HTML5 & Javascript

Software Design and Testing
(the really boring stuff)

Software and Scale

* Building small programs is easy (comparatively)
  * Of course you need to be interested, but the average programming book might take 2 weeks to work through from cover to cover (e.g. the Stoyanov book)
  * That probably covers most of what you need to know to write small Javascript programs using quite sophisticated techniques
  * If all programming was this easy, there would be no news reports of gross and expensive failures in software systems
* Serious problems start to arise when you try to build a big program
  * Almost everyone's first attempt will continue using the techniques that you learned to make small programs
  * Big mistake! That's a bit like saying that having built a garden shed, you're ready to try a skyscraper

Analysis and Design

* Before you try to build something big
  * It's a good idea to know ***exactly*** what you're going to build
  * List the requirements, and from them, write a detailed specification
  * This is Systems (or Software) Analysis
* Having decided what you want to build, it is a good idea to draw up a few plans
  * What are the major components?
  * How will they interact?
  * This is Software Design

Build a big (software) thing

* "Traditional"* steps in software development:

Step	Description
1.	Analyse the tasks the software is to perform – create a requirements list
2.	Create a design of the software in terms of major main functional features
3.	Create detailed designs of the core components (e.g. database, user-interface, …)
4.	Write code for the major components, testing each in isolation to ensure it meets its spec.
5.	Integrate major components, testing at each integration stage
6.	Test the fully assembled system.

* * like your granny used to do

Problems with the traditional approach

* There are lots, but can be boiled down to:
  1. The separation of steps – e.g. complete analysis before design, complete design before implementation etc.
     - This is not how big projects get done – teams tend to work on parts of a system that can progress individually
     - Too many people on a project would have to 'hang around' waiting for output from others
  2. "Products" arrive too late
     - Even a medium sized project could take 3 months of analysis, 6 months of design ***before there is anything that can be tested***
     - Analysis and designs are PLANS. Plans don't always work
  3. The end user is rarely well represented in old-style software projects
     - They are the people who really know what a system needs to do – the obvious people to consult
  4. Requirements change
     - Often before the ink is dry on a specification of system requirements, the requirements have changed – business processes get modified, additional data becomes available, new legislation comes into force etc.
     - Any sensible development regime that expects to take several months (or even years) to build a system needs to respond to these changes

Recent and Current Software Development Processes

* Current trends are to try to reduce the formalism of older methods
  * Documentation was always the output from earlier stages (Analysis, Design) - that simply generates a lot of "soon to be out of date" paperwork
    * Code (properly written) IS documentation
* Ordering a project as Analysis->Design->Code->Test puts the only concrete measurement tool at the end
  * Too late to be of use
  * Recent trends are for Test-Driven-Development or Behaviour-Driven-Development
    * The software specification is the set of tests the software is to pass
      * Tests make very good statements of what is ***expected*** from software
    * Write the tests first, the software second
* Reduce formalism to get the development team on-side
  * The best people to define working practices are the people who will do the work
* Involve the customer at every stage
  * Stating requirements, defining tests, reviewing the current state of a system etc.

See " Agile Software Development" in Wikipedia and at http://agilemanifesto.org/

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Two newer development paradigms

* Test-Driven-Development (TDD) (Kent Beck)
  * Write software specifications by writing tests
    * e.g.: after executing `bankAccount.deposit(50.00)`, the account balance ought to have increased by £50.00
    * See http://www.agiledata.org/essays/tdd.html
* Behaviour-Driven-Development (BDD) (Dan North)
  * Modification of TDD, with particular focus on behavioral specification – specification aimed at business people instead of developers
    * e.g. As a bank customer, I want to be able to withdraw cash from my account:
      * Scenario 1: If I try to withdraw £50.00 from my account and there is at least £50.00 in my account, I should get the cash and the balance should reduce by £50.00
      * Scenario 2: If I try to withdraw £50.00 from my account and there is less than £50.00 in my account, I should not get the cash and the balance should not change
    * These scenarios should be written in the business language of the user (here a customer), not technical language

Testing and Tools

* TDD is expected to be tool-driven
  * The standard toolkit is xUnit (JUnit for Java, NUnit for .NET, jsUnit for Javascript etc.)
    * Developed by Kent Beck
    * Easy (for a developer) to write tests which are automated
* BDD was not oriented towards tools (but by defining an area of the software industry, guess what…
  * Cucumber, is the current standard toolkit for BDD
    * See http://cukes.info/ for a good overview
    * More end-user oriented
* Jasmine is a testing framework for Javascript code
  * It claims to be BDD, but more like half-way between TDD and BDD (you need to be a developer, because Jasmine tests are written in Javascript)
  * See http://pivotal.github.com/jasmine/ for a good overview.

A typical Jasmine Test Suite

* E.g. for a Bank Account Deposit:

Specification: Bank Account

1 : 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

```
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);
    });
  }
);
```

It is normal to write these tests before the BankAccount type code exists

The TDD/BDD Process

* Write a test
* Run-it : watch it fail?????????
  * Yes – otherwise your test could be flawed so that it passed in any circumstances
  * In practice – run the test before the software that is to be tested exists!!!!!
* Make small changes between test runs
  * Very much smaller chance of doing anything that is too clever
  * e.g. Create an *empty* BankAccount type and test it
  * Add a property (e.g. balance) and test it
  * Add code to update the property and test it, etc., etc.
* Once the test passes, *make a secure copy of the software that passed* and add the next test
  * Version Control software (e.g. Git, Subversion) is used extensively in TDD/BDD, because you can easily save the intermediate steps without amassing a huge number of complete copies

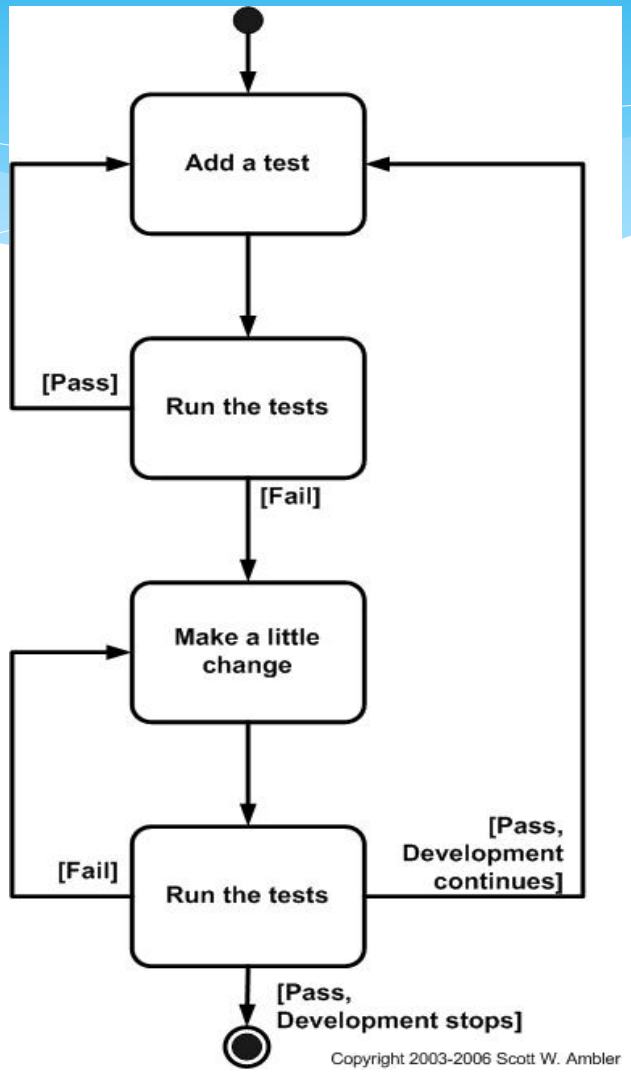


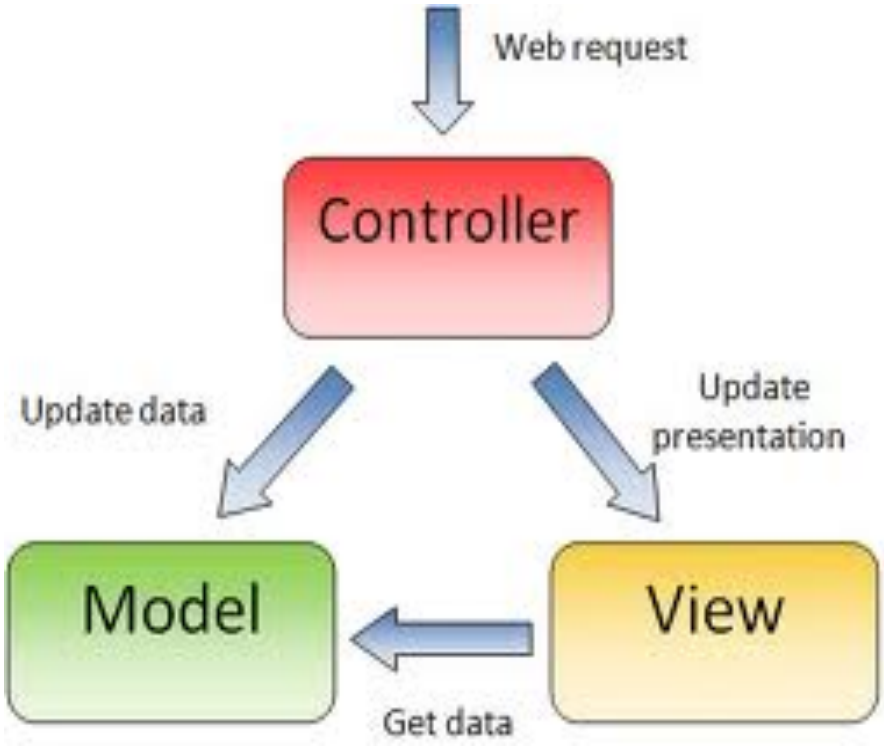
Figure 8.4: Test-Driven Development (source: www.agiledata.org)

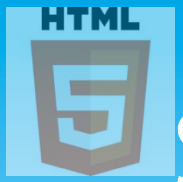
That's the core objects dealt with – what about the User-Interface?

* Common practice is to separate out software elements into 3 types of component
  * Model – the objects that make up the system you are trying to build
    * BankAccount, Appointment, Contacts, GameBoard+Pieces, etc.
    * The whole working model will include some general purpose components (e.g. collections, databases etc.)
  * View – the objects needed to display the model on the screen and interact with it
    * Typically, user-interface elements: e.g. an HTML <form> that displays bank account data and lets the user interact with it
  * Controller – typically the top-level command system that controls an application or major component
    * In a web-app, this is typically the set of event handlers that let the user decide which command to execute now – withdraw cash, deposit cash, view a filtered statement etc.

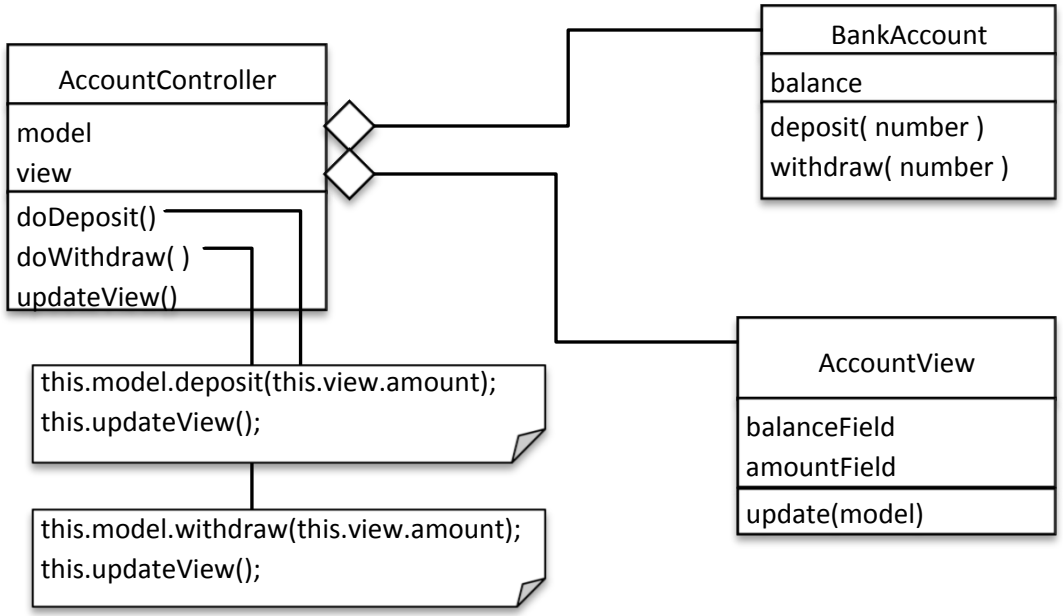
Model-View-Controller (MVC)

* This is a very common development principle for building interactive software
* Microsoft ASP.NET now operates entirely under MVC principles (see http://www.asp.net/mvc)
* Almost all web applications benefit from MVC principles, because of the normal separation of the core data (typically on a database on a server) from the pages used to view it, and the commands (typically URL handlers) used to access it.





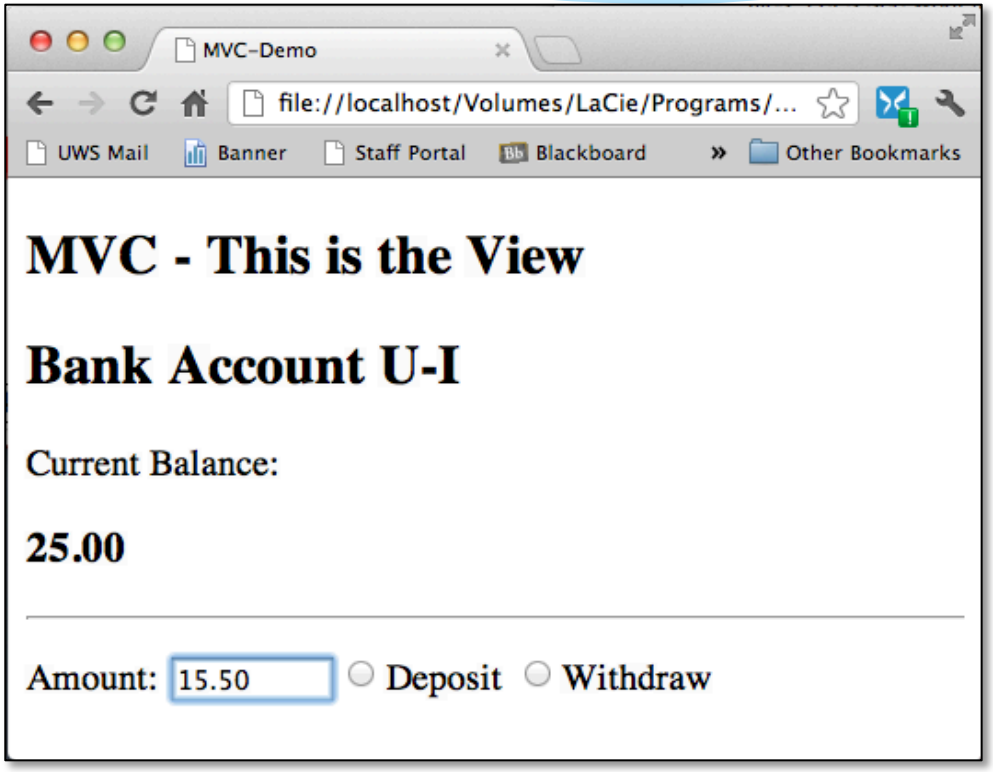
Separating out the concerns



* This is a UML class diagram
  * Shows the types involved, their properties and methods
* Here, AccountView and AccountController are defined as types that support the Model
* Intentionally, all defined methods are trivially simple
  * Otherwise the combination might get complex
* By separating out M, V & C, the same Model could be used for different versions of the app
  * A desktop app
  * A web app
  * A mobile client

Demo app

* Find this at Moodle
  * Samples and Demos (last folder)
    * MVC Demo
* Worth a look but don't get too involved unless you're particularly interested
* This is not expected for your project work
  * It would be nice, and a good implementation would attract good marks – but not necessary



Frameworks

* While the MVC demo shows how all of the components are developed in gory detail, this is not normal
* More usual to use a pre-defined framework
  * E.g. the Microsoft ASP.NET MVC Framework
    * Does much of the "wiring up" work for you, leaving you to concentrate on the model details
  * For Javascript, look into JavascriptMVC, Backbone, SproutCore
    * Don't use these lightly – typically frameworks have a steep learning curve because the assumption is that you are already familiar with the language and the principles