eXtreme Solo

A Case Study in Single Developer eXtreme Programming

Gareth Cronin

University of Auckland

Table of Contents

Introduction

Aim

The aim of this project is to use the lightweight software engineering methodology known as "eXtreme Programming" (XP) to produce a casual human resources management system for a university. XP was developed by Kent Beck and is a fine-grained, iterative method of development, relying on constant developer to customer communication and frequent working releases of code [Beck 1999]. This paper sets out to demonstrate that XP holds great advantages over more traditional top-down approaches to software development.

While XP was conceived as an ideal system for small to medium sized teams of developers, it also makes an ideal system of development for the solo programmer. A number of the team-centric caveats were of course omitted, but the intentions and spirit of XP were used throughout.

eXtreme Foundations

XP draws on many earlier ideas for solutions to the problems that have faced software developers since software development began. The solutions were a reaction to the problems inherent to the "waterfall" model of traditional software processes that has been a major influence since 1970 [Boehm 1988]. In the last decade software engineers have begun to realise that software diagrams and system documentation do not amount to design. Source code is design, "building" is design, debugging is design and testing is design [Reeves 1992].

Kent Beck states that the basic problem with software development is failure to manage risk: failure to keep to schedules, high levels of defects, failure to meet original requirements, failure to cope with changing requirements and complete failure: project cancellation [Beck 1999a].

Risk can be confronted by growing software rather than building it, the system should be made to run even if it does nothing useful at first [Brooks 1986]. An XP project is built in small increments with continuous feedback from customers. Changing requirements are absorbed in the process, and missed requirements are picked up quickly and easily. Defects are managed by writing tests before writing code and giving the highest development priority to ensuring that the suite of unit tests created during development always run at 100% [Beck 1999b].

Project visibility is a key factor in risk management. If the progress of the project is visible at all times, it is obvious if things are not "on track". Ron Jeffries, who worked on the C3 project (mentioned in **Previous XP Development** p.**Error! Bookmark not defined.**) tells of the failure of a new payroll engine to meet requirements on its first run. The customers had been inspired with so much confidence by the visibility provided through XP that the customers chose not to cancel the project but to broaden it:

> **"Imagine that. The worst foulup of your entire project, and your customer hardly even notices! They're so confident that they cheerfully broaden the scope of your effort.**
>
> **At the time, it was a surprise. But we understand how it happened. It happened because the Extreme Programming values, Feedback, Communication, and Simplicity lead to Confidence." [Jeffries 1999]**

Approaching the most difficult tasks first within an iteration also mitigates risk. If a task is breaching time estimates it can be reviewed before it causes serious delays in the project.

Values, Principles and Practices

XP is based on values, principles and practices. The four values are:

- Communication
- Simplicity
- Feedback
- Courage

"Communication" is about involving customers in the testing and always consulting directly with the customer when ambiguities arise. In a team situation it is also communication between programmers, in particular pair-programming. "Simplicity" is always taking the simplest approach to a problem.

"Feedback" is programmers receiving feedback from their programs through constant unit-tests, and high project visibility ensuring that the customers have constant feedback. "Courage" is about changing, refactoring or just throwing away code without fear [Beck 1999a].

The basic principles are:

- Rapid feedback
- Assume simplicity
- Incremental change
- Quality work

Rapid feedback is an expansion on the feedback value above – feedback is important, rapid feedback is even better. Assuming simplicity and incremental change involve designing only for today, not attempting to include plans for the future. Instead, XP programmers refactor and move in small steps to cope with change. Quality work is considered essential at all times.

Kent Beck also outlines more general principles of honesty, responsibility and speed. Minimising the number of artefacts means that an XP programmer can travel light and move fast.

The actual practices of XP described next realise these values and principles.

The Planning Game

The planning game is the process of eliciting requirements from the customers, estimating the time involved in each set of requirements and the associated risk of misestimating. The speed at which development advances is known as the *velocity*. The velocity for a team of one is the speed at which the solo programmer completes tasks. The requirements take the form of features that the customer wishes to have in the system – each feature is captured as a "story" on an index card. The customers select a priority level of 1 to 3 for each story and the highest priority stories that fit into the iteration length are the stories that are implemented.

Small Releases

Release cycles are kept as short as possible to maximise feedback to the customers and have a working system as quickly as possible.

I worked using three-week iterations with a release at the end of each.

Metaphor

The metaphor is the single concept guiding the whole system. In the case of my system it is that of a roster and a pay schedule as they are traditionally presented on paper: a grid for the roster and a grid for the schedule.

Simple Design

Always taking the simplest design possible minimises development time. It is very cheap to refactor to accommodate changes in the future. I did not find this to be a problem, but a case study of a software engineering class using XP at the Universität Karlsruhe in Germany found that students struggled with the concept of not planning for the future after years of training to the contrary. The authors of the study go so far as to suggest that withholding future requirements from programmers is the only way to prevent them designing for the future [Müller 2001].

Testing

XP demands the writing of tests before the corresponding code. Writing tests before code forces the programmer to think through what the corresponding code is supposed to provide. Test-first writing makes realising interfaces easy and ensures that a test is written for every unit. By developing a suite of tests incrementally, testing is much less of a chore than writing an entire suite of tests at the end of development.

The study at the German university mentioned above found that 87% of the participants felt that test cases strengthened their confidence in the code and planned to continue the practice after the course. The study found that:

> **"Writing tests forces software engineers to distinguish between the functionality to implement and the base conditions under which the implementation has to work." [Müller 2001]**

Refactoring

This fits in with the simple design practice. Refactoring is the art of changing parts of the existing program to make adding new features more straightforward. Often this involves generalising several similar classes to a single class or breaking a class into a generalisation with several specialisations.

Pair Programming

This is the most controversial of the XP practices, and unfortunately not possible as an XP solo programmer. Pair programming means that all programming is done as a pair of programmers at a single terminal. One programmer codes while the other programmer reviews code, the programmers exchange roles whenever they need to. This means that all design decisions involve at least two brains and there is less chance of one person straying from the discipline of the XP process [Miller 2001].

Collective Ownership

This is also a team concern and does not apply to my project. Collective ownership is a reaction to traditional models of individual ownership or no ownership:

> **"In XP, everybody takes responsibility for the whole system. Not everyone knows every part equally well, although everyone knows something about every part. If a pair is working and they see an opportunity to improve the code, they go ahead and improve it if it makes their life easier." [Beck 1999a]**

Continuous Integration

In a team situation this refers to integration within a few hours. A tested unit is integrated into the main build and the entire system is tested until the tests run at 100%. In my project as the sole developer I was able to maintain integration at all times. I ran the entire suite of unit tests as soon as a single unit was ready. When I compiled, I compiled the entire system.

40-hour Week

Kent Beck suggests that no one is capable of feeling fit and ready for work every day if they work a 60-hour week. He also suggests that overtime is a symptom of serious problems in a project. This was no problem for me as the most time I could afford to spend working on the project next to my other studies was around 25 hours a week.

On-site Customer

Communication and feedback at the necessary speed and frequency are only possible if the customers are accessible. In an XP project, Kent Beck recommends that a representative customer relocates to the XP team's building, making them available for face-to-face conversations whenever a question needs to be asked.

Two of my customers, Amber and Deborah worked in a building some distance away from my office, but Emma-Jane was located a floor below me, and Elspet worked in a cubicle in my office. With one of each of my main types of stakeholders easily accessible, I felt that I met this requirement.

Coding Standards

Collective ownership and pair programming means that with different programmers changing the same code all the time it is impossible to work successfully without a single set of coding standards.

This is another team concern, rather than an individual one. Nonetheless, consistency across code is important, and I created naming conventions to use in all parts of my design.

Related Work

The first "full" XP project was the large-scale C3 payroll project developed at DaimlerChrysler between 1996 and 2000. C3 was originally started in 1995 using traditional development processes, but it failed to meet requirements. At Kent Beck's suggestion, the company dumped the original design and began again using the XP process. The company has stated that the only time they fell short of their promises to the customers was when they strayed from the XP principles [Beck 1999b].

A number of university-environment case studies have been carried out and documented [Müller 2001][Kivi 2000], and testimonials of XP-based successes in industry are emerging all the time [Beck 1999b].

XP has been used successfully under a number of different project structures, even in a fixed-price scenario with a system for modelling a viral infection scenario [Pelrine 2000].

Hundreds of businesses and thousands of people are now using XP as their standard development process.

XP is not the only "lightweight" software process; open source development [Raymond 2001] shares many of the same ideas, although it lacks the discipline of XP. A process that is often compared to XP is Feature Driven Development (FDD), it is also an iterative customer driven approach [Fowler 2000].

Background

Top-Down Design

A month or two before I began the XP development, I made a classical top-down analysis of requirements for the system. In hindsight this was useful, as I have been able to see the errors that I was blundering into before the change to XP. Before I turned to XP, I had completed a use case analysis with event flow narratives and some high-level object analysis and design. By the time I had completed the first XP iteration, I realised that my original model was unnecessarily complex and that the requirements were confused.

Defect after Defect

Before this project I had developed two major systems for Student Administration at the University of Auckland. In each I used classical top-down analysis and design. I met with the customers, I listened to their requirements and I interpreted and noted down these requirements "in my own terms". I went away to my office and returned some weeks later with the "completed" system. I employed no incremental testing and designed no formal acceptance testing.

The result of this style of development is predictable. I found myself locked into a long period of "code and fix" issue resolution. I had a werewolf on my hands [Brooks 1986]. I had missed major requirements and spent large amounts of time "bolting them on" to the system without stopping to refactor when structures became unnecessarily complex. Every time I fixed something it would break a number of other things, creating whole series of chaotic chain-reactions in what should have been a stable production environment. The design that I had created was almost too rigid to simplify and I was spending so much time supporting and fixing that I did not have time to backtrack and refactor anyway.

This is not to say that the systems do not perform their functions. They do and they are still used, but I have no confidence in the quality of the underlying code and its ability to withstand changes in the business practices.

The Casual Staff Management System

The Student Administration of the University of Auckland was facing a problem with management of its casual staff. While the new enterprise-wide PeopleSoft implementation had eased the burden of processing permanent staff, casual staff requirements had been somewhat neglected. I was asked to produce a system that could store rosters, produce pay schedules and eventually automate roster generation based on staff requirements and staff availability. The only technology-specific requirement for the system was that it was to be operated exclusively from a web interface. Initially the system would service the Call Centre and the Student Information Centre, but eventually it might be expanded to handle the other divisions of Student Administration: the Recreation Centres, the Accommodation and Conference Centre and the Maidment Theatre.

The Customers

The two business centres are in different physical locations and have separate cost centres, but they share staff. Almost all of the staff members are employed on casual contracts and most are full-time students, working part-time to support themselves through their studies.

The Student Information Centre is a central drop-in point for all student administrative matters for the University. It is located in the main Student Administration building. The centre has a large front desk where staff assist students with enquiries and receive forms for processing in various departments. The

centre also has computers set up as student access points to the University's web-based enrolment system. Staff support students at the computers as they carry out such tasks as enrolling, changing courses and programmes and finding course information.

The Call Centre is a telephone reception centre where operators receive calls from the University's toll-free number. The Call Centre answers general enquiries and provides support to students using the online enrolment system from home. The Call Centre also deals with emails to the University's general enquiries addresses and mails out publications such as handbooks and prospectuses.

There are three main stakeholders in the system: the supervisors of the Call Centre and the Student Information Centre, the Human Resources coordinator for Student Administration and my employer: the Group Manager of Information Systems for Student Administration.

Three customers wished to represent the first stakeholder:

Name	Role
Emma-Jane	Student Information Centre manager
Amber	Call Centre manager
Deborah	Call Centre team-leader

A single customer represented the second:

Name	Role
Elspet	Human Resources coordinator for Student Administration

My employer made it clear that he did not wish to be involved in the development – I was to give him a brief on the scope of the project and an estimate of the projected time to completion. I was also required to meet with the managers of the other Student Administration groups who may be interested in the system in the future to ensure that no essential requirements were ignored.

The first group of customers have a purely functional background but did have some experience in defining requirements and acceptance testing from time they spent during the implementation of a new enterprise-wide administration system that the University had been carrying out over the last two years.

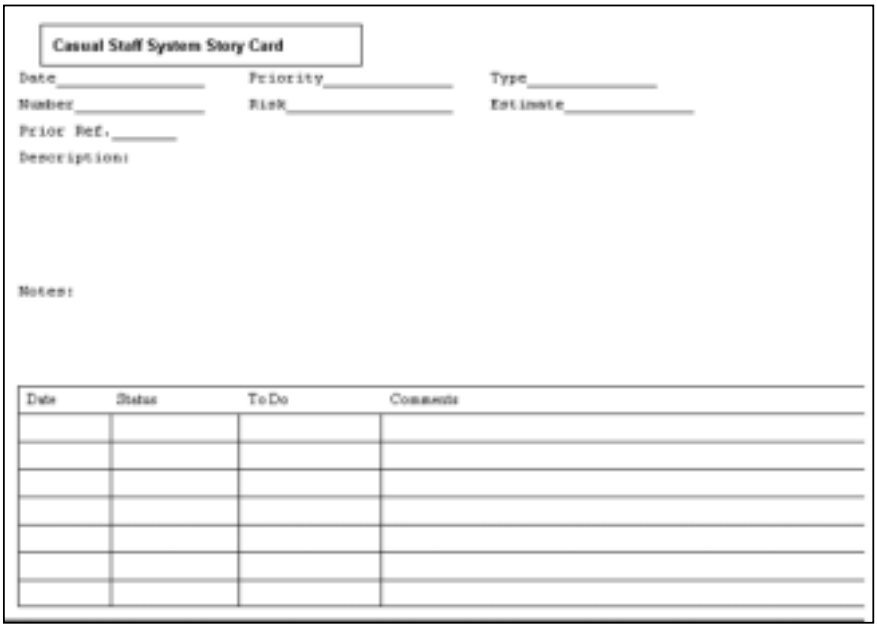
Elspet has a technical and functional background and been consulting on development and testing of the enterprise-wide system and coordinating training for it. She had a more active interest in the development as a whole, stating that:

> **"[I have] a background in Operations Management with a particular interest in business process redesign to align processes with strategic and operational objectives (which by my definition always includes customer focus and staff satisfaction). I believe that technology should support business processes and that care should be taken that the shaping of business processes is not be abdicated to any technology system - however technology can be used to drive change. I believe that testing of the human interface is often overlooked - i.e. if the system works it will be signed off but often there is no testing relating to what it would be like for someone to operate the system day after day"**

Preparation

My XP Toolkit

I closely followed the examples in Kent Beck's "eXtreme Programming Explained" to produce the index card to use as a recording device for customer stories [Beck 1999a].


Casual Staff System Story Card

Date_____ Priority_____ Type_____

Number_____ Risk_____ Estimate_____

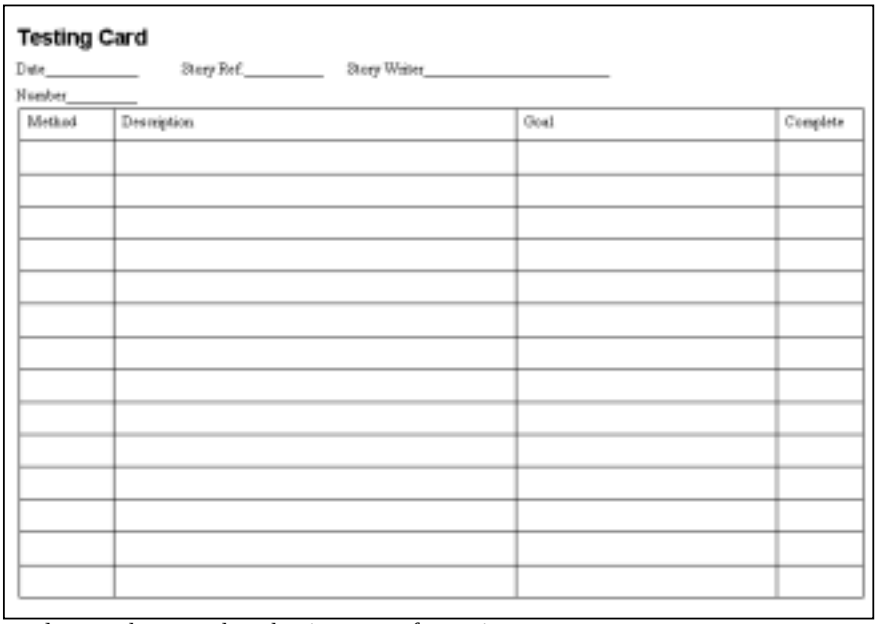
Prior Ref._____

Description:

Notes:

Date	Status	To Do	Comments

To aid in the writing of functional tests alongside my customers, I created a test-plan card that I could


Testing Card

Date_____ Story Ref._____ Story Writer_____

Number_____

Method	Description	Goal	Complete

staple to each story when the time came for testing.

The story card has room for the priority a story receives at the planning game, the estimate of how many "points" the story will take to complete, and the risk involved with the accuracy of that estimate. The cards also have room for a sequential reference number, a date, and a prior reference if the card is a modification of a prior story. Beneath these is space for the customer's story itself, my notes on that story, and the task grid.

I used the task grid to track the actual time it took to complete the story. In this way I could refer back to stories to improve my estimates as the project progressed.

I chose to use a system where one point represents a half-day of "perfect engineering time". *Extreme Programming Explained* suggests that a single point should be a perfect engineering week with a team of ten or so programmers, so I decided that the half-day is a suitable granularity for a solo developer.

Architecture

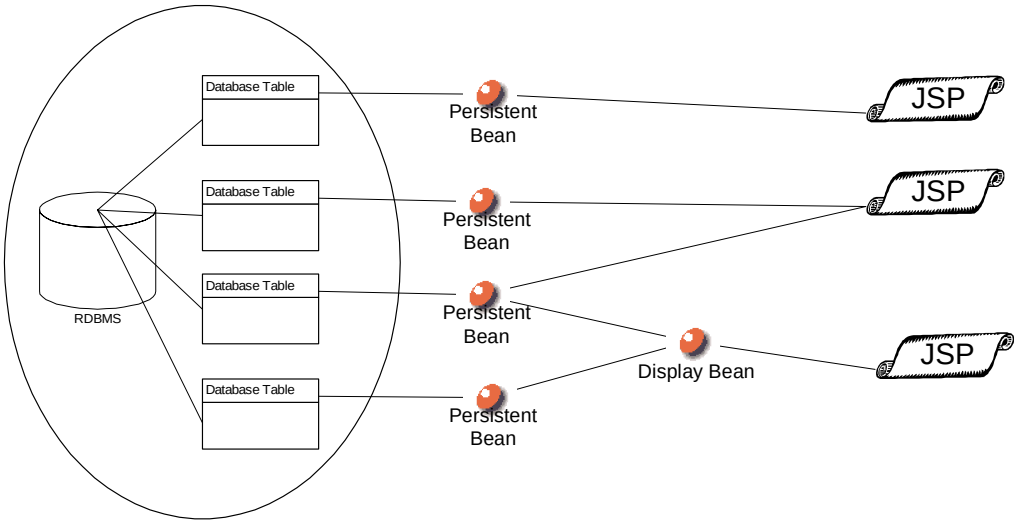
Although “architecture” is a dirty word in XP, I found it was necessary to at least define the set of tools and APIs that I would use in development, and the roles that they would play.

I considered using the Java 2 Enterprise Edition [Sun 2001] framework and creating a set of Enterprise Java Beans [Sun 2001b] as the foundation for the system. The complexity of configuration and initial design was such that I decided that given the small size of the system, JavaBeans [Sun 2001a] with hand-coded persistence would suffice. I call these “Persistent Beans”. There is a bean for each logical entity in the back-end relational database.

The sole requirement that my employer had made was that the system would present a web-interface. No additional client software could be deployed – I was to rely on the University’s Netscape 4.x [Netscape 2001] and Internet Explorer 5.x [Microsoft 2001] installations as the client.

I chose JavaServer Pages (JSPs) [Sun 2001c] as the presentation technology. JSPs provide a way to leverage the full power of Java within the web context. My employer was keen for me to take advantage of the zero-cost of open-source solutions, and so I decided to deploy the system on Tomcat [Tomcat 2001] – the open-source web application server.

The JSPs present and alter data either directly through the persistent beans, or through “Display Beans” that gather data from the persistent beans and perform some additional processing before the data is presented. An example of this is the beans that prepare arrays to be used in grid displays, eventually becoming HTML tables in the JSPs.

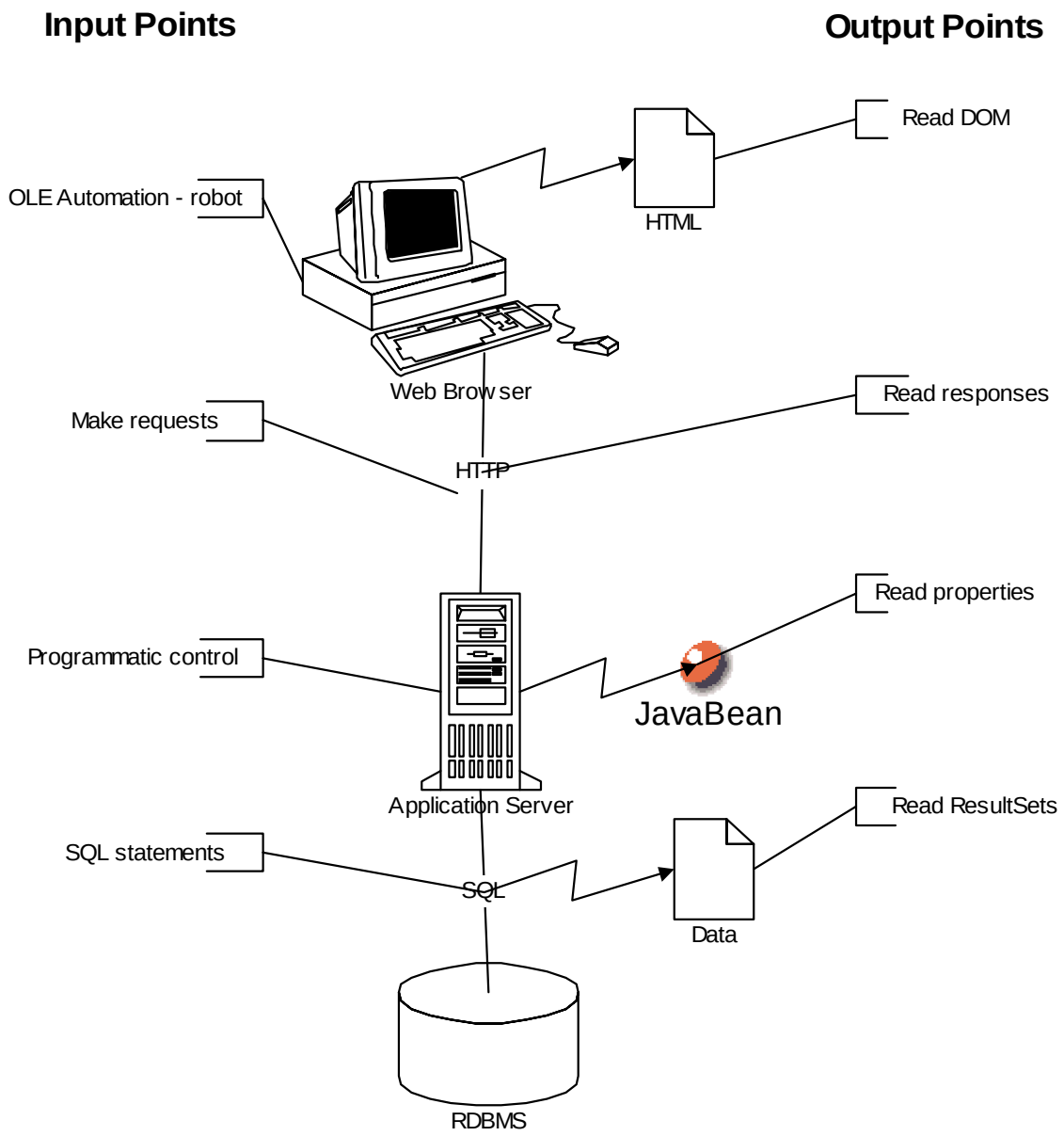


I chose to follow the advice of the writers of “Professional Java Server Programming” by using HTML only within the JSPs. At no stage was mark-up language to enter the JavaBeans [Allamaraju 2000].

Testing Levels

XP specifies that unit tests must be written before the code that they are to test and that unit tests must run at a 100% pass rate at all stages of development and deployment. XP also specifies that functional tests must be written by the developer and customer together to ensure requirements are being met.

My architecture exposes testable interfaces as below:



As I mention later, only some of these interfaces are practical and useful as testing interfaces.

Iteration One

This iteration took place between the 4th July 2001 and the 20th July 2001

The First Planning Game

The first meeting of the four customers and myself was stretched over two one-hour-long meetings. Once I had explained the concepts of the planning game, I passed out story cards printed on coloured paper. All four customers were eager to express their ideas. After forty minutes I already had a stack of twenty cards. After splitting and merging, I left the first hour-long session with twenty-four cards. I produced and noted estimates on these before the next meeting.

For the next meeting we used a room equipped with a white-board and beanbags. I spent the first twenty minutes of the meeting at the whiteboard. We brainstormed and I sketched ideas on the board. The customers had not worked with this method of requirements analysis before, so they tended to vocalise their ideas without writing them down – no doubt expecting me to interpret and write them down myself. After someone mentioned a feature, I would ask them to immediately write it down: "write a story!" In this way the stories began to build.

Even after reading the first few cards, I realised that I had introduced my own terminology to my earlier top-down OOA and OOD design, rather than using the customers' own terminology. With the natural conversational process of creating stories, the customers were able to lend their own business terminology to their processes. I could then take this and replace my somewhat artificial terms with more familiar ones.

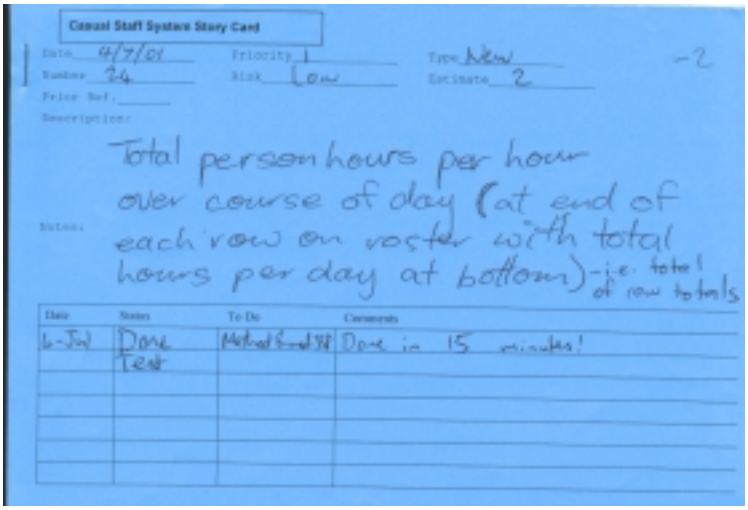
In the second session, it quickly became apparent that some of the tasks that I had previously dismissed as long-term goals were important to the customers at a much sooner stage. Amongst these was the implementation of an availability roster to electronically record when a staff member was available to work.

Thanks to the comfortable and spacious nature of the room, we were able to spread out all twenty-four of the original stories on the floor and walk around them, deciding on priorities. A certain amount of conflict arose as the customer who was to produce the pay schedules came up against the three customers whose main interest was in establishing an electronic roster: Emma-Jane, Amber and Deborah. They were able to resolve this by coming to a consensus that the first release should focus on an electronic roster, as the pay schedule would be dependent on this anyway. Elspet was happy to defer her stories until the second release.

When everyone was satisfied with the three priority piles, I added up my estimates for the eight priority-one cards and found that I had nine perfect-engineering days of development. I set a delivery date for ten days from the meeting and summarised what features the customers could expect. I supplied the customers with plenty of blank cards and encouraged them to write more stories and deliver them to me.

Implementation - The First Stories

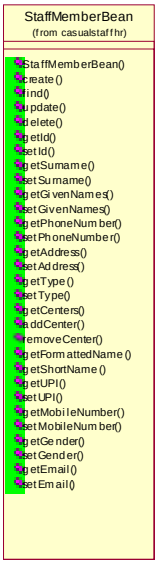
The following illustration shows a story from the first iteration.



The following table shows the stories included in this iteration.

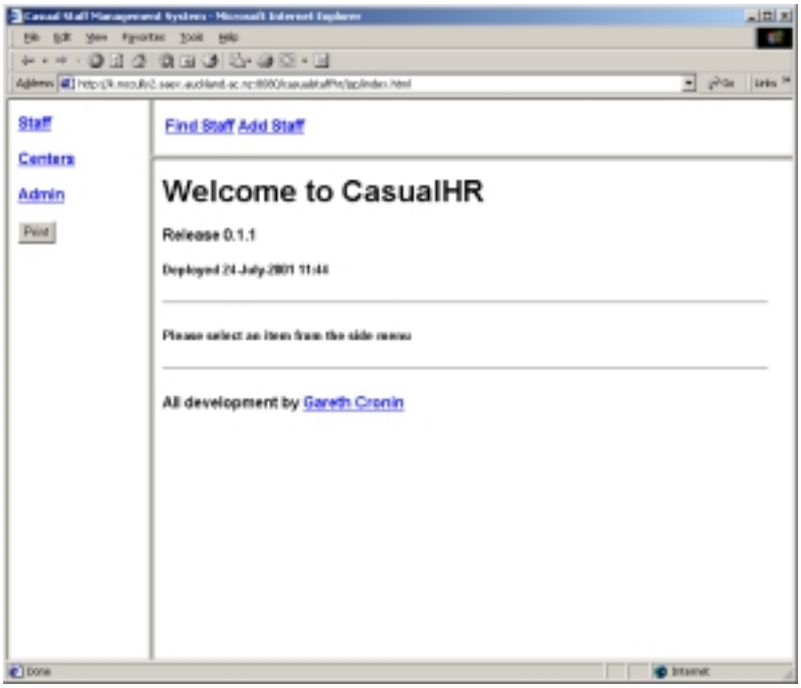
Number	Description
1/4	Need to be able to print a copy of the roster, and also a copy of the student details together or separately
1/5	To be able to see highlighted all staff members who are available by hours or days or over a month and print a list of this
1/6	Roster are easily updated daily so all daily staff changes can be entered into the roster. So, sick days show up as red, lateness shows up as blue, absenteeism as yellow.
1/10	Add an hourly pay rate to each staff member
1/15	Need to be able to add/change/delete a student name, address, DOB, id no, telephone no, cell phone no, M/F, start date, termination date, UPI and email address
1/24	Total person hours in each time block over course of day (at end of each row on roster with total hours per day at bottom)
1/25	Create a grid system that shows who is rostered in each area by hour
1/26	Need to be able to add lateness time, break time and leaving early time

Development began with story 1/15. 1/15 specifies personal data to be stored for the staff, so this meant implementation of the persistent beans to handle the staff data and the writing of JSPs to view, add and update the data. The following diagram shows the class I extracted from this story.

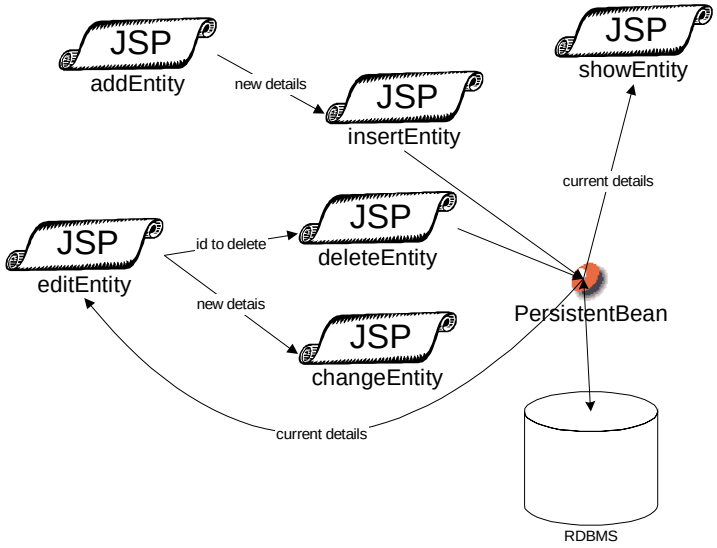


In the earlier top-down design before I began the XP process, I had planned to use a hierarchy with `StaffMember` as an abstract super-class and two sub-classes of `CasualStaffMember` and `PermanentStaffMember.` Under the XP paradigm, the simplest solution must be applied, hence the single `StaffMember` class. Further on in the development I realised that having a single class is a much better solution. I coded the persistency tests, coded the persistent bean, and created a `StaffMember` relation in the back-end database.

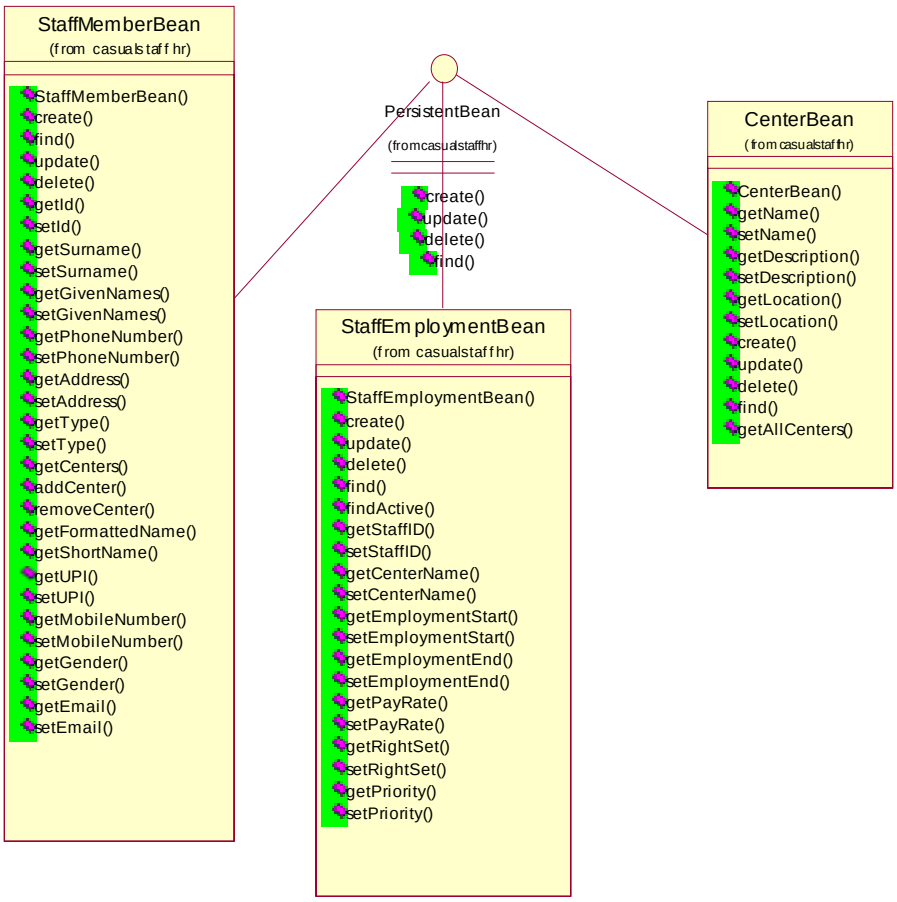
At this point I needed to implement JSPs for adding, editing and deleting the staff members. I decided to use a frameset with a side menu and changing top menu. The navigation for the system would consist of selecting a set of functionality from the side menu – e.g. “Staff” – that would cause the appropriate sub menu to appear in the top frame. Functions selected in the top frame would open the appropriate JSP in the main frame.



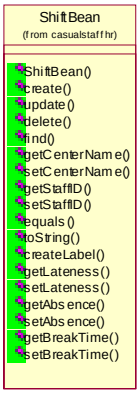
I developed a simple model of one JSP with a form for adding records, another with a form for editing or deleting records and three non-visual JSPs for carrying out the interaction with the persistence methods in the beans. The three non-visual pages were prefixed with “insert”, “delete” and “change”, and the two form-based pages were prefixed with “add” and “edit”. A final visual page was prefixed with “show” and used to display the contents of a record.



The next story was number 1/10 – adding an hourly pay rate. The pay rate applies to the centre that the staff member is employed at, so I needed to create a `Center` class and a `StaffEmployment` class. This exposed a common interface – `PersistentBean.`

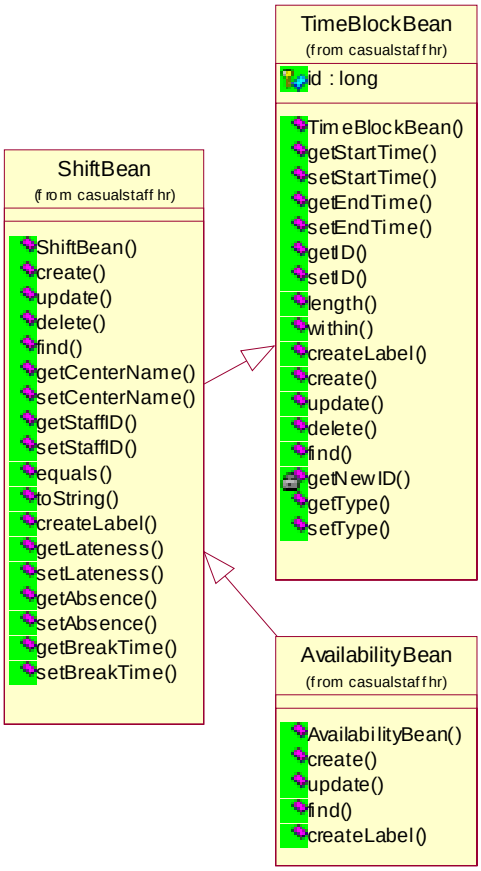


Story 25 requires a roster in a grid presentation. A roster requires a means of storing the shifts that a staff member works, so these classes had to be created next.

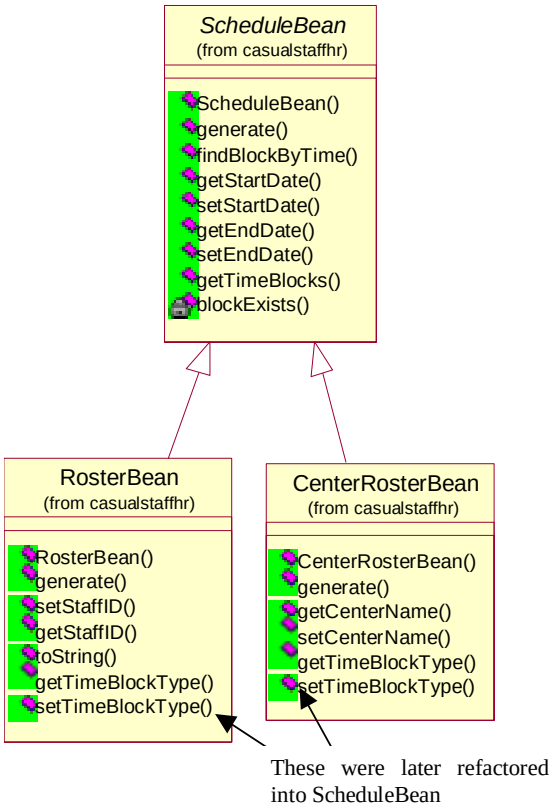


With a pool of `ShiftBean` objects, it is straightforward to produce a grouping spanning an arbitrary time period. This is important, as rosters are logically grouped over a working week, whereas pay schedules follow a pay cycle of a fortnight beginning on a Tuesday. Story 1/5 specifies being able to see a roster of availability. So I abstracted the `ShiftBean` out to a `TimeBlock` with the intention of having an `AvailabilityBean` that would also inherit from `TimeBlock.` However, I decided that `ShiftBean` had all the functionality that `AvailabilityBean` needed, so I made this the inheritance relationship instead. This is not a very natural representation, as I was left with much

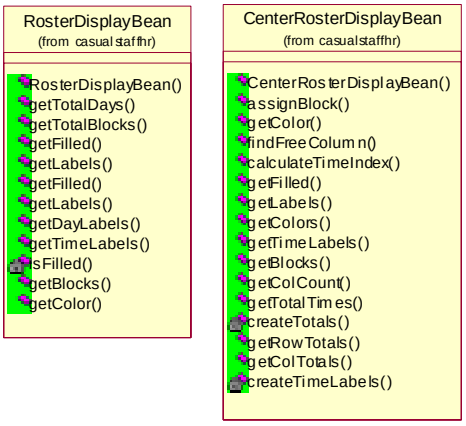
redundant data in `AvailabilityBean` but it was the simplest solution I could reach at the time, so I selected it.



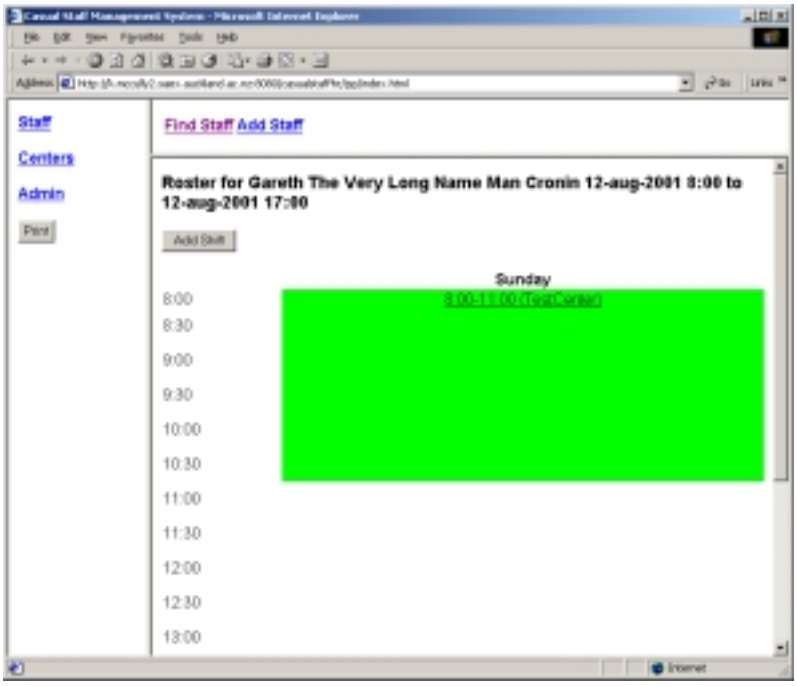
The rosters had to be shown by staff member or by centre, so for the underlying roster representation I abstracted a `ScheduleBean` with two subclasses.



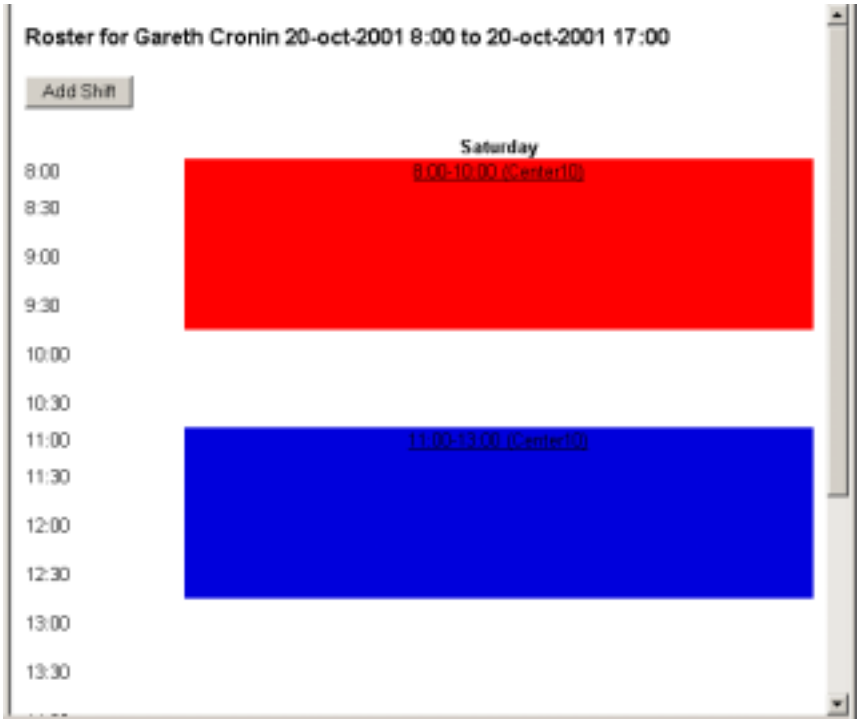
The visual representation really needed another layer, so I created two beans to hold a two dimensional array representation, ready for converting into HTML tables.



To finish story 1/25 all I needed was the JSPs to display the grids. Story 1/25 was finished right within my estimate of 4 points.



Story 1/26 required lateness and break time to be recorded somewhere for each shift. I added this to the `ShiftBean`. Story 6 required that a late shift show up on the roster grid in blue, and that absenteeism and sick days also be recorded for each shift, each appearing with a different colour in the grid. This was a simple matter of adding colour to the tables and a few more properties to the `ShiftBean`.



Story 4 called for a printable copy of the roster display. Because I was using a frameset in my application, there was difficulty involved for the user in ensuring the correct frame was active before printing. I added a “print” button to the side menu that used JavaScript to command the browser to print the contents of the “main” frame.

Unit Testing

For me, the most significant change to my development practice as a whole was writing tests before writing code. If a suite of thorough unit testing is in place, the confidence that one feels that a functional test failure is not due to a low-level failure within a class is one of the most valuable programming techniques I have ever come across. An enormous amount of time is saved during debugging at the functional level if a unit-level failure can be discounted immediately. When debugging my JSPs, I could put unexpected values down to an error within the JSP rather than further down in the JavaBeans.

Incremental unit testing is central to XP, and the XP website offers unit-testing frameworks for most popular languages. JUnit [JUnit 2001] is the Java-based offering. JUnit is a well-designed, clean framework of classes accompanied by thorough documentation on setting up a hierarchical testing system. While developing, the unit tests grow with the system creating a full regression testing system.

Because of the component nature of my architecture, the obvious strategy was to develop a testing class corresponding to each bean. Thus the StaffMemberBean has a StaffMemberTest class and the RosterDisplayBean has a RosterDisplayTest class.

The tests themselves can be divided into persistency tests, logic tests and monkey tests. The persistency tests check that the values that the properties of persistent objects are assigned stay the same through all phases of the objects life: creation, updating and retrieval, and that the object is impossible to retrieve following deletion. Logic tests check that the objects utility methods function correctly, e.g. that the `equals()` method has the expected behaviour. Monkey tests ensure that the class handles or raises appropriate exceptions when it meets unexpected values such as null property values.

Persistency

Testing persistency proved to be a challenge. I began by creating an interface for a persistent class test. The interface contained four methods:

- `testCreate()`
- `testRetrieve()`
- `testUpdate()`
- `testDelete()`

The creation test produces an instance of the class, sets its properties and calls the `create()` method to insert a row in the database table or tables associated with the persistent class. It then tests the properties to ensure they still contain the original values.

The retrieve test produces a new instance of the class, sets its identifier and calls the `retrieve()` method to set the properties of the class to the corresponding field values in the database. The properties are tested to ensure they again match the original values.

The update test sets the properties to a new set of entirely different values, calls `update()` to update the database and then takes a new instance and calls `retrieve()`. The properties are tested to ensure the update has taken place correctly.

Finally, the delete test calls the `delete()` method of the persistent class, then tests to ensure that `retrieve()` fails – indicating that the corresponding database data has been deleted.

My style of unit testing evolved further as the later functional testing revealed problems that I had not tested for earlier. An example of this is when I discovered that after functional testing involving the updating of staff details, all the staff members in the database had identical data. In other words, although they had different identifiers, their other details had somehow been cloned. I traced this to a missing “WHERE” clause in the SQL for updating staff details. An update on one staff member object was updating all the rows of the staff table in the back-end database.

To resolve this, I introduced a little formal experiment methodology by adding a “control” to my persistency tests. In the `setUp()` for each persistence test, I added rows to the appropriate database tables using known values. I deleted these rows in the `tearDown()`. In this way, there was a known set of data being introduced to the database for each individual test that shouldn’t be changed in any way by subsequent operations on objects. I added a `controlTest()` method to the persistence tests and called this method at the end of each of the create, retrieve, update and delete tests. The control test retrieved the control data from the database and tested that it still matched the original values.

Logic Tests

Many of the persistent beans in the system have `equals()` methods that have to be tested. This is not as simple as just creating two objects that are expected to be equal and asserting that this is so. Equality can go very wrong in very subtle ways. I like to enumerate all property combinations that should and should not be equal and then test each combination.

Take for example, in a bean with an "id" property and a "name" property. Two instances of the bean are set up. The necessary testing can be represented with the following matrix:

	`id1 == id2`	`id1 != id2`
`name1 == name2`	assert equal	assert not equal
`name1 != name2`	assert not equal	assert not equal

This means that testing has a complexity of n^2, where n is the number of properties in the bean that must be equal for two instances to be considered equal. Thus, unit test classes can swell to sizes considerably larger than the classes they are testing.

Monkey Tests

Kent Beck uses the term "Monkey Tests" to describe tests where effectively what is being tested is the system's ability to stand up to being used by someone with little or no knowledge of what values the system expects to be inputted. Monkey Tests in this respect are needed at the functional level, but testing that correct exceptions and exception trapping is present is also useful at the unit level.

The best example of this is the behaviour of persistent beans when handling null values. If a null value is concatenated into a string in Java, Java converts it into the string "null". This can cause havoc in persistent beans when the string in question is used in a SQL query string. Running a query with the word null in place of a numeric value or as a legal string value will cause unpredictable results depending on the domain constraints of the database relations being manipulated. It is desirable in most circumstances to convert the null value into an empty string if the property is a string type, or to raise an appropriate exception before the value is included in a database transaction. Each of my unit tests for a persistent bean includes a test where all the properties are set to null and each of the persistent operations are attempted against appropriate assertions.

Monkey tests can also be written to test that property values outside their domain constraints and nonsensical values are handled correctly. For example, numerical values that exceed pre-defined maximums or reversed start and end dates.

Test-first coding began showing its power when I was implementing the underlying objects for the grid displays. As I wrote each test, I wrote dummy methods in the class being tested, usually returning an array of the expected values. I was able to create a compiled and running test before I even began to think about how I was going to implement the code to generate the real expected values. When the test was complete, I simply replaced the dummy methods with the real code to do their job. This system forced discovery of all the necessary methods for the interface and left me with a unit test that I could rerun after completing each method.

Task Tracking

Thanks to the user stories I discovered a significant simplification was possible over my early pre-XP design. I was able to reduce the number of objects and database tables by half. I abandoned my original idea of having templates from which rosters could be generated in favour of simply copying established rosters from one date-range to another. I abandoned the idea of using persistent roster objects containing aggregations of persistent shift objects in favour of a pool of persistent shifts that recorded their own staff and centre data from which the "roster-ised" information could be extracted at runtime.

The idea of refactoring to allow for these changes was daunting, but following the XP values of *simplicity* and *courage*, I threw out the old code and tests and started again. The whole process took only half a day and greatly simplified the continuing design.

I noted the time it was actually taking me to complete the tasks for each story on the task-tracking grid on each card. I found that I had grossly overestimated the time for most of the stories, as I was being over-cautious having never tracked my time in this way before.

Functional Testing

XP requires that functional tests be automated. I looked into using JUnit's sister software HttpUnit [HttpUnit 2001] for the functional testing. HttpUnit takes the browser's place and provides sets of methods for creating HTTP requests and testing the responses. I felt that this testing was a little too far below the functional layer. I wanted a system where I could test through the same interface that the users would be using. I searched for a suitable zero-cost tool, but I was unable to find one. I decided that the best solution would be to write one. I included 2 points of engineering time in the first story's estimate to allow for this.

I developed a testing tool in Visual Basic for Applications (VBA) within a Microsoft Excel spreadsheet. VBA gives the programmer access to the spreadsheet application's objects and other application's objects through the OLE automation interface. Each application contains an OLE server that provides an API for manipulating the components of the application as though a user was doing so through normal mouse and keyboard actions. My tool created an instance of the Microsoft Internet Explorer application and used its OLE server to manipulate the browser.

Internet Explorer uses the Document Object Model (DOM) [W3C 2001] as an internal representation of the active hypertext document. I provided a "capture" facility where the HTML associated with the top-level nodes of DOM is recorded to a text file. I then created a simple scripting language that is entered into cells in a spreadsheet and provides facilities to:

- Navigate to a page
- Fill out text fields in a form
- Submit a form
- Compare a page to a previously captured page

The results of the comparisons are recorded in another spreadsheet as "pass" or "fail".

XP requires that the users participate in the writing of functional tests. This spreadsheet model with its simple language meant that the users could sit with me and approve or disapprove the tests that I set up for each story. The nature of an Excel workbook meant that the tests for each story could be placed on one spreadsheet each, nicely encapsulating the process components.

Another advantage of the direct browser manipulation approach is that the test can be observed, and errors can be easily diagnosed by pausing the process and observing the browser state.

Full details on the scripting language and use of the web robot can be found in **Appendix A – The Web Robot (p.61)**.

The Excel Web-Robot can be downloaded from http://casualstaffhr.sourceforge.net/excelwebrobot.zip

My web robot was not yet developed to a stage where the customers would feel comfortable scripting their own tests, so I asked them to sit with me and write them as I demonstrated the actions that the robot would need to take in order to complete the tests.

I attached a "Testing Card" to each story card and filled this out with a description and goal for each step of the test. For example, in the story for adding an hourly pay rate to each customer, the steps are:

Description	Goal
Retrieve staff member and add employment at a centre with a known pay rate	Successful addition of centre to staff member
Retrieve staff member's employment details	Correct display of known pay rate
Change staff member's pay rate to a second known value	Successful change
Retrieve staff member's employment details	Correct display of second pay rate

Release

The release date I had originally set coincided with the two centres' busier times as students began to enrol for the second semester. An outbreak of flu meant that the centre managers were unavailable until the initial release date to assist in writing functional tests. For these reasons I had to delay the release

by three days. The web archive file ready for deployment on Tomcat and a zip of the source code can be found at:

http://sourceforge.net/project/showfiles.php?group_id=28900&release_id=44320

The source code for this iteration with a revision tag of 2.0 is also available from the CVS repository at cvs.casualstaffhr.sourceforge.net.

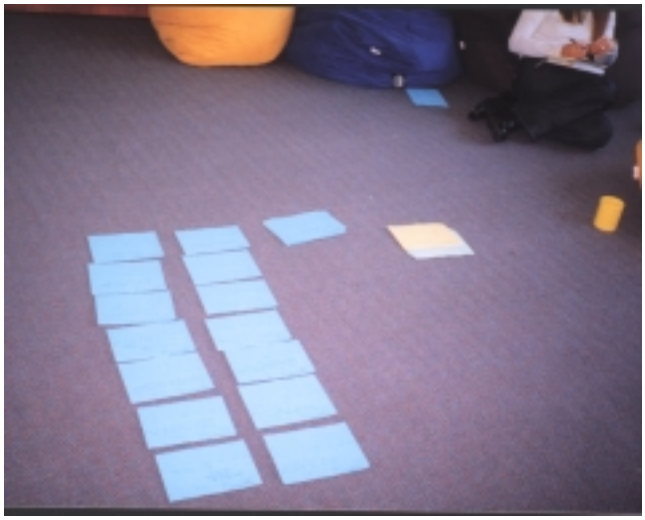
Iteration Two

This iteration took place between the 25th July 2001 and the 27th August 2001

The Second Planning Game

Given the advantages of the large room that we used in the first iteration, we used this room for the second planning game meeting. The customers presented me with stories they had written in the time since I had last seen them and I went through them and marked them with estimates of time to completion. Thanks to the stories of the first iteration, all the customers were well aware of the functionality implemented so far and were comfortable with the progress of the system.

Emma-Jane (left) and Amber (right) with stories on the floor



The stories being sorted into priorities

We held a brainstorming session where we mulled over various ideas for the second iteration. The customers were all keen to have the pay schedule generation working by the end of this iteration, but to safely handle pay details the system needed to be secured. I sketched out and explained options for security models and the difficulty of the varying implementations. The customers settled on a model involving a set of different security rights that could be applied to each staff member at each centre they worked at and a username/password login at start-up. This is explained further in the implementation section below.

By now the customers had all well and truly taken aboard the idea of writing stories, and needed little prompting to write them as we talked. They took turns writing the stories for shared ideas. As each story was finished I read through it and discussed details, jotting these on the card. It is interesting to

note that the XP lingo had entered the customers' vocabularies by this stage and they were using terms such as *stories* and *iterations* in a natural way.

Some of the stories that had been written previously were already covered by features implemented in iteration one, so I was able to put an estimate of 0 points and a type of "Already Done" on these.

I projected a release date of three weeks from the meeting and sufficient time to implement 22 points worth of stories. I was concerned that the security model could not be implemented as well as the pay-schedule within these time constraints and the customers suggested that so long as they could access the system with a single logon, they could limit access to the system to a select few and the more complex security model could be deferred. However, after adding up the highest priority stories, I found that we only had 17 points, so the security could be included after all. My estimates were considerably smaller this time, having over-estimated the time for almost all the stories in the first iteration.

The First Defect

During the first week of development of the second iteration, the customer associated with the pay schedule – Elspet – started using the first release, trying out the different functions. She called me over to her cubicle to show me the dreaded HTTP 404 error – "page not found". I assumed the server had ceased to function correctly, so I checked its status, but it was operating normally. I went to my JSP folder and looked for the page that should be there – it wasn't. I had completely left out one part of a story.

The roster for a centre was displaying correctly, but clicking the "add shift" link produced a 404 error. I had not created the JSP form for entering the shift details, or for creating the new object. Elspet asked me whether I had formal test plans. I replied that I did and showed her the test cards stapled to each story. On the test card for the problem story, I had written "similar to roster", a reference to the tests for a different story. In my rush to get the first release out, I had neglected to write and perform a full acceptance test for the story.

The only obvious defect arose where testing had not been conducted properly. This was a lesson I would not forget.

Implementation

The following table shows all the stories I included in this iteration.

Number	Description
1/9	Include/exclude total permanent staff hours with casual hours
1/14	Report hours worked total roster day/fortnight/month…
1/23	Timesheet report in the same format as HR
1/31	Any hours worked consecutively after 5.5 hours deduct ½ hour for lunch
1/32	Allow staff to indicate and have captured their preference of which centre they want to work in
1/35	Weekly overview of roster necessary
1/38	Comments field. Preferred area to work.
1/39	Ability to edit roster in the fly i.e.: after first pass of best fit make changes
1/40	Associate break with length of time for a shift
1/42	Requirements, availability, roster – copy from any time period to any other time period. Function to copy full week's roster to another place, for amendments.
2/1	Change to M T W TH F SA SU from full days
2/2	User ID auditing
2/3	Gender shouldn't default to female when adding new staff member
2/4	Single username and password
2/5	Security model: VC, UC, VS, US, AC
2/6	Total hours @ bottom of staff roster
2/7	Permanent staff hour to be noted on the roster and updated daily so that we can report an overtime hours worked etc. Need to be able to report on permanent hours as well as casual hours.
2/8	Staff daily sign in and sign out: staff member able to acknowledge they have worked those particular hours each day, daily roster print out which individual staff check and initial
2/9	Permanent staff salaries only incl. in time of reporting – not on rosters etc
2/10	Overtime is shared 50/50
2/11	Staff by name as well as ID number

Following the XP practice of "hardest things first" I decided to begin with story 2/5: the new security model. I investigated using the J2EE [Sun 2001] model, but it involved setting up complex configurations and I decided the most simple approach would be a bit-set of security rights (`RightSet`) assigned to each staff member for each centre they work at, and a single `SecurityBean` that could be held in the session context and supply access information to the JSPs as they are opened. The implementation of these classes was straightforward, but I had not counted on the length of time it would take to insert security-checking code at the top of each JSP that I had created.

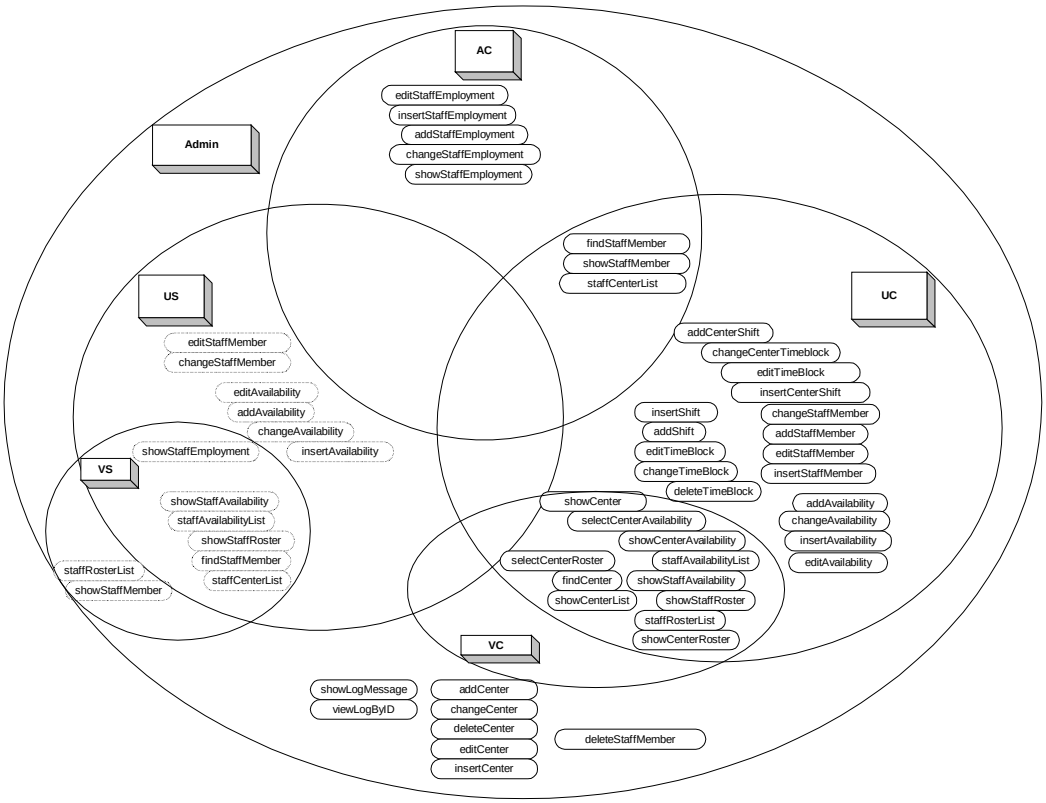
During the planning game, the customers and I had divided the security into five standard types and a special super-user "admin" type. The five standard types were to be:

- View Staff (VS) – view details relating to the logged on user only
- Update Staff (US) – update details relating to the logged on user only

- View Centre (VC) – view details relating to any staff at the centre the logged on user is employed at
- Update Centre (UC) – update details relating to any staff at the centre the logged on user is employed at
- Administrate Centre (AC) – use special functions relating to the centre the logged on user is employed at

Because the bit-set is attached to the employment entity, a different set of rights can be added to each staff member at each centre they work at. The special admin role is associated with the staff members themselves.

I then went through the existing JSPs and decided what rights should be necessary to use each one. The diagram below shows the possible domains. Pages in dotted lines refer to pages that can only be used to view or update information on the logged on user, not any other staff member.



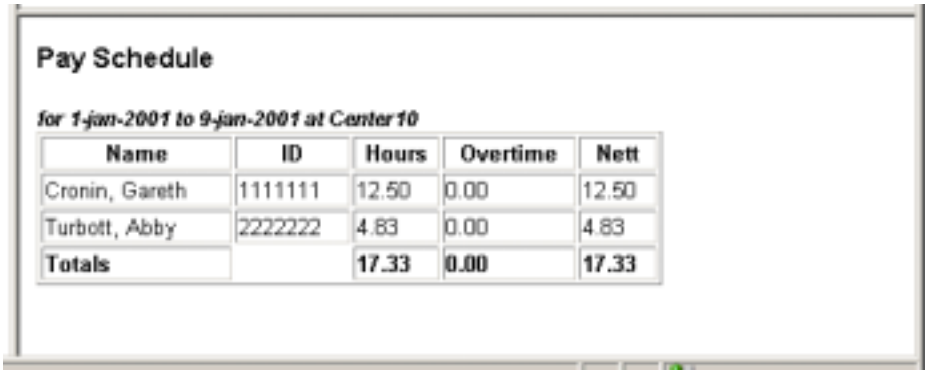
As I outline below, the security implementation took twice as long as my estimate, mainly due to having to design a new testing process for the security.

The next most difficult set of tasks was the set that arose from story 1/14 (which was merged with 1/11 and 1/36). We had merged these stories as they covered the selection and generation of a pay schedule and a reporting facility to show a breakdown of the time within that pay schedule.

The pay schedule is an alphabetical listing of all staff members for a particular centre or centres along with the number of ordinary hours they worked, the number of overtime hours, and the net total when the overtime hours are multiplied by 1.5.

I decided to build several layers of abstraction, as the story specified that reporting should be able to be by day, fortnight or month and for any set of centres. To get this sort of flexibility it would be convenient to have a class that represented a staff member's day at a centre, a class that represented the totals at one centre, and a class for the totals over any number of centres.

It was during this section of implementation that I realised the value of pair programming in non-solo XP programming. In a piece of code where the program iterated over the days within a pay period, I neglected to be wary of Java's system of always passing variables by reference. It took me 4 hours – nearly an entire point – to locate the cause of my unit test failures. If I had a partner programmer reviewing my code, I am sure they would have noticed this error straight away. It was only because I was buried so deep in the details of the code that I was unable to "stand back" and see it.



Pay Schedule

for 1-jan-2001 to 9-jan-2001 at Center10

Name	ID	Hours	Overtime	Nett
Cronin, Gareth	1111111	12.50	0.00	12.50
Turbott, Abby	2222222	4.83	0.00	4.83
Totals		**17.33**	**0.00**	**17.33**

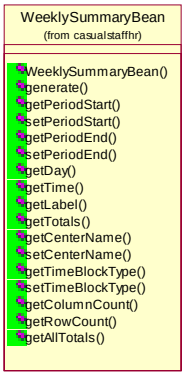
Story 1/23 suggested that the pay schedule should be in the same format as the manual schedule that is delivered to the Human Resources department at the end of each pay cycle. I had been following this format anyway, so the schedule was already in this format.

Story 1/9 called for the ability to remove permanent staff hours from a roster. This was achieved with a checkbox on the `selectStaffRoster` and `selectCenterRoster` JSPs and appropriate code to enable or disable the hours in the `CenterRosterBean` and `RosterBean`.

A business rule for all casual staff pay is that any shift of 5.5 hours or greater in length must include a half hour break. Implementing story 1/31 involved adding a check into the `PayDayBean` to force the break time to 30 minutes if no break of 30 minutes existed already and the shift was of this duration.

1/32 meant adding a new property to `WorkTimeBlockBean` (this is a class introduced in refactoring explained later) and corresponding fields in the `addShift`/`addCenterShift` and `editShift/editCenterShift` JSPs. It was at this point that I decided that perhaps the centre and staff versions of these JSPs could be refactored into one JSP each.

1/35 calls for a weekly "overview" of the roster. I had suggested in the planning game that this could be a grid that displayed the number of workers assigned to each block of time during each day over some arbitrary time period. A click on the column header could then take the user to the centre view of the roster for the corresponding day. This resulted in a new `WeeklySummaryBean` class and a couple of JSPs.



1/38 was another simple modification to the `WorkTimeBlockBean`. The customers had suggested that a comments field would be useful to annotate a shift or availability with free text. Here again I found myself repeating modifications as I altered the centre and staff versions of the maintenance pages for shifts and availabilities.

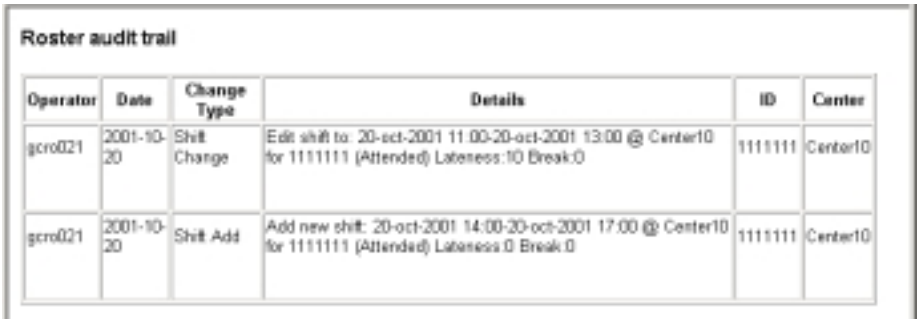
1/40 called for associating a break time with each shift, this had been covered in story 1/6. 1/39 was just the ability to edit the roster at any time; this was also an "already done" story.

1/42 called for the ability to copy a roster or availability schedule from one date/time range to another. I implemented a `copy()` method in `ScheduleBean` that takes the start time of a new date range as its parameter and creates new `WorkTimeblockBean` objects for the new date range. A JSP with selection buttons for whether to copy shifts or availabilities and fields for the date completed this functionality.

2/1 was a simple matter of changing the column headers on the roster displays from full day names to abbreviations. This was to ensure that the grid would fit into the browser window as easily as possible.

2/2 was a request for user auditing for changes made to rosters. I created a persistent bean called `Audit` and added code to the processing pages for shifts to add records containing the username used

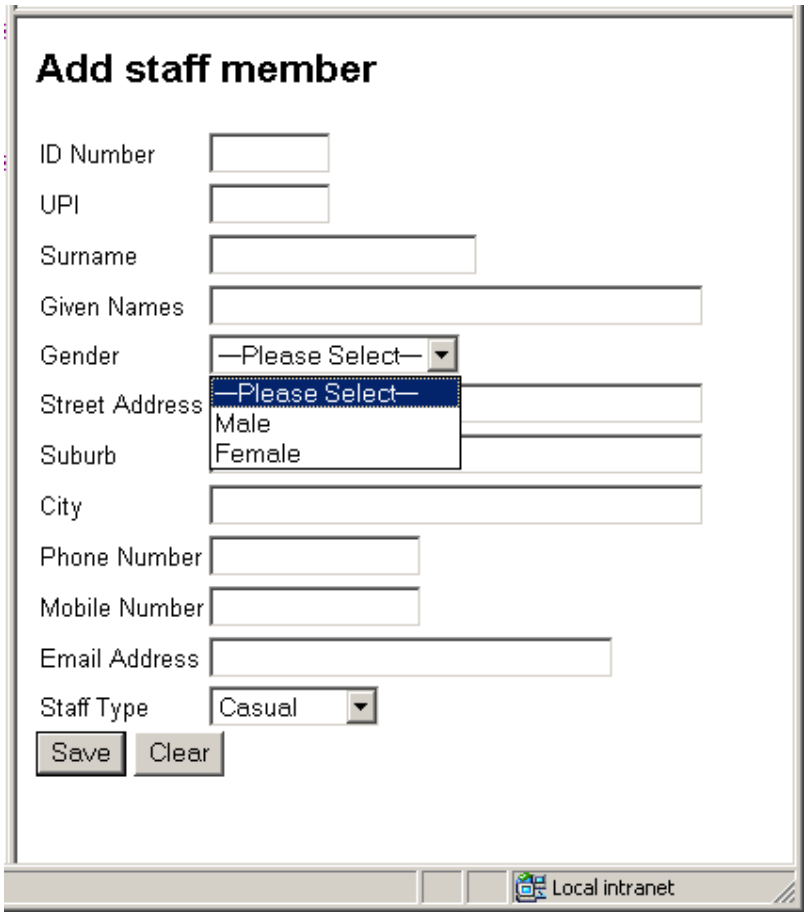
to make changes and what those changes are. I then added a link to the roster display to show audit information relevant to the schedule being viewed.



Roster audit trail

Operator	Date	Change Type	Details	ID	Center
gcro021	2001-10-20	Shift Change	Edit shift to: 20-oct-2001 11:00-20-oct-2001 13:00 @ Center10 for 1111111 (Attended) Lateness:10 Break:0	1111111	Center10
gcro021	2001-10-20	Shift Add	Add new shift: 20-oct-2001 14:00-20-oct-2001 17:00 @ Center10 for 1111111 (Attended) Lateness:0 Break:0	1111111	Center10

2/3 addressed a potential problem with a default gender for new staff members being entered. I placed a combo box with the two genders on the JSP for adding new staff, and the box was defaulting to "female". The customers felt that this would lead to people accidentally entering staff as "female" when they were not. I added a third category of "—please select—" and made this the default selection. I then added checking to the insertion processing JSP to reject a gender of this type with an error message.



2/4 was a temporary security solution that was a simple server configuration change to prompt for a single username and password to gain access to the system until the new security model was being developed. Live data was being entered into the test system in the meantime and needed to be secured in some way to conform to New Zealand's privacy laws.

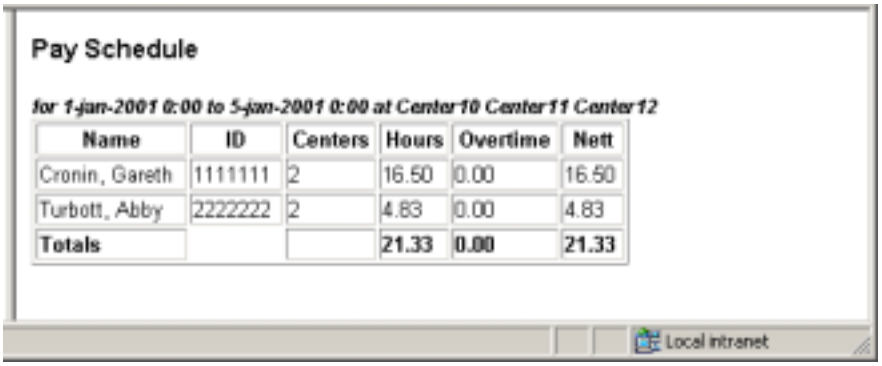
2/6 was the addition of a total number of hours to the bottom of the staff roster display.

2/7 requested the provision of discrimination between permanent and casual staff hours for reporting purposes later. The staff type was already part of the `StaffMemberBean` so I only needed to add controls for setting and changing the type to the staff JSPs. Casual hours were not be included in the pay schedule reports, but would sometimes be needed in other reports. I added a parameter to the reporting beans to select whether or not permanent hours were to be included.

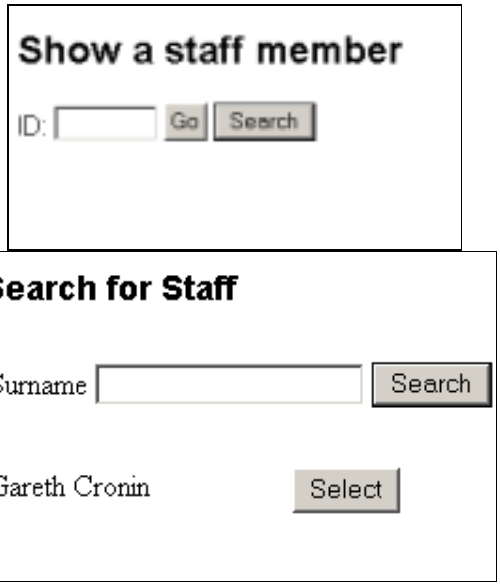
2/8 called for the daily centre roster to be printable in such a way that staff could initial their shifts to indicate that they had attended them. This required no extra development.

2/9 requested that permanent staff salaries were not recorded for the staff. This was just a matter of allowing a staff member to be assigned to a centre without a pay rate.

2/10 asked for the splitting of overtime worked on a 50/50 basis between the two centres. I added a centre-count column to the pay schedule to show whether a staff member had worked at one or two centres so this splitting could be calculated manually based on this.



2/11 requested the ability to search for staff by name rather than by ID number. I suggested that the best way to do this would be a pop-up search window that showed a list of staff that matched some given name and then closed and returned the ID number to the existing form. The customers were happy with this. I had not used pop-ups and the necessary JavaScript before, so I performed a "spike" to see how difficult this value passing between browser windows would be. It turned out to be straightforward, so I created a pop-up with a "fuzzy" search by surname. Running a search provides a list of matching staff with a button beside each name. The button closes the pop-up window and sets the value of the "parent" ID field to the corresponding ID number.



Task Tracking

In hindsight, it appears that I encouraged the customers into a far more complex security model than was necessary at this stage of development. I feel that this demonstrates how difficult it is to keep to this XP principle of simplicity. Because I started with the security, I could have pulled out of its development and reviewed it with my customers, but I decided that any delay would be short in any case.

Overall I had taken on far more stories than I should have in a single iteration. My underestimation in the first iteration had led me with much enthusiasm into overestimation in this one. I had also failed to consider the time I would need to spend on functional testing when estimating each individual story.

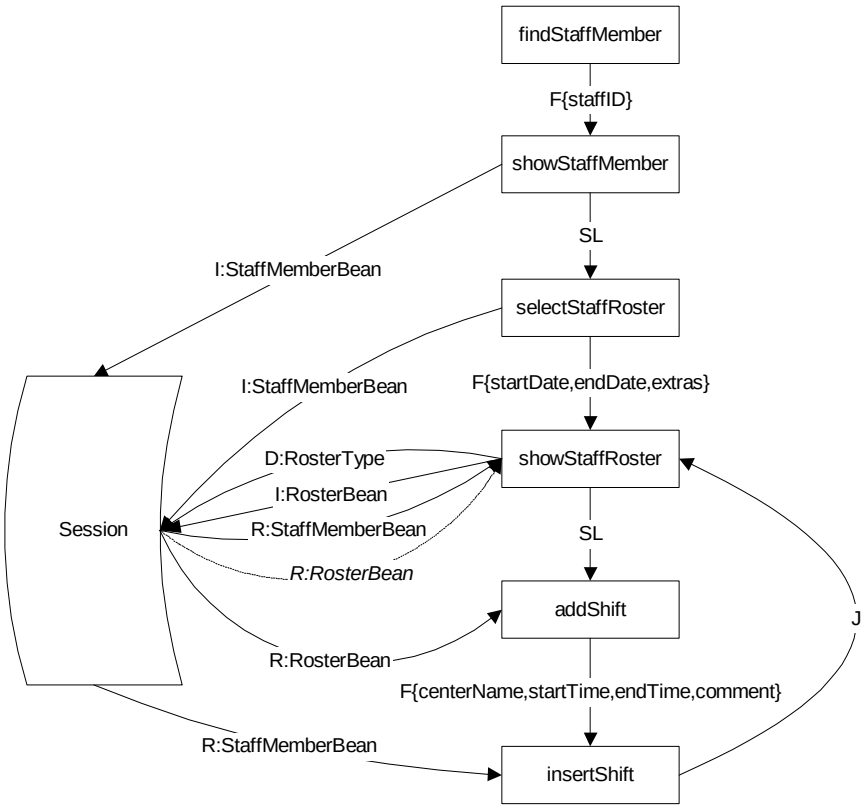
Refactoring

After the various bouts of refactoring in this iteration, I came to the conclusion that at least three or four points of development time should be set aside for refactoring when the implementation of the stories is finished. When the distraction of coding new developments is removed, it is much easier to review code and look for simplifications. This may be a consequence of not utilising pair programming – the reviewing that I felt was necessary at the end of implementation might have been carried out during implementation had I had a programming partner.

The major refactoring that was needed at the end of this iteration was in the JSPs. As my experience with writing JSPs grew I had changed the way that I passed object references between JSPs. I had been alternately switching between using object references held in the session context and using URL parameters. My coding standard was diminishing and I was duplicating code to compensate for these inconsistencies.

The best example of this is the add-shift cycle for staff rosters. Before refactoring, the design worked as in the diagram below. The table shows the abbreviations I have used in the notation.

Abbreviation	Meaning
I:	Instantiate and store in session
R:	Retrieve from session
D:	Store attribute in session directly
SL:	Static link
SL{p1,p2..}:	Static link with parameters
DL{p1,p2..}:	Dynamic link with parameters
F{p1,p2..}:	Form with parameters
J:	Jump (HTTP forward)

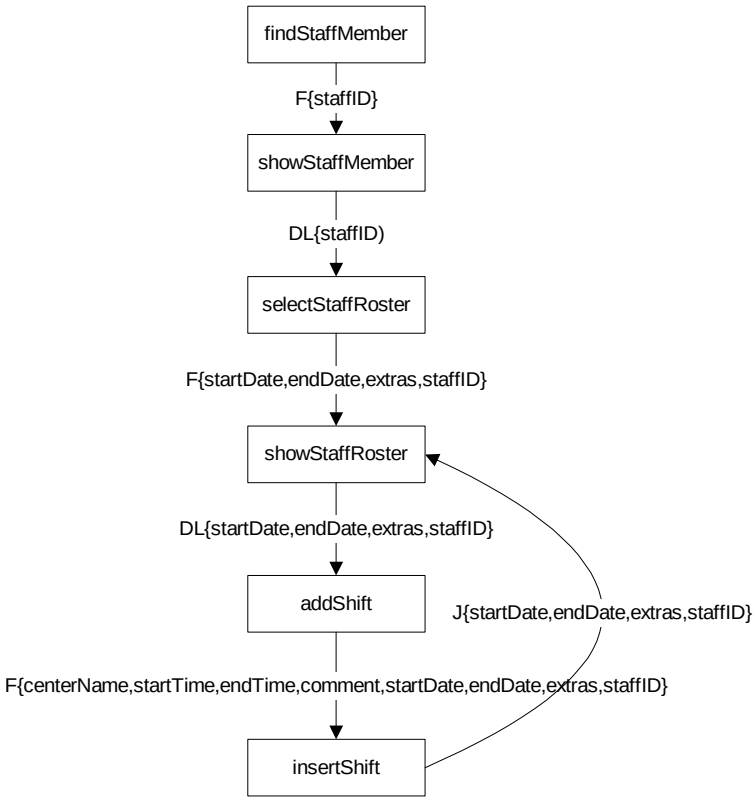


This system has the advantage of only having to instantiate each object once during the entire process. For example, the security checks at each page need to know which centres the staff member being viewed works at. Because the staff member being viewed is stored in the session, the object for the staff member can be retrieved without re-instantiation by `showStaffRoster,insertShift,` and `showStaffRoster.` There is also the advantage in the jump back to `showStaffRoster` where the roster does not need to be re-generated to be displayed.

However, these advantages must be balanced against the problems of navigation dependence. If a session times out while `showStaffRoster` is being viewed, navigating to `addShift` will raise a null pointer exception when an attempt to retrieve `RosterBean` from the session context is made. Binding a page in this way also makes it less reusable and more difficult to test. If pages depend on prior pages being visited, pages cannot be tested for their security individually. This means that the only thorough way of testing the security of dependent pages is to visit every permutation of all other pages first – this is clearly undesirable.

I decided to go through the JSPs and change all the navigation paths to a system of propagating identifying properties through URL parameters and re-instantiating objects where necessary. This is in effect a combination of URL re-writing and hidden fields to provide persistence rather than the built-in JSP session awareness, but security is still provided using the JSP session object.

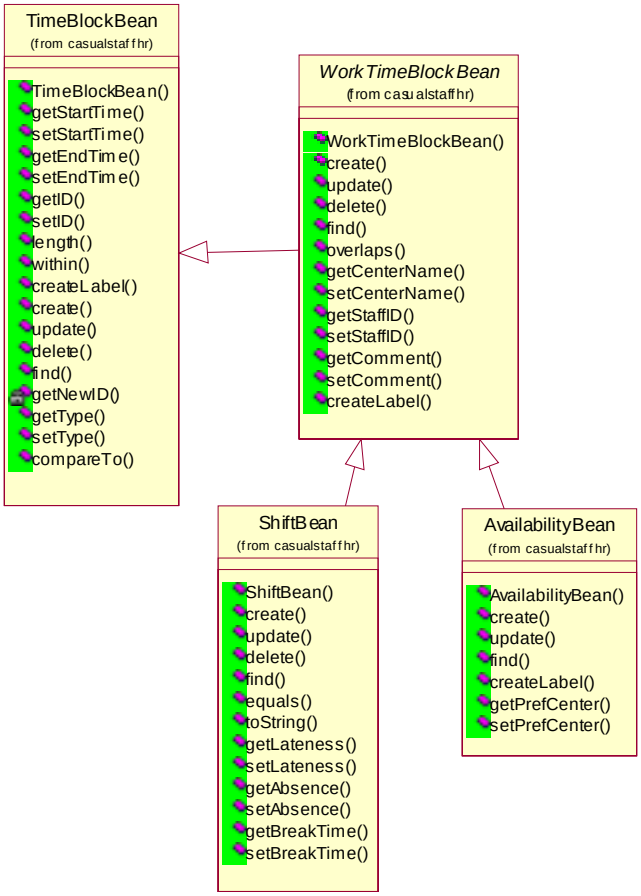
When I analysed the number of extra database accesses and objects that would be created, I noticed places where I had been using object properties rather than request parameters, even though parameters were available. By removing these I actually reduced the database activity. The final result produced only a tiny increase in I/O and memory use, gave me a lot more confidence in the robustness of the JSPs and had a simpler overall structure as can be seen in the following diagram.



I had effectively moved the storage of state from the server to the client. Even if the server was rebooted mid-session, the client could continue (after being prompted for a login and authenticating) as if nothing had happened. It could be argued that server-based state is advantageous if the client crashes, in that the session could be continued after a client restart, but this sort of session recovery requires complex IP matching (which is also potentially insecure) or a secure socket implementation that is beyond the requirements of this project.

Also during this process I found that I had separate JSPs for processing the database side of shift addition, deletion and updating for a staff member and for a centre. I was able to condense these into a single JSP for each operation.

I decided that with the JSP code becoming more controlled, it would be easy enough to refactor the `ShiftBean`, `AvailabilityBean`, and `TimeBlockBean.` As I mentioned in the first iteration, I had made `AvailabilityBean` a subclass of `ShiftBean`. Although they shared some functionality, other parts they did not, so I introduced a new abstract subclass of `TimeBlockBean`: `WorkTimeBlockBean`. I was able to move three properties out of `ShiftBean` for common use with `AvailabilityBean`, and leave three properties in `ShiftBean` that were not relevant to `AvailabilityBean`.



As mentioned in the implementation section, I was not satisfied with the way that the `addShift` and `addCenterShift` JSPs were performing almost exactly the same function. I looked into merging them, but I realised that the security checking was quite different for each JSP. The `addShift` JSP will allow a staff member with the US security right to use the page for their own record, whereas the `addCenterShift` JSP will only allow a staff member with the UC right for that particular centre to use it. Some sort of conditional flow would be required to check which security check should be applied. This did not seem too difficult, but it would also need to be applied to other pages such as `editShift`, which was in a similar situation. I realised that the sensible thing to do would be to refactor the actual security checking, placing each type of security check it its own file and including them in the head of the JSPs with a combination of conditionals and simple include directives. I decided to defer this refactoring until the commencement of iteration three.

Testing Issues

Because this part of the development concerned reports, the code contained a lot of summing, averaging and summing again. These types of methods are obvious candidates for unit testing, but I realised that I had to make methods public to test them, because I had put my unit tests in a separate package to the main classes, meaning that protected methods were not visible. I realised that in an object oriented language with visibility rules, unit tests do need to be placed in the same package, so I moved all the testing classes into the main package.

As the system became larger, I had to be more careful with encapsulating unit tests. Unit tests should not use domain objects; they should instead use only the classes that the class being tested uses. If tests stray into domain objects the tests become non-reusable and make re-factoring more difficult. An

overly interdependent test must be entirely re-written if the class being tested is generalised, this can be avoided if the testing is kept at a more atomic level [Canna 2001].

I became concerned with the rigidity of my "capture-and-compare" method of web testing as I enhanced the JSPs from the first iteration. There was barely a single page that had not been changed in some way, rendering the test conditions from the first iteration useless. Capturing the pages is a tedious and time-consuming activity, and I felt that I needed a better approach to the web testing. I was reluctant to use black box testing because of the advantages of a more visible method that I outline in the previous iteration.

A New Method of Testing

Developing the security model led me to the point where I wanted to test the correctness of the permissions attached to each JSP file. This is really unit testing of the JSPs themselves. My web robot was not really suited for the task because the test's sole function was to assert that an access violation had been detected, or assert that it had not under each test condition. I decided to turn to HttpUnit [HttpUnit 2001], the sister-software of JUnit mentioned in the previous iteration.

HttpUnit "black-box" tests a web server, making HTTP requests and responses. Methods are provided to examine these requests and responses, including methods for retrieving header fields. HTTP header fields are a way of extending responses by including custom name-value pairs in the header of the application protocol messages. The JSP I had built to inform the user that they do not have permission to access a particular page contained code to insert a header with the key "VIOLATION" in the HTTP. All I needed to do was create a system of tests that changed the permissions of the user currently logged in and make attempts on each of the possible pages, checking for the appropriate response each time.

I had already unit-tested the class that held the set of permissions for a user, so I was confident that setting and clearing bits produced the correct permissions without any unwanted side effects. I needed to set the permission for each security domain and test that domain, one page at a time.

The security model gives rise to eight separate domains if the intersections of access sets are extracted. I first attempted to write tests in Java code in the same manner as JUnit tests. This soon became a "copy-and-paste" process, so I wrote a small scripting language and a front end to perform the tests. The language provides commands to set and clear security rights and to check whether an access violation or a standard HTTP error has occurred.

I decided to drop the Excel Web Robot and expand my HTTP Black Box to perform all the functions that the Web Robot already provided. But, instead of capturing entire pages and comparing them with actual pages during testing, I would instead check titles of pages and test for the existence of specific pieces of HTML or text. In this way minor changes to JSPs would not render previously designed test cases inoperable. HttpUnit provides methods for examining the text in tables by cell reference. Because I used tables heavily in the design of the JSPs, almost all displayed data was contained in a table of some sort, and so examining these with HttpUnit was straightforward. I added commands to the scripting language for:

- Checking for error codes
- Asserting the existence or non-existence of particular HTML strings
- Asserting the existence or non-existence of particular text in tables
- Filling in and submitting forms

This is explained in more detail in **Appendix B – HTTP Black Box (p.62).**

Communication

As I prepared for the release and a new planning game, it became apparent that the business had made few attempts to use the first release. On questioning, the customers apologised, but stated that they had not had time to use it. This is unfortunate, as it meant I was not getting the rapid feedback I needed on the initial functionality. There seem to be two reasons for this lack of testing of the embryonic release.

1. The customers did not have a lot of "spare" time to devote to testing. Testing of the system was regarded as low priority compared to the day-to-day tasks of running the business. I had failed to communicate the importance of feedback from the customers to the developers.
2. The system only provided part of the functionality that the customers required. Although it would enable them to enter a roster and view it, they felt that it was easier to continue to use their existing manual system, rather than taking the time to learn a whole new system when there was no guarantee that the system was going to deliver what they needed. In this respect I had failed to emphasise that in fact the system *was* capable of doing that part of its job thanks to the correct prioritisation of the first-release features.

I believe that with some time spent on training the users in how to use the first release, this situation could have been avoided. This meant that in future releases, some more of the available engineering time would have to be devoted to training rather than development.

During her testing of this release, Elspet expressed irritation at having to handwrite the story cards. She felt that she would prefer to type the stories. This is not really feasible at the planning game, but in between games, she wanted to be able to enter the stories directly onto a computer. One of the best things about the cards, as opposed to electronic versions, is that the physical manipulation of them lends concreteness to the prioritising process in the planning game. Piles of cards make much more sense than the equivalent 'virtual' representations. I decided that it would be fine if she was to type into a template and then print the result out to create the "between game" stories, so I deployed a link to the cards as an administrative function on the actual system.

Release

With an explanation of the importance of testing the release from Elspet, Deborah started testing in earnest when the second release was made available. She entered a full day's roster and used the staff maintenance functions.

This raised some interesting issues. In our discussions during the planning game and in the stories themselves, no one had ever specified in detail how they would like to navigate through the system. This was left up to me, and I imposed my model of navigation. Deborah had some difficulty understanding my view of the data entry process; again some training would circumvent this. A defect was discovered immediately, where a roster was only ever displaying times from 8 a.m. to 5 p.m. On revision, this turned out to be hard-coded into the JSPs. I had added this hard-coding for convenience in the first release. The functional testing elicitation from the customers had not been thorough enough on my part to ensure that shifts outside these times were attempted.

The application files for this release can be downloaded from:
http://sourceforge.net/project/showfiles.php?group_id=28900&release_id=49976

The source code is available from the CVS repository cvs.casualstaffhr.sourceforge.net tagged as 3.0

Iteration Three

This iteration took place between the 30th August 2001 and the 20th September 2001

The Planning Game

We began the planning game with a discussion to attempt to clarify some business rules to do with reporting of statistical information. Emma-Jane and Amber wanted a report showing the amount to be paid to each staff member from each centre (either the Call Centre or the Student Information Centre). Elspet wanted a report that summarised the amount to be paid to each staff member as a total from both centres. Emma-Jane and Amber did not realise that the Human Resources Payroll Department did not distinguish between the individual centres, and that the amounts were actually debited from a single cost centre. In the manual system, Elspet was ignoring the breakdown and simply taking the totals to submit to HR. However, Emma-Jane and Amber needed the breakdown as a report to their manager. The per-centre totals were used by their group internally, recording the amounts in journals.

This is a classic example of the customers not actually knowing what they want from the system, and utilising an outsider's (my) perspective to have their business processes "held up to them" for examination. Thanks to their involvement in the requirements elicitation, they were experiencing an added advantage by participating in the functional decomposition of their processes, not just with a business analyst, but instead with someone who was trying to decompose them from a programmer's view.

This meant we could review the system of sharing staff overtime that they had been using. They had been assigning any overtime worked by staff that worked at both centres by "paying" for half of the time each. The reason for this crude calculation was that a monthly calculation of ratios worked at each centre by each staff member was a considerable undertaking with the existing manual system of recording the time. I suggested that we could now split the overtime fairly between the centres by calculating the ratio of time spent at each centre during the pay periods concerned. This led to a long discussion between Elspet and the other two as to an appropriate way to go about this. After some quarter of an hour we settled on the ratio calculation.

At the previous game, we had thought that we would be looking at the automatic rostering by this stage, but given the lack of use the last release had received, the customers decided that this release should concentrate on improving the interface and reports. There were two main reasons for this:

- To move from the manual system to the new system, the customers wished to have an interface they could use effectively
- The roster had already been manually created up to November, so the automatic rostering could be deferred with no negative effects on the business

Bearing in mind the gross overestimate of my velocity from the previous iteration, I decided that this time I could only manage 15 points. We gathered together the priority 1 stories and found that we had 10 points. Since the customers had not spent much time working with the releases, I suggested that the remaining 5 points could be filled during the development as the customers came across user-interface issues that they wished to enhance or add. They liked the idea of this, so 5 points was set aside for stories that arose in the next few weeks.

Implementation

Ten stories were originally selected for this iteration, the last two stories were added as part of the reserved 5 points.

Number	Description
1/17	Report overtime entitlement by person, by cost centre, total. By day, month, year, fortnight.
1/3	Block time per person when they're not available to work
3/1	Inactivate/delete staff member, where staff member is deactivated future shifts need to be deleted
3/2	"Are you sure now" before the staff member can be deleted
3/3	Work and home phone number

3/4	Date format – combo boxes rather than free text
3/5	Only allow shifts to be entered when ID number is valid
3/6	Daily backup
3/7	Menu layout – cascading
3/8	Edit an ID number (for typos)
3/9	Automatically split a shift into 2 parts when it is added to specify break time
3/10	Address format should follow NZ Post standard (address line 1-4 and so on)

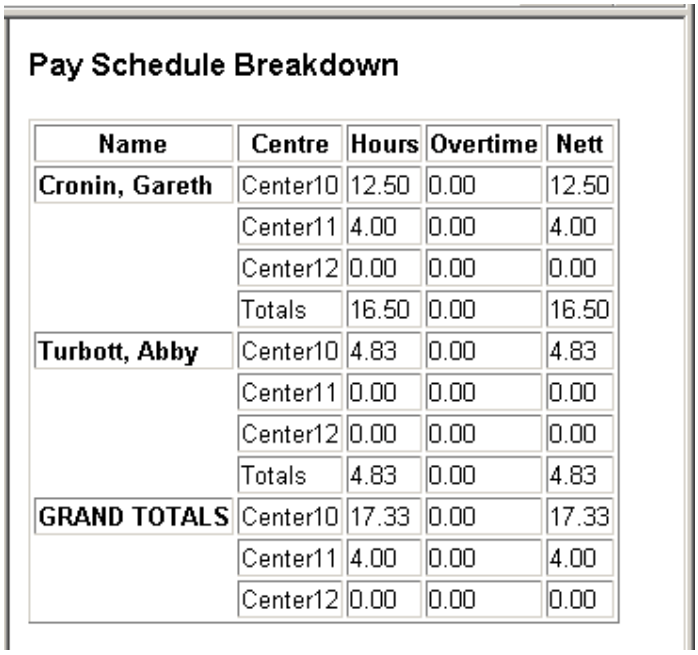
I began implementation with 1/17, as this carried the highest estimation of time and risk. The story required a more detailed pay schedule than the original pay schedule from the previous iteration. In the earlier one, I totalled the hours worked by each staff member during a pay period, regardless of which centre the hours were worked. I now needed to split these into a total for each staff member, for each centre. The earlier structure consisted of a hash table for each total into which the total keyed by the staff ID was entered. There was a hash table for the raw total, one for the net total, one for the break time total etc.

Key	Value
1111111	100
2222222	125
3333333	120

I decided to create a two-level hash system, where each staff ID would key another hash table, rather than just a total. The second hash table would provide an entry for each centre worked at. In this way it was a simple matter to provide either individual totals or split totals.

Key	Value	
1111111	*Key*	*Value*
	Center1	50
	Center2	50
2222222	*Key*	*Value*
	Center1	125
	Center2	0
3333333	*Key*	*Value*
	Center1	70
	Center2	50

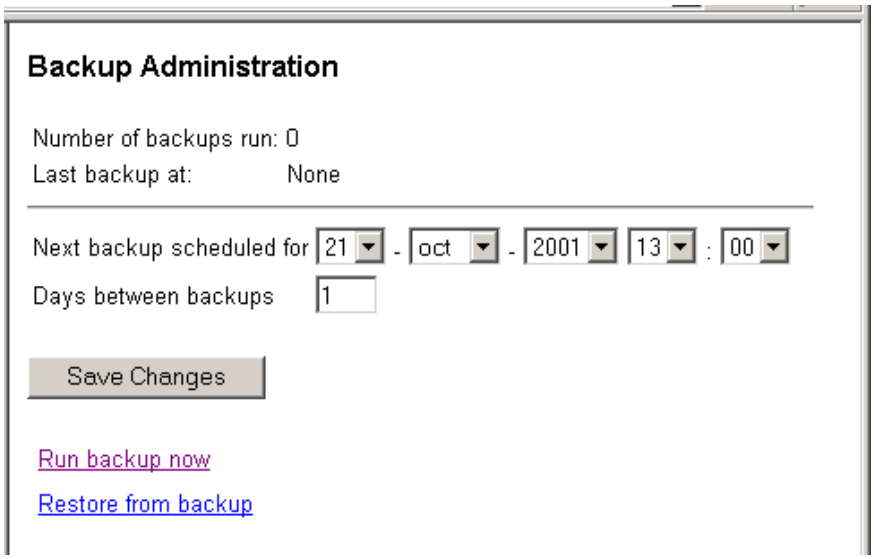
This produced the following schedule.



Pay Schedule Breakdown

Name	Centre	Hours	Overtime	Nett
Cronin, Gareth	Center10	12.50	0.00	12.50
	Center11	4.00	0.00	4.00
	Center12	0.00	0.00	0.00
	Totals	16.50	0.00	16.50
Turbott, Abby	Center10	4.83	0.00	4.83
	Center11	0.00	0.00	0.00
	Center12	0.00	0.00	0.00
	Totals	4.83	0.00	4.83
GRAND TOTALS	Center10	17.33	0.00	17.33
	Center11	4.00	0.00	4.00
	Center12	0.00	0.00	0.00

I had estimated a medium level of risk associated with the daily backup system as specified in story 3/6. The customers wished to have an automated daily backup of the data, with a method of restoring from that backup. I decided to create a backup procedure where the contents of each database table is dumped into a flat file. This allows for very simple backup, checking and restore. I created an administration JSP that allows the user to begin a manual backup and specify a time and interval for future backups. Starting the first backup spawns a background thread that waits until the appropriate time and runs the backup again. I provided a restore facility through another JSP.



Story 3/5 was a request for validation of the entry of a new shift or availability for a staff member. If the staff member did not exist or was not assigned to the centre to which the shift was being added, the customers wanted an error message generated. This raises an interesting point as far as how persistency is implemented and controlled. Several times through the development of this system, I questioned whether perhaps I should have made use of a generic persistency framework such as “container managed” J2EE Entity Beans, or an object-relational bridge. I went so far as to implement my own test framework using reflection and bean introspection. However, to implement this check for a valid shift, I simply added a call in the `WorkTimeBlockBean` to the `findActiveAt()` method of the `StaffEmploymentBean` . This method executes a SQL query that makes use of a complex

parenthesised expression to efficiently find out whether a staff member has an active employment at a particular centre. If I had used a generic persistence-manager, this sort of efficiency through placing the burden of calculation on the relational database itself would not have been possible. The objects would have needed to be de-serialized and iterated over at a substantially higher computational cost.

3/8 was a concern that Elspet had that a staff member might be entered with an incorrect ID number, and for the error to remain for some time before being discovered, at which point it would need correction. I suggested a "find and replace" system that could replace any instances of one ID in each of the database tables with a new ID. This was simple to implement, but did require some thorough unit testing to ensure that other data would not be damaged with such a potentially destructive action. I used SQL queries again here, as a one line query processed on the back end is far more efficient than the equivalent instancing and updating that a purely object based method would require.

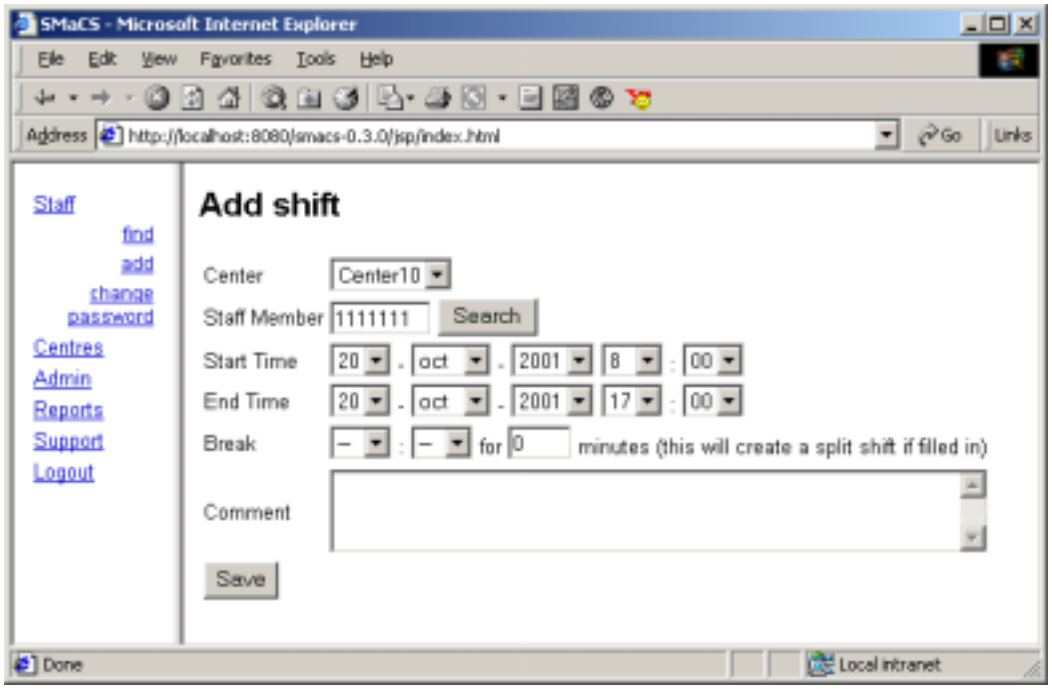
3/7 was a usability problem. My three-frame navigation system was unpopular for a number of reasons:

- Having to click on a side menu item and then move to a top menu item required constant large movements for the eyes and for the mouse hand
- Because the top menu defaulted to the staff sub-menu items, clicking on the staff side menu item at application start-up produced no visible effect, leading the user to think they had "done something wrong"

I changed to a system of "cascading" menus. I removed the top frame, and instead had a separate page for each main menu item, with the sub-items appearing below that item. The two examples below show the expanded "Admin" menu and the expanded "Staff" menu.

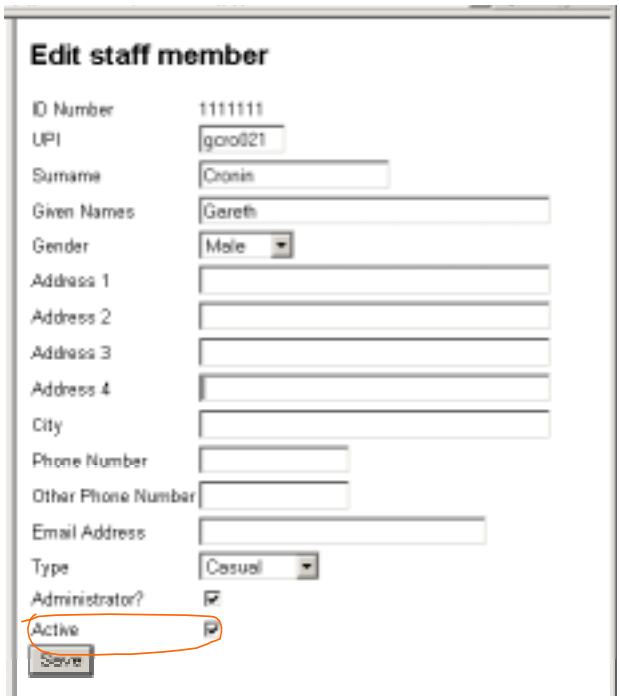
1/3 refers to the need for a staff member to indicate time they are "definitely not available for" as opposed to time where they could be asked to work if necessary. I made use of the database field in the TimeBlock table that I had been using to indicate whether a shift was attended to store the type of availability, represented with the string "yes" or "no".

3/9 was a solution to the awkwardness of adding a shift with a lunch break. As the system was implemented, it was necessary to add two separate shifts in two separate operations. I resolved this by adding new fields to the relevant JSP. The new fields provided space for entry of a break start time and break duration. The system inserts two shifts separated by the nominated break duration, beginning at the given time. The following screen-shot shows the cascading menus and the "add shift" page with the new fields.



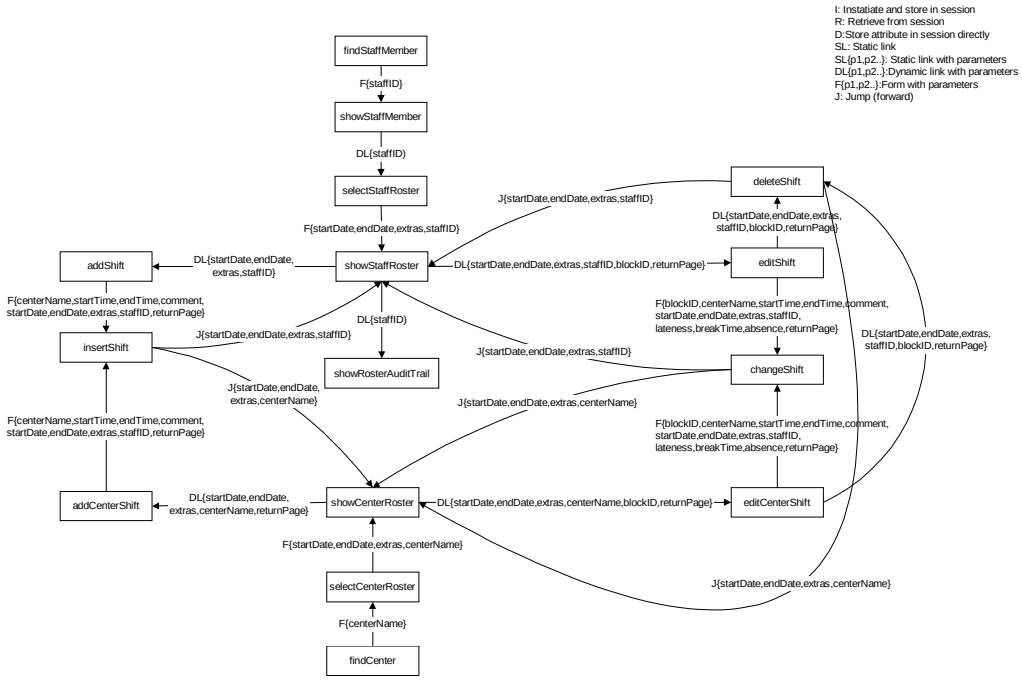
3/10 was a simple change to the address format to conform to the New Zealand standard of four address lines plus city and country.

3/1 was implemented to make data easier to handle in the environment at the centres where staff turnover is fairly high. Although the system already provided functionality to store start and end dates for contracts, staff members without a current contract would still appear in the search dialog when an appropriate search was executed. We decided to create an independent way of marking a staff member active or inactive. This consisted of an “active” checkbox on the staff details JSPs and a corresponding column in the staff database table. I added an “include inactive” checkbox to the staff search dialog to indicate whether or not inactive staff should be included in the search.

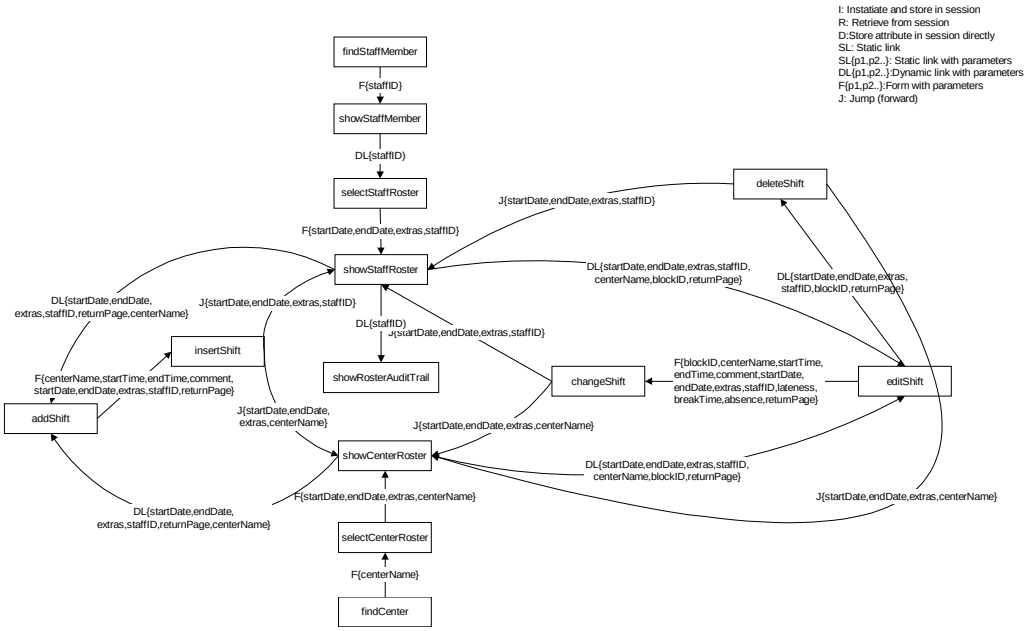


Refactoring

The first refactoring I performed was to remove the redundancy in the maintenance cycle for adding shifts and availabilities. I had two types of pages for each action of each type of time-block. One set of pages was for creating or altering blocks for staff, and the other set was for doing the same thing for centres. The only difference between the pages was the absence of a staff id field on the staff version (as it was assumed that this page had been navigated to from a staff roster view) and the absence of a centre name field on its counterpart. The following navigation diagram shows the original system.

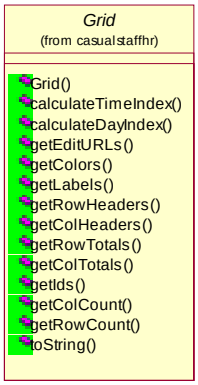


The following diagram shows the system after the refactoring, with two less JSPs. Because the same pattern is used in the availabilities, this change resulted in a total reduction of four JSPs. Having a single version of each of these pages makes the code far simpler to maintain.



At the time I was preparing the new detailed pay report for story 1/17, I realised that almost every display in the application was using a grid of some sort and that I was repeating code for column

headers, row headers, background colours for cells and various other parts of a HTML table-based grid display. I created an abstract Grid class that any display component can inherit from. The class has properties for the basic items such as the column and row counts and the headers, and also array properties to hold the text, colour and URL (if the cell is editable) for each cell.



I then built a “chunk” of JSP that displays the appropriate property values in HTML. The “chunk” can then be included in a JSP along with the necessary code to instantiate a concrete grid sub-class, dramatically reducing the amount of code necessary in each grid-based JSP and avoiding code duplication.

Functional Testing

Prior to this iteration, I had made a time to spend with each customer to help them write functional tests on test cards that I attached to each of the stories that they had written. This meant that I was guiding the test writing more that I would like to. If the customers are allowed to write their own tests independently, then those tests will be a true to test of the requirements specified in the story because they are unaffected by the developer’s view of the system. I decided that the customers knew enough about writing test descriptions to do so independently.

Rather than have the customers fill out the test cards, I deployed a set of JSPs on the production system to allow them to enter tests in the “description/goal” format that I had been using on the cards. I copied the details from each story for reference, and made fields available to add as many rows of tests as they liked to each story. From this information I was able to write the test scripts to be run by the black box testing application. The advantage of having these online was in being able to instantly read the stories written by the customers located away from my building.

Communication

I received an email and a phone call from managers in other departments of the University inquiring as to the progress of my system and when they would be able to “have a look at it”. My employer had been “selling” the system to other departments in an effort to increase the perceived value of the development in the eyes of other parts of the University management. Although this is a natural part of business, it presented some interesting problems. The nature of XP is that I was working to one small group’s requirements. I was now wondering how to handle disparate groups of customers when I was pressured to meet their requirements as well. Having four customers rather than just one already had its difficulties, so the question was whether to develop with a new group of customers while ensuring that the acceptance tests for the original group were still met, or whether to split the product into a new version for each disparate group. At the time this report was completed, I had not resolved this situation.

Around the time for the next planning game, I set up hour-long appointments with each customer to take them through the features of the system so far and explain how to use them. This made it easier for the customers to test the features, and also instilled enthusiasm for the system when they were made aware of the potential for saving time that the report generation could provide.

Usability

Following my concerns about the lack of attention we had given to the user interface when writing stories at the planning game, I decided to try a usability/story-writing experiment. I carried out a "thinking aloud" session with Emma-Jane, which also doubled as a training session. During "thinking aloud" the user carries out tasks while vocalising what they think they are doing and why at every step of the operations they perform [Nielsen 1992]. I took down a rough transcript, and questioned Emma-Jane further when problems arose.

We started with the simple task of finding and showing staff details. My rough transcript follows:

The "click here to login" is obvious
I enter my UPI and password and click "login"
I go to the staff menu – obvious starting point
I click find
I don't know the ID so I try the search button
The search asks for a surname, so I enter it
I try to use lowercase, but I get an error that says the first letter has to be uppercase, I click the back button and try that
I don't know what the "include inactive" checkbox means
I click "select" by the person's name
I click go

The only problem with this task was Emma-Jane's question about the "include inactive" checkbox's function. I asked whether she was accustomed to using "tool-tips" at all, and suggested that a tool-tip might be useful, she agreed that it would be, so I asked her to write a story for that feature.

We then tackled creating a roster from the staff view:

Found the staff member as before
Clicked the "rosters" link
Entered the start date for the roster
Very confused by the start and end times attached to the dates – what is their function?
Found "Add Shift" without a problem
Set start and end times for shift and submitted
Got error message that the staff member doesn't work at the selected centre – what action should I take now?

My system of having a precise start and end time does not seem to be particularly intuitive. To show a roster view over several days, the user selects a start date combined with the first hour they would like to see, then an end date combined with the last hour. E.g. using the values 1-Sep-2001 8:00 to 4-Sep-2001 17:00 will display shifts from 8a.m. until 5p.m. for each of the days 1st, 2nd, 3rd and 4th of September. The only simple way to improve this would be to group the times separately from the dates, but after a brief explanation, Emma-Jane understood the system and decided that a change would not be necessary.

The error message regarding the staff member not being assigned to the centre was clear, but it was apparent that it would be useful to have a link from the page displaying the error directly to the page for assigning that staff member to that centre, i.e. to a potential remedy for the error. This way if the staff member really was supposed to be employed at the centre, they could be assigned immediately. Emma-Jane wrote a story for this feature.

We then moved on to assigning a staff member to a centre:

Went to the centres menu, no sign of a function there
(I told E-J to go to the staff member's record first)
Found staff member and found "Centres" link with no problem
Questioned the start date – start of what?
No idea what the abbreviations for the security rights actually represent
(We pause at this point while I refresh the cache on the server, having received some miscellaneous error – this fixes the problem – however, the session has timed out and the login prompt appears)
Log in no problem
No idea that clicking the back button on the browser can be used to get back to what was being performed before the time out

Saved the details and then tried to edit them
Confused by the sudden appearance of an end date – this wasn't on the add employment page – assumes that it was probably there but didn't see it
Followed "back to staff" link
Thought that the "Availability" link was highlighted [it was actually just not in the visited link colour]

This activity raised quite a number of issues.

- It became apparent that I should have a link from the centres view to "assign employment to a staff member" in addition to the existing link from the staff view
- The caption for the start date should make it clear that it is the staff member's contract start date
- The session time-out re-login prompt should return the user to the place they were when the session timed out, rather than relying on them using the back button to achieve this
- The contract end date should appear on the add employment page, not just the edit page, as the contract end date is usually known at the time the staff member is assigned to the centre
- The visited link colour needs to be set to the same colour as the normal link to avoid confusion with different coloured links appearing

Stories were written for all these issues.

The last activity we tried was to add availability to a staff member:

Followed the "Add Availability" link from the availability view
Confused by the drop down box that shows all centres on the system

Emma-Jane wrote a story requesting that the drop down box only shows centres that the staff member is currently employed at.

Defects

There were only four defects in the production system. They were:

1. A rounding error caused the totals at the bottom of the roster to display the wrong number of hours where half hours had been worked.
2. The roster was only displaying up to 5pm.
3. The pay schedule was enforcing a half hour break where it shouldn't.
4. Blocks that extended outside the roster being viewed did not appear at all.

These defects were minor, and took only half a point to repair.

Release

The binary files for this release can be found at:

http://sourceforge.net/project/showfiles.php?group_id=28900&release_id=53737

The source code has the CVS version tag 4.0.

Iteration Four

This iteration took place between the 27th September 2001 and the 26th October 2001

The Planning Game

At this planning game we began to look at the possibilities for automated rostering and some improvements to the user interface. Deborah was concerned that roster entry had taken her a long time when she entered a full week's roster for the Call Centre. Amber suggested that entry directly into the grid might be a better idea. This is of course difficult in an HTML table, so I suggested a system where users could click on a cell in the grid and have the time and date they had clicked on used as the default date and time for the shift entry. Amber also wanted to see a list of names in each cell, rather than one name per column as in the existing display. I had not considered this possibility before, but it seemed like it would be no problem to implement.

We had a prolonged discussion about the implementation details of automated rostering. My project supervisor attended the meeting and noted afterwards that this level of detailed discussion on implementation actually steps beyond the definitions of the planning game. However, I do not think that it is necessary to completely hide implementation details from the customers, particularly where the details could influence how closely the requirements of a story will be met. In the case of the rostering, I thought it would be useful to explain how a best-fit algorithm would work and attempt to gauge whether it would suffice. In a previous system that I developed, which included assignment algorithms, a lot of time could have been saved if I had discussed the implementation details with the customers from the beginning.

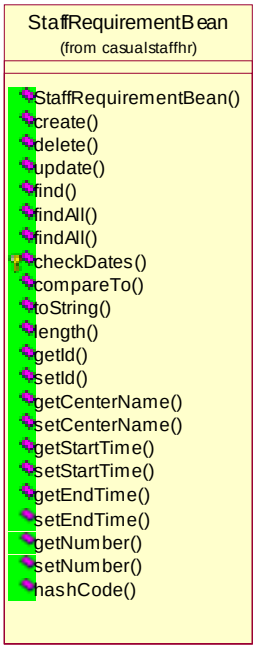
Implementation

The stories for this iteration were as follows:

Number	Description
1/29	Allow staff to give maximum number of hours possible to work [each week]
4/1	Need to be able to add different staffing requirements to cover peaks and troughs
4/2	Automatic rostering -> best fit: ordered by 1. priority (experience and ability), 2. longest shift (availability)
4/3	Daily cap of 8 hours shift per person [in automatic allocation], maximum number of hours 37.5/wk: but will need to be able to override this if necessary or just override when manually updating roster
4/4	Save original version of roster (baseline) and be able to update baseline (to compare with actual)
4/5	Option to make amendments to roster and then have this change apply to all future weeks
4/6	Create a report that can be used as a wall chart that can also be used as a reference for rostered hours
4/7	Warning message to come up if we manually override and make someone over 37.5 hours, need to know this is overtime
4/8	See borders between blocks, grid system to make reading times easier
4/9	Do data entry direct into grid
4/10	Sign in sheet

The obvious story to begin with was 4/2, the best-fit rostering algorithms. I spent some time planning how to apply functional decomposition best and settled on a set of methods that could retrieve a time block from a set of blocks that was the closest match to another particular time block, within a given tolerance. I then created methods to "subtract" one time block for another, adjusting the start and end times, splitting or removing the block according to the attributes of the other. The basic idea of the algorithm for rostering is to find the best-fit availability for a staffing requirement and adjust the availability so that the requirement's time is removed from it, and then to adjust the requirement so the filled portion is removed.

Before I could do this, I needed a way to store the requirements, so I created a new database table and persistent bean.



Unfortunately, the due date for this report arrived before I could finish the implementation for this iteration.

A Talk with Kent

I was fortunate enough to be able to meet and talk with Kent Beck when he visited New Zealand for a conference on the 9th of October. I asked him a few questions directly addressing problems I had been having, and I attended a general address he delivered on XP.

Kent's initial comment was that XP solo does not really work, but it should work better than any other solo development process. I agree that solo programming itself is very difficult, and in the future rather than accepting a solo project, I will consider demanding sufficient funds for an additional developer. I believe that XP for two could be an ideal development environment for very small projects.

During conversation, Kent reiterated the underlying concept of XP, to take a big problem and make it small. Certainly many parts of this project that seemed daunting at first were reduced to much smaller problems by taking a disciplined approach to always selecting the simplest possible solution and "refactoring mercilessly".

When I asked him whether making customers write acceptance tests was really an essential part of the process, his response was absolute: the customers must write the tests, and the tests must be automated. I expressed my concern at the customers struggling with the syntax of the testing language, and he asked me whether any of the four customers could cope with it. I suggested that Elspet would be able to, and Kent suggested that there was nothing wrong with having a "tester" on the business side, whose responsibility is to write acceptance tests. Furthermore, this person could teach the other customers how to write the tests so that they could also participate. I decided to implement this in future iterations.

I also asked for his opinion on unit-testing JSPs. When he realised that I had static HTML mixed in with logic inside the JSPs, he suggested that perhaps using the JSPs to produce XML and testing the XML, then transforming the XML to HTML with XSL would be a good idea. I explore this further in Technical Considerations p.58.

Refactoring

Yet again the functional and unit tests showed their value when it was necessary to add a parameter to the `generate()` method of the `CenterRosterBean` to control the ordering of the `TimeBlockBeans` within the roster's collection. This was the first step in preparing for automatic rostering, as I required the collection of `TimeBlockBean` objects to be ordered according to the staff member's priority at the corresponding centre, rather than the ordering required by the roster displays of staff member then time. The most efficient way to do this was to create two different SQL queries and let the database do the ordering, rather than retrieving the blocks one by one and having to search through the collection for the appropriate position or by re-ordering the entire collection after retrieval. After I had added the parameter, I ran a build of the system and found no errors, but unit testing identified 4 breaks in the classes and functional testing revealed 5 breaks in the JSPs that the change to the `generate()` method had created. Without the tests, these problems could have remained unidentified until the users stumbled upon them.

Release

The release for the fourth iteration is due on Friday the 26th of October 2001. The source code will be tagged with version 5.0.

A test system is available within the University of Auckland network at:

https://mantaray:8444/smacs/jsp

Final Feedback

As an attempt to gain an overall feeling for the success of XP in the development of SMaCS, I surveyed each of the customers on their impressions of the development process. I tried to get them to compare this experience to their previous experiences in a development environment.

These are the survey questions:

Planning and Development

1. Have you been involved in the implementation of a new computer system before? If so, in what capacity?
2. Have you been asked to provide requirements for a new computer system before, if so at what level?
3. Have you been in direct contact with the programmers of a system before? If so in what capacity?
4. Have you been involved with the functional/acceptance testing of a system before? If so in what way?
5. Have you been asked to write tests for a system before?
6. Did you find writing tests difficult, if so in what respect?
7. Did you feel that the "planning game" was a useful forum for expressing the features of the system that you required, if not why?
8. What suggestions (if any) do you have for improvements to the process of gathering your requirements for the system?
9. How useful was it to have access to a **working release** of the system at the end of **each iteration**?

(Circle)

Not at all	Vaguely Useful	Useful	Very Useful	Essential

10. How closely did you feel that the system met the requirements in your stories **at each release**?

(Circle)

Not at all	Vaguely	Satisfactorily	Very Satisfactorily	Perfectly

XP

11. Would you recommend eXtreme Programming as a development process to other people, if not why?
12. Please give any comments on the development process as a whole:

Conclusions

Engineering Considerations

Risk

The part of development that brought the most risk was the automatic rostering. Had I not used XP, I suspect that I would have started this development earlier and run out of time to bring the more important functionality to the quality that is was required to reach. Through the planning game process, I was able to measure the risk and time that the automatic rostering would introduce and make my customers understand the implications of devoting development time to it. This led to concentration on making the other parts of the system as usable as possible thus removing the risk of complete project failure.

One of the concerns expressed to me by Elspet when I first proposed using XP was whether the system would scale well. The risk of failing to meet performance requirements was mitigated by the technologies I employed. While I used a zero-cost back-end database and web container with my own very simple connection pooling, the system will run on any combination of servers that conform to the J2EE web container standards. This means that it can be deployed on a multi-CPU high-performance container with a high-end database and connection pooling system to scale to much larger numbers of users. I present more details on performance in the evaluation section below.

This project has progressed to a point that the customers and myself are happy with. The project is within its initially estimated time limits. This compares very favourably to my earlier systems where they have run well beyond their schedules, particularly in the case of one system, which is now a year outside the original timeframe.

Requirements and Usability

Meeting requirements is emphasised in every part of the XP process. The planning game produced well-contained sets of features and filtered out features of lesser value through the prioritising by business. A “cute” feature that was proposed in the first planning game was the inclusion of a pop-up message to remind the managers when staff members’ birthdays were approaching. While everyone was initially enthusiastic about this feature, it is of little business value and was continually pushed to the bottom of the priorities. I cannot help but think that without this process I would have implemented it early on at the expense of more important components.

Functional/acceptance testing was the area in which I struggled most. I initially had difficulty acquiring and creating suitable tools for web interface testing, and then I found difficulty with persuading the customers to write the tests.

The user interface was very much neglected during the first three iterations, despite its importance in the final product. A “thinking aloud” usability test where the customer writes a story to remedy each sticking point in the test is a very useful device to address the user interface issues.

Reuse

The classes and their unit test counterparts are sufficiently well encapsulated to provide a useful set of software components for future development. The JSP architecture itself and its accompanying automatic generation tools will certainly be useful in the future. I am already using the architecture and the testing tools I developed during this project for other systems.

The technologies that are available now, such as the “web container” that Tomcat provides for running JSPs promote reuse on a grand scale. I did not have to worry about the details of handling HTTP connections or messages at any stage.

XP Values and Principles

Assume Simplicity, Simple Design, Refactoring and Courage

Using the simplest possible solution certainly improved my velocity. My code remained readable and easy to maintain thanks to the absence of irrelevant “future-proofing” code.

I learnt a great deal about refactoring during the development of this system. At many times during the first two iterations, I felt very nervous about applying substantial refactoring to the code. However, the results were always better than I expected, and came with much less effort than I had thought they would require. Taking the courage to refactor at the first sign of unnecessary complexity is an effective way of avoiding overly complex designs and redundant code.

I found that improving the code through the application of design patterns is much easier during refactoring than during design itself. “Standing back from the code” reveals areas in the design where design patterns could bring benefits.

Rapid Feedback and Communication

The planning game is a very effective forum for communication between development and business. Not only was it beneficial for me as far as the elicitation of requirements went, but the business also appreciated the chance to examine their business processes. Elspet commented to me that she thought that this detailed examination of business rules was a major benefit of this project.

My under-emphasis of the importance of functional testing was a problem. Involving the business in the testing from the very beginning and explaining its importance should be a paramount concern. Where I did have acceptance tests running properly, I found them an invaluable form of “on-tap” rapid feedback.

The user-interface must be addressed from the very beginning. User training is required to ensure that the customers feel comfortable trying out the implemented features, in order that they reach a stage where they can provide constructive criticism on the human-computer interaction and transform this feedback into stories.

As I mentioned earlier, I found that disciplined unit testing and test-first coding was the greatest benefit that XP brought me personally as a developer. A suite of unit tests that can be executed at any time gives an invaluable gift of confidence in one’s code. Tracking tasks and calculating velocity provided me with much improved estimating techniques.

Quality

The very low defect rate throughout the project demonstrates the power of using thorough testing to provide software quality assurance. I spent no more than a point of development time in any iteration repairing defects. In the previous systems that I have developed, production releases have required days of debugging. Because these earlier systems had no regression tests, the debugging would often produce further defects that were not discovered until the users found them.

Above all, the testing, the feedback and the close relationship between developer and business resulted in an application that I can take pride in, and that I am not ashamed to pass on to other developers to improve and maintain.

Evaluation

Survey

All four of my customers stated that they had been involved in the implementation of a new system before. However, Emma-Jane commented that she was not involved until the “end” of the University’s PeopleSoft implementation, and therefore felt that she was not involved in the implementation per se at all. This perception probably comes from the isolation that heavyweight processes can inflict on users, undervaluing their potentially valuable feedback.

No one had written test plans of any kind before, but all had performed some ad hoc testing before. All of the customers commented that coming up with test plans was time-consuming and therefore difficult to fit in to their schedules. Amber mentioned that it was difficult to “...pin down exactly what it is that you require it to do...”

Amber stated that she felt we had not paid enough attention to the existing manual systems early enough in the project. We did concentrate heavily on the stories the customers wrote during the planning game, without putting a lot of effort into making sure that we gathered all useful resources such as existing paper reports before the games.

The customers were unanimously positive about the planning game as a forum for requirements elicitation. Further to this, Emma-Jane commented that the group conversation amongst the customers

themselves was very useful, and that building on each other's ideas enhanced the process. This reaction supports the idea of having multiple customers.

All of the customers were satisfied with the way each release of the system met requirements, and all felt that having a working release was either useful or very useful.

Performance

I adapted my HTTP Black Box for performance testing by implementing a multi-threading "spawn" command to set up simultaneous sessions with the web server and perform a task. It is possible to set the number of connections along with a set of standard tests to carry out. For the evaluation I decided to use a simple task where a roster is selected and viewed for a centre.

The test was run with the web server running on Windows 2000 on an AMD Athlon 900 with 256 MB RAM connected via a 100Mbps Ethernet LAN to the backend database running on Redhat Linux 6.2 on an AMD K6-400 with 128MB RAM.

I present below my findings with Borland Interbase 6.0 as the back-end. The times shown are in seconds, and are the average of the completion times of each individual thread.

	Trial number					Average	Total
#threads	*1*	*2*	*3*	*4*	*5*		
1	1	0	1	1	1	0.8	4
2	1	0.5	1	1.5	1.5	1.1	5.5
4	3.5	3.75	3.75	3.5	3.5	3.6	18
8	5.75	7	7	7.5	13.75	8.2	41
16	29.25	53.75	40.625	18.4375	29.125	34.2375	171.1875

The first thing I noticed was the serious performance degradation after a number of trials. I restarted the back-end database between trials to try and compensate for this. I monitored the memory during the running both on the web server and in the test program, but the problem appears to originate with Interbase. I suspect that the special Java-based server that marshals the JDBC requests is consuming resources without releasing them. If the server is left alone for some time, it seems to right itself, therefore I assume there is a problem with garbage collection or memory handling in general.

This issue aside, the response times are a little worse than linear, in fact almost $n \log n$ up to 8 users, after this the performance declines dramatically.

I was disturbed at these results, because although I was only expecting around 10 simultaneous sessions in the production environment, the system would quite possibly need to scale in the future. I decided to re-run the tests on the same hardware with Postgresql [Postgres 2001] as the back-end database. This produced the following results.

	Trial number					Average	Total
#threads	*1*	*2*	*3*	*4*	*5*		
1	0	0	0	0	0	0	0
2	0	0	0	0	1	0.2	1
4	1	1	1	1	1	1	5
8	2	2	2	2	2	2	10
16	5	5	5	5	5.5625	5.1125	25.5625
32	11	11	11	11	10.84375	10.96875	54.84375
64	22	21.21875	22	21.1875	21.57813	21.59688	107.9844

These results confirmed my theory that the back-end was at fault with the performance degradation. As can be seen, the response times increase linearly above 2 threads, indicating that the system scales well. A task completion time of 22 seconds is acceptable for 64 simultaneous sessions. I tried one further test for 128 threads, which produced a time of 43 seconds.

With high-end commercial programs, the system could scale even further, but for the requirements set down for this system, i.e. around 10 simultaneous sessions, the performance is more than satisfactory.

Technical Considerations

Persistence

I see-sawed between ideas on how to manage persistence in my design. On one hand I could make sure that properties in persistent classes returned instantiated objects, e.g. `public CenterBean getCenter()` or I could just return the object's relational identifier: `public String getCenter()`. The former method is more object-oriented, but it is grossly inefficient when the executing code is only concerned with *which* instances of `CenterBean` are involved, rather than being concerned with any of the values of `CenterBean's` properties. For example, to retrieve a list of centres that a staff member works at, it is only necessary to retrieve the values of the `CentreName` field in the `StaffCentreAssignment` table where the `StaffID` contains a particular value, rather than having to instantiate a group of `StaffEmploymentBean` objects and retrieve the value of the `CentreName` property in each of these. For this application, using relational identifiers minimises the amount of data that is passed between the application and the database, so this is the method I settled on.

Abstraction of Web Presentation

Finding the line between unit tests and functional tests for the JSPs in this system was a difficult task. Testing a JSP is made difficult by the fact that it can hold both logic and presentation. As suggested by Kent Beck, using JSPs to produce Extensible Markup Language (XML) and transforming the XML with Extensible Stylesheet Language (XSL) to create the presentation HTML could potentially solve this problem [Beck 2001]. The XML could easily be unit tested using the standard Document Object Model (DOM) and walking through the XML tree. Presentation would be restricted to the results of the XSL transformations, and be unnecessary to unit test.

The Future

Electronic Cards

While physical index cards are useful from the point of view of sorting into piles by priority, or time estimates etc, they are inconvenient for searching through. When portable devices such as 'web-pads' are more readily available, I believe writing stories electronically and printing hard copies to be manipulated in the game could improve the planning game process. This way an electronic copy would always be available, and be easily searchable and distributable later on.

Going Solo – Final Thoughts

As Kent Beck himself commented, XP solo is not a satisfactory way of developing a system, but it's better than any other solo process. It is useful to consider which XP practices had the most effect on the development of this system, which had little effect, which were ignored as team practices, and in what way ignoring those practices detracted from XP's benefits.

"Growing" unit tests into a regression testing suite and test-first coding were the practices that lent the most benefits to this project. The benefit can be gauged by comparing the results of this development with the previous systems that I mentioned at the beginning of this report.

It is now difficult to imagine developing a system without a set of regression tests. Any change to a system, no matter how small carries an enormous risk of breaking another part of the system. My regression tests alerted me to and allowed me to repair frequent breakages during this development, often at times when I would not have thought such a small change would have such an impact. Changes to my earlier systems still result in breakages that take me days to find and repair, often with negative consequences for business.

I believe that test-first coding corrects what could be considered an age-old problem of “programming back to front”. No matter how carefully a software design diagram is created, transforming this so-called “design” into code will result in errors. In my previous programming efforts, these problems do not become apparent until the code is tested. It therefore makes sense to write the test first and match the code to the test, this way the effort is effectively halved through writing the code correctly in the first place. Furthermore, a separate effort for writing tests later is unnecessary, and the tests document the original requirements for the corresponding piece of code. Compared to my earlier systems, my development speed was increased by writing tests, rather than decreased as one could expect given the extra time spent coding unit tests. I believe this is because any new code written was always higher quality than my previous efforts due to the test-first coding, and therefore I spent less time repairing defects.

Small iterations with fixed time frames ensured that I always had a goal to work towards. I had used vaguely iterative processes in the past, but without a time frame to work in, the feedback was not sufficiently frequent, and the project visibility was very low. The visibility during this development was excellent at all times.

Having a customer on-site was nothing new for me, although I did manage to only use face-to-face discussion rather than email contact as I have in the past. The fact that no serious ambiguities or misinterpretations arose during development can be attributed to having on-site customers.

It is unfortunate that I could not take advantage of pair programming. One of the major outcomes of this development for me is to identify areas where pair programming could have helped immensely. I have embraced design through simplicity, but refactoring was made very difficult by not having a partner to review the code while I was immersed in the details of programming. Tedious tasks such as scripting acceptance tests could have been made much easier by splitting the load between two people. Another developer would quite possibly have discovered much sooner the occasional important tests that I was slow to address.

As a solo developer, changes to the “load factor” of project velocity are dramatic. The load factor refers to the percentage of “perfect development time” (the points measurement, in my case half days) that is actually spent on development. My other responsibilities frequently got it the way of development and slowed my velocity markedly. Having other developers or even just one other developer could have mitigated this problem.

With the success of this project over my past projects, I intend to use XP for all my future solo development. XP will also be a serious draw card for me when seeking work in team development.

Appendices

- Web Robot
- HTTP Black Box
- Other Tools

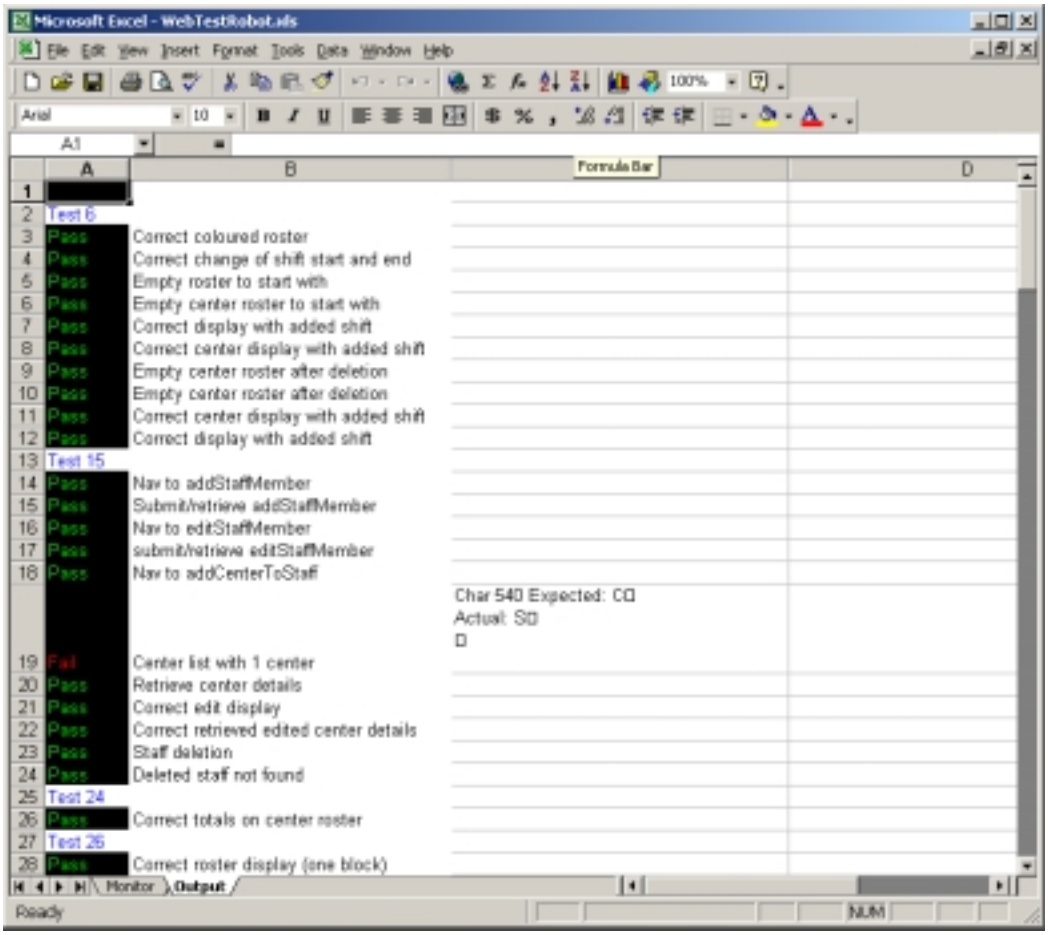
Appendix A – The Web Robot

Although I only used the web robot in the first iteration, it was a useful tool. The syntax is as follows:

Command	Parameters	Purpose
Pause		Stops the execution of the test – useful for checking the current state of the browser
Navigate	Page	Navigates to the page
Comment	Text	Sends the text to the output window
FillText	Field, Value	Fills the form field with the given name with the given value
Submit		Submits the form
Expect	Text, Comment	Compares the current browser page HTML with the given text and outputs the comment next to the result
FileExpect	Filename, Comment	Compares the current browser page HTML with the given file contents and outputs the comment next to the result
SQL	ODBC String, SQL	Sets up an ODBC connection with the given string and executes the given SQL
Quit		Quits the active browser window

Excel makes an Object Linking and Embedding (OLE) connection using Visual Basic for Application (VBA) code to an instance of the Internet Explorer browser. Excel can then control navigation and read the contents of pages. A separate “capture” command captures the contents of an active browser window to a file or a spreadsheet cell. The “expect” command compares these captures where needed.

The following screenshot shows a completed set of tests.



Appendix B – HTTP Black Box

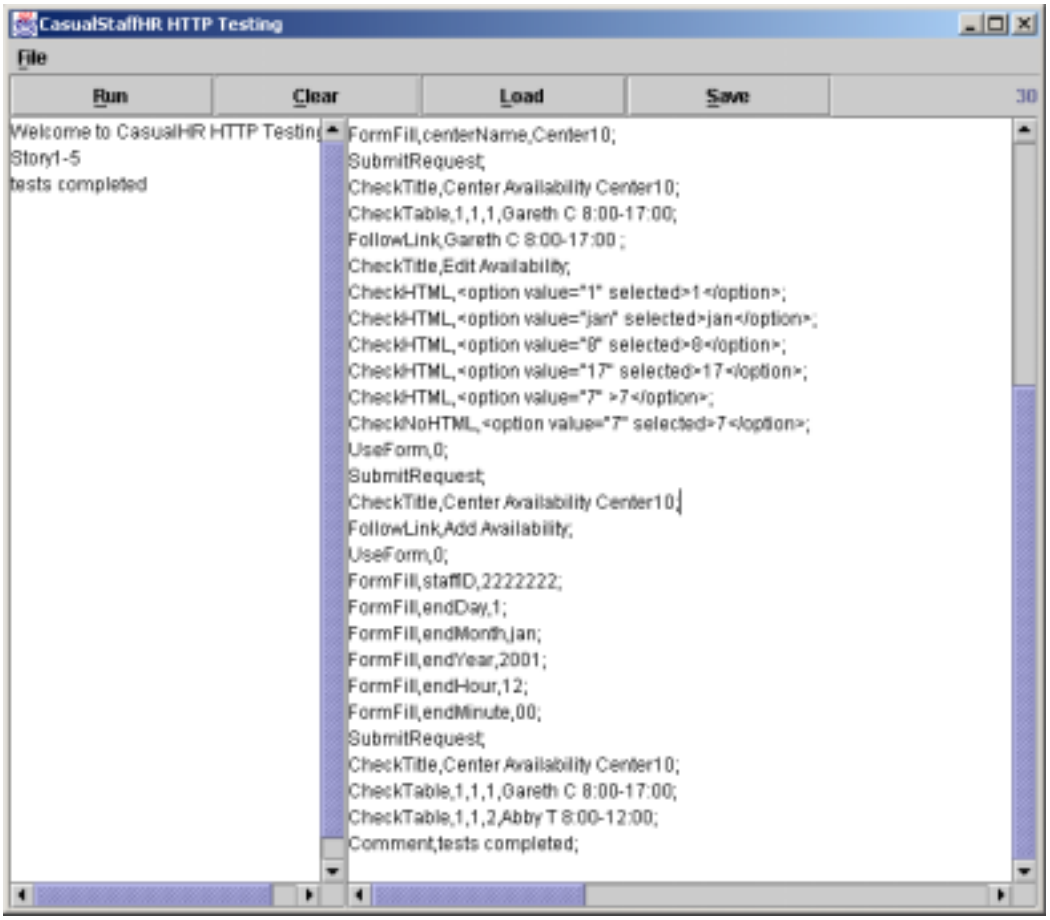
My front end for HttpUnit comprises an output frame and a simple text editor. The contents of the text editor can be saved to and loaded from files. The available set of commands are entered or loaded from file into the text editor and executed by pushing the “run” button. The commands exploit HttpUnit’s features such as analysing the contents of tables, filling in form fields and checking for the presence of certain strings.

The commands are:

Command	Parameters	Purpose
Setup	-	Runs the basic test fixture that sets up a sample data set for testing in the database
Teardown	-	Removes the fixture from the database
ResponseTest	Comment, Page	Navigates to the page and outputs the comment if there is an error together with the error message
UseForm	Form#	Makes the form the active one (index base of 0)
FormFill	Field, Value	Fills the named field of the active form
CheckTitle	Title	Asserts that the title of the active page is that given
CheckHTML	String	Asserts the presence of the given string

CheckNoHTML	String	Asserts that the given string is not present
CheckTable	Table#, Row, Column, Value	Asserts that the row/col reference in the table specified contains the given value
SubmitRequest	-	Submits the active form
FollowLink	Name	Follows the link with the given name
Comment	Text	Outputs the given text
SetAdmin	User, Boolean	Sets the given user to admin or not admin
SecurityCheck	Page, Boolean	Attempts to navigate to a page and throws an exception if the boolean is true and a security error is met or vice versa

The following screenshot shows a completed test.

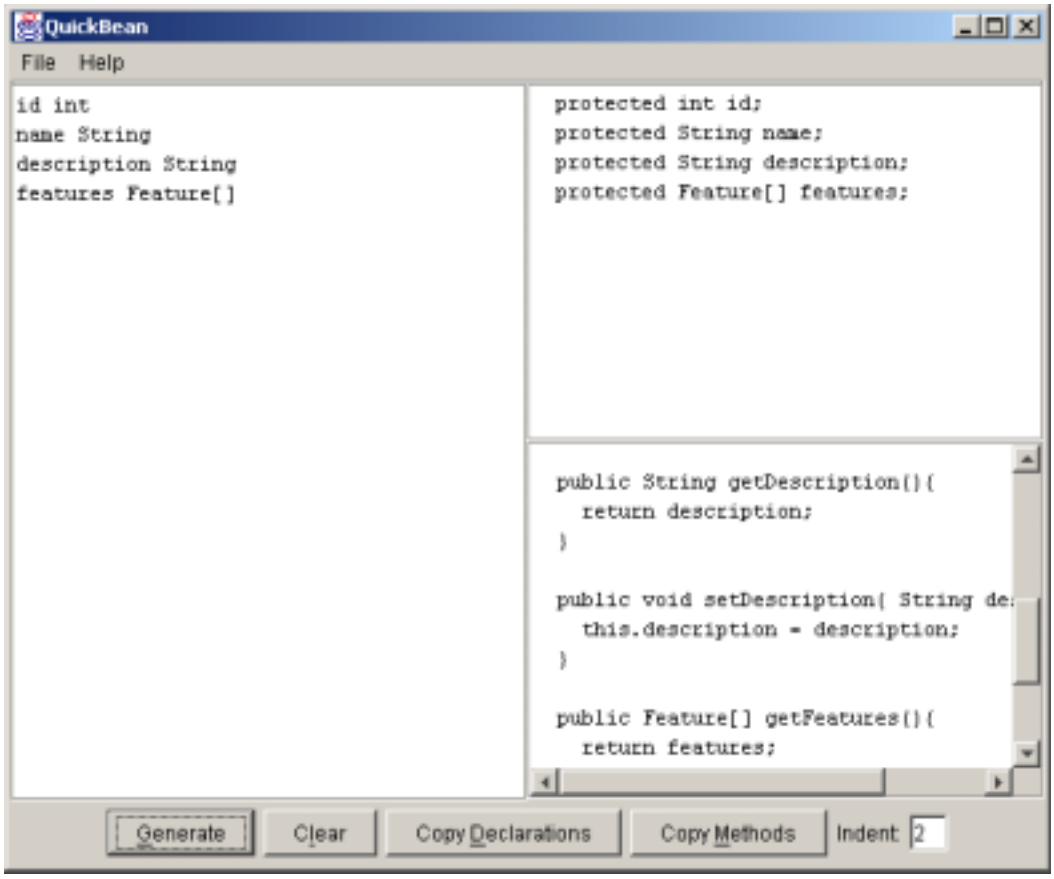


Appendix C – Other Tools

QuickBean

I used JBuilder Foundation Edition as my development environment for Java. The freeware Foundation edition lacks the rapid bean design tools of the more advanced editions. I decided to write a tool to take care of the bean code generation. I developed a very simple tool that takes a list of property names and types and generates a set of declarations and getter/setter methods for those properties. The tool outputs these to separate text areas and provides one-click copying to the clipboard. I estimate that this reduced

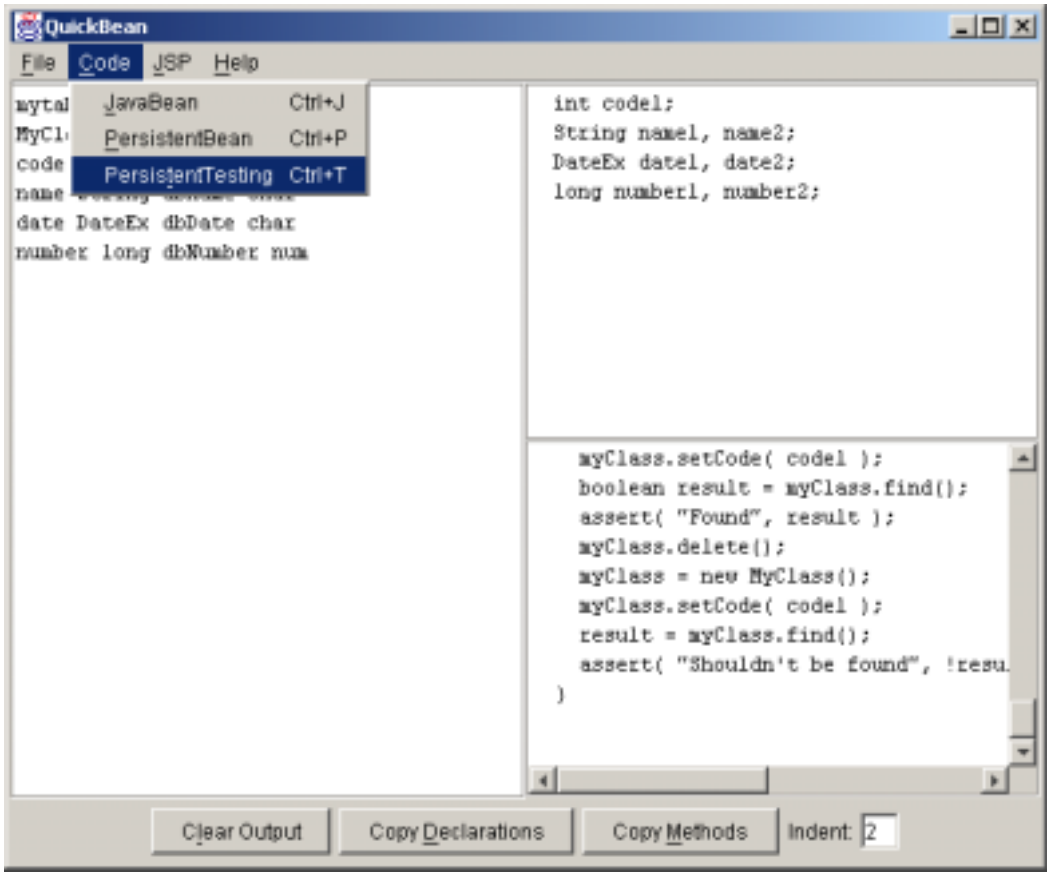
the development time for an average JavaBean from up to 10 or 15 minutes of tedious coding to less than half a minute.



QuickBean can be downloaded from http://casualstaffhr.sourceforge.net/QuickBean.zip

QuickBean II

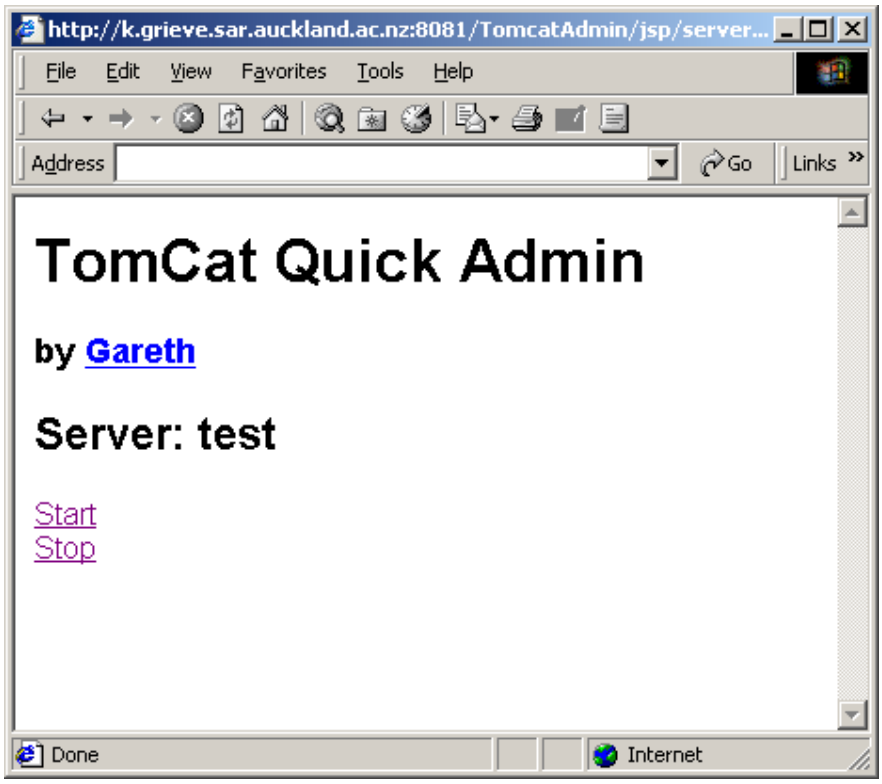
When I had spent some time with my JSP/JavaBean architecture, I realised that it would be reasonably simple to generate the code for persistence with a tool. I added functionality to generate the methods for the persistent JavaBeans. It then became apparent that the unit tests could be generated, so I added features to create the classes for testing in JUnit. Finally, I added the ability to generate the HTML/JSP code for the add/edit/delete/view pages themselves. This took the time for the build/test cycle from several hours down to a matter of minutes.



QuickBean II can be downloaded from http://casualstaffhr.sourceforge.net/QuickBean2.zip

TomcatAdmin

The version of Tomcat that I used has no support for remote administration. I developed a simple (and rather crude) tool that provides a web interface to start and stop any Tomcat server instance running on the same machine by using the Java Runtime class to execute the Tomcat-provided start-up and shutdown scripts.



TomcatAdmin can be downloaded from http://casualstaffhr.sourceforge.net/TomcatAdmin.zip

Acknowledgements

I would like to thank my employer John Holley, Group Manager of Information Services for Student Administration at the University of Auckland, for giving the opportunity to study a real situation and earn an income during the process.

I would also like to thank my customers Emma-Jane, Elspet, Amber and Deborah for their enthusiasm and support during the development.

References

[Allamaraju 2000]
S. Allamaraju, Karl Avedal et al. "Professional Java Server Programming". Wrox Press Ltd, Birmingham, USA, 2000.

[Beck 1999a]
Beck, Kent. “Extreme Programming Explained, Embrace Change”. Addison Wesley, USA, 1999.

[Beck 1999b]
Beck K. “Embracing change with extreme programming”, *Computer*, vol.32, no.10, IEEE, USA, October 1999, pp.70-7.

[Beck 2001]
Beck. K. Interview with the author, 8th October 2001.

[Boehm 1988]
Boehm BW. “A spiral model of software development and enhancement”, *Computer*, vol.21, no.5, May 1988, USA. pp.61-72.

[Brooks 1986]
Brooks, Frederick P. “No Silver Bullet, Essence and Accidents of Software Engineering”. *Information Processing ’86*, H.-J. Kugler ed., Elsevier Science Publishers B.V., North Holland, 1986.

[Canna 2001]
Canna, Jeff. “Testing, fun? Really?” http://www-106.ibm.com/developerworks/library/j-test.html, http://www.-106.ibm.com/developerworks, 2001.

[Fowler 2000]
Fowler, Martin. “Lightweight Methods”, *Software Development*, vol. 8, no. 12, CMP Media, USA, December 2000.

[HttpUnit 2001]
http://httpunit.sourceforge.net

[Jeffries 1999]
Jeffries, Ron. “Extreme Programming – An Open Approach to Enterprise Development”. http://www.xprogramming.com/xpmag/an_open_approach.htm, http://www.xprogramming.com/xpmag, 1999.

[JUnit 2001]
JUnit. http://junit.sourceforge.net. 2001.

[Kivi 2000]
Kivi J, Haydon D, Hayes J, Schneider R, Succi G. “Extreme programming: a university team design experience”. *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era*. IEEE. Part vol.2, 2000, pp.816-20 vol.2. Piscataway, NJ, USA.

[Microsoft 2001]
Internet Explorer V5.x. http://www.microsoft.com/windows/ie. http://www.microsoft.com. 2001.

[Miller 2001]
Miller, Roy W. and Collins, Christopher T. “XP Distilled”. http://www-106.ibm.com/developerworks/library/j-xp, http://www.-106.ibm.com/developerworks, 2001.

[Muller 2001]
Müller MM, Tichy WF. “Case study: extreme programming in a university environment”. *Proceedings of the 23rd International Conference on Software Engineering*. IEEE, pp.537-44. Los Alamitos, CA, USA. 2001.

[Netscape 2001]
Netscape Communicator V4.x. http://browsers.netscape.com. http://www.netscape.com. 2001.

[Nielsen 1992]

Nielsen, J. “The Usability Engineering Life Cycle”. *Computer*, Vol 25. No 3. pp 12-22. IEEE, USA, March, 1992.

[Pelrine 2000]
Pelrine, Joseph. “Modelling infection scenarios – a fixed-price eXtreme Programming success story”. *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications*, Pages 23 – 24, USA, 2000.

[Postgres 2001]
Postgresql. http://www.postgresql.org. 2001.

[Raymond 2001]
Raymond, Eric S. *The Cathedral and the Bazaar*, http://www.tuxedo.org/~esr/writings/cathedral-bazaar.

[Reeves 1992]
Reeves, Jack W. “What is Software Design”, http://gosh.ex.ac.uk/~cs98abh/archive/documents/what_is_software_design.html. Originally C++ Journal, 1992.

[Sun 2001]
Sun Microsystems. *Java 2 Enterprise Edition.* http://java.sun.com/j2ee. http://www.sun.com. 2001.

[Sun 2001a]
Sun Microsystems. *JavaBeans*. http://java.sun.com/products/javabeans. http://www.sun.com. 2001.

[Sun 2001b]
Sun Microsystems. *Enterprise JavaBeans*. http://java.sun.com/products/ejb. http://www.sun.com. 2001.

[Sun 2001c]
Sun Microsystems. *JavaServer Pages*. http://java.sun.com/products/jsp. http://www.sun.com. 2001.

[Tomcat 2001]
The Jakarta Project, Apache Software Foundation. *Tomcat*, official reference implementation for JavaServer Pages and Java Servlets. http://jakarta.apache.org/tomcat. http://www.apache.org. 2001.

[W3C 2001]
Document Object Model. http://www.w3.org/DOM. http://www.w3.org. 2001.

[XP 2001]
eXtreme Programming Software. http://www.xprogramming.com/software.htm. http://www.xprogramming.com. 2001.