Validation for the Vazels GUI

Unit Testing

What did we do?

Due to fact that our project was a GUI, unit testing took a back seat during development until a later stage than would normally be expected in developing a large project. The “logic” of the program is simply a series of ways to communicate one type of user input (on our graphical front-end) into another (as command-line input generated for Vazels by our software).

Use of our program is split into two parts, which we dub a “setup phase” and a “running phase.” Unit testing the functional code for the setup phase has been easy - we run a set of set-up commands and check that the configuration stored by the server matches what we expected. Unit testing the code for the running phase was more difficult, as it is difficult for us to construct test data on which to operate - the running phase analyses output from Vazels and sends it back to the user. However, having prepared a series of dumps of real test data, we have been able to make some steps towards this.

We were skeptical about spending too much time trying to get good test coverage of code in the user-facing front-end - the easiest way for us to test this is to start our program up and “see whether it looks right.” Having prepared scripts (“user stories”) to be followed when testing the UI, we have a consistent and rigorous method for manual testing.We were skeptical about spending too much time trying to get good test coverage of code in the user-facing front-end - the easiest way for us to test this is to start our program up and “see whether it looks right.” Having prepared scripts (“user stories”) to be followed when testing the UI, we have a consistent and rigorous method for manual testing.

What did it tell us?

Actually, very little - which we will discuss. The portions of our code that we were able to unit test well (the server-side code) were written in Python - a language in which it is easy to write clear, concise, and functionally correct code. Some disadvantages of Python are that there are no compile-time checks (it is an interpreted language) on things like typing (it is a dynamically typed language), and it is tempting for the coder to use built-in structures (such as “dictionaries”) which allow slightly more flexibility (and fewer checks on correctness) than if we had made custom classes to hold our data (as we would have done had we been writing in Java or C++).

Between these factors, it is easy to understand code written in Python, and easy as programmers to see what we “should” be doing, but it is also easy to make typos which often go unnoticed until run-time, although they affect the function of the program. This would seem to make unit tests more important than ever - since we don’t have a compiler checking things through for us - but in fact our unit testing caught only one such error throughout the writing of our code (specifically, data was being stored at the wrong place in a hierarchy of data, as the code author had missed out an array access that they should have made). It’s difficult to say whether this outcome was because we didn’t write good enough unit tests, or

whether it was because our code was - at a low level - simple enough that unit tests weren't required. It's worth noting, we feel, that given the time scale of our project, much of the coding was focused on writing new features, which were merged into the master once they had been manually verified, and then for the most part left alone (aside from refactoring). Perhaps over a longer time scale, there would have been many more opportunities for introducing regressions into our code - and we continue to write new tests despite their lack of success at finding errors for us so far, because we feel that they may help maintenance in the future.

General Validation

User Interface Validation

We won't discuss grey-box GUI testing separately, as to build a GUI was the aim of the project, and a feature in a GUI couldn't be said to be complete until it has been tested in as part of the running program, in as many possible use cases as possible.

User interface decisions

The aim of the project is to produce a usable graphical front-end to a command-line application. This means that the user interface was perhaps the most important obvious goal of the project - perhaps, indeed, the only obvious goal. That said, because of the nature of the application we have been designing an interface to (Vazels - a command-line tool for managing large scale testing of systems across a distributed test bed such as Planet Lab[1]). We also had to address how we could build a meaningful user interface to interact with a remote system (the main Vazels program will in many cases be situated on the same infrastructure as the test-machines - a remote host). Originally, the tool would have been controlled using SSH, so the differences between working remotely and working locally would be few; for us, however, the difference has been significant. This is because delivering a usable interface to a non-local user presents its own set of difficulties. After some discussion, we chose to go with using a web-based UI, feeling that this would deliver the most flexible interface to users without needing to configure more involved methods such as X-tunnelling (a method which allows standard user interfaces to be transmitted to remote users).

Validation

The decision to produce a web-based UI was discussed at the time with our supervisor, and his input was taken on-board during the design of the architecture. Whilst the concepts of AJAX/REST-based UI and running an interface to a service on a remote machine took a little explaining to some members of the group and to the supervisor, we feel that we eventually settled on what was probably the most practical solution.

The design of the UI itself was not the key concern during the development phase of the project. In this phase, the priority was overcoming the difficulties of interacting with a command-line service from our web-server, and of producing the front-end capabilities necessary to properly interact with our web-server. This may seem counter-intuitive, since the main aim of the project was to produce a usable GUI, but it seemed to us that a GUI would be useless if it could not properly control the

[1] http://www.planet-lab.org/

service it was designed for, and for that reason we focused on the functional aspects of our project until a relatively late stage, when we assigned a group member to produce a visual prototype of how the project might eventually look to the user.

To validate this design, we first considered it as a group, then discussed it with our supervisor, who asked certain questions about the way, for example, lists of machines involved in the experiment would scale. Following this discussion, we refined some of our user stories, and compared them to our design. The aim here was to check that users would be able to easily navigate the interface and achieve the tasks we predicted for them, and we found this method of validation gave us confidence to proceed.

Toolkits

Having decided to produce a web-based front-end, we wanted to avoid some of the challenges associated with producing efficient, professional-looking web applications. One of the problems here is that it can be difficult to produce consistent GUIs, and ECMA-script (“Javascript,” or “JScript,” depending on the browser) can be a difficult language to verify and test, because of its very dynamic nature. Whilst there exist many libraries and frameworks to help us work with ECMA-script, (such as jQuery, Dojo, or YUI) we wanted to try something new - so we decided upon Google Web Toolkit (GWT), which also includes a way to specify the user-interface design in XML. GWT allows us to write the functional code of our program in a variant of Java, which it then translates into ECMA-script optimized for different browsers. The toolkit comes as a plugin to Eclipse, so we could code our web application in a familiar IDE with many of the advantages of static type checking, code completion, and a language with which we are all familiar. This was a novel approach to web development for all of us, and whilst it may not have been the most productive (given the time it took for us to learn the toolkit), it was certainly a worthwhile learning experience.

GWT comes with some profiling tools built in, but given the nature of our application, we didn’t find it necessary to make use of them. None of the tasks performed within the user-interface are resource-intensive, mostly just being sending configuration data (or occasionally files) to the server. Whilst we tried to make sure we wrote performance-aware code, we thought that clarity of ideas and design was more important than speed, and that it would be counter-productive for us to spend time worrying about it. Additionally, ECMA-script comes with memory management, which is fast and efficient in most modern browsers, so memory profiling was not a priority either.

Code inspection

Each feature was coded on a new branch using the Git revision control system, and each branch was reviewed by at least one other team member before it was moved into the master branch. This meant that every piece of code entered into the main project had been looked at by at least two separate people. Nonetheless, some bugs slipped through, so in these cases a third person would fix the issues, and before the fixes were merged back in to the master, they too would have to be reviewed. Whilst we were aware of the time constraints involved in this, we thought

it made sense for everyone to be aware of what everyone else was doing, so this review process not only helped us catch bugs, but also improved general awareness of parts of the project that we weren't necessarily working on.

Stress testing

Our project being a front-end to an existing service, the performance limitations are mostly those imposed by the underlying systems. Points where our performance could be affected were, for instance:

- A slow network connection, impairing communication between the user's front-end and the back-end server (this is separate from the concern of bandwidth within the network where the test is actually running; that would be a limitation of the Vazels system),
- High network traffic / many concurrent requests,
- Poor server performance,
- Large file uploads - the user had to upload parts of the system under experiment to the server.

We actually subjected our solution to stress testing for only one of the above concerns, the reasons (and outcome) being in order:

- A slow network connection is an infrastructure problem, and there is only a limited amount we could do about it if we were to address this problem. Given the low volumes of traffic involved for most parts of the application, a network connection slow enough to cause us any real problems would also be one that prevented the user from, for example, using their email accounts or collaborative document tools like Google Documents - indeed, it would have to be sufficiently slow to prevent them from making any real use of the Internet at all, quite aside from administering a remote test bed. We have no intention of planning for this eventuality; the user must have a usable Internet connection in order to carry out remote testing.
- The front-end is designed to be used by one user at a time. The only source of requests should be from our web-based front-end; it would be extremely difficult for one user to generate requests too quickly for the server to handle them.
- It is difficult for us to simulate performance issues on the machine where the server back-end is running, and besides this, any performance issues affecting our server would have an even greater effect on the Vazels system it is controlling. Since the limiting factor here is the system with which we interface, we don't find it productive to address the problem on our end.
- We did test performance when uploading large files (such as a large system under test): Our server is built on top of a "Restlite" back-end - an LGPL (open source) web server implementation. This existing solution is able to handle large file uploads, although our server takes a slight performance hit when dealing with them, as we expect parts of the system to be compressed - decompressing large files is resource-intensive.

Managerial Documentation

Management tools

We used Git for version control, so that we could all work on separate or overlapping parts of the project without having to deal with conflicts. Git's intelligent and quick merging algorithm meant that solving merge conflicts was relatively painless once all members of the group got used to the tools. To begin with, any large operations in Git had to go through one member of the group who had most experience with the system - which was something of a bottleneck - but now all members of the group are able to deal with the basic functionality. This has been a useful tool in allowing us to work quickly and independently, whilst keeping track of the work we've all been doing.

We started off using a hosted solution on Project Locker[1], which provided us with a hosted Git repository, as well as a Wiki and Trac issue tracking. We originally went with this solution as it allowed us to keep our code private while making use of a fully-featured hosting option with a good issue tracker. Using Trac was a good way for us to keep an eye on outstanding tasks, as well as measuring progress, and the Wiki provided a good place for knowledge pooling (so that each member of the group could document what they'd done all in one place). However, we had some issues with the host, including high latency, and a period when our files became totally unavailable. Obviously this was a problem for us, so we took the decision to move our files across to Github, and start using the issue tracker, repository hosting, and wiki provided there. Moving between hosts mid-way through the project caused a slight issue; however, since Git is designed to work with multiple remote hosts anyway (being a distributed version control system), the transition was smoother than expected. Now that the project is being hosted at an open-source location, the need for good documentation and an up-to-date wiki is more pressing (members of the public, and indeed potential employers can potentially see our code).

Google Calendar and Doodle have both been invaluable tools for scheduling meetings and time management.

Management policies

Code review played an important part in deciding which code did and did not eventually become part of the project. Prioritising tickets on the issue tracker helped us prioritise which tasks group members spent their time on, although for the most part group members were trusted with being able to decide their priorities for themselves. Aside from the occasional reminder that deadlines needed to be met in time to demonstrate features to our supervisor, this approach worked well, and allowed people to spend time on what interested them. We set a small number of "Core goals" for each person between one meeting and the next, but not enough to use up all the time they would spend working, so that they would be free to do whatever they felt was important. This meant that, for example, Andy could spend time going back and refactoring hastily put together "deadline meeting" code, while James could spend time writing documentation which wouldn't normally have fit into our work schedule.

---

[1] http://www.projectlocker.com/

Knowledge Management

The online Wiki did not prove as valuable a tool as we might have hoped. The problem here was perhaps that there was little incentive or reminder for people to go back and update it, unless they were specifically assigned that task. This is a problem that we would each want to address if we decided to use a Wiki for knowledge management in a future project.

What we did find worked effectively was coding in small groups (two or three people), which we did on many occasions, since we all share a workspace (the Department of Computing labs). “Extreme Coding” turned into tutorial sessions on parts of the project that group members were unfamiliar with (this was actually quite positive, and a good way to learn), and answering questions on the working of code helped group members justify their design decisions (as well as flagging up issues to be returned to in the future).

Code review was also an important knowledge management tool, as it meant every team member had to be familiar with the rest of the project. Also more useful than expected were meetings with our supervisor, where group members had to explain parts of the project. This ensured that all group members had a consistent view of ideas involved.

Meetings

The following is a list of formal scheduled meetings. We had at least two informal meetings each week in which we discussed progress, blocking issues, workloads, and so on. Additionally, group coding sessions covered management topics, and these happened between certain group members on a near-daily basis, but were not worth recording - they mostly resulted in the production and assignment of tickets to the people present. We did not adhere closely to methods such as Scrum, instead picking the parts we thought would be appropriate (such as regular progress reports given to the whole group) and discarding those that were not suited to our team structure and schedules (the “Sprint” methodology didn’t make sense for us; while requirements were set from an early stage, implementation details changed regularly, and different working paces of different members meant we had to regularly adjust the work being done - on a basis much more frequent than for example a week’s sprint).

- October 14 - Initial meeting to discuss the nature of the project with Prof. Wolf. All group members attended.
- October 15 - Group meeting to discuss where we’d like to take the project & how we might go about producing a remote user interface. All group members attended.
- October 19 - Group meeting to discuss progress and thoughts so far. Discuss understanding of the Vazels system. All group members attended.
- October 20 - Meet Prof. Wolf to discuss ideas on implementation, and further discuss how Vazels works. All group members attended.
- October 22 - Meet to discuss experience with toolkits chosen, and to finalise which ones we’re working with. All group members attended.
- October 25 - Meet Prof. Wolf to finalise project key requirements. All group members attended.

- October 26 - Meet to work on Project Report One (Project Inception). All group members attended.
- October 27 - Meet to finish Project Report One collaboratively. All group members except Simon attended.
- November 1 - Meet Prof. Wolf to discuss first project report and progress. All group members attended this, and a meeting immediately afterwards to discuss onward direction.
- November 5 - Show Professor Wolf a very basic implementation of the project's "Setup Phase" (please see first report for details of how the project was split into phases). All group members attended.
- November 8 - Meet to discuss Project Report Two (Progress). All group members attended.
- November 9 - Meet to continue collaborative work on Project Report Two. All group members attended.
- November 12 - Meet to finalise Project Report Two. Simon not present - instead he gave good input remotely on the evening previous. James and Chris stayed on afterwards to finalise everything and submit.
- November 15 - Meet to discuss work for the next week. All group members attended.
- November 19 - Meet Prof. Wolf to discuss Project Report Two and progress so far. All group members attended.
- November 24 - Meet to discuss progress over previous week and future progress. All group members except Simon - who was at work - and Andy - who dialed in - present.
- November 30 - Meet to discuss progress with Prof. Wolf and discuss the schedule over the next month. All group members attended.
- December 7 - Meet to discuss difficulties with Project Report 3 (Validation). Simon, Chris, and James attended
- December 8 - Meet to discuss Project Report 3. All group members attended.
- December 10 - Meet to make progress on Project Report 3 and make progress on internal client code. All group members attended at various points in the day.

Hourly Workload

We put our weekly milestones on Mondays, so we say the week ends on the Monday, and the entries are for the work done since the previous Monday. Some technical understanding may be required for the details, although we have tried to keep it mostly understandable.

Week Ending 25/10

- Andy 2hrs: Learning how Vazels works, 18hrs: Debugging Vazels - initial problems involving infrastructure in Department of Computing's lab facilities.

- Andreea 3 hrs: Learning about Vazels. 1hr: Learning about similar tools on PlanetLab.
- James 1-2hrs: Looking for tookits, (GWT, Git, svn). 1-2hrs: Going through GWT tutorials. 10hrs: Working with Vazels & trying to run test experiments. 3hrs: Finding a hosting solution for Git and investigating issue tracking systems. Getting those set up. 2hrs: Formalising initial project aims and writing to-do lists from discussions in emails. 2hrs: Going through Output Parser code with Simon.
- Simon 4hrs: Learning to use Vazels, running experiments and understanding how outputs are formatted. 4hrs: Writing test code that could parse the Protocol Buffers provided by Vazels. 4hrs: Writing a python module that could eventually be called to provide the output as a JSON object. 3hr Following git manuals and getting written code put into the repository. 2hrs: Going through code with James.
- Chris 3hrs: Learning about Vazels. 2hrs: Looking at GWT and Git

Week Ending 01/11

- Andy 1-2hrs: Researching toolkits (Restlite, GWT), 35hrs: Writing a basic server and client, including a persistent configuration manager and authentication server side. Writing a configuration editor, local cache and AJAX request helpers client side. Also writing a proxy server for GWT to forward requests to our real server during testing (GWT has a “testing” phase which complicates SOP restrictions on modern web browsers).
- Andreea 5hrs: Researching toolkit (GWT). 1h: Researching Git (trying a tutorial) 3hrs: Investigating transactions (a Vazels concept) and their use for our project. 3hrs: Project Report 1.
- James 13hrs: Writing Project Report 1. Less than one hour: Putting administrative documents into the Git repository. 1hrs: Code review. 2hrs: Enabling starting and stopping of Vazels with basic error checking (client and server-side).
- Simon 3hrs Writing the handler for the rest server to connect with the output parsing python module.
- Chris 5hrs: Researching GWT, and installing compatible Eclipse plugin. 3hrs: Learning about HTTP Post requests, and the necessary server interaction. 3hrs Project Report One. 1hr: code review.

Week Ending 08/11

- Andy 4hrs: Solving toolkit version issue (I downloaded the toolkit three days prior to everyone else and as a result had a different version). 3hrs: Moving settings manager (client) to use a graphical UI builder. 4hrs: With James, working out who destroyed the repository and fixing it. 9hrs: Getting our testing proxy to handle multi-part form data. 3hrs: Code review.
- Andreea 4hrs: Dealing with toolkit version issues (incompatible with Andy’s). 2hrs: Further familiarising with Git. 8hrs: Coding the Allocate Groups Widget (Groups are a Vazels concept). 2hrs: Learning how to use the UI Binder (building interfaces in XML instead of procedurally).

- James 2hrs: Moving documentation into the Wiki. 4hrs: Running Vazels experiment projects and capturing output for testing by Simon. 4hrs: Merging branches, fixing merge conflicts, fixing issues with the repository. 5hrs: Designing the server-side interface to groups administration.
- Simon 2hrs: Taking others' Vazels experiment data and validating that the output module was working. 1-3hrs: Began to think through how it would be possible to display data to the user.
- Chris 4hrs: Creating a GUI which uploads workloads, using GWT. 1hr: Setting up and familiarising self with Git.

Week Ending 15/11

- Andy 1hr: Documentationdy 2hrs: Learning how Vazels works, 18hrs: Debugging Vazels - initial problems involving infrastructure in Department of Computing's lab facilities.
- Andreea 3 hrs: Learning about Vazels. 1hr: Learning about similar tools on PlanetLab.
- James 1-2hrs: Looking for tookits, (GWT, Git, svn). 1-2hrs: Going through GWT tutorials. 10hrs: Working with Vazels & trying to run test experiments. 3hrs: Finding a hosting solution for Git and investigating issue tracking systems. Getting those set up. 2hrs: Formalising initial project aims and writing to-do lists from discussions in emails. 2hrs: Going through Output Parser code with Simon.
- Simon 4hrs: Learning to use Vazels, running experiments and understanding how outputs are formatted. 4hrs: Writing test code that could parse the Protocol Buffers provided by Vazels. 4hrs: Writing a python module that could eventually be called to provide the output as a JSON object. 3hr Following git manuals and getting written code put into the repository. 2hrs: Going through code with James.
- Chris 3hrs: Learning about Vazels. 2hrs: Looking at GWT and Git
- Week Ending 01/11
- Andy 1-2hrs: Researching toolkits (Restlite, GWT), 35hrs: Writing a basic server and client, including a persistent configuration manager and authentication server side. Writing a configuration editor, local cache and AJAX request helpers client side. Also writing a proxy server for GWT to forward requests to our real server during testing (GWT has a "testing" phase which complicates SOP restrictions on modern web browsers).
- Andreea 5hrs: Researching toolkit (GWT). 1h: Researching Git (trying a tutorial) 3hrs: Investigating transactions (a Vazels concept) and their use for our project. 3hrs: Project Report 1.
- James 13hrs: Writing Project Report 1. Less than one hour: Putting administrative documents into the Git repository. 1hrs: Code review. 2hrs: Enabling starting and stopping of Vazels with basic error checking (client and server-side).
- Simon 3hrs Writing the handler for the rest server to connect with the output parsing python module.

- Chris 5hrs: Researching GWT, and installing compatible Eclipse plugin. 3hrs: Learning about HTTP Post requests, and the necessary server interaction. 3hrs Project Report One. 1hr: code review.
- Week Ending 08/11
- Andy 4hrs: Solving toolkit version issue (I downloaded the toolkit three days prior to everyone else and as a result had a different version). 3hrs: Moving settings manager (client) to use a graphical UI builder. 4hrs: With James, working out who destroyed the repository and fixing it. 9hrs: Getting our testing proxy to handle multi-part form data. 3hrs: Code review.
- Andreea 4hrs: Dealing with toolkit version issues (incompatible with Andy's). 2hrs: Further familiarising with Git. 8hrs: Coding the Allocate Groups Widget (Groups are a Vazels concept). 2hrs: Learning how to use the UI Binder (building interfaces in XML instead of procedurally).
- James 2hrs: Moving documentation into the Wiki. 4hrs: Running Vazels experiment projects and capturing output for testing by Simon. 4hrs: Merging branches, fixing merge conflicts, fixing issues with the repository. 5hrs: Designing the server-side interface to groups administration.
- Simon 2hrs: Taking others' Vazels experiment data and validating that the output module was working. 1-3hrs: Began to think through how it would be possible to display data to the user.
- Chris 4hrs: Creating a GUI which uploads workloads, using GWT. 1hr: Setting up and familiarising self with Git.n. 4hrs: Migration of UI components to XML-based specification (a major feature of GWT). 7hrs: Various minor fixes. 5hrs: Modularising server components. 5hrs: Created SSH key generation code, and got the server to save configuration automatically. 2hrs: Refactoring. 1hrs: Project Report 2.
- Andreea 2hrs: Coding the client side of the Mapping Restrictions
- James 8-9hrs: Project Report Two. 1hrs: Merging & Code review. 4hrs: updating server-side Groups management.
- Simon 6hrs: Experimenting with storing output data client-side that could be updated incrementally as data was received - eventually decided against. 4hrs: Project report 2. 1hr: Code Review.
- Chris 6hrs: Allow obtaining of list of workloads previously uploaded to the server. 3hrs Project Report Two

Week Ending 22/11

- Andy 4hrs: Refactoring & Documentation on the Wiki.
- Andreea 2hrs: reading on good practice on building GUIs. 2hrs: Sketching the design of GUI on paper. 8hrs: Creating first visual draft of GUI.
- James 3hrs: Refactoring, documentation & trivial code changes. Less than 1 hour: Redesign of global configuration management client-side. 5hrs: Updating server-side control of Vazels to monitor output and do more robust error checking/handling. 1hrs: Adding ability for the user to upload the System Under Test (a Vazels concept). 3hrs: Adding ability to properly start the server with all files extracted to appropriate directory.

- Simon 6hrs: Researching new ways to display/layout each GWT widget on the page. Implemented a tab layout.
- Chris 5hrs: Working on uploading workloads to groups - including creating singleton Workloads class to have a local store of data. 2hrs: GUI to upload actors.

Week Ending 29/11

- Andy 5hrs: Fixed a number of unnoticed regressions to do with deployment (not caught by tests due to the GWT test harness functioning slightly differently to the deployed product). 2hrs: Refactoring & improving authentication methods server-side.
- Andreea 4hrs Creating the Main Setup Menu Tab (restructuring the UI to fit with the new design)
- James 1hrs: Trivial code changes and refactoring. 1hrs: Code review.
- Simon 1hr: Code review.
- Chris 2hr: Creating a Workload manager in same style as the (since recreated) Group Manager, 1hr: Code review.

Week Ending 6/12

- Andy 12hrs: re-implementing client-side global storage for configuration data. 2hrs: Rewriting code to make better use of existing Python libraries rather than duplicating functionality. 8hrs: Refactoring the code I wrote this week.
- James 2hrs: Code review.
- Simon 4hrs: Started Output Data widget.
- Chris 4hrs: Interfacing with new Group Manager to easier upload to server

Week Ending 13/12

- Andreea 15hrs: Applying visual styling.
- James 12hrs: Project Report 3 (Validation). 5hrs: Updating server code to query running experiment for results, as well as monitor status of running hosts. 1hr: Refactoring.
- Simon 3hrs: Project Report 3. 5hrs: Building Output Data widget.
- Chris 4hrs: Uploading of workloads to groups - finish. 3hrs: Project Report 3.