University of Glasgow | School of Computing Science

Level 3 Project Case Study Dissertation

TP3 Team E Project: A Market for Industrial Symbiosis

Robert Allison
Ross Anderson
Joe Frew
Michael McKay
Ben Procter
Nicholas Saunderson

10th April 2016

Abstract

Many businesses and organisations produce vast quantities of waste material as a byproduct of their industrial processes, which are often of a usable quality to be recycled. Many (often small) businesses do not recycle their waste, often because it is impractical and costly. In fact, not recycling waste can be extremely damaging to the environment and can cost more in the long term [**?**].

Our Team Project 3 software team designed and implemented a "proof of concept" system of waste trading, where businesses or individuals can buy and sell scrap metal in a stock-market style website, meaning highly competitive pricing and a practical, flexible and anonymous "order matching" algorithm.

Introduction

The purpose of this essay is to document the entire software development process of our team throughout the Team Project 3 course, from the initial meeting and requirements gathering sessions with the customer to the final demonstration of our project.

Although the team had a wide range of technical skills and experience, and some members had worked in professional software teams during internships and smaller hobby projects, we didn't have much experience with the formal software development methodologies, techniques and practices covered in our Professional Software Development course.

We divided our software process into six main areas: project planning, requirements engineering, source control management and versioning, software engineering, quality assurance and testing, and process improvement. We have written a chapter of this dissertation on each area discussing our experience, the challenges faced and how we solved them, and the techniques we used to improve in each area.

Our customers for the project were Liz Brooks and Mike Bonaventura, both from the Crichton Carbon Centre, tied with the University of Glasgow. They specialised in environmental governance, environmental justice and climate change. They required a prototype of a "stock market" style system that would allow small, medium and large scale businesses to trade waste: not only saving them money but also encouraging re-use of products, components and materials - a step towards a Circular Economy [**?**]. This would hopefully make a significant positive impact on the environment by reducing the energy expended creating or extracting new materials, and also reducing the impact on the Earth's atmosphere caused by the destruction of waste. The project was to be called "AMIS" - A Marketplace for Industrial Symbiosis.

Part of their requirements for the first iteration was for us to decide which types of waste to explore in our prototype. After some research we decided to focus on scrap metal, mainly because it was easily classified by type, use and quality. Scrap metal is generally sold to dealers who re-sell to other businesses or to the council who can reduce it. AMIS would hopefully allow businesses to trade directly - saving them money and allowing rural businesses to trade with each other directly without needing to transport the waste to a dealer. Although we based our prototype on the needs of scrap metal traders, we focused on keeping it as extensible and general as possible allowing the site to be extended to all different types of waste.

Case Study Background

Our customers specialise in a variety of fields - both academic and applied work - and they investigate how certain techniques can influence the carbon landscape of Scotland and how to implement these techniques into real world scenarios. These specialisms are far-reaching and cover topics from Sustainable Rurality to Ecosystem services and Land Management [**?**].

The Crichton Carbon Centre came to us with the original concept for AMIS. They knew that many small to medium sized businesses in Dumfries and Galloway that were involved in the manufacturing industry, or those that used raw materials, were often left with large quantities of unneeded material that would simply go to a scrap yard - an action that was both costly to those companies in terms of profitability but also to the environment. AMIS would provide a competitive local market for buyers *and* sellers.

Initial Requirements

At our first meeting with the clients, it was asked that we only design the site for one material market such as metals, food waste or lumber products. It was also explained that any site designed would have to have several key requirements:

- The system would need to be easy to use and navigate, as users could range from small business owners to tradespeople, with a wide range of computer skills.
- The site would not only need to be able to find suitable revenue to be self sustaining in the long term, but also grow and be expandable to other countries and markets.
- We would need to be able to show that it was compliant with the relevant waste handling legislation.
- User data must be kept secure at all times in line with the Data Protection Act (1998), and users allowed to view all data stored about them and be able to update it if necessary.

To tackle these requirements we created mockups of webpage layouts, as well as investigating different financial models for websites and social enterprises.

We dedicated one team member, Ben, to research any essential legislative requirements, especially laws that were different in Scotland from the rest of the UK because this would give us an insight into how we could extend AMIS into other countries or markets.

During our requirements gathering, the customer suggested a generic stock market where users are matched automatically by the system. We liked this model because demand on a particular market could be used to influence prices and would in turn keep the system competitive.

Because our project's customers weren't going to be the main end users of the site (unlike many software projects), they helped us profile and investigate potential waste traders. They suggested that the main users of the website could be small business owners (wishing to make a profit on surplus, or buy at a low cost), tradespeople and hobbyists, who might be more flexible in the specification of metal they required.

Progress on the website itself was quite slow during the first two iterations, the complex idea of the "stock market" model meant we spent much time planning and designing how the system would work - turning an abstract notion of a stock market into a system of physical transactions was quite a difficult task, and we made many key design decisions during these sprints.

During the next three iterations we prototyped the main features of the site: the user's dashboard, market pages (with price graphs), credentials submission features, user profile pages, the order matching algorithm, and features fulfilling the other main functional requirements. After Iteration 5 we had implemented all of the key functional requirements (and most of our non-functional ones), created a test harness and relatively comprehensive set of unit tests, and investigated the site's potential business value. We dramatised a scrap metal transaction during our final customer demonstration which we felt demonstrated the usefulness and value of AMIS.

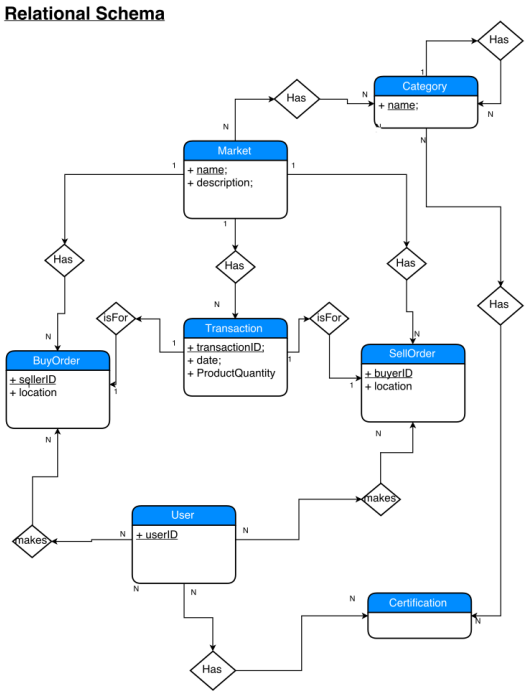
Project Planning and Team Organisation

From the start of the project, the team recognised the need for a concrete planning system. Some of us had experience from internships of "Kanban", a work management system using queues of work items, inspired by the Toyota Production System.

Although the entire team agreed that a Kanban style system could be helpful, after

the Professional Software Development lab on Trac (a project management and bug/issue tracking system) we decided as a team that the "ticket" functionality provided by Trac could replace Kanban while maintaining a structured method of creating and distributing work items.

To actually plan out the project from the beginning, we had to visualise the overall structure and components of the system, so we created several UML and relationship diagrams to break our initial idea of AMIS into easily digestible pieces. r0.5



To try and limit the scope of our project, we used the MoSCoW strategy to separate the features we intended to implement into four levels of priority. This also enabled us to estimate when features would be completed, by prioritising the "Must-Have" features in earlier iterations.

Our weekly Agile stand-up meetings allowed us to discuss what we had achieved in the previous week and what we planned to do next. After this we would usually split into smaller groups and share our expertise. We also decided to make an effort to eat together after our stand-ups to get to know each other and discuss any more informal aspects of our group project. We felt this was important to create a

positive team atmosphere, and enabled us to discuss informal ideas for the project in a more relaxed setting.

We assigned ourselves roles at the start of the project based on our strengths and experience, although upon learning our customers wished a web-based application Michael (formerly chief architect) switched roles with Joe (formerly project manager), as Joe had real life experience with Django and web frameworks during an internship.

We also set up an instant messenger "group", where team members could quickly communicate with each other to discuss team administration. The "instant" nature of the messaging service was useful however we did encounter problems with the system later on, as we realised the tone of the discussion was often very informal, and items we discussed were difficult to find again later or relate to our current work. After exploring the possibility of using Tea Rooms or Slack, we decided to stick to our previous instant messenger for the push notification function, convenience, portability and file sharing functionality, but also agreed to take anything important we discussed there and write it up as a more detailed post on the Trac forum. One of the reasons we liked the Trac discussion forum plugin was that we could use WikiFormatting, and include pictures, videos and make hyperlinks to tickets, commits, and other forum posts.

Our use of Trac and the discussion forum was part of the "paper trail" planning strategy we adopted. To leave a detailed history of our project work for review and reference we decided consistent, frequent documentation was key. After the second iteration, our client asked to be kept informed about our major design decisions, so we created a rolling document detailing our progress, client specifications and any other relevant information that we kept constantly updated through the project. This was easily done using the "instant update" feature of Google Docs, so our changes could be updated in real time for easy reference.

Each new issue or task in the project was opened, updated and resolved as a ticket on the Trac system, giving a concise list of past and current tasks. We later found that we could link our SVN commit messages and tickets to each other, and so were able to document which updates to the system fixed which issues, making it even easier to keep track of our progress.

We decided to use the 4L ("Learned, Liked, Lacked, Longed For") Agile retrospective activity to lead our retrospectives at the end of each iteration: the concept seemed intuitive and we could always use a different activity if we felt the 4Ls technique was ineffective. After feedback from our demonstrator, we decided to end our retrospectives with a list of concrete goals for our software process for

the next iteration.

Requirements Gathering and Engineering

The team elicited requirements mainly through face-to-face requirements gathering sessions with the customer at the start of each development sprint.

After each face-to-face requirements gathering meeting we would follow up by email confirming any nuances we had encountered while documenting the meeting. We tried to be flexible since we learned from the Agile methodology that we should favour "customer collaboration over contract negotiation" and listen to our customers to build a fair set of requirements for everyone.

We refined the structure of our meetings during the first two sessions and settled on the following pattern:

1. Recap the previous iteration's requirements.
2. Demonstrate how we fulfilled them, or how we planned to deal with them if we hadn't. We would also walk the customer through any new workflows in the site.
3. Suggest possible new features, based on technical feedback from our team and what would be easily developed.
4. Invite the customer to comment on the direction they saw the site going and listen to any new ideas they had.

We believe this structured way of requirements engineering worked very well - our customers were busy and we feel the refresh and walkthrough of the site helped them to be creative.

After the meeting, we would classify and organise the requirements (using MoSCoW if we needed), prioritising them, analysing how we would validate the new features from the requirements, and then documenting the meeting and requirements on Trac. This loosely followed the "discovery, classification & organisation, prioritisation, validation and documentation cycle" of requirements elicitation we learned about in our Professional Software Development course [**?**].

Sometimes in the meetings we had to negotiate requirements directly with the

customer, if we felt some features might take too long or be infeasible. In one instance we needed to negotiate to perform less financial modelling of the system as nobody was experienced enough in finance to do this.

Because our system was a "proof of concept" and our customer was so flexible and open to ideas, they didn't supply us with many non-functional requirements, so we drew these up ourselves, basing them on what we imagined potential users of the system might need. We validated these ideas with our customer during our meetings.

Source Control, Versioning and Change Management

Change Management is key to a successful software project, especially in a project like AMIS with continually changing requirements and customer needs, a team of developers mainly working from home, and a variety of skills and experience within the team.

Our customer had given us broad scope with the functional requirements of our project, and helped us shape the site as it grew in each iteration. We found that as we developed more of the site our customer would modify the requirements as they saw new features emerge.

The team quickly started creating Trac tickets to manage work and the assignment of tasks. Most of the team had previous experience with Change Management Software (mainly "git") and the "update, resolve, commit" cycle [**?**, Chpt. 2], which helped us immediately begin committing changes and making progress with the project. This also allowed us to quickly analyse changesets to determine the causes of any bugs we discovered in the software.

We recognised the importance of descriptive commit messages early on, and we soon started linking them to tickets (and linking tickets to commits). As the project expanded we found after a long session of programming we sometimes gave in and committed short or informal messages. Our feedback for Iteration 4 commented on this so we discussed possible solutions. One option was to create a minimum length for commit messages as a pre-commit hook, but we agreed this would just be frustrating, and the only real solution would be developer discipline. For example, we reminded ourselves that we should avoid the "svn ci -m" flag and type a longer commit message in a proper text editor.

The nature of the "iterations" meant our releases were deadline-driven (and not milestone driven) so it was important we managed our time and work queues carefully as to ensure the completion of key requirements by each iteration. Time estimation was one area we could have performed better in, as the volume of work seemed to increase exponentially as we got closer to the deadlines, and we agreed that in hindsight, we could have used a technique like Planning Poker to more accurately estimate time allocations for each task.

Throughout Iteration 4 we suffered some setbacks with the test suite where we had significant bugs and issues getting the tests to pass, and we rarely had a working copy available that would fully pass the test suite. Just before the Iteration 4 customer meeting, we realised we had each focused on new functionality too much and that we didn't have a stable, working copy of the site on our change management system to show to our customer, and we had to use a local copy on a developer's laptop to demonstrate the site for the customer meeting.

This prompted us to immediately rethink our change management, and set up a proper branching strategy. During our 4th iteration retrospective we brainstormed and agreed that a "Branch-when-needed" [**?**] system was best, where we would try and keep "trunk passing the test suite and do major development work in small, testable branches.

We felt disallowing any direct modification of trunk could induce Development Freeze [**?**], so we agreed that if only minor changes were made the risk of breaking the main branch would be acceptably small. We formalised our new branching strategy by creating a tutorial in our developer how-to forum. .

Our change communication was mostly through our weekly stand-up meetings, and minor changes (such as changes to private implementation) initially conducted over our instant messenger although in hindsight this was quite ineffective, and we eventually realised that the temporary nature of instant messages didn't ensure everyone was fully informed of the change which could have lead to clashes. We found the "Timeline" feature of Trac was a useful tool enabling us to quickly update ourselves on all changes by other developers since we'd last committed, and we also used the Trac forum to host a section on "General Discussion", which we used for some change communication.

Our change communication with the customer was via the "rolling document" we'd set up to communicate major design decisions, and published features of the software. This worked really well, and Liz was happy with the progress we were making, although we did make fewer changes to the rolling document during the

third and fourth iterations as we'd finalised the design of most major functions of the site and were mainly working on the test harness and smaller non-functional requirements.

Many source control paradigms recommend taking "tags", "releases" or "snapshots" of the project. We started to do this after Iteration 3, where we back-tagged our previous iterations and began tagging the project during new ones. This was useful as it let the team see exactly what progress was being made between each iteration.

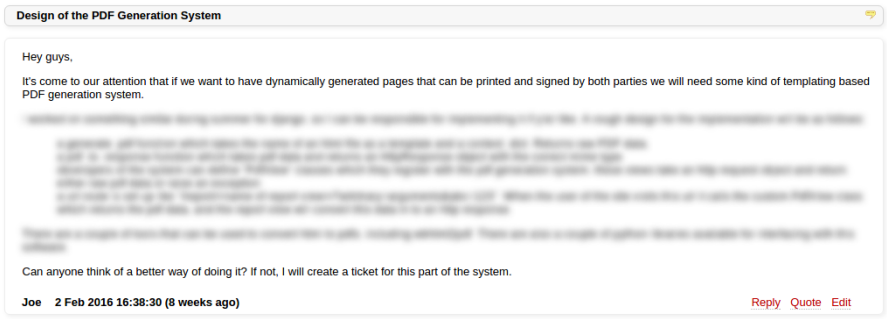
Software Engineering and Development

Designing for Modularity

Before the start of the software development process, the team were aware of the concepts of high cohesion, low coupling and modularity when designing good software systems. The project significantly reinforced our understanding of why these concepts are important when working with complex systems.

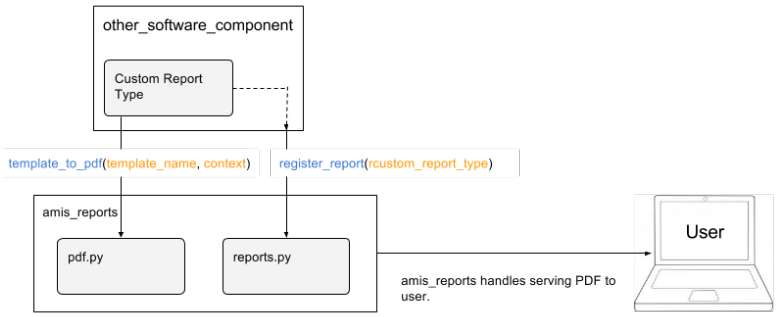
Since our project briefing emphasised that AMIS was a proof of concept and it would be subject to a lot of change, one of our main development goals was to ensure a highly modular and flexible design. To accomplish these design goals, the project was split into a number of distinct apps which implemented different parts of the site's functionality. This design made for good separation of concerns and cohered closely to one of the Django framework's main design philosophies [**?**] which is *loose coupling* between software components.

For ach app (such as amis_reports or amis_legislation), the team discussed the general design of the interactions between it and the rest of the software. In the case of the amis_reports app, the topic was brought up at the 3rd iteration retrospective as a feature for the upcoming 4th iteration, and ideas were exchanged about how best to implement it. We installed a discussion forum plugin on Trac as a means of recording the ideas discussed more formally:



Example discussion post on Trac

This discussion acted as an initial specification for the interface which other developers would use to interact with the report generation component. In the case of the amis_reports app, Joe implemented the internal API. Another developer (Michael) then tested the effectiveness of the interface by creating a new report type and registering it with the system. This strategy proved that the interface was effective as Michael, with no previous experience of the interface, was able to rapidly and successfully prototype a new report class.



Since the team thought it would be good practice to adhere to Django standards and conventions, we decided to use design patterns common to the framework. For the amis_reports app, we chose the "registration" design pattern which is seen throughout Django, for example in the django_admin app [**?**]. This pattern allows a developer to register a class with custom business logic with the amis_reports component.

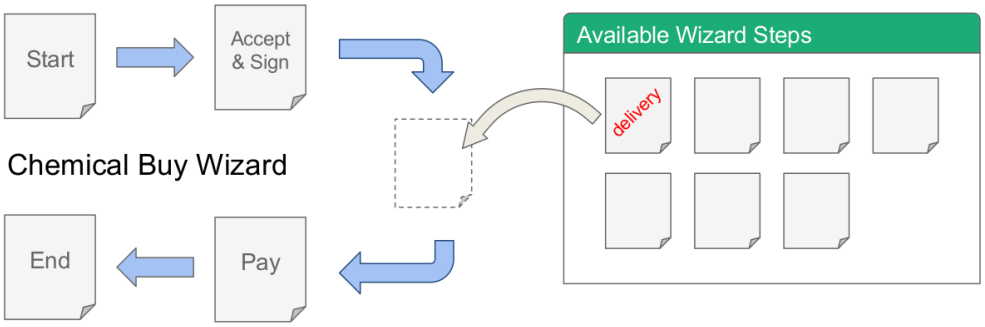
The benefits we enjoyed by designing the amis_reports app in a loosely coupled way sharing the conventions of the Django framework definitely reinforced the importance of modular design. It was due to the success of this component that we decided to design others in a similarly modular way.

Architectural Patterns

Another example of a component designed to be highly modular is the transaction wizard manager. The system allows developers to create "wizard steps" with custom business logic and page layouts which administrators can view and use to construct wizards (composed of multiple *wizard steps*) which they assign to markets (each market has one *buy* wizard and one *sell* wizard). This allows the process of confirming a buy and sell order to be totally customisable to particular users or markets, and allows administrators to change these processes without having to consult developers.

We found that the use of well-known design patterns coupled with descriptive method names like "register" acted as "beacons" [**?**, p. 397-709] to other members of the team and allowed them to more quickly grasp the structure of the software component and make use of it more efficiently.

The Django framework offers interchangeable middleware classes which process a request before it reaches the view ("controller" in MVC - Model, View, Controller [**?**]) logic. The middleware pattern is similar to the well-known *Chain Of Responsibility* and *Pipeline* patterns as seen on Vince Huston's (a highly regarded author on software engineering) site [**?**]. Inspiration was taken from this design pattern when creating the transaction wizard. The diagram below the pattern in use with the *registration* pattern to allow admins to build wizards from a library of wizard steps:



Some of the advantages of the middleware/chain-of-responsibility architecture are:

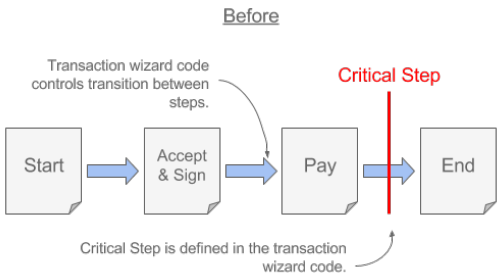
- Modular steps offer good separation of concerns so it's easy for developers to add new functionality and remove old functionality without any knock-on effects.

- Wizard and individual Wizard Steps are reusable by admins at the object level [**?**, p. 214] reducing repetition in the codebase in adherence with the DRY ("Don't Repeat Yourself") principle [**?**].
- Configurability via the admin panel makes it highly flexible and easy to use.
- All wizard state is stored in the database, so Django maintains its statelessness which means it can be scaled horizontally.

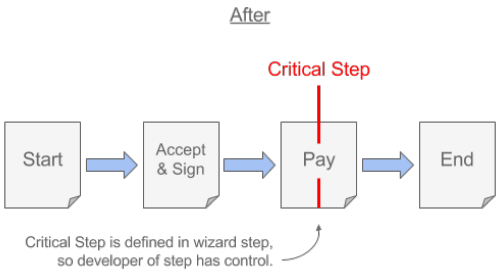
Through the design and implementation of the transaction wizard the team learned the importance of the use of design patterns when creating modular and maintainable systems. We also learnt the advantages of inspecting existing codebases to identify patterns which we could later use.

Software Refactoring

r0.5



We designed the transaction wizard for any amount of complexity in the individual wizard steps, since they could contain arbitrary business logic, such as the user having to complete a form or upload documents. When implementing a "critical step" functionality (which would ensure that both users in the transaction had reached a certain step before either could continue), Michael decided to create this logic in the transaction wizard framework code rather than within a wizard step.
r0.5



At the team's weekly standup meeting, Michael updated the team of his progress. Joe suggested that to keep the transaction wizard system simple and flexible, this functionality could be moved into a wizard step. On the right is a representation of the "Chain of Responsibility" for the original implementation, followed by a diagram of the implementation after refactoring.

The team decided to refactor this logic into a wizard step, which had no outward effect on the usage/behaviour of the app during unit and user acceptance testing, but made the design of the system less complex and more flexible and maintainable. It also enhanced the usability of the app from an administrator's viewpoint when designing transaction wizards, because the complexity of the critical step is abstracted away from them to be handled by developers instead. This also allows for "glass-box" extensibility, where new features can be added by only extending the original source code, not modifying it [**?**, p. 1].

This design decision conforms with the YAGNI ("You aren't going to need it") principle [**?**] since it delays the implementation of the critical step functionality until the point where it was actually a required feature.

This refactoring step could probably have been avoided with a slightly improved team process. Fuller documentation on using the new feature could have been provided by Joe to improve the team's understanding of the capabilities and limitations of the system. Although Trac was used effectively for ticket management, team members could have included other developers in tickets (using the "CC" functionality, similar to email) where they thought they may have had expert knowledge on the system (In this case, Michael could have CC'd Joe). Despite this, the software refactoring step was a success; we learned that code documentation and good developer change communication are key to a good team process.

Quality Assurance & Testing

Planning the testing process

During the initial planning stages of the project, we used the MoSCoW system to prioritise and document initial functional requirements. We translated the functionality from this document to create the behavioural testing specification, which would lead our testing development.

A further behavioural testing document was then created based on this information, which would be the foundation of our black box testing specification. This fitted in well with the Agile development process we were practicing, so the team decided early on in the project that we would continually test as the project progressed using the behavioural test specifications.

The following is an example behavioural test, using the "Given, When, Then" technique, a common Agile method of designing behavioural tests:

```
Feature: Place Orders for Matching
              Given(setup)
                 A registered Buyer.
              When(trigger)
                Buyer places order.
              Then(verification)
                Order is displayed to seller so that seller can view.
```

We chose to use this technique because web applications are typically event driven and this helped us translate our behavioural tests into site functionality.

Our unit testing process involved setting up drivers and stubs to test the core objects of the system [**?**]. The aim of this was to test the individual components of the site, the units being the various functions and objects within the models, views and controllers in our site's MVC pattern. For example when we wanted to test the functionality of a Buy Order, we would create a driver that would set up a user profile within the database, and the required stubs such as the Market and the Transaction Wizards. These tests were a good indication that our application was executing correctly in regards to object behaviour. We did encounter some difficulties when setting up some of the more complex objects, as they were largely dependent on a number of attributes being set correctly in order to execute. This was resolved by removing some of the unnecessary dependencies.

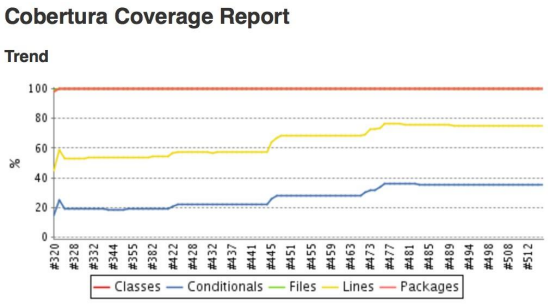
It was a difficult process to capture the application's core functionality into accep-

tance tests because the customer gave us a lot of freedom over the site interface and functionality. We were continually adapting the site to the customer's requirements after every iteration and then demonstrating the site during customer days.

We did deliver a testing process based on the user acceptance testing model (UAT). The plan was to observe general users as they interacted with the system. We prepared and conducted a study to evaluate the system, which involved gathering sample users and tasking them to explore the system and perform tasks, like registering and logging in to the site and uploading credentials, then quizzing them on how easy they found the tasks. This proved to very effective as it gave a clear indication that our application was achieving the project's usability requirements.

We aimed to cover all of the core components of the project during development, which worked quite well in the beginning, but as the number of core components expanded we found the number of required test cases increased at a much higher rate. In software testing this is regarded as combinatorial explosion [**?**]. The main reason for an increase in the number of components was partly due to requirements gathering during the customer days, as the project moved forward so did the scope. Towards Iterations 3 and 4 of the project the team decided to focus less on expanding new features of the site, and instead focus on developing a more comprehensive test suite.

r0.5



We evaluated our test suite by generating coverage reports using a code coverage tool named "Cobertura". Cobertura analyses the code base by both test coverage and cyclomatic complexity, and generates reports with new Jenkins builds, logging failures as they arise.This was very useful as it gave an indication of how comprehensive and effective our test suite was, and which areas needed improvement.

Pictured is an example graph from the Cobertura report generation: We can see that because we focused on testing towards the later iterations the test coverage

was expanding with every new build. If our customers had been more stringent about their needs and requirements we could have employed Test Driven Development, and the test coverage would have been generally constant over time. However, in a proof of concept system, changing requirements are to be expected.

Although the line and conditional coverage of the source code was increasing, the quality of the test suite could not be validated by this, as it was just an indication of how much of the code was actually being tested.

Once we started to better understand the system's behaviour on a larger scale and how components of the system depended on one another, we were able to optimize the test cases to achieve a more accurate and comprehensive test suite. Overall as a team, we gained a great deal of experience in the process of testing. We also gained confidence in our application, knowing that we were delivering a high quality bespoke piece of software.

Software Process Improvement

Throughout our project, we used Agile methods of process improvement as part of our quality assurance process. This was achieved by analysing our work and team process after each iteration in the form of a retrospective.

We decided the most appropriate person to lead our retrospectives was our project manager, who would hopefully have good general knowledge of the project and the team. Our process refinement generally followed the same pattern: a theme board. The results of our discussion were placed on either sticky notes on a wall or written on a whiteboard. Team members discussed and reached a consensus on which notes should be added to the board. As discussed in "Project Planning and Team Organisation", this theme board followed the 4 L's philosophy (things we lacked, learned, liked, and longed for).

Following the process improvement "change, measure, analyse" cycle [**?**, Chpt. 2.4], during the retrospective we asked questions related to the development process such as what, in our process, was working well and should either be continued or carried out more often. We identified and analysed faults with the process such as what wasn't working and should be stopped, along with identifying things we could start to do. We then transcribed the board onto Trac. We translated issues into tasks for the next iteration, and made sure that we had at least one person responsible for following each issue through. This continuous evaluation was extremely useful in identifying, and finding solutions to, issues that cropped up throughout the

development of AMIS.

Our retrospectives reassured us that our published change communication with the customer was effective, so we kept the format of our customer meetings the same and continued using the rolling document for published changes. We also identified that the customer was happy with our ability to listen, accept feedback and make suggestions on the features of the site, and discuss and implement new requirements for the site. Features (such as an administrative panel and flexibility in weight units) that we agreed to try and implement in previous requirements gathering sessions were not forgotten, as we documented the sessions effectively following the retrospective.

Issues with our work management process were also identified. We noted in Iteration 2 that the allocation of tasks was too laissez-faire and that led to tasks being forgotten or assumed to be covered by other team members. We adopted a more rigorous approach to ticket generation on Trac along with eventually making fuller use of Trac's ability to link to commits and tickets.

Using PyDoc for internal documentation gave us the benefit of automatic HTML generation of documentation, and also allowing the project architecture to be understood by people of varied technical ability.

As a result of demonstrator feedback, we evaluated our use of the retrospective process itself. In this evaluation we found that we focused too much on the product without giving due attention to the process needed to test and validate that it met the customer's requirements. Our primary means of communication regarding features were found to be too unstructured and informal, so we began to use Trac to allow us to have more documented and formal discussions regarding these decisions. Another theme with our process improvement was the focus of the topics within the retrospective discussion. We tended to use this only to evaluate how the meeting with the customer went. We identified this issue and geared our focus towards process improvement in Iterations 3, 4, and 5.

Our use of Retrospectives, and the lack of structured discussion caused us to introduce a new system of tickets linked with Trac forum posts (to discuss the retrospective) after the third iteration. We also created a custom report in Trac which allowed us to view all retrospective tickets at once, and created a "retrospectives" Wiki page to list actions to be taken after each retrospective. We found this worked well for the fourth iteration.

Conclusion

Throughout the course of the project, we all gained a wealth of technical experience in areas like Python, Django and web development and improved our understanding of the software engineering process and customer interaction.

We agreed that the part we learned most about was dealing with the customer: requirements elicitation and negotiation. Being computing science students, we discovered our usual approach to software engineering was very technical and we had to refocus ourselves on the product when communicating with Liz and Mike- always reminding ourselves: **"How does this feature make the product valuable to the customer?"**.

For example, we spent a lot of time ensuring the site was flexible, extensible (even by non-developers) and scalable, and considering how the customer could customise the site to include other markets of waste, or expanding the transaction process. This gave the customer maximum value for money by enabling the site to be grown and developed with minimal technical refactoring. We agreed that giving the customer a flexible system is important for software projects in general, although not too much: as we learned with the Transaction Wizard Critical Steps, the YAGNI ("You Aren't Gonna Need It") principle can avoid time, resources and money being wasted on unneeded functionality.

We compared our system with iLearn, a system developed for the Scottish Government as a Digital Learning Environment [**?**]. iLearn was also supposed to be flexible and open. The developers of this project seperated the project into "Integrated" and "Independent" services [**?**], similar to our separation of feature applications, although we didn't have a need for "independent" services (ones where there would be no direct programmatic interface between the services, relying on the user to navigate and operate between services manually).

We also found that using common design patterns in the project source code not only increased our team productivity by allowing developers to quickly pick up the intent of a piece of functionality, but also minimised time and effort implementing these functions: implementing features using regular design patterns is quick and efficient.

A third area in which we improved was testing and our use of continuous integration (CI): initially none of us had any experience with a CI system. While it was frustrating when our CI build would fail, we learned many important lessons from it, and it helped us improve our branching strategy.

Our CI technology also allowed us to analyse *how well* the site was being tested. While automated unit test coverage is not a reflection of test quality, it indicated which areas and modules of the project required more rigorous testing, and which were almost tested fully. For example, we found that our views ("controllers" in MVC) had near complete coverage, but our "forms" for user input had quite deep nested logic (which is a potential source of bugs) and relatively poor test coverage, so we shifted our testing focus accordingly.

We read in John B. Goodenough and Susan L. Gerhart's 1975 article [**?**] on Software Testing that they believed that most software errors result in failing to imagine or deal with all possible conditions of use of a program. We found that this was true for our project, the deep nested logic in our forms validation indicated that there were many potential "edge cases" we couldn't think of to test fully. In hindsight using a technique like Test Driven Development [**?**] could have helped us design code which would be easier to *comprehensively* test. This paper also recommended a systematic method of formal testing using condition tables (to ensure a fully comprehensive test suite) [**?**], a method we agreed could be interesting to try out.

All in all, we agreed we enjoyed our software engineering team project and feel that we all benefited greatly from our experience both technically and professionally. We hope that we can further develop our software engineering skills from this project and even apply our new skills in our wider professional software engineering careers.