test of a bigger system, we're likely to need a lot of the diagnostic information to make sense of what happened.

We can fix this test failure easily by providing a BankAccount constructor function. For now, an empty one will do:

```
// Define BankAccount as a constructor...
var BankAccount = function(initialDeposit){};
```

Listing 8.3: The test subject – at this stage, it is as minimal as we can make it

The code in listing 3 would normally be added to its own file (BankAccount.js) and referenced by another <script> statement in TestRuner.html. To re-run the tests, simply press the refresh button on the browser. Jasmine will generate a new report, but now problem that is reported looks like this:



```
Failing 1 spec
1 spec | 1 failing

BankAccount Test Suite contains an amount equal to the initial deposit amount.

Expected undefined to equal 100.
ined to equal 100.
pectationResult (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:102:32)
file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1194:29)
s> (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/BankAccountSpec.js:20:37)
execute (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:1024:15)
next_ (file://localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2025:31)
//localhost/Users/alistair/WebstormProjects/BDD-Tests/lib/jasmine-1.2.0/jasmine.js:2015:18)
```

Figure 8.3: A section of the Jasmine report showing the error and (highlighted) the code statement that it occurred in

Note that the message under the test suite name starts with "expected" – i.e. one of the expects() statements we put into the tests has failed (things have improved since the previous test run). If we look into the test suite code (BankAccountSpec.js) at line 20, character 37, we'll find that the expectation that failed was:

```
expect(account.balance).toEqual(initialDeposit);
```

This is to be expected – we have defined a BankAccount class but have not provided a property called ***balance***.

Test Driven Development, or in this case, Behaviour Driven Development (the main difference is in how the tests are described in the first place), works in this way:

1. write a test
2. run it and watch it fail
3. amend the code and re-run the test until it passes
4. identify a new tests to add, and return to step 1

Changing the code and adding new tests incrementally is less stressful to a developer, and is also much more likely to result

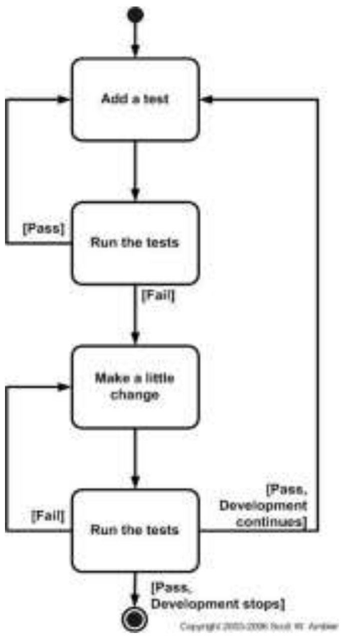


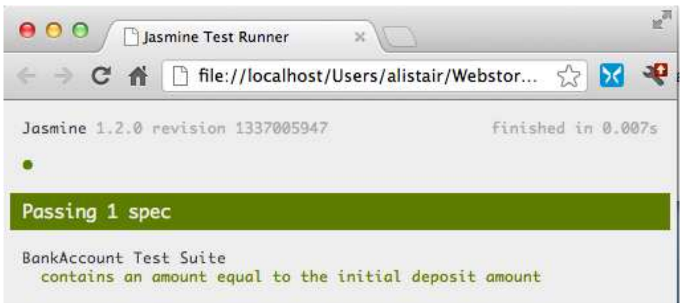
Figure 8.4: Test-Driven Development (source: www.agiledata.org)

in correct code that writing a passage of code and then writing tests for it would. Starting the process by writing a test that will fail makes it more likely that the test actually does test what it is supposed to. Figure 8.4 shows this process as a flowchart.

To make the previous test pass, (hopefully) we just need to add a ***balance*** property and make sure that it is set to ***initialDeposit*** in the constructor. However, it is important to point out here that the assumption I've made here (about adding a ***balance*** property) is for the purposes of illustration only. In general, assumptions are bad – you write tests and change code to pass them as a process of investigation. As soon as you adopt the attitude "I know what to do here to make this work", the test cycle gets sidelined and the code is driving the tests, rather than the (correct) other way round.

This may sound like a philosophical detail, but it is one that is central to TDD and BDD. Programmers do a bad job when they make assumptions – always!

Continuing with the example, I should now amend the BankAccount code and re-run the test.

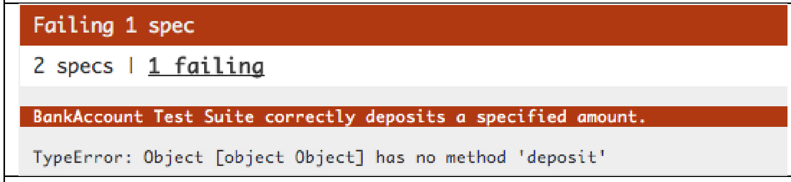
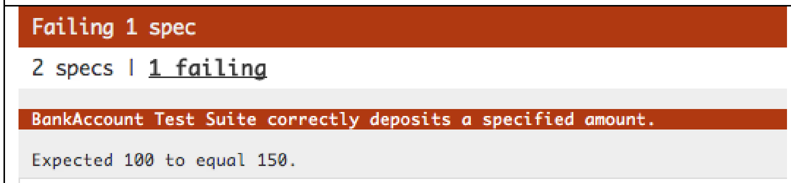


```
Jasmine 1.2.0 revision 1337005947          finished in 0.007s
●
Passing 1 spec
BankAccount Test Suite
  contains an amount equal to the initial deposit amount
```

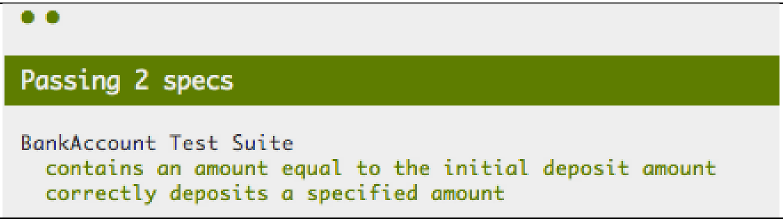
Figure 8.5: A passing test in Jasmine

The test (that a **BankAccount** can be created) has passed, and we can add the next test. To complete the example, I'll just show the additional tests, the changes made to the **BankAccount** type and the test results (with some explanation) and so on until we have a working **BankAccount**. Remember that the original BDD specification is shown in listing 8.

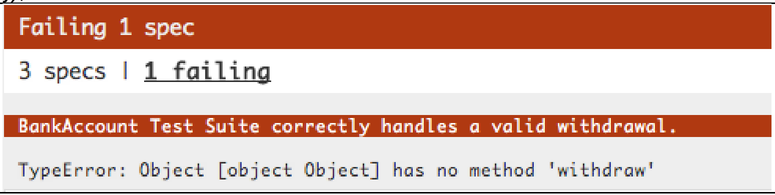
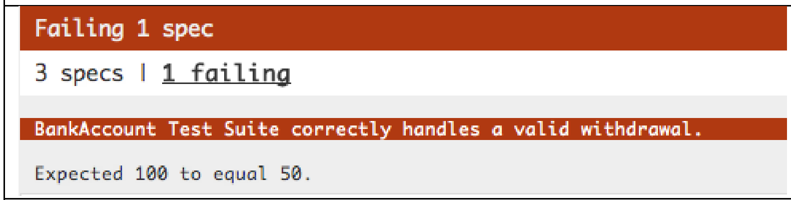
it("correctly deposits a specified amount", function(){
    var depositAmount = 50;
    var result = account.deposit(depositAmount);
    expect(account.balance).toEqual(initialDeposit + depositAmount);
});
	This is a test of the deposit method (BDD spec 2), added after the first it() test. Result is…

 Failing 1 spec 2 specs | 1 failing BankAccount Test Suite correctly deposits a specified amount. TypeError: Object [object Object] has no method 'deposit'	There is no deposit method.
BankAccount.prototype.deposit = function(amount){};	Add an empty deposit method. Result is…
 Failing 1 spec 2 specs | 1 failing BankAccount Test Suite correctly deposits a specified amount. Expected 100 to equal 150.	The deposit method does not credit the account.

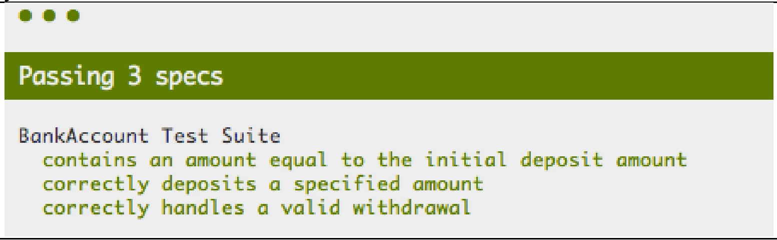
BankAccount.prototype.deposit = function(amount){
    this.balance += amount;
	Update balance property in deposit method. Result is…

};	
 ● ● Passing 2 specs BankAccount Test Suite contains an amount equal to the initial deposit amount correctly deposits a specified amount	The deposit method passes its test.

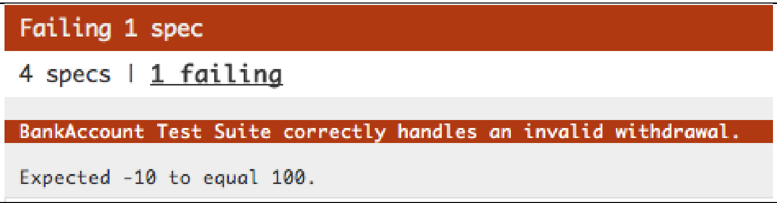
it("correctly handles a valid withdrawal", function(){
var withdrawalAmount = 50;
var result = account.withdraw(withdrawalAmount);
var remainingBalance = initialDeposit - withdrawalAmount;
expect(account.balance).toEqual(remainingBalance);
});
	Add a test for a successful withdraw action, where the account balance is adequate. Result is…

 Failing 1 spec 3 specs | 1 failing BankAccount Test Suite correctly handles a valid withdrawal. TypeError: Object [object Object] has no method 'withdraw'	There is no withdraw method.
BankAccount.prototype.withdraw = function(amount){};	Add an empty withdraw method. Result is…
 Failing 1 spec 3 specs | 1 failing BankAccount Test Suite correctly handles a valid withdrawal. Expected 100 to equal 50.	The withdraw method does not debit the account.

BankAccount.prototype.withdraw = function(amount){
this.balance -= amount;
};
	Add code to the withdraw method. Result is…

 ● ● ● Passing 3 specs BankAccount Test Suite contains an amount equal to the initial deposit amount correctly deposits a specified amount correctly handles a valid withdrawal	The withdraw method debits the account correctly.

it("correctly handles an invalid withdrawal", function(){
var withdrawalAmount = 110;
var initialBalance = account.balance;
account.withdraw(withdrawalAmount);
expect(account.balance).toEqual(initialBalance);
});
	Add a test for an unsuccessful withdraw, where the account balance is not enough. Result is…

 Failing 1 spec 4 specs | 1 failing BankAccount Test Suite correctly handles an invalid withdrawal. Expected -10 to equal 100.	The withdraw method allows a withdrawal the account does not have funds for.

BankAccount.prototype.withdraw = function(amount){
if(this.balance >= amount){
this.balance -= amount;
}
};
	Add code to the withdraw method to test for adequate funds in the account before attempting a withdrawal. Result is…

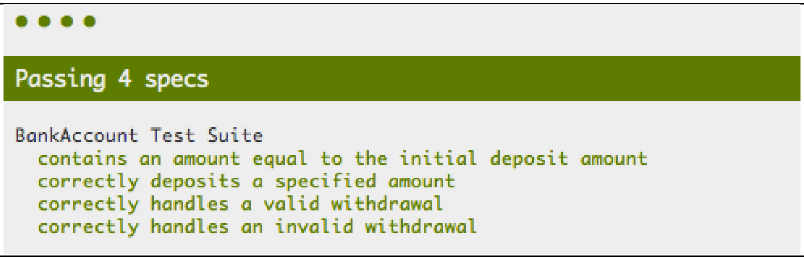
 •••• Passing 4 specs BankAccount Test Suite contains an amount equal to the initial deposit amount correctly deposits a specified amount correctly handles a valid withdrawal correctly handles an invalid withdrawal	The withdraw method does not allow amounts beyond the account balance to be debited.

Table 8.2: Completing the BankAccount type using BDD Tests

Of course this is not a complete implementation of a BankAccount; if you bank just sent you a note every now and again with one figure on it, the balance, you'd probably be pretty annoyed with them. To complete it, we would have to add another test:

When the user requests a statement of the account transactions
Then a complete list of deposits, withdrawals, charges and interest payments for the past XX days is generated and displayed.

This would be a massive test to implement, since in doing so we'd have to change how the entire BankAccount class developed so-far worked; it would have to record each deposit, withdrawal etc. (along with the date of the transaction, how it was done - e.g. ATM, Bank Counter, internet transfer etc., and the account balance after the transaction) in a transaction register. This in turn would mean re-engineering how deposit() and withdraw() methods worked, how the balance was calculated etc.

This is a bigger problem, and one that I'll not show here. However, it is worth keeping in mind that the tests we have implemented so far ***would still have to pass***, and so even after a complete re-working of ***BankAccount*** we would still have evidence that the code met the earlier specification. The strength of TDD & BDD is that both approaches provide us with evidence of the software working, a full specification (and, for BDD, one that is in terms an end-user can follow easily), and a suite of regression tests that we can re-apply any time the software is amended.

Behaviour Driven Development – its true value

If you've done any coding before now, you're probably thinking that this is a lot of work to do for a very small amount of code. However, let's look at the process objectively:

- There was never any stage beyond the initial BDD specifications in listing 8.1 where anything complex was attempted. If the code you are writing **must** work, this is an important benefit
- The tests did not take long to set up. Certainly my explanation here was a bit long winded, but much of it was explaining how to set up tests that are initially odd looking, and stepping through the process in some detail so that you could follow a complete example
- The tests are not a part of the BankAccount object type. They are placed in a separate file and so when we finally deploy the BankAccount software, the tests can be kept away from the final software. This means our testing process does not bloat the software
- The tests are a faithful representation of the BDD test specs in listing 8.1, which are written in a way that ensures that anyone can understand them. End users can follow these tests and create their own if they identify cases where more testing is needed
- We only ever added tests; none were removed, so when we add a new test of a newly specified feature, and then add that feature so that it passes the test, we get a built in assurance that nothing in the original set of features has been damaged
- The tests can be stored and run again at any time. This is a major benefit – we can always go back and test if the program still meets its original spec. We can even change entirely how

the software works and the tests remain a real indication that the software still meets its spec. ***Regression testing***, the process of ensuring that changes to software don't do anything to change its specified behaviour, is a very important capability for a project that has to last over several revisions. There is a good reason why Jasmine displays test progress as a series of green dots – we get an indication of the tests that have passed as well as the tests that have failed and that gives us a progress indication for the project as a whole.

User-Interfaces and Testability

BDD and TDD both describe processes that let us build software via the definition of tests. From what we've seen of Jasmine (a BDD tool that is not radically different from jsUnit, a TDD tool), we can see that all we need to do is arrange to ***call*** the software under test in controlled circumstances.

Ok – now think back to the last time you used any software that operated from you, the user entering function calls; my guess is that (unless you're above 45 years old or a diehard Linux user) you've never used this type of software. The Graphical User Interface (or GUI) has been the standard form for interacting software since the late 1980s.

This provides us, as diligent software developers, with a problem. We've seen lots of good reasons for using test to specify software. We now need to apply this approach to software that is going to be driven by a GUI, which in itself provides no callable interface.

The solution to this conundrum is, in fact, well known and has been used in various forms by well-organised software companies for years. Build the operational parts of a program using TDD or BDD, and then use a GUI to act as a front for it. Thinking about that for a minute, you can see exactly the same approach taken in many real-world things that are not software.

Very rarely, while driving along the road, will you see anyone balanced precariously on top of a car engine adjusting the screw that increases the amount of fuel flowing into the engine, or using a foot out of the door to slow it down like Fred Flintstone. A car has a dashboard, pedals and a steering wheel. These provide the linkages to the engine, the brakes and the angle of the front wheels and all of the other ***inputs*** that affect how the car runs.

Using a sensible (and quite common) design process, you can create the same sort of arrangement for a piece of software. In chapter 6 we had a brief look at HTML5 forms and form elements; these can be used to create the 'dashboard' of our computer program. This is ideal because the available HTML controls come fully formed and ready to use (a bit like going into a parts depot and picking up a steering wheel and a few switches and levers). Not only that, but they are built to be easy to retrofit on to an existing piece of software with the minimum amount of code.

A well-established paradigm (i.e. a model or pattern) exists to make the connection of a working software model to a user-interface a straightforward task. This is the Model-View-Controller pattern, which is often described in this diagrammatic form:

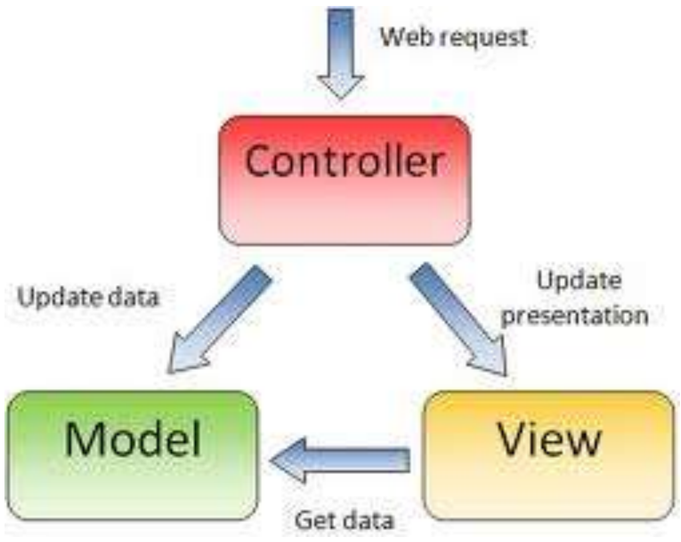


Figure 8.6: The MVC Pattern

What this pattern of software describes is the relationship between the software that has been built to a specification (i.e. the Model), the software that processes user-input and causes this to affect the Model (the Controller) and the software that displays the results to the user (the View). Bear in mind that the situation it describes is one where, having completed the functional part of the software, you still need to create the parts that the user interacts with. However you approach a programming task, you will almost always have to provide these other parts. What MVC does is to describe a way to do it with minimal pain and effort.

The View and Controller parts of an MVC system are not easy to test. However it is easy to argue that a) if you have good components available they are easy to build (so less in need of testing) and b) their operation, unlike the operation of a complex model, is so obvious that testing is unnecessary. Given the shortage of software for testing user-interfaces, it is just as well that this argument holds water.

The model, on the other hand, is easy to test, as I hope the BDD development of the BankAccount type demonstrated. By separating out an application into the three components, the most important of them immediately lends itself to a development process that is simple and effective.

The View and Controller elements could do with more formal definitions:

- View: this renders the model (or parts of it) into a form suitable for interaction. Most often, this is a user-interface element such as a Form. In a web application, the View is the HTML page
- Controller: this component is the one that processes and responds to the user's interactions on the page – i.e. the event handlers

Continuing with the simplified BankAccount example, we can put together a View and Controller that would allow a web user to interact with it. Please bear in mind that this is a ***very*** simplified example, and nothing like the implementations you are likely to use while you are internet banking (at least I hope not).

Let's start with a conceptual picture of the BankAccount MVC implementation:

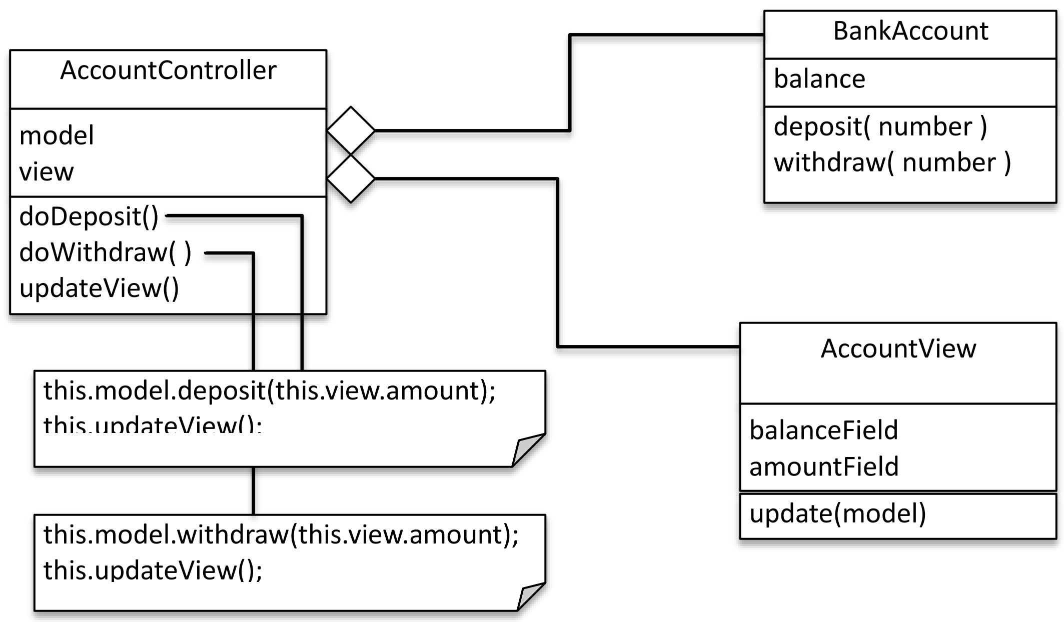


Figure 8.7: An overview of a Model-View-Controller Bankaccount system

The system shown in figure 8.7 is clearly a lot more complicated than a simple ***BankAccount*** object. On the other hand, a BankAccount object says nothing about how it will appear on the display (browser) or how a user will interact with it. This is ***a good thing***, since if the ***BankAccount*** code included all of the information, it would be very difficult to use it within any context other than a web application.

The benefits of the MVC way of doing things are:

- Separation of concerns:
  - the model does not need to know how it will be used
  - the view does not need to know anything about hoe the module is implemented
  - the controller can be ignorant of what is in the model and how it is displayed. It only needs to be aware of the major operations that the system must support
- Ease of testing
  - the model can be fully tested without any of the complexity of a user-interface
  - the controller is very easy to test interactively since all it does is issue commands
  - the view can be tested in isolation since all of its functionality is managed by the controller
- Ease of maintenance
  - if it is decided that the model should be re-designed (e.g. to support transactions so that a statement can be generated), this can be done without any change to the View or Controller. The View and Controller will probably need to be changed in new functionality is to be added to the model, but these upgrades can be done separately, once the new model has been shown to work with the existing view and controller

  - if it is decided to create a new type of user-interface (e.g. for a desktop application or a smartphone), the only changed needed will be to the view
  - the controller will only need to be changed if the basic functionality of the model is changed, and then it can be done after the model is shown to work with the existing controller

One of the core principles of the MVC pattern is that the interfaces between objects should change as little as possible. The design of an MVC system, like that shown in figure 8.7, is organised to be very general purpose, so that it would work for a BankAccount system or a graphical rendering application. Interfaces between the various components are restricted to the ones that exist in the pattern already – the controller updates the view and the model, the view is able to ask the model for data, the user only interacts with the controller.

In effect, the controller is the thing that orchestrates how the model and view interact. In a web application it is nothing more than a container for references to the model and view, and a manager for the event handlers that indicate what the user wants to do. Here are the three components in Javascript code:

```
var BankAccount = function(initialDeposit){
   this.balance = initialDeposit;
};

BankAccount.prototype.deposit = function(amount){
   this.balance += amount;
   return true;
};

BankAccount.prototype.withdraw = function(amount){
   if(this.balance >= amount){
      this.balance -= amount;
      return true;
   } else {
      return false;
   }
};
```

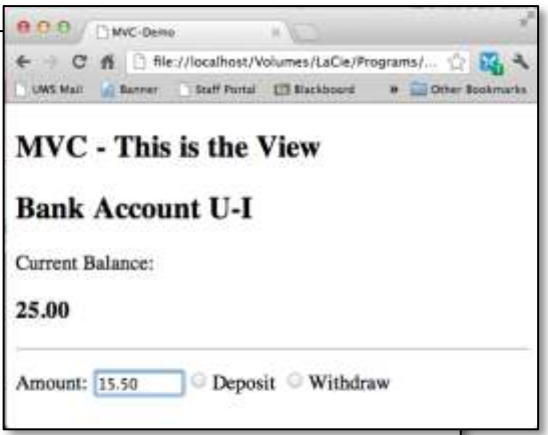
Listing 8.4: The BankAccount (a trivial implementation) type. This has a single property, balance, and two methods, deposit() and withdraw().

```
var AccountView = function(){
   // This function will create a view object, providing access to the
   // html user interface elements that belong to it.
   this.viewForm = document.getElementById("viewForm");

   // The amount box..
   this.amountBox = document.getElementById("amount");

   // The balance element...
   this.balanceElement =  document.getElementById("balance");

   // The buttons...
   this.depositButton = document.getElementById("deposit");
   this.withdrawButton = document.getElementById("withdraw");

   // These two methods update the presentation on the screen.
   // Sets the deposit & withdraw option buttons to un-checked..
   this.clearOptions = function(){
      this.depositButton.checked = false;
      this.withdrawButton.checked = false;
   };
   // Update the display to match the current account balance…
   this.update = function(model){
      this.balanceElement.innerText = model.balance.toFixed(2);
      this.amountBox.value = "0.0";
   };
};
```

Listing 8.5a: The AccountView type. This contains only properties and functions that update the presentation of the model.

Note that this class expects there to be various components in the hosting HTML document (the first five assignments in the class are references to HTML elements. A suitable HTML page is shown in listing 8.5b

```
<!DOCTYPE html>
<html>
<head>
   <title>MVC-Demo</title>
</head>
<body>
   <heading>
      <h2>MVC - This is the View</h2>
   </heading>
   <form id="viewForm">
      <h2>Bank Account U-I</h2>
      Current Balance: <h3 id="balance"></h3>
      <hr/>
      <p/>Amount: <input type="text" id="amount" size="10">
      <input type="radio" id="deposit" name="transaction">Deposit</input>&nbsp;
      <input type="radio" id="withdraw" name="transaction">Withdraw</input>&nbsp;
   </form>
</body>
<script type="text/javascript" src="js/model.js"></script>
<script type="text/javascript" src="js/view.js"></script>
<script type="text/javascript" src="js/controller.js"></script>
</html>
```



MVC - This is the View

Bank Account U-I

Current Balance:

25.00

Amount: 15.50 Deposit Withdraw

Listing 8.5b, and the resulting web page

```
var AccountController = function(){
   alert("New account");
   this.model = null;
   this.view = null;
   var me = this;

   this.updateView = function(){
      this.view.update(this.model);
      this.view.clearOptions();
   };

   this.runDeposit = function(){
      var amount = parseFloat(me.view.amountBox.value);
      me.model.deposit(amount);
      me.updateView();
   };

   this.runWithdraw = function(){
      var amount = parseFloat(me.view.amountBox.value);
      if(!me.model.withdraw(amount)){
         alert("Insufficient balance in account for that transaction");
      }
      me.updateView();
   };

   this.init = function(){

      var initialBalance = parseFloat(prompt("Enter new account initial amount:", ""));
      this.model = new BankAccount(initialBalance);
      this.view = new AccountView();

      this.view.depositButton.onclick = this.runDeposit;
      this.view.withdrawButton.onclick = this.runWithdraw;

      document.getElementById("withdraw").onclick = this.runWithdraw;

      this.updateView();
   }
};

window.onload = function(){
   var ctrl = new AccountController();
   ctrl.init();
};
```

Listing 8.6: The AccountController type. This sets up event handlers (on the view), and provides functions that call the various model & view methods. It also kick-starts the application (using window.onload in this case).

While there is a lot of code in the overall MVC implementation, it is not much less than if we had decided to do the whole thing in a single JS file, and it has all of the benefits listed earlier. In most real-world circumstances, the amount of code would be less because a standard MVC framework would be used (see JavascriptMVC: http://javascriptmvc.com, Backbone.js: http://documentcloud.github.com/backbone/, SproutCore: http://sproutcore.com/ for some popular examples).

Summary

We've covered a lot of material that goes beyond simply programming in Javascript. Hopefully the issues covered in this chapter have made sense for anyone who has an interest in going beyond the basics of Javascript. It is worthwhile having some knowledge of TDD and BDD as a working methodology since many companies now use these as core development principles. MVC is a pattern that gets implemented in many big websites, and for good reason since the developers of these sites want to spend as little time as possible making changes or fixing a site when problems arise.

Questions

1. Requirements analysis is done so that software developers know exactly how to implement a system – true or false?
2. Which is the more up-to-date software development process model – Waterfall or Agile?
3. When is the best time to create tests for a software system?
4. Behaviour Driven Development is concerned with creating highly technical descriptions of how a software system should behave – true or false?
5. It is important in test-driven development (and behaviour driven development) for a new test to be seen to fail before the software under test is updated to make it pass. Explain why this is so.
6. Explain why the Model-View-Controller pattern makes it easier to create working interactive systems.
7. Describe the purpose of a View.
8. Describe the purpose of a Controller.