Using SXP in the Software Hut

Version 5.0.0

10th February 2009

m.holcombe@dcs.shef.ac.uk

Web site: http://www.dcs.shef.ac.uk/~wmlh/Software%20Hut%202008-9.htm

Based on versions 1-4 of the Genesys XP Guide by David Vivash, Chris Thomson, Marian Gheorghe, Mike Holcombe

Introduction

This document details how the SXP (Sheffield XP) method will be used within the Software Hut. It is deliberately short to encourage you to read and understand it – further reading advice is given at the end of the document. (Yes I know it seems like a lot of reading, but this saves you reading about four books so I think it will be worth it!)

IMPORTANT NOTE: YOU ARE EXPECTED TO WORK FOR *15 HOURS ON AVERAGE PER WEEK* ON THE SOFTWARE HUT. Be aware that you will have deadlines for other modules and this will mean that you should front-load your effort on the Hut to compensate. You may also wish to be prepared for extra effort around delivery time but if you pace yourselves correctly this should not cause too many problems.

Frequently Asked Questions

To ask a question regarding this document please email m.holcombe@dcs.shef.ac.uk.

SXP Introduction

Extreme Programming (XP) is a relatively new invention, intended to make software engineering a more successful process. Kent Beck invented the method as a backlash against the more documentation-oriented approaches, which were deemed too bureaucratic and difficult for an average team of programmers and which could not adjust to the client's changing requirements. XP is an *iterative* approach – systems developed using XP are developed in *short* iterations and released at *frequent* intervals.

The version of XP presented is known as Sheffield Extreme Programming (SXP) and is similar but not identical to that presented by Kent Beck. The most useful reference for this method is Mike Holcombe's book "*Running an Agile Software Development project*" published by John Wiley. See also: http://agile.genesys.shef.ac.uk/

This guide is intended as a summary to kick start you using SXP, you will find that you will have questions. You may find answers in the textbook, if however, your question is not answered there please ask your manager.

SXP General Principles

There are some general principles that apply for the whole of the project when you are using SXP.

Courage

Some of the following sound scary, stay with them and you should have a successful project at the end of the semester.

Speed of Delivery

SXP is all about getting a system up and running quickly, this means that you don't want to spend weeks building up a set of requirements. You should produce your first code after the client meeting in week 3. You can then use this to help the client understand his/her possibilities and you to structure your own thinking. Hopefully the early work won't get thrown away but it should be if it turns out that it isn't right!

So you will work on an **incremental delivery strategy**, in the first few weeks as you start to collect the requirements you will also be writing the code. However if they change don't worry, re-use what you can and throw the rest away.

Documentation

You do not need to produce lots of design documentation, so do not! However **you will need user guides** which you should write towards the end of the semester when your product is most stable. You also are not excused from adding comments; these should follow the form of your **coding standards** (see later). At the end of week 5 you will produce a brief **requirements document** which will be 'signed off' by your client.

Pair Working

Whatever part of the project that you are working on, most of the time you should work in a pair. If working at a single computer (which is recommended) you should swap regularly to allow both people to "have a turn".

Pair working allows continuous peer review of the task. If you swap pairs regularly it also serves to propagate knowledge throughout the group. This avoids having a project dependent on any one person's knowledge, which is essential for code maintenance in SXP as there is very little documentation for you examine apart from the stories, the code and the tests.

System Metaphor

The system metaphor is an overview of your project. We represent this is an html file in the root directory of your project. This should have hyperlinks to all the main parts of your project (each code file, different types of documentation etc) and explain what they do and how they relate to each other. In particular it should mention which story each bit of code relates to. **This file should be updated at least each week.**

Coding Framework – TRAX

PHP is Server-side HTML embedded scripting language. It provides web developers with a full suite of tools for building dynamic websites:

The **TRAX** framework allows for the rapid development of dynamic websites with database 'backends' that are easily tested and in a suitable technology for hosting on CICS and many ISP environments. All of the current projects will be using this technology.

Testing is done using **PHPUnit** an automated unit test system. We need to ensure that the test sets have maximal coverage.

Celerity . The tool used at epiGenesys for functional test automation is Celerity and its website can be found at:

http://celerity.rubyforge.org/

There is also an API documentation area of the website that shows all the classes and methods that are available for use with the Celerity API, this can be found at:

http://celerity.rubyforge.org/yard/

The first step to using Celerity is to create a feature file for each User Story that is to be tested. A lecture will expand on this.

Cruise Control is a continuous integration builder. A document containing how to change the settings to be able to get Cruise Control to work for each individual project will be made available

.

Each team will set up and maintain a WordPress Blog with the following

- Progress
- Overview
- Objectives
- Achievements
- Documentation
- Resources

Further details later.

Feature File Testing - A document outlining how to use the FeatureConverter tool to generate the ruby steps file from the feature files will be available.

Every team will have an account on the epiGeneys servers where all the data will be kept.

We will be using a CVS system called svn (Subversion) - a version control system used to maintain current and historical versions of files such as source code, web pages, and documentation.

SXP Step by Step

SXP is all about making software development more manageable and easy. Over the course of this project we expect you to develop high quality software. So whilst SXP attempts to reduce your work load in producing complex documentation, you still need some minimal amount stories, XXMs, tests are the main ones – see below.

You should follow this process step by step. However at any time you can drop back to an earlier step and repeat it, but you must then **update the files created in the previous steps, although this may not be in order** they should be updated within a week of changes elsewhere.

Step 1: Initial business analysis

What are the business objectives of the client's organisation, are they identified and articulated, how does the system support these, are there any hidden issues? Use techniques like Mind Maps to list issues and their general relationships. Try to

identify the key business processes involved and interactions with other parts of the organisation, other software etc.

Who are the stakeholders and how are they involved?

Step 2: The client meeting step 1: write story cards

System requirements are captured in SXP using "stories". A story is a high level description of a particular function the system should provide - a sort of "lightweight use-case". Y**ou should also record your story cards on the group web tool every week.** The edges of the XXM are components of stories, some may form complete stories on their own, and others may be so related that you put them on a single card.

Each story should go into enough detail to give the intention of the function, but should be high level enough that the customer understands it. A story should be detailed enough for you to make an estimate of the amount of time required to implement it and to define the tests required to show that it works.

When writing a story, a particular problem is the amount of detail that the story should go into. Ideally a story should define a single operation from the uses perspective, so that once completed it can be easily demoed to the client. **However as a guide each story should take no longer than two weeks for one person to implement.**

Stories should be created using a template which includes the story name, what operation it carries out, list of related stories and *how the stories will be tested in outline.*

You will use the *StoriPost* system to manage your stories. This requires the inclusion of information on the business value of a story. This will enable us to more effectively consider client objectives when planning and prioritising work. It will also encourage a stronger focus on developing business value rather than just developing functions. The basic tests or acceptance criteria for each story should be included.

Details later.

Step 3: The client meeting step 2: pick story cards

This is where it is decided which user stories will be implemented in the next iteration. It is up to the customer to decide which stories will be implemented next; the programmers give time estimates on each story based on their current knowledge of the system.

Step 4: Iteration preparation: System design with Extreme Machines

Extreme X-Machines (XXMs) are a simple model for forming something which is similar to the graphical use case model which you should be familiar with from UML. This is the first tool that you should use as you approach the development, and can be done either before or after the first client meeting. The machine should show how the system fits together and will define your integration tests.

Creating your XXM

The best way to create a XXM model is as a team, as you will all have different ideas about the system you will be creating. You may also use it in a meeting to stimulate your discussion with the client if he does not know what he wants yet.

We have provided a plug in as part of the Eclipse software package to help you draw XXM, and generate test sets from them. There is some information about using the tool in the appendix. You need to remember to update your XXM whenever you change the behaviour of the system. Examples of changes include: adding new screens, and adding new transitions between screens (possibly with extra buttons or internal decisions).

An XXM consists of a number of screens (pictorially circles), internal decisions (pictorially squares), directed labelled edges (directed arrows with text labels that connect the states) and some designated start and finish states (shown by smaller circles). Figure 1 shows a simple example of such a machine. This XXM demonstrates a simple login system. To define what the machine does we have given each edge the name of some user action, for example: “click(register)”. This machine then represents the behavioural model of the system.

Of course in a large system there may be many such internal decisions and screens. In order to keep the diagram simple and understandable we decompose it. Any edge, screen, or decision can be decomposed into a complete XXM. This means that you can start with a simple XXM at the top level and expose an edge with a name like: “click(wizard)”. This can then be decomposed in another machine which shows each of the possible stages of that function. In practical terms the sub-diagram is decomposed such that: In the case that a transition function is decomposed the initial start and finish functions are left unlabeled. And in the case that a screen or decision is decomposed the start and finish functions are named the same as those respectively entering and leaving the original state. Don’t have more than 7 or so states in a single diagram.

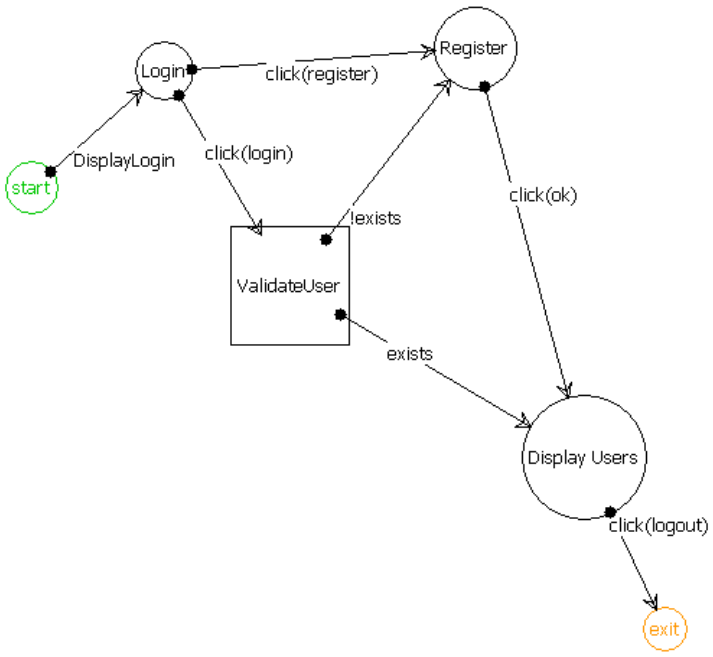

Figure 1: A simple XXM machine with four states

Step 5: Write some tests

Now is when you need to writing your tests. For each set of tests you should construct a test document. This will describe how to run the tests, the inputs required, and the expected outputs. There will be a class on automating tests using the local facilities. If you run the tests manually you should also record when you ran the tests, and if they passed or failed. An example document is in the appendix.

Unit Testing

A unit test is a test of a single function or method. It is recommended that you test everything that could possibly break. For an object-oriented language, this means that simple accessor and mutator methods do not need to be tested, anything more complex should be.

Unit tests ensure that methods do what you expect them to do. By ensuring that the unit tests actually run, it means a complete check of all functions can be (and is) carried out all of the time. This is so that code that is already tested is protected from new code. If new code breaks old code we know about it as soon as we run our tests.

Unit Testing Frameworks

Unit testing frameworks provide a mechanism for managing and running unit tests. A unit testing strategy needs to be agreed on by the project team before any production code is written, so it's important that everyone in the team understands how to create, manage and run the set of unit tests right from the start, you may want to experiment with the test server to confirm how to do this.

Integration and Behavioral Testing

The XXM plugin generates a useful set of these tests for you. The format of the tests is shown in figure 3. The format is fairly straightforward, the items in brackets are the screens or decisions that the program makes, between the chevrons (>>) are the functions to be executed. This example system generated three test cases to be run. In order to test them you must construct a test document that includes these cases, and the inputs to the system that will allow the path to be followed. When you test the system you must record whether the correct path was followed, i.e. the screens are shown in the anticipated order.

(start) > DisplayLogin > (Login) > click(register) > (Register) > click(ok) > (Display Users) > click(logout) > (exit)
(start) > DisplayLogin > (Login) > click(login) > (ValidateUser) > !exists > (Register) > click(ok) > (Display Users)
(start) > DisplayLogin > (Login) > click(login) > (ValidateUser) > exists > (Display Users) > click(logout) > (exit)

Figure 3: Example integration test set produced by the XXM plugin.

The point of integration testing is to check that all aspects of the story have been correctly implemented. It also helps the programmer understand exactly what the customer may mean by a particular story.

Step 6: Write the code

At this point you start to write the code, don't be put off by all the things that you don't yet know about. Instead, write a simple solution that addresses the story card you are implementing, but if you know that something else is coming write your code with this in mind.

Once you have some code, and you are implementing new stories and functions you may find that your code starts to look a bit like spaghetti. This is when you should **refactor**. You should add this as a new task to the web site, and then set to work. Refactoring is all about making the code better designed. Because you had no formal design your code will start to get into a mess, by moving functions into different classes and reducing cut and paste you will get better code. Martin Fowler's and Eric Freeman's books may give prove helpful, the Freeman book is easier to get into but is focused on design patterns.

.

Step 7: Run your tests again

Now use the test software to run your tests, they should now pass, if not return to step 5.

Step 8: Quality Assurance

Finally you need to check that the code conforms to the coding standards and is of good quality.

Coding Standards

When you write code you are expected to write it to a **coding standard,** this describes, amongst other things, the type of comments required and how to name variables. You should use a coding standard for all code that you write during the software hut. Please make it clear which standard you are using (the whole team should use the same standard).

A set of coding standards PHP is on the web site.

Step 9: Next iteration

Now return to step one and start the next iteration. As you are working as a team you may find that as a team you are working at different stages on different stories, this should not be a problem. Just use the *svn* system to organise things properly.

Step 10: Integrating

We use the Cruise Control to add new stories to the system built so far.

We plan to deliver the first Release in week 3

Recording progress

Each team will use a website (SUPPLIED) that will contain information about the project, what meetings are held (with their brief minutes), who has done what and a record of the different processes carried out each week.

Further reading

Books and Web Sites

The biggest web site dedicated to XP is the "Extreme Programming Roadmap", linked to from http://c2.com/. This site is quite hard to navigate, but has everything there is to know about XP. A slightly easier site is at http://www.xprogramming.com, which includes various articles exploring XP in more detail.

Hut site

http://www.dcs.shef.ac.uk/~wmlh/Software%20Hut%202008-9.htm

XP

Kent Beck: "Extreme Programming Explained", Addison Wesley, 1999.

Martin Fowler: "Refactoring improving the design of existing code", Addison Wesley 2000.

Eric Freeman and Elisabeth Freeman: "Head First Design Patterns", O'Reilly, 2004.

Ron Jeffries' book *XP Installed* is available on his website:

http://www.xprogramming.com.

Demystifying XP:

http://www-106.ibm.com/developerworks/java/library/j-xpcol.html

SXP

Mike Holcombe's book: "*Running an Agile Software Development project*" Wiley

http://agile.genesys.shef.ac.uk/

XXM

Chris Thomson: "Story cards and extreme machines":

http://www.dcs.shef.ac.uk/~cthomson/seefm03/cthomson4.pdf

PHPUnit

Sebastian Bergmann: "PHPUnit Pocket guide":

http://www.phpunit.de/pocket_guide/3.0/en/index.html

Web pages

There are links to various useful web pages on Chris Thomson's software hut del.icio.us page:

http://del.icio.us/smogit/softwarehut

Other documentation can be found here: www.dcs.shef.ac.uk/~cthomson/sh2007

Appendix

Writing Tests in PHPUnit

PHPUnit tests are very similar to those in JUnit. There is a gotcha that you need to be aware of however, and that is that PHPUnit can only tests class files in PHP and not the more 'normal' files which are a mixture of PHP and HTML. Therefore you should design your PHP using class files for the main functionality reserving the other type for defining the interface and testing it. You may also find that PHP Smarty can help with the interface elements (http://smarty.php.net/).

This example is taken from the PHPUnit (3.0.1) distribution it is a model of a simple bank account. To run these tests on the command line please install PHPUnit on your own computer following the instructions on:

http://www.phpunit.de/pocket_guide/index.en.php

Or use Eclipse in the lab using the PHPUnit plugin, which can also be downloaded for your own use, you'll need the plugin from the plugins directory and the modified version of PHPEclipse. They are available from:

http://www.dcs.shef.ac.uk/~cthomson/sh2007/

You should also download and install the Zend PHP plugins:

http://www.zend.com/de/pdt

Create a new PHP project in Eclipse and create the following files. The first file is the one that we will test, this models a simple bank account.

BankAccount.php

```
<?php
class BankAccount
{
    private $balance = 0;

    public function getBalance()
    {
        return $this->balance;
    }

    public function setBalance($balance)
    {
        if ($balance >= 0) {
            $this->balance = $balance;
        } else {
            throw new InvalidArgumentException;
        }
    }

    public function depositMoney($amount)
    {
        if ($amount >= 0) {
            $this->balance += $amount;
        } else {
            throw new InvalidArgumentException;
        }
    }

    public function withdrawMoney($amount)
    {
        if ($amount >= 0 && $this->balance >= $amount) {
            $this->balance -= $amount;
        } else {
            throw new InvalidArgumentException;
        }
    }
}
?>
```

The second file tests the first one. A function is created to test each of the original functions. The assert methods do the actual testing, and if the assertion fails you will receive an error when your run the test.

BankAccountTest.php

```
<?php
require_once 'PHPUnit/Framework.php';
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase
{
    private $ba;

    protected function setUp()
    {
        $this->ba = new BankAccount;
    }

    public function testBalanceIsInitiallyZero()
    {
        $this->assertEquals(0, $this->ba->getBalance());
    }

    public function testBalanceCannotBecomeNegative()
    {
        try {
            $this->ba->withdrawMoney(1);
        }

        catch (Exception $e) {
            return;
        }

        $this->fail();
    }

    public function testBalanceCannotBecomeNegative2()
    {
        try {
            $this->ba->depositMoney(-1);
        }

        catch (Exception $e) {
            return;
        }

        $this->fail();
    }

    public function testBalanceCannotBecomeNegative3()
    {
        try {
            $this->ba->setBalance(-1);
        }

        catch (Exception $e) {
            return;
        }

        $this->fail();
    }
}
?>
```

Lastly the final file collects several test files together, in this case we only have one, but when you test your projects you can use this method to collect all your tests together. To do this just an extra require_once statements and addTestSuite calls.

AllTests.php

```
<?php
if (!defined('PHPUnit_MAIN_METHOD')) {
    define('PHPUnit_MAIN_METHOD', 'AllTests::main');
}

require_once 'PHPUnit/Framework.php';
require_once 'PHPUnit/TextUI/TestRunner.php';

require_once 'BankAccountTest.php';

class AllTests
{
    public static function main()
    {
        PHPUnit_TextUI_TestRunner::run(self::suite());
    }

    public static function suite()
    {
        $suite = new PHPUnit_Framework_TestSuite('PHPUnit');

        $suite-> addTestSuite('BankAccountTest');


        return $suite;
    }
}

if (PHPUnit_MAIN_METHOD == 'AllTests::main') {
    AllTests::main();
}
?>
```

Now we need to set eclipse up to work with PHPUnit (you should only have to do this once). Select the Window→Preferences menu, in the preferences dialog select "SimpleTest" from the list on the left. For the PHP exe file enter:

"P:\eclipseplugins\php5\php.exe"

For the PHP ini file enter:

"P:\eclipseplugins\php5\php.exe"

For the PHPUnit path enter:

"P:\eclipseplugins\PHPUnit-3.0.1"

Now select the Run→Run menu and double click on the PHPUnit3 item on the left hand side. Now select your project and the AllTests.php file, and click on run. The tests results should appear in a new view and in the console view. Sometimes if there is an error in the PHP files no errors and no test results will appear, just rerun the tests until they do.

Further examples and fill documentation for PHPUnit can be found at:

http://www.phpunit.de/pocket_guide/index.en.php

Using the XXM Plugin for Eclipse

Overview

This XXM design tool is a basic Eclipse plug in which allows the creation of simple XXM type diagrams within the Eclipse framework. The tool produces output files in XML format for easy editing and use by other applications

The current data format only allows of a description of the connections of the XXM diagram. This is limiting as it does not allow the explicit definition of the inputs/outputs and memory normally associated with the XXM machine, however this follows from the philosophy of the "The Book of Genesys" which aims for ease of use above complete definition. The tool allows basic test sets based on a transition cover to be created.

Basic Usage

The basic use of the XXM model in the tool is highly configurable to the application at hand as the format of labels is currently unchecked. The recommended use of the diagrams is as follows: (This guidance should allow for the creation of useful and maintainable diagrams.)

- There are three types of states:
    - *Circular states should represent screens in the system*. The name of the state should match the name of the screen. The comment should match the name of the implementing class, a colon, and the method that displays the screen.
    - *Square states should represent some internal data state,* and a decision based on it. The comment should match the name of the implementing class, a colon, and the method that does the comparison.
    - *Start and End states should represent when control enters and leaves an XXM*. Typically only one state of each type should be used in each diagram, however for easy of presentation multiple states of this type are allowable. If this machine breaks down a higher level XXM, then the comment should contain the name of this XXM.
- Functions can only exist between states. There are three basic types: (All types should be labelled with a comment should match the name of the implementing class, a colon, and the method that implements this function. This method would normally be in the class that contains the originating state.)
    - The first corresponds to user input. These are fired on a user action and take the form: "click(ok)" where "ok" is the name of the action, and "click" is the type of the action. This type of function is only present originating from screen (circular) states.
    - The second corresponds to a decision on the data and take the form of a Boolean expression. They are always present in pairs (the expression and the negated expression) originating from an internal (square) state.
    - The final type is that which originates from start states or ends in a end state. These functions should be lifted from the calling XXM if it exists, or the underlying system otherwise. As such they would tend to fall into one of the above two types.

- In principle any aspect of the XXM can be broken down in a hierarchical function:
    - Any screen (circular), or internal (square) state can be broken down into a XXM which has a final number of functions equally to the number of functions leaving the original state. These should be labelled in the same way as the functions in the top level diagram.
    - Any function can be broken down, however it must always have exactly one final function pointing at an end state and labelled in the same way.
    - The strength of the generated test set is potentially compromised by breaking down a diagram in this manner. In order to maintain the strength is necessary to combine the test sets of all the diagrams produced in a cross product style way. However in most cases it would be sufficient to test the implementation of the XXM in isolation.

Benefits

By creating a XXM of your system you should be able to easily define a behavioural model what ever language you are using. You can not currently automatically generate this; however we are considering a Java implementation. The example system that is downloadable from the site gives one demonstration of this.

The tool currently produces a test set to allow you to integration test you system to ensure that correct user actions are possible. The generated test paths provide a way to show that all functions/transitions in the system do in fact exist and are connected between the correct states. The set provided is considered the minimal set required to show this. The tool does not identify test data for the test set, it only tells you what to test, and you have to work out how to test it. For each state and function you need to define inputs that will cause the function to be taken.

In the case of screen states it is easy to define these inputs as typically you will have already defined them as an action, for example clicking an OK button. In the case of the internal (square) states a requirement is placed on the memory of the system. In order to ensure this you may have to manipulate a database externally or set up the memory though an interaction through the interface, or more commonly both.

Using the plugin and Eclipse in the Lewin lab:

First thing, you will need to do:

- Go to "My computer", open tools menu -> Map Network Drive
- Assign a driver letter "P:" (or any other kind of letter, that is available for the you)
- Map to "\\holly\public"
- Make sure, the "reconnect at logon" is ticked.
- And then click finish.

Then, then you will need to create a shortcut.

- On the desktop, right click -> new -> shortcut
- Type in the location to the eclipse.exe (e.g. “P:\eclipse\eclipse.exe”)
- Then add an “-data”, to the end of that location. (e.g. “P:\eclipse\eclipse.exe –data”). The “-data”, will allow the program to write to the designated directory, (such as “H:\Workspace”). Without the “-data”, it will write back to

the directory on the "holly\public", which causes the program not to work properly. Since you do not have write access to "holly\public"

When you start the program, it will take at least 5 minutes to load up, and just appends how many people are loading it at the same time, which could take even longer. When the first pop up appears "Workspace Launcher", make sure the workspace location is "H:\eclipse\workspace", and tick "use this as the default, and do not show again" and then OK. The next time you load it up is should be a bit faster!

To install on your own machine

1. First get Eclipse. You need either version 3.0.1 or 3.1.0 (Platform SDK), currently we have only tested on MS Windows and these versions of Eclipse.
2. Download the plugin.
3. Unzip the contents of the zip file to the root of your Eclipse install taking care to preserve folders.
4. If you want download the example system, a small Java application developed in Eclipse, with XXM diagrams.

Creating your first XXM in Eclipse

1. First get to know Eclipse.
2. Next make sure the plug in is installed, see above.
3. Open File > New > Project, Using the wizard create a project for the type of application that you are going to write.
4. Open File > New > Other, select Sheffield Extreme Programming in the list, and then XXM Designer.
5. Click the browse button next to container, select the project you just created. Change the filename to one of your choice ending in XXM. Click Finish.
6. On the tab at the bottom of the file editor, click on Extreme XMachine.
    - You can now create states by double clicking anywhere in the editor, or Opening the XXM menu > Add State.
    - You can add functions by first clicking on a state, holding the shift key and clicking on another state.
    - To edit the names, comments and State types. Click on the state or function that you want to change. And select Window > Show View > Other, click on Basic, and select properties. You will now see the properties window that allows you to edit these properties.
7. To view the test set, click on the test tab.

Further Information

Flash demo of using the XXM plugin inside Eclipse:

http://www.dcs.shef.ac.uk/~cthomson/xxm/XXMEclipseTool.htm

Download the plugin, example and latest release notes:

http://www.dcs.shef.ac.uk/~cthomson/xxm/xxmplugin.html

Test document example

System: Example System

Subsystem: Login

Unit tests:

Use Eclipse to run LoginTest, all tests should pass.

Check manual tests on story cards.

Integration and Behavioral tests:

1 → *(start) > DisplayLogin > (Login) > click(register) > (Register) > click(ok) > (Display Users) > click(logout) > (exit)*
Run application, click on login, click on register, type "bob" in field, click on OK, click on logout.

2 → *(start) > DisplayLogin > (Login) > click(login) > (ValidateUser) > !exists > (Register) > click(ok) > (Display Users)*
Run application, click on login, type "bob" in field, click on login, type "bob" in field, click on OK.

3 → *(start) > DisplayLogin > (Login) > click(login) > (ValidateUser) > exists > (Display Users) > click(logout) > (exit)*
Run application, click on login, type "bob" in field, click on login, click on logout.

Tests run:

Date	**By**	**Result**
31/01/05	CThomson	*Unit tests:* Fail line 28, possible error in test? SC 12,13, user does not appear to have been added to database. SC 13, unable to locate 486dx2 to test QA. *Integration tests:* Pass. *Other observations:* Possible fault on the display user screen, no users displayed.

Extreme Programming Practices

No	Practices	*Activities/Products*			
1.	**Planning games**	Write stories cards- identify priority, composition of releases and date of releases	Estimate tasks and hours need to complete the task	Estimate cost	Plan to release main stories as soon as possible.
2.	**On-site customer (if possible)**	Write stories (optional)	Write simple user interface test sets	Discuss stories, user interfaces with client	Immediate feedback – discussions, email, phone
3.	**Pair programming**	2 persons – changing the roles after a few hours	Swapping partner after several weeks	Swapping partners – the 'experts' will be able to understand and check the programs	
4.	**System metaphor**	Common vision (used in discussion with client and among team members)	Shared vocabulary	Analogies	Architecture
5.	**Testing**	Test cases- developing test cases before coding	Test suite	Unit testing	Functional testing
6	**Coding standard**	Identify standard	Naming convention		
7.	**Simple design**	Simple user interfaces	Simple programs to facilitate future maintenance	XX-machine-to- Story cards-	
8.	**Refactoring**	Reuse the existing codes			
9.	**Continuous integration**	Incremental integration			
10.	**Short/ Frequent release**	Continuous review with client – to identify client's satisfaction as the project progress	Feedback from clients- to identify changes as early as possible	Changes in requirement	
11.	**Collective ownership**	Repository- CVS	Test partner- different pair acts as tester	Swapping subprogram	
12.	**Forty-hour week**	Less activities during final months	Completed projects	Tested projects	Integrated software

Courtesy of: Sharifah L. Syed-Abdullah