The University of Auckland
Library Thesis Consent Form

Author of thesis	John Patrick Downs
Title of thesis	Ambient Awareness of Build Status in Collocated Software Teams
Name of degree	Master of Science
Date	February 2, 2010

Please tick the boxes that apply

Print Format

✓ I agree that the University of Auckland Library may make a copy of this thesis available for the collection of another library on request from that library.

✓ I agree to this thesis being photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.

Digital Format

I certify that the digital copy of my thesis deposited with the University will be the same as the final officially approved version of my thesis. Except in the circumstances set out below, no emendation of content has occurred and I recognise that minor variations in formatting may occur as a result of the conversion to digital format.

✓ I confirm that my thesis does not contain material for which the copyright belongs to a third party.

or

☐ I confirm that for all third party copyright material in my thesis I have either:
  a) obtained written permission to use the material and attach copies of each permission
or
  b) removed the material from the digital copy of the thesis; fully referenced the deleted materials and, where possible, provided links to electronic sources of the material.

Signature of Author: ______________________ Date: February 2, 2010

Comments on access conditions

Graduate Centre only: Digital copy accepted ☐ Signature Date

Ambient Awareness of Build Status in Collocated Software Teams

John Patrick Downs

Under the supervision of
Dr. Beryl Plimmer and Professor John Hosking

A thesis submitted in fulfilment of the requirements for the degree of
Master of Science in Computer Science, The University of Auckland, 2010.

February 2010

Abstract

This research explored the use of ambient devices for the purpose of monitoring build status. Within agile software development teams, particularly those employing continuous integration procedures, the build process is the most frequent opportunity to assess the source code and the system under development. Ambient devices present dynamic digital information directly within an environment in an easily perceivable manner. The separate channel provided by ambient devices ensures that users perceive their contents as separate and distinct from their standard workflow. A number of teams within the software industry anecdotally reported using these types of technologies to monitor their build processes, but no formal research has been conducted to assess their efficacy or to delineate the requirements for such a system.

We conducted a case study with a software team using a linear action research framework. An initial set of observations of the team, and interviews with each team member, found that the build process was a dominant part of the team's daily workflow. While team members recognised the importance of the builds for a variety of purposes, broken builds occurred frequently. Team members reported difficulties with their processes for becoming aware of broken builds, of determining who may be responsible for a given broken build, and in applying adequate social pressure to ensure that the builds were fixed rapidly and did not result in adverse effects.

We designed a build awareness system to address these issues. Nine specific and measurable hypotheses were constructed to evaluate the effects of the introduction of this system. Two types of ambient device were provided to the team: a central light indicated the overall state of the team's current builds, and individual Nabaztag Wi-Fi rabbits were provided to each developer to indicate when a developer may have committed code which caused a build to fail. Three prototype behaviours for the systems were evaluated.

Team members reported that the devices gave them an increased sense of awareness of, and responsibility for, broken builds. Additionally, some individual developers noted that the system dramatically changed their perception of the build process and made them more cognisant of broken builds. Despite these comments, there was no measurable change in the proportion of builds which broke following the introduction of the ambient devices. However, the overall number of builds increased substantially, and the duration of broken builds decreased. Additionally, the properties of the ambient devices – separation of channel, simplicity, glanceability, and redundancy of encoding – were validated as core components of a successful ambient awareness system.

Acknowledgements

Firstly, to Beryl and John, my supervisors – thank you for all your advice and encouragement, as well as all the practical assistance. I would also like to thank the HCI group in the Department of Computer Science at the University of Auckland, especially Andrew, Nilu, Paul, Rachel, and Samuel, for their encouragement and for being a very helpful sounding board for ideas.

I would also like to express my appreciation to the participants for this study, and to their employer for generously allowing them to spend some of their work time with me.

This project would have been significantly more difficult if it were not for the contribution of the University of Tampere in Finland. Not only did they create the excellent jNabServer, which I used extensively in this project, but they also generously loaned me two Nabaztag rabbits to use for the study. In particular I would like to thank Jaakko Hakulinen for his assistance. Similarly, I would like to thank Murray Niederer for his help with ordering and shipping several of the rabbits from London. Both provided a great deal of assistance with maintaining my sanity when all other avenues of finding “those damn rabbits” were closing! (To anyone else considering a similar project: buy all the devices you need before the company has a chance to go out of business.)

To everyone who read chapters of this thesis, or even the thesis in its entirety – particularly Andrew, David, Matt, and Mum – thank you for helping make this a better piece of work.

I would also like to thank my friends and family for their support. Matt – thank you for all your input, your encouragement, and for being there. And finally, my parents have been consistently supportive of my academic endeavours, and I truly appreciate everything they have done. Dad – I don’t know if you realise it, but you were one of the main reasons I went into this field. Your struggle with technology has made me realise how unnecessarily difficult all of this “high tech” stuff is, and over time I hope to make a small dent in that.

Table of Contents

List of Figures

List of Tables

Chapter One
Introduction

The process of developing commercial software within a team is time-consuming and complicated. A large number of subtle problems can creep into the process because of miscommunication, human error, misunderstanding of the requirements, and myriad other reasons. The sooner these problems can be detected the easier they are to fix – and the lower the cost of fixing them (Saff & Ernst, 2004).

A build process provides regular checkpoints at which a system can be evaluated against objective criteria for quality, stability, and performance. Build processes are increasingly complicated, bringing in sub-processes and information sources such as code compilation, unit testing, static analysis, and even functional and performance testing. However, they are also becoming increasingly automated, with build servers automatically performing these tasks, recording the results, and saving and distributing the build artefacts.

Because of this degree of automation, builds are able to be run very frequently. Continuous integration procedures, for example, run the build process every time any source code changes. Depending on the team's practices and development methodology, this could result in many builds occurring each day. Compared to more traditional practices of irregular or even once-daily builds, continuous integration results in a frequent sampling of the status and health of a development project and provides a regular opportunity to assess any problems or issues which may arise.

Once a build has occurred, team members must be made aware of the outcome of the build and given the opportunity to correct any issues which may have been found. However, the need for this awareness must be balanced against the other tasks that a software developer will be performing. Additionally, a number of social factors are at play in any team and must also be considered. For collocated teams, a range of techniques can be employed that take advantage of the physical context and environment in addition to the shared knowledge that team members possess.

Ambient devices present dynamic information in an at-a-glance manner and have low attentional requirements. Such devices could be as simple as a light bulb in the corner of a room which turns on or off depending on a given criterion, or as complex as a large-screen display presenting charts or multiple dimensions of information. In this thesis, the use of ambient devices is explored within the domain of build status monitoring.

The remaining sections in this chapter describe the motivation for this research as well as its objectives and contributions. Specific terms used throughout this thesis are defined in section 1.4.

1.1. Motivation

Within the software engineering and HCI research communities, attention has often been focused on promoting awareness of project and code status to developers. Biehl, Czerwinski, Smith, and Robertson (2007) constructed FASTDash, a system to passively aggregate and display information about developers' activities to each other on a large screen within their work environment. Similarly, Palantír (Sarma, Noroozi, & van der Hoek, 2003) provided developers with information about the types of code changes being made by other developers in real time, and presented this information within the developers' IDE. Both systems were predicated on the assumption that increasing awareness of other team members' activities would improve the process and output of software development.

Due to the frequency of builds and the large volume of information generated by the build process, modern software development teams are looking for the best ways to monitor these systems and to become proactively aware of issues that may be uncovered during builds. Monitoring systems can include automated emails at the conclusion of every build, publishing results to Twitter or an RSS feed, or even just the provision of a web page for team members to view build results on demand. Some teams within industry have also constructed various types of ambient hardware devices to present build information in an easily perceivable and low-impact manner. For example, Clark (2004) describes the well-known example of a lava lamp: when a build breaks, the lamp is turned on through a series of hardware controllers, and is thought to provide both practical and social cues to correct the problem as quickly as possible.

Although various types of build monitoring devices and systems have been described on industry blogs, in industry books, and in other publications, little work has been conducted to formally assess these devices and to determine whether they have any measurable impact on the practices of the team and the quality of the builds. In fact, there has been little speculation on the definition or nature of these effects, or even what these devices might reasonably be expected to do.

This research focuses on the design, construction, and evaluation of a set of ambient build monitoring devices. In particular, this project provides a framework for gathering the requirements for such devices, makes specific predictions about the effects of introducing a build monitoring system, implements a series of prototype systems, and evaluates those prototypes based on the hypothesised effects.

The project was conducted as a case study with a ten-person software team in Auckland, New Zealand, using a methodology based on the linear action research paradigm. A preliminary analysis was conducted to determine the requirements for a build monitoring system. This analysis was based on the existing literature and on observations and interviews conducted with case study team members. From these requirements a prototype build monitoring system was designed and implemented, and then evaluated and refined with the case study team over a period of several months. A range of quantitative and qualitative data were collected to explore the effects that the system had on the team's activities in the area of build monitoring and awareness.

1.2. Thesis Objectives

This thesis provides a framework to explore ambient monitoring of build status information, using a single case study software team as an example. The overall process by which these issues are explored fits within a linear action research paradigm. To achieve the overarching aim of providing this type of framework, five sub-goals were established.

1.2.1. Evaluate and Compare Ambient Device Technologies

A large number of display devices have been constructed and are in use within the area of ambient awareness. In order to identify modalities and technologies that are appropriate for use in an ambient monitoring system, the properties and constraints of these devices must be examined.

An initial goal of this thesis was to form a taxonomy of a set of devices, classifying them along a range of relevant dimensions. Based on this list these devices could be compared, and the appropriate devices selected to meet the project requirements.

1.2.2. Determine and Develop Requirements

Different software teams employ different methodologies and different strategies for managing their daily workflow. In addition, given the paucity of research in the area of build monitoring, few evidence-based best practices have been established.

An important goal of this work, therefore, was to determine a set of requirements for a build monitoring system for a specific team. These requirements were based on the team's structure and workflow, and on the issues and concerns that the team had with their existing practices and processes. While a single case study can only provide one set of requirements, which may be unique and particular to that team, over time a larger corpus of studies could be established to validate and extend the requirements to other teams and other processes.

1.2.3. Establish Measurable Hypotheses

Action research methodologies require that a project include objective and verifiable measurements to assess the success of any interventions. In the case of this project, hypotheses could be developed based on the team's requirements for a build monitoring system and on the effects that could reasonably be expected to be observed when those requirements are met.

A set of nine hypotheses was generated based on the requirements gathering phase of the study. The hypotheses are specific and can be assessed using concrete and objective data that can be readily derived from build server logs. The hypotheses are used to evaluate the effects of the system and to evaluate its success in addressing the issues and concerns that the team members had previously experienced with builds and build status monitoring.

1.2.4. Design and Implement Prototype System

In order to explore the possibilities of ambient build monitoring systems, such a system must be designed and implemented. The process of designing and implementing a system requires a number of hardware, software, and behavioural decisions to be made. Drawing on the comparison of the ambient devices described above, appropriate hardware devices were selected based on the requirements of the case study team. Additionally, software components were constructed to interact with the team's build server as well as to perform the logic and processing required to transform the build log data into ambient device output.

In this project, two separate classes of ambient build monitoring device were planned and constructed. As the project evolved the behaviour of the devices was changed to allow for the team's work practices to be more closely followed and to compare multiple types of behaviour.

1.2.5. Evaluate Prototype against Hypotheses

Finally, once the prototype had been designed and implemented, it could be evaluated and measured according to the hypotheses previously described. In this project, the case study team was provided with ambient build monitoring devices for a period of several months. During this time a number of quantitative metrics were calculated, and these metrics were then evaluated against the research hypotheses. Additionally, qualitative information was obtained both through informal conversations with team members and through semi-structured interviews towards the conclusion of the study.

1.3. Thesis Outline

Chapter Two provides an overview of relevant previous research. First, the areas of software development processes and the types of information used by professional software developers are explored. This is followed by a summary of some previous studies intended to understand and improve the ways in which software team members communicate and stay aware of relevant information. This chapter also describes the principles of ambient awareness and lists numerous ambient devices and technologies which can be adopted when presenting information in an ambient form.

Chapter Three describes the methodology used in this study. Specifically, the underlying principles of case study research and action research are discussed, as is the decision to use this type of methodology for this study. Next, the chapter outlines the steps followed to perform the study, including the list of the individual components of the study from the initial preparation through to the data analysis.

In Chapter Four, the two initial study components are described. First, an observational component was conducted to gather information about the ways in which the members of the case study team interacted and performed their daily work. Next, in-depth interviews were performed with each team member to gain more information about these topics and others. Chapter Four outlines the ways in which these study components were designed, and describes the results and analysis performed on the data.

Chapter Five builds on these results, as well as the literature described in previous chapters, and outlines the core requirements that an ambient build monitoring and awareness system should meet if it is to successfully address the issues and concerns outlined by the team members. This chapter also discusses potential technologies and practices that could be used in such a system. Finally, the chapter describes some concrete and testable hypotheses for changes in team behaviour that can be objectively observed following the introduction of an ambient build monitoring and awareness system, and can be used to judge its success and impact on the team.

The implementation used in this study is described in Chapter Six. A series of hardware and software components were required to enable the appropriate information to be obtained from the build server, to be manipulated and aggregated, and then to be provided back to the development team in a useful form. This chapter outlines the steps followed to implement each of these components and describes the overall architecture of the system.

Chapter Seven outlines the evaluation of this system. The evaluation was conducted along several dimensions. First, a series of metrics were calculated to test each of the hypothesis outlined in Chapter Five. The methods used to calculate these metrics, and the results of these calculations, are provided. Next, a set of qualitative feedback was provided by participants both in the form of communications with the researcher during the study and in follow-up interviews towards the end of the study. This information is also provided in Chapter Seven.

Chapter Eight discusses the implications of the results, including a discussion of their support for the hypotheses described in Chapter Five and a comparison of the devices and prototype behaviours. Finally, Chapter Nine concludes this thesis and provides some suggestions for future work in the area of ambient build monitoring and awareness, and ambient awareness more broadly.

1.4. Definition of Terms

Table 1. Definition of terms.

Term	Definition
Ambient device	Any device intended to provide information within an environment in an unobtrusive manner, and which does not require attention to be paid to it in order for that information to be perceived.
Ambient display	An ambient device that presents information on a monitor, screen, or other display, as opposed to through LEDs, movement, etc.
Awareness	"An understanding of the activities of others, which provides a context for your own activity" (Dourish & Bellotti, 1992).
Broken build	A build which has failed for any reason, such as invalid source code or a unit test failure.
Broken build chain	A contiguous series of broken builds within a plan, terminated by the next successful build.
Bug	A type of work item that represents a specific problem or defect in a system.
Build	A single instance of a build process. A build process may include some or all of the following actions: checking out source code; validating source code; compiling source code; running unit tests; running other types of tests.
Build server	A software system responsible for managing builds, plans, projects, and build artefacts.
Changeset	The set of source code changes associated with a build.
Continuous integration	A development practice in which a build is automatically triggered whenever a source code change is made and committed to the repository.

Term	Definition
Glanceability	A property of a device which describes that information presented through that device can be perceived at a glance.
Issue	See *Bug*.
Plan	Within Atlassian Bamboo (Atlassian Pty Ltd, 2009c), a plan is a representation of a source code tree and associated metadata specifying the frequency and settings which should be used to build that code.
Project	Within Atlassian Bamboo, a project is a logical grouping of plans for administrative purposes. More broadly, a project is a set of work performed by one or more people in order to achieve a specified aim.
Repository	A software component which manages a set of source code artefacts and provides access to these artefacts to team members.
Situated display	A device that presents information within a physical environment through the use of a monitor, screen, or other display.
Work item	A feature, bug (issue), requirement, or other actionable task to be performed on a system by a developer or other software team member.

Chapter Two
Related Work

This chapter examines the literature relating to ambient and passive awareness of software project information. Section 2.1 describes the literature on software development processes, and section 2.2 discusses the types of information that software teams typically use when developing a software product. Section 2.3 describes previous work conducted to understand and enhance the communication and awareness of information within software teams. Finally, section 2.4 examines the myriad technologies and methods that can be used for ambient awareness, both within software teams and more generally, and presents a taxonomy of various devices that have been developed for this purpose within multiple domains.

2.1. Software Development Practices and Processes

Software development has evolved from a collection of ad hoc activities to a set of formalised processes and evidence-based procedures. Different sets of procedures emphasise different types of information and require team members to have varying levels of awareness of that information. This section discusses the relevant literature in the area of software development processes and best practices, particularly the move towards agile software development and the associated test, build, and feedback processes that support agile methodologies.

2.1.1. Prescriptive and Agile Methodologies

A wide body of literature outlines the various software development processes that are in use within the software industry. Pressman (2005) summarises the overall direction of the software engineering community and notes that most of the methodologies in wide use fit into two distinct categories: classical prescriptive methodologies and agile methodologies.

Prescriptive methodologies, also known as plan-based methodologies (Ceschi, Sillitti, Succi, & De Panfilis, 2005), are the more traditional processes for software development. The waterfall model prescribes a set of well-defined phases for software development: requirements gathering, design, development, testing, and deployment. A project moves through these phases in a sequential manner, and each phase builds upon the information and artefacts produced in the preceding phases (Pressman, 2005). The spiral model is an iterative process in which a project is divided into a series of sub-projects, with each sub-project going through the same general phases as in the waterfall model.

Agile models were designed largely as a reaction to the perceived inefficiency and bureaucracy of prescriptive models (Sommerville, 2007). While a number of distinct agile development methodologies are used within industry, there are several general principles that apply across the range of agile methodologies (Beck et al., 2001). In particular, agile development models are designed for flexibility and rapid development, with an emphasis on keeping developers continually aware of the project's changing status and giving team members the capability and mandate to adjust their actions according to these changes.

A number of methodologies have been developed to provide guidance for implementations of agile processes. Common examples of agile methodologies include eXtreme Programming (XP) (Jeffries, Anderson, & Hendrickson, 2001) and SCRUM (Beedle, Devos, Sharon, Schwaber, & Sutherland, 1999; Schwaber, 1995). Depending on the specific methodology in use, teams will emphasise different aspects of the software development process and may conduct tasks in different ways or in a different order. For example, SCRUM mandates that developers work within relatively short 'sprints', each corresponding to an atomic set of deliverable functionality. In XP, team members work in pairs to write code, and a strong emphasis is placed on testing that code early and frequently. The following section describes various testing practices employed within XP as well as in other methodologies.

2.1.2. Unit Testing, Regression Testing, and Test-Driven Development

A common strategy for software component verification is unit testing. This involves a developer writing code to exercise and evaluate a software component, such as a method, object, or other code which can be tested in an atomic manner (Pressman, 2005). The test code documents the expected output of the component and compares it with the observed output. If the actual output is equivalent to the expected output the test is considered to have passed; otherwise, the test has failed.

Unit testing can be integrated into developer workflows in a number of different ways (Pressman, 2005). Developers can construct a library of these unit tests and manually execute them at various times during the coding process. An advantage to this approach is that as new tests are added to the library, the overall proportion of the system tested by the unit test library will increase. When developers make changes to existing code, or refactor or otherwise rework the system, they can re-execute the unit test library and verify that their changes have not caused any existing functionality to break. In this way, unit tests are used as a common form of regression testing.

Test-driven development (TDD) is another approach to unit testing and is particularly popular among XP and other agile developers (Beck, 2002; Beck & Andres, 2004). In the TDD approach, developers write unit tests before implementing a function or otherwise making a change to the primary codebase. Once the test has been written it can be executed and observed to fail, thereby establishing a known baseline point. Only once this baseline is established will a developer begin to write the code to actually implement the function or change. Following this implementation the developer will again run the unit test, verify that it passes, and finally may run the full suite of regression tests before considering the change to be complete.

Nagappan, Maximilien, Bhat, and Williams (2008) list a number of advantages of TDD and of unit testing in general. Specifically, the process of writing unit tests as part of the main development workflow becomes part of the design process for the component, and also implicitly documents the functionality of the system as it is actually implemented. The suite of unit tests becomes invaluable when developers are required to modify existing code – if the unit test coverage is sufficiently high (i.e. a majority of the system code is tested by unit tests), developers have the confidence to make changes, knowing that any regression errors will be detected quickly by the test suite.

2.1.3. Software Configuration Management and Build Processes

Irrespective of the development and testing methodologies used by a team, a software configuration management (SCM) system will often be used to manage the source code, control changes, and enforce team policies and processes. A single SCM system may be comprised of multiple independent components, potentially integrated together in some way, each performing a specialised task. For example, a source code repository such as Subversion (CollabNet, 2009) can store and manage source code artefacts; a build server such as CruiseControl (ThoughtWorks, 2009) can build the source code into executable code; an issue tracking system such as Trac (Edgewall Software, 2009) can provide a centralised location for team members to store, process, and analyse work items; and a dependency management tool such as Maven (Apache Software Foundation, 2009) can be used to automatically resolve version conflicts resulting from dependencies between different libraries and external frameworks.

A key part of software development is the build process. This process is ultimately responsible for translating human-readable source code into binary executables. In team environments a build server will often be used to provide dedicated resources for the build process. The build server will generate and store build artefacts and reports, and will enable tracking of a project's complete build history over time. Depending on a number of contextual factors, a given build will be considered successful – indicating it meets a minimum set of standards relating to code quality and functionality – or a failure (Fowler, 2006).

Developers can use a number of strategies to minimise the likelihood of build failures. Continuous compilation is one such strategy. This technology, a feature of many modern integrated development environments (IDEs) such as Eclipse (Eclipse Foundation, 2009), builds code silently and in the background of the developer's work environment, enabling rapid feedback in cases of coding or syntax errors and reducing the need to invoke the compiler and build process manually in order to find these errors.

Continuous testing is another strategy. Saff & Ernst (2004) constructed a system for continuously running regression tests on developers' workstations. This system enabled rapid feedback on the impact of code changes, and in an experimental evaluation was found to significantly affect the success rate on a programming task. The authors cited a number of benefits to frequent testing:

The longer a regression error persists without being caught, the larger its drain on productivity: when the error is found, more code changes must be considered to find the changes that directly pertain to the error, the code changes are no longer fresh in the developer's mind, and new code written in the meanwhile may also need to be changed as part of the bug fix.

In both continuous compilation and continuous testing procedures, Saff & Ernst (2004) argue that the goal is to minimise the duration of a problem. The duration of a problem can be split into the ignorance time (the time from the introduction of the problem to the developer becoming aware of it) and the fix time (from when the developer becomes aware of the problem to when they correct it). Continuous compilation and continuous testing aim to decrease the ignorance time, thereby decreasing the overall duration of the problem.

Continuous integration (CI) is another procedure intended to provide frequent feedback (Fowler, 2006). CI systems ensure that whenever code is checked into the source code repository the entire project is built as soon as practicable. Like the other procedures, CI is intended to decrease the ignorance time – although CI acts on the entire codebase rather than on an individual developer's changes.

CI systems often run a number of other tasks as part of their automated build process in addition to source code compilation. These may include automated unit testing, static analysis, dynamic analysis, profiling, load testing, and a number of other processes (further described in section 2.2). If any of these tasks fail the build is considered to have failed. Teams employing CI systems will typically consider a failing build to be their highest priority to fix (Holck & Jorgensen, 2004): until the build passes, the system is considered to be in an inconsistent state. Because of these properties of CI it is a popular technique among agile teams to ensure that the project remains in a stable state (Bowyer & Hughes, 2006; Duvall, Matyas, & Glover, 2007).

Holck and Jorgensen (2004) summarise the primary advantages of CI. They argue that CI provides motivation to team members (since they can see a working system, including their own individual contributions, on a regular basis), and also that CI reduces the risk of an integration requiring large-scale fixes in order to successfully complete, since these integrations occur frequently and therefore have smaller changes. Karlsson, Andersson, & Leion (2000) describe an example use of a daily build process and cite the advantages of receiving regular feedback, isolating errors with builds, and allowing testers to work with fresh builds on a frequent basis. CI systems will typically build code far more frequently than once per day, so these advantages can be reasonably expected to be multiplied.

Despite these advantages, little work has been conducted to quantify the benefits and effects of employing CI procedures. Kim, Park, Yun, and Lee (2008) describe their experience implementing a CI system as part of a broader set of integration and quality control procedures; however, they do not describe the evidence leading them to select CI. Ebert (2001) describes a case study of sixty projects in which CI was introduced as part of a series of interventions to improve validation processes and practices. Following their interventions, they found that the number of defects was reduced, and those remaining defects were detected earlier. However, their study was correlational in nature and given the number of simultaneous interventions it is difficult to isolate the effects of CI alone. Despite this lack of published evidence for its efficacy, Fowler (2006) argues that CI is a fundamental part of modern software development procedures.

2.2. Sources of Information in Software Development Processes

At any stage within the software development process, irrespective of the methodologies in use, a software team will have collected a large set of information describing the project requirements and state, as well as artefacts such as source code.

This section outlines the main types of information that are collected with regards to the project state. Section 2.1.1 examines the purely technical information sources, such as the build artefacts and statistics, and section 2.2.2 outlines the use of knowledge management systems such as work item tracking. Section 2.2.3 describes some work that has attempted to look at the social metrics that can be calculated from software development project, correlating these metrics with other aspects of the project in question. Finally, section 2.2.4 describes the usefulness of overall project status metrics and of maintaining a regular awareness of the project status.

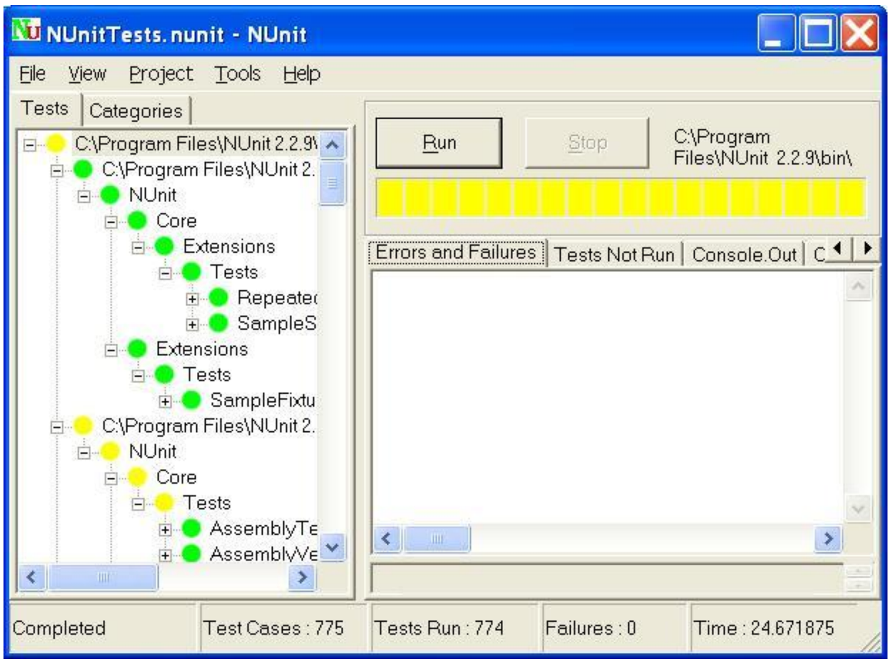
2.2.1. Technical Information

Technical information within a software project is the information which relates to the functionality of the end product. This information can be derived directly from the source code, specifications, and other primary artefacts, or from the output of various tools which operate over the artefacts. Table 2 lists some examples of technical information sources within software development projects.

Table 2. Examples of technical information derived during the software development process.

Information Source	Summary
Unit testing	The creation and execution of test code by developers to evaluate small pieces of functionality within an application.
Code coverage analysis	The process of monitoring the execution of unit tests to ensure that every component and line of code is adequately tested.
Customer acceptance testing	Similar to unit testing, but tests are designed to be written by end users or customers to ensure that the functionality behaves as expected.
Profiling	Monitors the execution of a running application to assess its use of system resources such as memory and CPU time. Often used as part of a performance improvement strategy.
Smoke testing	Simple functional testing to ensure that a build meets a minimum set of requirements. Can be manual, partially automated, or entirely automated.
Static analysis	Examines source code, usually to assess compliance with best practices, naming conventions, organisational requirements, and so forth.
Dynamic analysis	Monitors the execution of a running application to assess its behaviour or perform runtime tests.
Load testing	Simulates high levels of activity on a system to test the system's capability to handle high load.

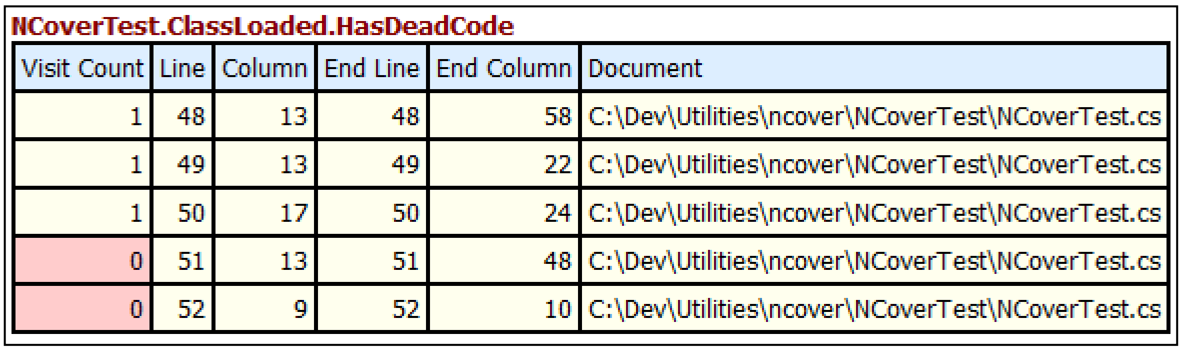
The previous section described unit testing and noted that it is a common technique employed by software developers to ensure their code is functionally correct. Dedicated unit testing tools such as NUnit (NUnit.org, 2009) will result in output similar to Figure 1, showing a view of the tests which have passed, failed, or not run. The test results can be directed to a file or streamed into another application for further processing.



```
Completed    Test Cases : 775    Tests Run : 774    Failures : 0    Time : 24.671875
```

Figure 1. Example unit testing output from the NUnit unit testing tool (NUnit.org, 2009).

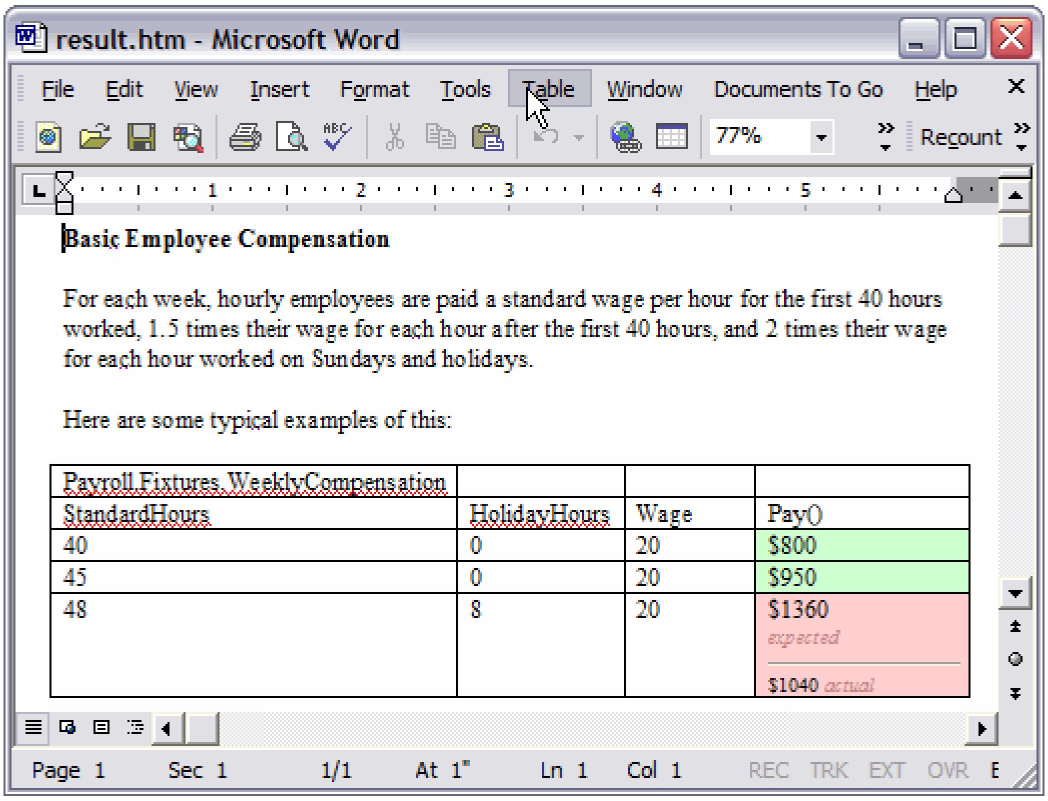
While unit testing can be extremely useful to ensure that specific functions and sets of features within a software application work correctly (Peled, 2001), if only a small proportion of the system is tested in this way, the overall product may not be adequately tested. Code coverage analysis tools such as NCover (2004) monitor the execution of the tests and detect whether branches of code have been executed or not; based on this information, a precise report (similar to Figure 2) can be generated listing the components, or even lines of code, which have not been 'visited' by the unit testing framework and therefore have not had unit tests written for them. As with unit test reports, code coverage reports will often be transferred to a machine-readable format such as XML, or transformed to a human-readable report.


NCoverTest.ClassLoaded.HasDeadCode

Visit Count	Line	Column	End Line	End Column	Document
1	48	13	48	58	C:\Dev\Utilities\ncover\NCoverTest\NCoverTest.cs
1	49	13	49	22	C:\Dev\Utilities\ncover\NCoverTest\NCoverTest.cs
1	50	17	50	24	C:\Dev\Utilities\ncover\NCoverTest\NCoverTest.cs
0	51	13	51	48	C:\Dev\Utilities\ncover\NCoverTest\NCoverTest.cs
0	52	9	52	10	C:\Dev\Utilities\ncover\NCoverTest\NCoverTest.cs

Figure 2. Example code coverage output from the NCover tool (NCover, 2004)

Another type of testing that is useful in some development projects is customer acceptance testing. While similar in principle to unit testing, customer acceptance testing frameworks such as Fit (Cunningham, 2007) are designed to be used by end-users or customers rather than developers. Users create test cases using common tools such as word processors and spreadsheets, and the testing framework executes the test cases and presents the results in a format similar to unit testing reports (Figure 3).



Basic Employee Compensation

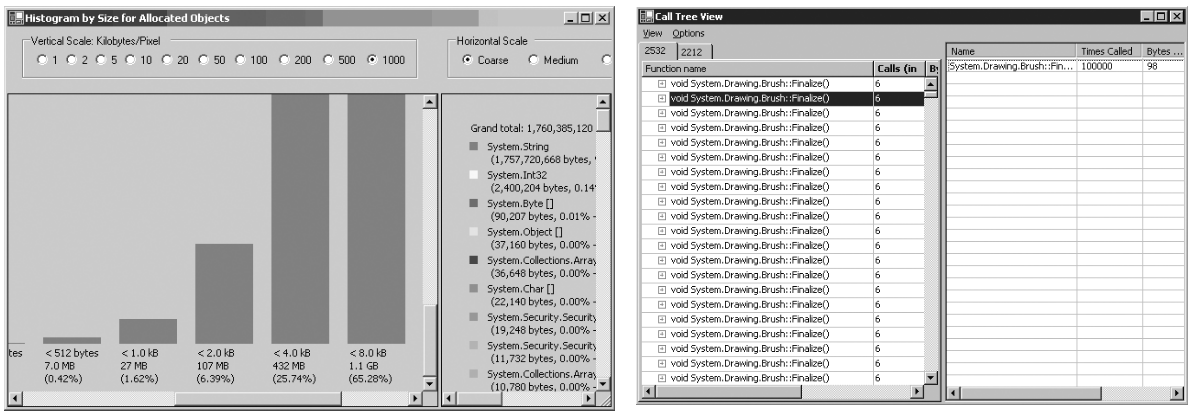
For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 *expected* $1040 *actual*

Figure 3. Example customer acceptance test format (Cunningham, 2007)

In addition to tools which test functionality, software engineering processes typically include tools to analyse the performance of a system under development. Code profilers monitor the execution of an application and measure the memory and CPU usage, the memory allocation behaviours, and other low-level details relating to how the software is executing (Eick & Steffen, 1992). Profilers often provide a series of reports and charts with detailed information about the profiled application (for example, Figure 4). Developers can then work to optimise the application if necessary.



```
Grand total: 1,760,385,120
System.String
  (1,757,720,668 bytes,
System.Int32
  (2,400,204 bytes, 0.14
System.Byte []
  (90,207 bytes, 0.01% -
System.Object []
  (37,160 bytes, 0.00% -
System.Collections.Array
  (36,648 bytes, 0.00% -
System.Char []
  (22,140 bytes, 0.00% -
System.Security.Security
  (19,248 bytes, 0.00% -
System.Security.Security
  (11,732 bytes, 0.00% -
System.Collections.Array
  (10,780 bytes, 0.00% -
```

Function name	Calls (in	B
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	
void System.Drawing.Brush::Finalize()	6	

Name	Times Called	Bytes ...
System.Drawing.Brush::Fin...	100000	98

Figure 4. Example output from the Microsoft CLR Profiler (Meier, Vasireddy, Babbar, & Mackman, 2004)

Finally, smoke testing is a technique used to verify a build meets a minimum set of criteria. Typically this will include successfully booting the application and running a series of basic functional tests to ensure the application is usable and in a consistent state. Smoke testing can be conducted with varying degrees of automation. If it is entirely automated, smoke testing may be employed as part of a continuous integration procedure in order to verify each build is in a state where it can be further tested or demonstrated (McConnell, 1996).

2.2.2. Knowledge Management Systems

The process of developing software requires a great deal of coordination between different stakeholders. Knowledge management systems are commonly used to track work items, goals, project news and updates, and as a repository for specifications and other project documentation (Pressman, 2005).

Work item tracking is a particularly common type of knowledge management system used with software projects. In these systems, a central repository (such as Trac (Edgewall Software, 2009)) stores information about customer requirements, bugs, enhancements, and feature requests. These work items can be assigned to developers and their status tracked over time. Additionally, as noted in section 2.1.3, if work item tracking is integrated with other systems (such as source code repositories), commits of source code can be linked to a feature request or bug report, enabling advanced project status tracking and reporting.

2.2.3. Social Communication

Commercial software development is almost exclusively conducted as a team exercise. Whether the teams are collocated or distributed, team members require regular contact in order to coordinate activities and plan work.

Within a distributed team, Wolf, Schroter, Damian and Nguyen (2009) studied the intra-team communication which took place through the comments placed on the work item tracking system and found that the more communication that occurred during a development iteration, the more likely that iteration would result in a successful build. In principle, this type of metric could also be calculated for collocated teams and included as a project status metric.

2.2.4. Project Status Information

A final general category of information used within software teams is project status. Each organisation will have different metrics to track the status of a software development project; fundamentally, however, project status metrics could be considered to measure the team's ability to deliver the project's functionality on time and within budget.

A number of different project status metrics and tools can be used. Prescriptive models suit the rigidity of Gantt charts, which track the critical path of a project and can measure the likely impact of change requests and unanticipated developments. Within agile teams, and especially teams employing the SCRUM methodology, Sutherland (2001) argues that burn down charts are more useful. Burn down charts compare the estimated sprint completion time with the actual sprint completion time, and can be tracked throughout the sprint. Ideally a burn down chart will display a downward-sloping straight line, indicating steady progress towards the end of the sprint and the completion of the sprint's goals.

Proactive monitoring of a project status is a critical part of software development. Keil & Robey (2001) studied a variety of software projects and found that team members typically did not want to 'blow the whistle' on a failing or off-track project, even though this would be valuable information. They concluded that "*better communication of project status, especially bad news, can reduce the incidence of software runaways*".

2.2.5. Summary

This section has described the wide variety of information which is used by software development teams as they go about their work. Some of this information – such as test results, code metrics, and work item information – is a standard part of a software developer's toolkit, while others – such as social communication measures and overall project status metrics – are less commonly used.

Every organisation and team will place different value on each type of information. For distributed teams, social communication is likely to be an important area for team members to specifically focus on, while for collocated teams it may occur naturally and thus not warrant additional attention. Different project methodologies will also emphasise different types of information. For example, test-driven development requires that unit tests be used extensively as part of the developer's daily workflow, while a team adopting a more traditional waterfall model might instead be concerned with the degree to which the code conforms to an original specification.

2.3. Awareness and Coordination within Software Teams

With so many distinct sources of information and so many parallel activities being conducted, it is important for team members to coordinate their efforts, avoid unnecessary duplication, and reduce conflicts between their work (LaToza, Venolia, & DeLine, 2006). Staying aware of the status of the team and the project is a critical part of this.

Within the fields of computer-supported cooperative work and software engineering a number of studies have examined the ways in which software teams stay aware of relevant information and coordinate their activities accordingly. An overarching theme of this work is that awareness and coordination are inextricably linked: in order to coordinate activities and work, team members must have awareness of the activities of other team members and of the project as a whole. This section summarises the previous work in the area of software team awareness and coordination, including interventions designed to improve aspects of these behaviours.

According to Dourish and Bellotti (1992), awareness can be defined as "*an understanding of the activities of others, which provides a context for [one's] own activity*." Multiple types and forms of awareness have been posited, and Cadiz, Fussell, Kraut, Lerch, and Scherlis (1998) distinguishes between two broad categories of awareness. Active awareness is purposeful and conscious, such as asking specific questions to elicit information (e.g. Sillito, Murphy, & Volder, 2006), while passive awareness is distinct in that it "*involves keeping track of events of interest without making a conscious effort to do so*" (Cadiz et al., 1998). Gutwin and Greenberg (2001) make an analogous distinction between intentional and incidental awareness in their Workspace Awareness Framework.

Intentional or active awareness of information is an important aspect of teamwork. Curtis, Krasner and Iscoe (1988) argue that one of the major challenges in the software design process is the "*constant need to share and integrate information*", and suggest that "*the communication necessary to develop a shared vision of the system's structure and function, and the coordination necessary to support dependencies and manage changes on large scale projects are team issues*". Despite the fact that their study was conducted in 1988, these issues are still concerning software teams and researchers today. For example, in a case study of a global software team, Damian, Izquierdo, Singer, and Kwan (2007) found that a broken build could be traced directly to communication breakdowns between team members.

More broadly, Gutwin, Penner, and Schneider(2004) studied how distributed software teams communicate awareness information and found that a level of commitment was required on the part of the developers to ensure the communication channels continued to be used and that the information was exchanged frequently. In studying the various communication modalities in use within three open source projects, such as mailing lists, commit logs, and chat, they noted that the time required to communicate using each modality substantially affected the degree to which that modality was used.

Incidental or passive awareness is an alternative awareness strategy that does not rely on purposeful and intentional communication. According to Gutwin & Greenberg (2001), one of the primary characteristics of awareness is that it is often a secondary goal, with team awareness information being collected incidentally as part of a more broad task. Because of this, team members do not set out to gain awareness of a system, project, or situation, but develop this over time without any specific purpose or intention to do so.

In an attempt to use these principles to enhance the awareness and coordination of team activities within a case study team, Biehl et al. (2007) constructed FASTDash. Information from software team members was passively collected within their IDE, aggregated within a central middleware server and database, and visualised on a large screen situated within the developers' physical work environment. FASTDash was intended to display a visual representation (Figure 5) of the source code files opened by team members, the specific parts of the file they were working with, which files they had checked out from the repository, and whether there were potential conflicts between two developers' work.

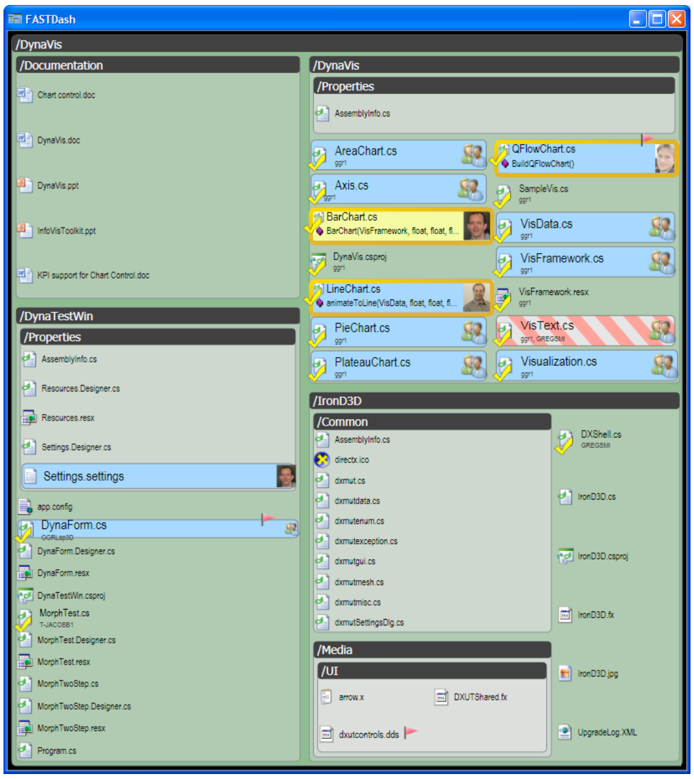


Figure 5. FASTDash user interface (Biehl et al., 2007).

Similarly, Palantír (Sarma et al., 2003) extended an SCM system in order to provide developers with information about the artefacts used and changes made by other developers on the team. As developers worked on their own tasks, they received notifications if another developer began making changes which might affect their work and were presented with an estimation of the impact that these changes may have. This information was presented unobtrusively within the developer's workspace, to avoid unnecessary distractions or otherwise drawing attention from the primary task of writing code.

2.3.1. Summary

Although each of these studies focused on different tools, different sources of information, and different ways of communicating or visualising that information, there is a common recognition that within a team environment it is critical for team members to be aware of each other's activities, and of the activities of the team as a whole. In addition, real time or near-real time feedback presents substantial advantages over post-hoc reporting (Biehl et al., 2007). Awareness is a core mechanism used by teams to dynamically adapt and coordinate work between team members, and ultimately to improve the quality of the team's work.

2.4. Ambient Awareness and Situated Displays

Ambient awareness is strongly related to a shift from ‘technology use’ to ‘technology presence’, described by Ross (2007). Rather than requiring conscious and purposeful use of a device, ambient technologies are designed to be directly or indirectly available within a physical environment, providing awareness information in a background and unobtrusive manner. Additionally, the mere presence of a technological artefact within an environment conveys some contextual information about the artefact, its purpose, or its context of use. These concepts fit within the pervasive and ubiquitous computing paradigm, and other ‘beyond the desktop’ paradigms.

This section discusses the literature within the ambient awareness and situated display fields, first by examining the general principles that form the basis for these technologies, and then by describing some previous studies which have used ambient awareness and situated displays within a variety of contexts. Finally, this section concludes with a discussion of some previous projects that have been conducted within software teams to provide build status monitoring information using these technologies.

2.4.1. Ambient Devices and Situated Displays

Ambient devices are one general category of device specifically intended to provide visual representations of digital information within a physical environment. Typically the visual representations are simple and easy to perceive. Ambient devices are designed as a form of calm technology, described by Greenberg (1999) as:

‘calm’ [technology]... recedes into the background, becoming almost invisible unless some event attracts attention to it or when a person consciously decides to bring it into the center of their attention.

A clock is an example of such a device, fading out of attention for the most part, but presenting information in an easy-to-perceive form at a glance when necessary (Sellen, Eardley, Izadi, & Harper, 2006). Greenberg (1999) provides a series of design guidelines and conceptual issues for consideration when implementing calm technologies. These guidelines emphasise the importance of selecting an appropriate type of device for the message, and of positioning the device within an environment in a meaningful way. Ambient devices convey information through their programmable parameters (LEDs, screens, and so forth) as well as the context and environment in which they are used.

Situated displays fit within the same general category as ambient displays, but do not necessarily include the ‘calm’ element that ambient devices typically possess. A situated display is an electronic display placed within an environment, but differs from an ambient device in that it may present detailed information which requires a great deal of attention to be perceived or used. While both types of device take advantage of their physical context and will become embedded into an environment (White, 2008), situated displays will be more likely to contain in-depth information for consultation or analysis while ambient displays will convey simple information which can be easily perceived, even in the absence of direct and overt attention.

The previous section described the difference between active awareness and passive awareness (Cadiz et al., 1998). Within situated display technology, a similar differentiation can also be used to distinguish ambient devices from non-ambient devices. Situated displays as a general category will provide digital information to a user, but only ambient displays – a subset of situated display technologies – do so in a passive and non-invasive manner.

In their discussion of the use of passive awareness within distributed work teams, Cadiz et al. (1998) provide a set of guidelines for the design of passive displays. These include asynchronicity (avoiding presenting information as part of the user's main workflow), proportionality (when visualising data, more important information should be physically larger), and aggregation (providing high-level summaries of information).

Ambient devices fit these criteria well. They are specific and separate devices, and thereby avoid presenting information to the user directly - instead allowing the user to pay attention to the device when they are able to do so. Their visual parameters can be manipulated, and they are typically designed to provide simple visual information that can act as a summary or overview of more detailed information. Shneiderman's information seeking mantra (1996) is "*overview first, zoom and filter, then details on demand*". Ambient and situated display devices can provide an overview of a situation, system, or information source. Depending on the context of use this may be sufficient; if not, further exploration of the data can be conducted.

2.4.2. Feedback on Dynamic Systems

Regardless of the type or amount of attention required to perceive information presented through a digital display, these devices are ideally suited to provide real-time feedback and status information relating to dynamic, changing systems and data. Clarke, Hughes, Roucefield, and Hemmings (2003) describe the awareness processes in use in a hospital ward with regards to monitoring bed usage, and conclude that any electronic system to augment or replace existing procedures should account for the ever-changing nature of the hospital environment. Within the domain of software teams, Biehl et al. (2007) note that the information most frequently used by developers is also the most frequently changed, and argue that the development of situated electronic displays, updated dynamically and in real time, is necessary to capture and provide feedback on the ever-changing process of software construction.

Dourish and Bellotti (1992) introduced the principle of shared feedback – that is, feedback information passively collected without any additional effort or reporting required on the part of team members, and presented to the team within a shared workspace. To Dourish and Bellotti, a shared workspace consisted of synchronised GUI applications running on individual users' workstations, essentially forming a logically shared workspace within physically individualised devices. However, when dealing with collocated teams, situated displays and ambient devices provide a physical manifestation for such a workspace and allow team members to passively share feedback information using a single shared physical display.

2.4.3. Taxonomy of Ambient and Situated Display Technology

A large number of projects have explored the use of ambient awareness and situated display technology in a number of different domains. These projects have used a variety of different devices, form factors, and behaviours. This section provides a taxonomy of a representative sample of these technologies.

Each technology has been classified along a number of dimensions (Table 3). The dimensions were selected to provide an assessment of the most important aspects of the technology relating to their ability to be used for ambient awareness. Many classifications within the dimensions are subjective and are provided on a continuum, but are intended to enable a general comparison of these devices. The taxonomy of devices is provided in Table 4.

Table 3. Dimensions used for categorising ambient device technology.

Dimension	Description
Visibility	The ease by which the device is able to be viewed within a physical environment. Devices which are large, highly luminous, centrally located, or very obvious are considered to have high visibility. Personal devices (such as those on a person's desk), and devices which are small, dim, or present information in a subtle way are considered to have low visibility.
Degree of Attention	The amount of attention that must be expended in order to perceive the meaning of the information that the device is presenting. A device which has a single function, or which has its function made obvious by its context or identity, is considered to have a low degree of attention required. A device which requires the user to examine its contents, browse through multiple

Dimension	Description
	levels of information, or perform complex cognitive operations such as mapping between domains, is considered to have a high degree of attention required.
Information Type	The type and complexity of the information which can be displayed. We have adopted the categorisation structure proposed by Elliot, Watson, Neustaedter, and Greenberg (2007), who identified the most common types of information presented on ambient devices as: • **Binary**: displaying values representing either logical true or false. • **Discrete**: displaying one (or more) of several discrete or distinct states. • **Continuous**: displaying a value within a continuous range. • **Textual**: displaying purely textual or numerical information. • **Multimedia**: displaying images, sound, video, links, or other multimedia content.
Resolution/Granularity	The amount of information which can be presented by the device at any given time. A device which can only present a binary state (such as a device consisting of a single LED) is considered to have low resolution. A device which can present textual information, multiple binary states simultaneously, or presents information on multiple dimensions is considered to have high resolution.

Table 4. Taxonomy of example ambient device technologies.

Type of Technology	References	Description	Visibility	Degree of Attention	Information Type	Resolution/Granularity
Single LED		Simple small LED turns on and off.	Low	Low	Binary	Low – 2 values
Single light bulb		Light bulb turns on and off.	Medium	Low	Binary	Low – 2 values
Text LCD	Elliot et al. (2007)	Simple two-line text display. May include a button (which can be used to scroll).	Low	Medium	Textual	Low-medium – 2 lines of 20 characters.
Lava lamp	Clark (2004), Rogers (2008), Savoia (2004)	Lava lamps turn on and off. There are often 2 lamps (one red, one green).	Medium-high Take several minutes to warm up for full visibility	Low	Discrete (on/off, red/green)	Low – 4 values (both off, red on, green on, both on)
Glow Lamp	Elliot & Greenberg (2004)	Rotates to display one of 5 coloured panels.	Low-medium	Low	Discrete (5 colours)	Medium – 5 values
Ambient Garden	Elliot & Greenberg (2004)	Flowers' stems can grow/shrink. Each flower has an LED (either red or green).	Low-medium	Low-medium	Discrete (LED) and continuous (height of flower stem)	LED: low (off/on, different colour) Stem height: medium (limited by physical space available)
Ambient Beads	Elliot & Greenberg (2004)	Vertical location of beads on a monitor changes depending on a variable.	Low	Low	Continuous	Medium (limited by physical space available)
Ambient Orb	Ambient Devices (2009)	An orb-shaped light that can change colour and pulse at different frequencies.	Medium-high	Low	Continuous	Medium – range of colours, and can pulse at different frequencies

Type of Technology	References	Description	Visibility	Degree of Attention	Information Type	Resolution/Granularity
Digital photo frame	Downs & Plimmer (2008, 2009)	Small display that can show any arbitrary information, with some technological limitations.	Low-medium (small screen) Suitable for small team or individual cubicle	Depends on application	Multimedia	Fairly high. Small screen, but wide scope for displaying information.
Dedicated PC monitor	e.g. Woodward (2007)	Medium display that can show any arbitrary information.	Medium Suitable for entryway, small team, or in parallel	Depends on application	Multimedia	Medium-high. Fairly big workspace (depending on how big the information is). Wide scope for displaying information.
Large LCD/plasma display/projector		Large display that can show any arbitrary information.	High Suitable for most team sizes, assuming they are collocated	Depends on application	Multimedia	High. Wide scope for displaying information and plenty of space.
Chumby	Chumby Industries (2006)	Small (3.5") LCD colour touchscreen, similar to a digital photo frame.	Low-medium (small screen)	Depends on application	Multimedia	Fairly high. Small screen, but wide scope for displaying information.
Nabaztag	Violet (2009a)	The Nabaztag is a Wi-Fi enabled device in the form of a rabbit. It has multiple independently controlled dimensions of output:				
		Movable ears	Medium	Low	Continuous	Medium-high. Ears movable within 180°.
		5 LEDs (255 colours)	Medium	Low	Discrete	Medium (on/off; colour variation).

2.4.4. Situated Displays and Ambient Devices in Software Teams

Software developers necessarily use their workstation for the majority of their day-to-day work. However, situated and ambient display technologies can be used to support and augment developers' work processes where appropriate. Given the consideration of asynchronicity described by Cadiz et al. (1998), dedicated displays would certainly provide a separate channel of information to the standard workflow that developers engage in. Accordingly, some previous work within the software engineering community has examined the role of situated and ambient displays within software teams.

As discussed in section 2.3, FASTDash (Biehl et al., 2007) provided a situated display, passively updated with information relating to developers' work in progress and current status, and was intended to provide developers with the information they would need to coordinate the complex work of constructing software. However, due to these multiple and detailed information sources, the complexity of the FASTDash interface (see Figure 5, p21) largely precludes its use as a passive or ambient display. In order to obtain useful information from the FASTDash system, developers must pay conscious attention to it.

A similar project was undertaken by da Silva et al. (2007), who designed a set of large displays to support coordination between distributed team members. A number of visualisations were provided on separate displays, including a world map with information about team status within each geographical area and a 3D view of task activities in progress. Some of these visualisations were interactive, providing team members with the ability to 'drill down' into items of interest. Again, however, while these displays were physically situated within, and took advantage of, their physical environment, they were not ambient due to the volume and complexity of the information, and degree of interactivity that the displays required in order to be useful.

Dashboards (e.g. Few, 2006) are a final example of an application that could make use of situated display technology. Dashboards typically provide an overview of a system or project, providing one or more key performance indicators (KPIs) or metrics which can be used to proactively and reactively monitor such a system. The extent to which a dashboard requires attention depends substantially on the design of the dashboard's user interface.

Within industry, ambient displays and other ambient devices have been used to provide feedback on build processes to team members. Such devices have been referred to as 'continuous monitoring' and 'extreme feedback' devices (e.g. Rogers, 2008), and are particularly common among agile teams practicing continuous integration procedures.

A canonical example, described by Clark (2004), is a lava lamp. In this example, a software team will set up a lava lamp with some sort of programmable electronic controller, and will turn the lamp on or off depending on the most recent build result. A number of such projects have been attempted within industry and described on blogs and articles. In many cases the device behaviours were implemented somewhat haphazardly or without any clear design objective. In addition, no evidence has been provided to illustrate their effectiveness, and in most cases no specific hypotheses have been developed to describe the expected outcomes.

Table 5 lists a number of examples of such systems from publicly available web sites, blogs, books, and other industry sources. We include the extent to which each device is visible across a room as well as any benefits or evidence that has been described to support the use of the system or project.

Despite the use of these devices within industry, no academic research has been published, and no formal analysis has been performed, to explore the usefulness of these devices, to articulate the problems they are attempting to solve, and to evaluate whether they have any measurable effect on the team's behaviour or code quality.

Table 5. Examples of ambient build awareness technology used by industry members.

Title	Description	Visibility	Benefits or Evidence Described
Continuous Monitoring tutorial (Rogers, 2008)	Discussion of continuously monitoring build status (and other forms of software team status information) using ambient devices. No specific project or technology was discussed.	N/A	Potential benefits are cited (e.g. to quality, performance, stability, profitability) but no evidence provided.
Lava lamps (Clark, 2004; Savoia, 2004)	Describes 'extreme feedback devices' such as red and green lava lamps to display overall build status.	High	Anecdotally reports that they are a "*very useful tool*".
Ambient Orb (Swanson, 2004)	Adopts the Ambient Orb technology to display overall build status.	High	Anecdotally reports that they are useful: "*…once you have on a project with [it]... you won't want to work on a project without it*".
Build Wallboard (Woodward, 2007)	Uses a computer monitor as a dashboard. Intended as a demonstration of the capability of build server APIs.	Medium	None.
Brian the Build Bunny (Woodward, 2008a, 2008b)	Uses a Nabaztag Wi-Fi rabbit as a shared build notification device for an entire team.	High	None.
BetaBrite LED Sign (Atwood, 2005)	Comments that the use of an LED display has a different meaning to a standard monitor.	High	None.
Nabaztag (de Morlhon, 2009)	Uses a Nabaztag device in a very similar way to Woodward (2008b).	High	None.
Snowman (Harrison, 2007)	Created an ambient device from a USB snowman which could change colour.	Low	None.
Dell XPS LED 'Ambient' Lights (Quibell, 2006)	Use programmable lights on a laptop.	Low	None.

2.5. Summary

This chapter has examined the literature relating to the processes and information used within software development, including the role that awareness plays in team-based development processes. This chapter also discussed the ways in which awareness information can be presented through the use of situated displays and ambient devices.

Within modern software teams, builds are a core part of the team's everyday workflow. Builds present the most frequent opportunity for the team to assess the entire codebase and to provide this assessment as feedback to team members. Builds can be configured to provide a summary of a wide variety of information, although a typical agile team build will at least include code compilation (thereby checking its syntax and accuracy), unit testing, and some form of smoke testing.

A common theme throughout the literature on modern software processes and awareness is the importance of continuously monitoring and being aware of the status of the team's activities, the codebase, and of the project as a whole. Agile methodologies, and the numerous tools and interventions described in this chapter, emphasise the importance of providing feedback to team members frequently and rapidly. However, this must be carefully balanced against the negative effects of distraction and attention-shifting.

Ambient awareness technologies provide a useful mechanism to enable continuous feedback of status information to software developers without requiring overt attention be expended or constant distractions be endured. Various industry members have recognised the advantages of these technologies and practices, and have built custom ambient awareness systems to monitor build status information and present build feedback quickly and frequently. However, these projects have been conducted somewhat haphazardly and without any formal analysis of team practices, framing hypotheses, or evaluation of success.

The following chapters describe a project to implement and evaluate build status monitoring technologies, adopting the principles of continuous feedback and ambient awareness.

Chapter Three
Methodology

The previous chapter discussed the overall theme that awareness of build status, and software team status information monitoring more generally, has largely been neglected within the literature. Build status monitoring technologies have been used within industry but, to date, no systematic analysis of their optimum design, or any evidence of their effectiveness, has been provided or published.

This chapter provides an overview of the approach that was used to study build status monitoring. Section 3.1 provides an overview of the general principles of case studies and action research within information systems, including the linear action research paradigm, and discusses the decision to use a case study approach rather than an alternative type of study. Section 3.2 then describes the components of this study, including the initial requirements gathering phases and the design and implementation of ambient build monitoring devices.

3.1. Case Studies and Action Research

Within the field of information systems a number of empirical study methodologies are at a researcher's disposal. Pure experimental and quasi-experimental approaches are useful for evaluating interventions that address specific problems and have a quantifiable and testable set of hypotheses (e.g. Saff & Ernst, 2004), yielding quantitative data suitable for inferential analysis. Case studies are typically used for exploring detailed ideas or hypotheses which do not lend themselves to a quantitative approach, or for ethnographic research.

Case studies can be viewed as "*research in the typical*" (Nagappan et al., 2008). Instead of isolating variables of interest by precisely controlling the environment and circumstances in which a study is performed, case studies are conducted within a real-world situation. This has positive aspects as well as drawbacks. On the one hand, case studies permit deep analysis of a single situation, allowing a broader set of behaviour to be observed and giving greater scope for interacting with that behaviour. A well-chosen case study team can provide a great deal of practical and useful information. However, case studies are done within real-world environments, with all the variability in circumstances that entails, and do suffer from issues of reliability, validity, and difficulty in replication. A single case study is rarely sufficient to draw general conclusions, but can provide a solid basis for further study, or can be analysed as a part of a wider set of case studies.

Case studies can be conducted informally or within some sort of methodological framework. Action research provides one such framework. As its name suggests, action research involves the blending of formal research with practical and real-world interventions. As an applied discipline, information systems can take particular advantage of this type of research. McKay & Marshall (2000) describe the use of action research within information systems research as:

...ideally suited to gaining understanding of whether technology or methodology is perceived useful and helpful in practice, what problems and issues are perceived to arise, and to identify how practice can be improved within the value system of the problem owner.

Action research involves the design, implementation and evaluation of an intervention, or set of interventions, to address a specific problem or set of problems. These problems may be known *a priori* or may be discovered iteratively throughout the action research process. Baskerville & Wood-Harper (1996) illustrate the action research cycle as an ongoing process of diagnosing areas of concern, planning action based on these areas (with a strong connection to previous work), taking actions, evaluating those actions, and formally specifying and reporting the learning attained from the process (Figure 6). These processes take place within the context of a 'client-system infrastructure', the set of principles and agreements between the research team and the organisation describing how the study will proceed.

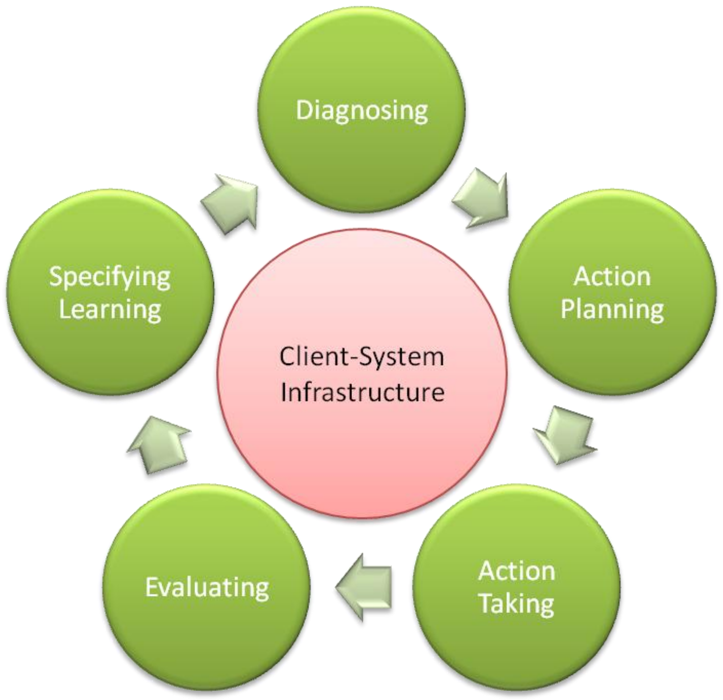


Figure 6. Action research cycle (Baskerville & Wood-Harper, 1996).

There are a number of characteristics separating action research from mere engagement in a consulting project. Two of the most notable of these characteristics are that actions must be situated within a theoretical framework, and that interventions must be empirically evaluated to assess their success.

A theoretical framework for any given action research project will usually be based on existing literature and previous case studies. If a relevant formal theory or model is available this will often be used as a starting point. If no such theory exists, some attempt can be made to abstract the core concepts and ideas from existing work in a bottom-up manner. In addition, the 'diagnosing' phase of an action research project can provide a large amount of knowledge to integrate into such a framework. From this theoretical framework a problem domain can be identified and narrowed, and a set of specific actions or interventions can be formulated.

Once an action has been decided upon and implemented it must be evaluated. Rather than haphazardly trialling ideas and drawing conclusions, action research requires that a researcher outline specific and measurable hypotheses to test, and to do so before an intervention takes place to reduce the risk of experimenter bias: "*above all, rigorous researchers pronounce the measurement approach before the intervention*" (Baskerville & Wood-Harper, 1996)

Any given action research project will be based on a single, or very few, case studies. Nevertheless, McKay & Marshall (2000) argue that action research should contribute to knowledge more broadly. Action research should ultimately allow for conclusions to be drawn between a set of interventions and the results that are observed, at least to the extent that non-experimental methodologies allow. This is acknowledged to be an incremental process with theories emerging from a set of similar case studies, in a process of gradual triangulation (McKay & Marshall, 2000).

3.1.1. Linear Action Research

Action research, as originally conceived and as described in social science literature, is iterative and cyclical in nature. A given project employing action research methodology will follow the cycle outlined in Figure 6 for several iterations, refining interventions and improving theoretical understanding after each iteration. However, within the field of information systems, it has been recognised that a diversity of action research approaches are appropriate (Baskerville & Wood-Harper, 1998). Recognising the differences between information systems research and social science research, Baskerville & Wood-Harper (1998) distinguish between traditional iterative action research, reflective action research, and linear action research, and argue that maintaining the essence of action research – analysis of a complex social situation, formation of hypotheses, planning and implementing actions, and evaluating their success – is more important than adherence to any given methodological paradigm.

For any given information systems research project, a number of decisions must be made. These decisions must necessarily take account of the project's budget, time constraints, and other resources. In the case of an action research project one such decision is whether to perform multiple cycles or a single cycle. With finite resources, a multiple cycle approach will likely limit the scope of each cycle and thereby limit the amount of information that can be elicited. A single cycle will allow for a deeper exploration into the issues and results uncovered during that cycle, but will also limit the overall scope of the project.

Linear action research provides a guiding methodology suitable for conducting a single cycle. Irrespective of the use of linear or iterative action research, the action research cycle still requires the core components outlined above (diagnosis, action planning, action taking, evaluation, and specification of learning). The linear paradigm simply diverges from the iterative paradigm in that it does not require multiple cycles in order for useful information to be obtained.

3.1.2. Use of Action Research in This Study

For this study, a number of study approaches were considered. Each approach was evaluated to determine if it would allow an adequate exploration of the concepts and problems inherent in build status monitoring within software teams. Ultimately the decision was made to adopt a linear action research-based framework. This decision was made for several reasons.

First, the limited amount of previous work in the area of build status awareness and monitoring meant that, before designing or implementing any interventions or devices, we would have to first identify the actual problems that real-world teams face in this area. While surveys or other quantitative methods would permit this to some degree, in-depth one-on-one interviews with team members allowed us to fully explore these issues without constraining the discussion to a set of specific questionnaire items.

Second, a key element of the study was to evaluate an intervention or set of interventions over a long period of time – ideally, several weeks or months. This would permit a more comprehensive analysis of the impact of these interventions, thereby increasing the reliability of the results. It would also allow us to track the interventions over time: a given intervention may be useful for monitoring build status in a short timeframe, but not necessarily over an entire project. A formal experiment takes place within a very short amount of time, not allowing such flexibility.

Third, it was assumed that software teams would be in a better position to make useful comments and suggestions, and would be far more able to give this feedback, with an extended opportunity to use and trial the technologies.

Action research provides the opportunity to explore a problem in detail, to evaluate different interventions, and to construct a set of concepts and results which could then be used for further research as well as in practice. It became apparent that an action research methodology would be ideally suited to our research objectives. However, the project's constraints – particularly with regards to its timeframe – precluded the possibility of conducting multiple detailed cycles of the action research process.

We opted to work within a linear action research framework. However, we viewed this methodology as a guiding rather than prescriptive framework; in particular, we made the adjustment that we would design and implement several prototypes, but without invoking entirely new cycles of the action research process for each prototype. Ultimately the study was conducted within an overall action research paradigm, but with adjustments as illustrated in Figure 7.

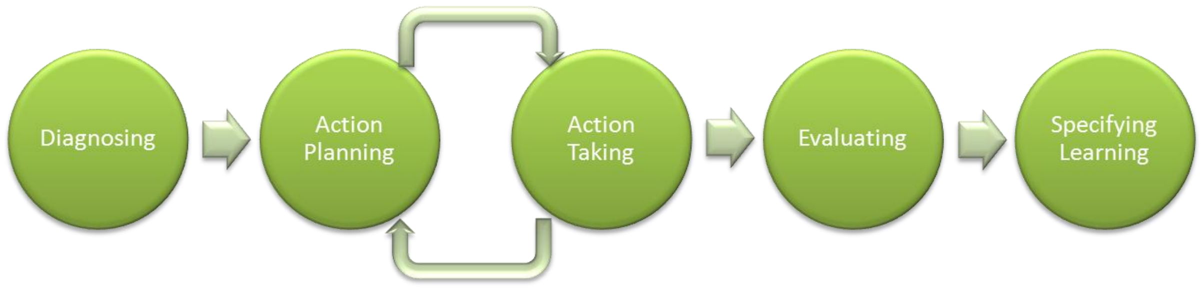


Figure 7. Hybrid cyclical-linear action research paradigm used for this study.

3.2. Study Plan

The study was divided into six component phases (Table 6). The initial phases were intended to gather information on problem areas within a team and to assist in the design and planning of concrete actions to address these areas. Later phases allowed these interventions to be performed, evaluated, and assessed. These later phases were conducted in an iterative manner so the interventions would be refined and multiple prototypes evaluated.

Table 6. Components of the case study.

Component	Description	Duration
Initial design and preparation	The study protocol was designed and a company was approached for a possible case study. Ethical approval was also obtained for the project, and consent was obtained from the case study company and its employees.	2 months
Observation	By working in the same physical environment as the case study team, it was possible to observe the types of social interactions that took place, including meetings, discussions, and team status updates.	1 week
Interviews	Detailed one-on-one interviews were conducted with members of the team. This component was performed in parallel with the observational component.	2 weeks
Design and implementation of prototypes	The information collected from the observations and interviews was analysed, as well as related literature. Based on this information, a prototype was designed and then implemented. This involved sourcing and using ambient device hardware and other infrastructural components.	3 months
Collection of baseline data	Qualitative information was collected to act as a baseline before any prototype interventions were conducted, enabling comparisons to be made between conditions.	5 weeks (overlap with implementation)
Evaluation of prototypes	The prototype system was installed into the team, and quantitative and qualitative data continued to be collected in order to evaluate its effectiveness.	3 months (some overlap with implementation)
Final interviews	Short one-on-one interviews were conducted with team members to discuss the study and the prototype technology.	1 day
Analysis	The quantitative and qualitative data obtained throughout the study was analysed and reported.	2 months

A software development organisation based in Auckland, New Zealand agreed to participate in the research.

Due to the nature of this study, approval was required from the University of Auckland Human Participants Ethics Committee (UAHPEC). The UAHPEC was provided with a full description of the research protocol, including the ethical issues which had been identified and a plan to address each of these issues (Table 7).

Table 7. Ethical issues identified.

Ethical Issue	Description	Addressed By
Power relationship between employers and employees	Employees may not feel able to discuss their opinions freely if they believe their employer is aware of their comments.	Managers were not told if any employees chose not to participate or withdrew during the study. Managers were not present in any interviews and were not given any interview recordings or transcripts, or any other identifiable information about participants.
Confidentiality	Some information provided by the participants, particularly during the interviews, may be sensitive.	All personally identifying information was removed from each participant's results before analysing and disseminating these results.
Commercial sensitivity	The information collected throughout the study could reasonably be viewed as commercially sensitive as it related to products the company was working on.	The research team was subject to a non-disclosure agreement that prevented us from discussing the identity of the company, details of the team, or information about the software product they were working on.
Informed consent	Team members should not feel compelled to participate in the study.	All participants received a full explanation of the research project and were able to withdraw at any time.

Participant information sheets and consent forms were constructed both for the managers of the organisation and for the individual employee participants. These documents are provided in Appendix A.

In addition to the approval from UAHPEC, the university's commercialisation organisation (UniServices Ltd.) required an agreement with the case study company to cover confidentiality and intellectual property issues. These processes were completed before the study commenced.

At the beginning of the study we worked with the software company to select an appropriate team. The company was in the process of transitioning from classical waterfall-style development teams to more agile approaches. They selected a team which had successfully implemented agile practices. The team worked as part of the wider organisation but on a product that was relatively independent from the rest of the company's work.

Initially, the team was comprised of ten software developers (two junior, five intermediate, and three senior, one of whom also was the team software architect), three testers, and a team leader. During the later parts of the study the team was restructured to only include developers and a team leader.

3.2.1. Observational and Interview Components

We conducted an initial one-week observational component in which the author visited the team's workplace. The purpose of this component was to obtain an overall idea of the team's communication and the daily work patterns that team members engaged in. The author also attended team meetings over this week.

In addition, interviews were performed with each team member lasting approximately one hour each. Most of these interviews were conducted concurrently with the observational component and the remainder were conducted in the following weeks. The interviews were performed in a semi-structured, conversational style and were intended to elicit information about the team's practices surrounding status information, particularly focusing on build status, and the problems and concerns they faced in these areas.

Observational data were collected and analysed for basic patterns. Interview recordings were transcribed and analysed using a bottom-up approach based on qualitative thematic analysis (Boyatzis, 1998), intended to identify and group common themes and items of concern to team members.

3.2.2. Design Concept Prototyping and Evaluation

Based on the findings from the interviews and the observational component, as well as the related literature, a set of core requirements was assembled. Based on these requirements a list of candidate technologies was assembled, and a set of specific, measurable hypotheses was established so the success of the technologies could be evaluated.

An initial prototype was implemented with two types of technology. One technology was intended to act as a central build status notification device, shared among the whole team, while the other was specific to each developer. Several iterations of these technologies were constructed in order to compare different behaviours.

To evaluate the technologies two main sources of data were used. First, build logs were collected for the team's builds, both during the study and for a baseline period in the month before. A number of metrics could then be calculated from these build logs. Second, qualitative information was collected in the form of emails and other communication with team members as well as follow-up interviews with each developer.

3.3. Summary

This chapter has outlined the methodology that was employed for this study. We described the foundational principles of case study-based research in general, and action research in particular, before outlining the procedures that were followed to investigate build status monitoring devices within the case study team.

The remainder of this thesis describes the study itself, beginning with the observational and interview components in Chapter Four. In Chapter Five the core problems and requirements are described, as is the design of the prototype system that was constructed to address these requirements. In addition, a set of hypotheses – which were used to evaluate the prototypes – are presented. Chapter Six discusses the detailed implementation of the prototypes. Chapters Seven and Eight describe the evaluation procedures and results and discuss the overall study findings.

Chapter Four
Observations and Interviews

This chapter discusses the initial study components performed with the case study team. These components were intended to gather information about the team and their daily work practices. Section 4.1 discusses the observational component conducted over a one week period, and section 4.2 describes the in-depth interviews with team members that were performed concurrently with the observational component.

4.1. Observations

We conducted the observational component of this study over a week-long period in May 2009. In total we observed the team for approximately twenty seven hours. The remaining time in the week was spent conducting interviews with team members. The observations and all of the interviews were conducted by the author.

The primary intention of the observational component was to discern the frequency and methods by which the team members communicated amongst themselves, as well as to get an impression of the reasons for these communications. This information was gathered by coding all face-to-face and verbal interactions between team members. We used the coding listed in Table 8 for this purpose as well as informal impressions of the types of communications which occurred. The number of participants and the duration of the communication were also recorded for each instance of communication behaviour.

Table 8. Behaviours recorded and coded within the observational component.

Behaviour	Description
Talking	At least one team member physically moved to another team member's desk to participate in a discussion.
Calling to Person	A team member called out to at least one other team member, without physically moving to another location.
Meeting	A meeting occurred between at least two team members at a location other than a team member's desk, such as a whiteboard, meeting table, etc.

Table 9 provides a summary of the average number of communication instances per hour over the study for each type of behaviour. A full set of results is provided in Appendix B.

Table 9. Average instances of communication behaviours per hour.

Behaviour Type	Average Instances Per Hour
Talking	5.60
Calling to Person	3.79
Meeting	0.19
Total	**9.62**

We were struck by how infrequently team members engaged in verbal and face-to-face communication activities. On average there were only 9.62 communication instances between team members per hour. Given that there were 14 individuals in the team, and with the heavily team-based work they were performing, this seemed remarkably low.

We also observed some seemingly unusual behaviour. For example, on several occasions the entire team simultaneously stood up and left the room for a meeting without anyone announcing that the meeting was beginning. Similarly, a number of team members started laughing at the same time even though nothing had been verbalised.

During the interviews it became apparent that the entire team was engaged in an on-going instant messaging conversation which was the primary means of communication. This is discussed in detail in further sections.

On the occasions when team members did communicate verbally, this was typically between a small number of participants (on average, 2.36) and was for a specific purpose such as a code review or for advice or assistance with a problem.

Because of the emphasis the team placed on instant messaging for most of the team's communication, the observational information was of limited use. The interviews were a much more useful source of information about the team's activities.

4.2. Interviews

Interviews were conducted with all fourteen members of the team, including developers, testers, and the team leader. Interviews lasted between approximately thirty and ninety minutes and were semi-structured in nature. Questions were designed to elicit information about a range of themes, including the participant’s background in software development, their day-to-day activities, the tools they used on a regular basis, and their experiences with problems in the development process, particularly relating to communication of the project status, project status tracking, and broken builds. The list of guiding themes used in the interviews is provided in Appendix C. Some questions were designed to obtain information regarding exceptional or unusual situations, such as irregular broken builds, a technique adapted from the Critical Incident Technique (Flanagan, 1954).

Interviews were recorded and transcribed, except in cases where participants asked not to be recorded. Transcripts were coded using NVivo analysis software (QSR International, 2009) and were then analysed using qualitative thematic analysis (Boyatzis, 1998). This section will focus on the results of our analysis of the interviews with the team's software developers. Comments from testers and the team leader are included where appropriate, but this section is mostly concerned with the developers.

A number of key themes emerged from the analysis: the daily work practices of the development team; the communication methods used by the team; the multiple and varied types of status information the team used in their daily work; and the process around and impact of broken builds. The following sections discuss each of these themes in more detail.

4.2.1. Team Methodology and Tool Support

The team used the SCRUM (Schwaber, 1995) development methodology, and accordingly worked within 'sprint' iterations. However, team members admitted they did not always strictly follow the methodology: in particular, team members reported a somewhat ad hoc approach to planning sprints, with issues and work items often being added to a sprint part-way through. As noted in later sections of this chapter, this caused some unanticipated problems with monitoring and reporting on the sprint progress.

While the team worked on developing a single distinct product, this product was broken into three components and any given developer typically worked on two or more of these components.

Sprints were typically two weeks in duration although the length varied depending on the circumstances. At the beginning of each sprint the customer and team leader would select a list of issues to be completed within the sprint. The development team would then meet to estimate the approximate time required for completing each issue. Following this planning meeting the team would begin working on these issues on a daily basis until the end of the sprint.

The team used a range of technologies to support their work practices and the SCRUM methodology, listed in Table 10.

Table 10. Tools used by the case study team to support their work practices.

System	Tool	Description
Issue management system	Atlassian Jira (Atlassian Pty Ltd, 2009b)	Allowed work items to be created, managed, grouped into sprints, and monitored individually or overall.
Build server	Atlassian Bamboo (Atlassian Pty Ltd, 2009c)	Managed the integration and nightly builds, performed the compilation process, executed unit tests, and tracked and reported on these build results.
Centralised source code repository	Subversion (CollabNet, 2009)	Stored and managed the codebases for each component.
IDE	Eclipse (Eclipse Foundation, 2009)	Provided the work environment for the developers to write code and unit tests.
Web-based time tracking system	Proprietary system	Enabled timesheets to be entered and reported on.

These tools were integrated together in various ways. For example, the issue management and time tracking systems were automatically linked, recording the actual time to complete a work item. This information was then used as part of the sprint monitoring process (see *Sprint Status* in section 4.2.4), and for subsequent sprint planning and estimation.

4.2.2. Daily Work Practices

The developers' work day structure was reasonably consistent and followed the same pattern irrespective of the specific component they were working on or the current sprint. The main work of the development team was cyclical and followed the same overall pattern for all developers, regardless of their seniority or the individual issues they were working on.

At the beginning of each day, most developers began by checking and replying to their email. Some also took responsibility for checking, analysing and discussing the results of the nightly builds that occurred the previous night. This brought the team to approximately 9.30am, when their daily stand-up meeting occurred (discussed in more detail in section 4.2.3).

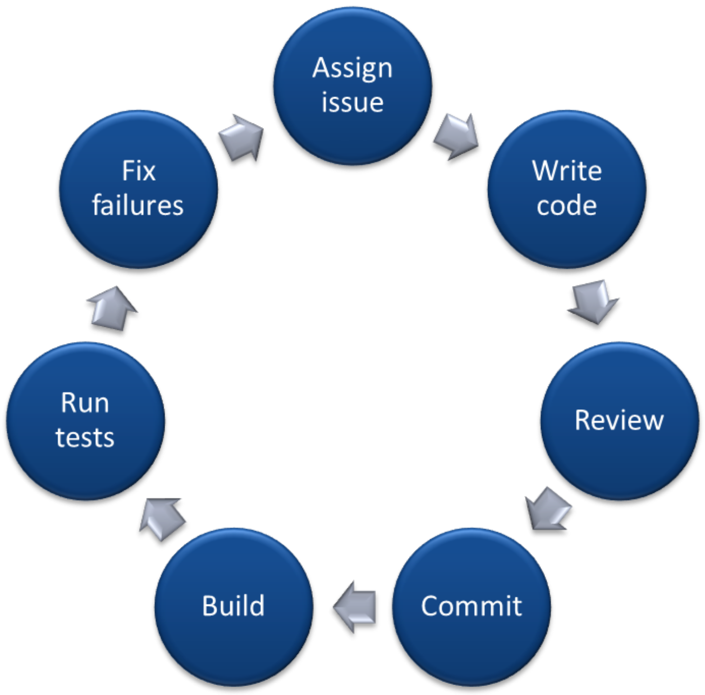


Figure 8. Standard daily development cycle.

Following the stand-up meeting individual developers performed a regular sequence of actions (Figure 8). First, a developer would use the issue tracking system to examine the list of outstanding issues in the sprint. The developer would find an issue (either a bug or a new feature) that they believed they could work on. There were no explicit rules or top-down assignments used for this process; instead, the developers would base their choice on their experience with the features or areas of code that the issue related to, and their own preferences. This process required a certain amount of self-governance on the part of the developers, although the team leader and senior developers monitored this process and ensured that the sprint goals were met. Having selected an issue, the developer would assign it to themselves so that other team members could see the task was underway and to avoid contention when other developers chose issues to work on.

The developer would begin working on their chosen issue, which typically involved writing code and unit tests and may have also required consultations with other developers on the team (or other teams within the organisation) for assistance or guidance. Depending on the size of the issue, this coding and testing process could take several hours or several days. In some cases, developers would create secondary issues, which flowed from the issue they were working on, in order to maintain task separation and to better plan and estimate large work items.

After completing work on their assigned issue, team policy required another developer to review the changes. This was performed at the developer's workstation and involved an examination of the source code changes with a verbal explanation. These reviews were intended as a preliminary check on the code quality, and also to ensure that another developer was familiar with the code in case maintenance was required. After the code review was completed, developers could commit their changes to the source code repository. The repository ensured that the reviewer's name was provided in the commit comments, thus enforcing the team's code review policy.

Once the code was committed to the repository, the build server triggered an automatic build. This would often take ten to fifteen minutes to complete. At the conclusion of the build the developers on the team were sent an email with the build results and after some time the build status monitor display would be updated (see *Getting Notified* in section 4.2.5 for further information). If a build failed, a developer – typically the developer who had committed the problematic code – would be responsible for correcting the problem.

At the conclusion of this process, the developer would select another issue to work on and the process began again.

Build Types

The team used a continuous integration process to automatically build the source code on a regular basis. Two separate build schedules were used: integration and nightly.

Every time code was committed to the repository, the build server would trigger an integration build. This automatically checked out the latest code from the repository and compiled it. If the compilation step completed successfully the integration build script also executed the set of unit tests for that project.

In addition to the integration build, a regular nightly build also occurred, irrespective of any changes that may have been made during the day. The nightly build was an extension of the integration build; as well as compiling the code and executing the unit tests, the nightly build performed a series of integration tests which would deploy the code to a staging server and automatically run test scripts. Because of these additional steps the nightly build typically took several hours to complete.

Nightly build results and artefacts were kept separate from integration build results. The testers on the team used the nightly builds as their test platform, so the team considered these builds to be important: a failed nightly build could result in delays in testing the latest features or bug fixes.

Project Components and Branches

The development team was responsible for three separate but related components. Developers on the team would often work on more than one codebase concurrently, although most developers reported they were most familiar with a single component. Despite the dependencies between the components, the codebases were separated within the source code repository and within the build server. As such, modifications to a single component would not result in an integration build for the other components.

In addition to the separation by component, the repository was also branched to allow the team to work on two versions simultaneously. While some developers worked on improving or fixing the current shipping version of a component other developers would be working on the subsequent version.

Because of these separations, the team used a total of six separate codebases within the source code repository, and twelve separate build plans within the build server, as shown in Table 11.

Table 11. Components and build plans.

Component Name	Version	Integration Build Plan Name	Nightly Build Plan Name
Component A	2.0	A20-INT	A20-NIGHTLY
	2.1	A21-INT	A21-NIGHTLY
Component B	1.0	B10-INT	B10-NIGHTLY
	1.1	B11-INT	B11-NIGHTLY
Component C	1.0	C10-INT	C10-NIGHTLY
	1.1	C11-INT	C11-NIGHTLY

4.2.3. Team Communication

In the one-on-one interviews, team members were asked about the ways in which they explicitly and implicitly communicated with each other. A number of distinct communication mechanisms were used within the team. Some, such as stand-up meetings and reports, were driven by the SCRUM methodology the team was using. Others, such as IM, seemed to be artefacts of the company and team culture. This section describes the main communication modalities team members reported using.

Stand-Up Meetings

The entire team – including testers – met at a regular time each morning for a 'stand-up' meeting. This type of meeting was in keeping with a fundamental tenet of SCRUM.

Typically these meetings were conducted outdoors (weather permitting), and a rugby ball was thrown to the person who was speaking. All team members were expected to speak, at least briefly, about the progress they made the previous day, their plans for the coming day, and any other issues or concerns they had. In addition, the team leader often gave an overview of the project status and provided updates and general guidance.

Overall, the majority of the team members found these meetings useful. Developers reported using the stand-up meetings to ask for help with particular technical issues. Several team members commented that the meetings provided an opportunity to stay aware of each other's work, thereby implicitly providing information about their availability and ability to assist with specific types of problems. Testers said they found this to be a valuable opportunity to stay up to date with the team's overall progress. Additionally, team members often found these meetings reassuring:

> *About every two days there's something that I'll want to jump in on, and you know, somebody says something and I think, "oh, that's interesting"... yeah it's good to have that dialogue and to figure out that everybody's on the same page and doing the same sort of thing.*

Despite the meetings' advantages, some team members expressed frustration with the way the meetings were conducted. A recurring comment was that the meetings often lasted much longer than they needed to: in some cases, meetings lasted twenty to thirty minutes. This was claimed to be a direct result of the team size. After our initial observations were completed, the team leader decided to temporarily split the team into two sub-teams, each working on separate branches of the product and each with their own stand-up meetings. Team members reported the meeting durations decreased to approximately five minutes – and that this was a more manageable meeting length.

Electronic Work Tracking Systems

The team used electronic work item tracking and time management systems. The two systems were integrated together and a number of dynamic visualisations were available from the data. The two main visualisations used by the developers were the burn down chart and the task board. In both cases the data for these visualisations would change on a semi-regular basis; while the issue tracking systems were updated regularly throughout the day, noticeable changes to these visualisations would only occur over a period of days. Both visualisations were available as web pages within the organisation's intranet.

The burn down chart provided a visualisation of the amount of work scheduled for the current sprint and the work completed by the team. Ideally, a burn down chart should show a downward trending straight line indicating progress towards the end of the sprint.

Different team members had very different ideas about the usefulness of the chart. Some senior team members argued that the chart was extremely useful:

> *Oh they're definitely really useful. Really useful. As long as everyone logs their time properly and updates the estimates properly.*

Others – particularly less senior developers – said they found the charts almost useless due to the way the team worked and the lack of strictness in the team's application of the SCRUM methodology:

> *We're not disciplined with... as soon as you start shunting stuff in and out of the sprints during the sprint they become fairly useless, and we're very, very badly disciplined at that.*

The task board was an electronic display that showed a list of all issues scheduled for the current sprint in three categories: issues yet to be started; issues started but not completed; and issues marked as completed.

About half of the developers noted that they used the task board as one of their primary resources for understanding the sprint's status. Several commented that they found this visualisation much more useful than the burn down chart. This was partly because this visualisation was less sensitive to changes in the scheduled items for the sprint. Additionally, some developers found it provided a useful at-a-glance method to track progress due to the simplicity of the visualisation:

> *You might look at the task board and go "oh my goodness, we've got half the work left to do, but we're already two thirds of the way through or 90% of the way through".*

Email

Team members often described their lack of use of email. Many developers commented that they only opened their email client two or three times each day. The amount of system-generated email also made this medium a source of some frustration, as discussed by one developer:

> *We get a lot of [issue tracking] spam which is, like, whenever anyone comments, or creates a ticket, or resolves a ticket, or anything like that, we get an email, which was a real pain originally, um, but I've kind of got my mail filters down now, so most of the [garbage] gets quite well filtered away.*

Many team members had set up similar filters to ease the barrage of emails they were receiving. The large number of issues and builds logged for the team resulted in an equally high volume of messages which quickly became overwhelming for individual team members. However, some commented that this also meant they sometimes missed valuable information.

In the case of build notifications, some developers were not even certain of the emails they even received, again emphasising the lack of importance these team members placed on email communication. This is illustrated by a comment from another developer:

Various different people get different emails for build notifications, so I certainly get them when there's a [Component A] build fails, but I'm not sure I get them when there's a [Component B] build fails.

Email was also considered problematic in that messages often took several minutes to arrive, even if they were only sent within the team.

Instant Messaging
As noted in section 4.1, during the observational component we observed that while team members did communicate verbally, this was typically between a small number of developers for a specific purpose. There were only a small number of occurrences where a developer called out to the entire team or made an announcement. However, team members appeared to coordinate and time activities with one another. During the interviews, team members explained that the entire team participated in a group IM conversation.

It quickly became apparent that this IM conversation was the primary form of communication within the team and within the rest of the organisation. A senior team member commented:

So the mode of communication has changed in this scenario, right. People don't need to talk.

The IM conversation acted as the main communication modality for a wide range of purposes, including on-going status updates, team collaboration, and for addressing technical questions, as well as occasional social or non-work related messages. Messages were visible to all participants even if there were only a small number of active participants. The developers typically found this a useful feature of the chat.

It's good as a broadcast mechanism. So maybe only two people are discussing it, but you know what went behind that reasoning which was came up.

Some team members commented that the extent of the conversation, and its detached and somewhat impersonal nature, had some unintended side effects. One developer noted:

I don't actually understand how I worked before without having this much communication, so I think it kind of really gets ingrained in you. It's like, "oh I don't know how to do something, I'll ask someone else," you know, instead of spending a few moments to work it out yourself.

Similarly, other developers found that the expectation that they would actively and constantly participate in the chat created problems for their ability to work:

But, on the other side, it could be quite interactive, like, breaking your concentration, like, constantly things keep coming up.

Despite these issues, the vast majority of developers on the team said they found the IM conversation a helpful and desirable communication modality and resource, even though the IM conversation logs were not archived or otherwise available for later conversation. Interestingly, many of the reasons cited by team members who preferred not to use email – interruptions, high volumes of messages, and broadcast messages not intended for a particular recipient – were also cited as features of the IM conversation, but were not generally considered problematic enough to discourage the use of IM.

Face-to-Face Conversations

Relatively little communication occurred in the form of face-to-face conversations. This form of communication was supplanted by IM for most purposes. However, some face-to-face communication did occur. For example, inter-developer code reviews typically took place at a developer's workstation, lasted around 15 to 20 minutes, and involved the developers discussing the changes. In addition, some other discussion occurred between developers who were seated close together.

4.2.4. Status Information

Another theme that was explored was the ways in which status information was obtained and communicated within the team. During the course of the interviews we found that the concept of 'status' had distinctly different meanings to different team members. Upon further questioning and reflection, it became apparent that developers distinguished between multiple levels of status information. Higher levels of status related to the project's delivery date, budget, and resource requirements, while lower levels related to development tasks, issue tracking, status of the source code, and other day-to-day concerns. While these categories are broad and may not be used by other stakeholders within or outside the team, the following three categories of status provide a useful framework for understanding how the developers thought about and used status information.

Day-to-Day Status

At the most fundamental level, developers were concerned with the day-to-day construction and building of the code. The primary metrics of interest were whether all the developers on the team were checking in source code regularly, whether the unit tests were passing, and (particularly) whether the nightly build could successfully run. In order to obtain this information developers used build server logs, the issue tracking system, and communication with other team members.

Sprint Status

The next level was concerned with the sprint that the developers were working on. Specifically, developers were concerned with the number and type of work items that had been planned for completion during the sprint or were added after the sprint began. There was some interest in time estimates given for work items, and in ensuring that the work items roughly followed the estimates laid out by the team in their sprint planning, but this was not emphasised.

At this level, the primary sources of information were the task board, the burn down chart, and shared knowledge about the current work items and progress towards the end goal. One developer commented on the use of burn down charts and explained that they were intended to be integrated into the developers' daily work:

> *It's difficult to generalise but I think in theory someone does look at the burn down chart and tells everyone what's going on at each morning meeting.*

Project Status

The highest level of information concerning the team was the overall project. This related to the ability of the company to deliver the project's specified features on time and on budget. The developers strongly emphasised that this was not a day-to-day concern for them; for example, one developer commented that:

> *Yeah I think the project is something that developers don't really have any influence over, so it's not really up to us to, you know, monitor that sort of thing, although when the team leader says "OK, we have to throw away all those changes and do these and... cos the requirements have changed", um, you know, they need, they're the ones that need to, like, justify that and explain to us, you know, the project status has changed for these reasons, but it's not our responsibility to actually go to the meetings and figure out what's going on in regards to the project.*

Some developers commented that they had an interest in the project status, and would try to follow it to a certain extent, but that they did not actively seek out this information. Instead, the information tended to be presented to the team by the team leader, by management, and through informal 'water cooler' discussions between team members and teams.

4.2.5. Broken Builds

A key theme which emerged from the team conversations was the regularity of, and process surrounding, broken builds. Because the team used a continuous integration model in which code was automatically built and tested as soon as it was committed to the source code repository, builds occurred several times each day. If code failed to compile, or if any of the unit tests failed, the build was considered broken.

Reasons for Broken Builds

The integration build process included two main steps: compilation of the source code and unit testing. Problems in either step would result in a broken build. These problems could occur for a number of reasons including inconsistencies between the tools and infrastructure used by the developers and by the build server, or a legitimate mistake or regression error.

Source code compilation was performed on the build server by using an Ant compilation script which used the standard Sun Java compiler. The Java compiler used by the build server behaved differently than the Eclipse-based compiler used on the developers' computers, and some developers discovered these compilers had different levels of support for some Java features and different degrees of strictness for the code they would consider valid. As such, code that compiled successfully on a developer's workstation may not have compiled successfully on the build server:

> *There are still some people who still commit code late in the afternoon and then go home. Uh, that's a bad habit that I would try and break people out of, but it's, uh, it's very tempting to check stuff in, you know, you finished it, you want to get it into the nightly build, so you check it in anyway. Um, and it works perfectly well in Eclipse so you assume it's going to work perfectly well...*

Unit testing was used primarily as a form of regression testing. In many cases failing unit tests were an indication of an unintended or undiscovered issue with code changes. In other cases transient problems with the unit test infrastructure, such as intermittent network outages or temporarily unavailable servers, caused some or all unit tests to fail inappropriately. Because of this, if all the build plans suddenly generated failing builds, the problem was likely to be related to this unit testing infrastructure rather than the tests themselves.

While developers could ensure their check-in would be valid before committing their code to the repository – both by manually compiling the code using the official Java compiler and by executing the set of unit tests – few did so. One developer, seemingly frustrated by unnecessary build failures, noted:

> *Um, and, you know, nobody checks in things that they think will break stuff, um, but it's a matter of how hard they considered that maybe.*

Another developer noted that the team members who tended to be responsible for broken builds were often more experienced:

> *The people who tend to be a little bit more senior, too, and uh intermediate developers, cos they tend to think that "oh yep, I should know what I'm doing, so this should be ok".*

A key point raised by a majority of the developers, however, was that while broken builds should not occur frequently, they were generally not a major problem as long as they were fixed in time for the nightly build:

The golden rule is that you don't break the nightly. It's OK to check stuff in that breaks the build, you shouldn't, but it's OK to check stuff in that breaks the build as long as you then fix it for the next integration build. There's no sin in misjudging whether something's going to, as long as you've considered it, and you check it in, and "oh no, that's broken it".

Negative Effects of Broken Builds

While developers recognised a broken build was not a serious issue if fixed quickly, almost all developers on the team also shared an understanding that broken builds were generally best avoided. One developer commented that builds often breaking might promote a laissez-faire attitude among the team:

If people treated it as a bigger deal than people currently do then I think that would go a long way.

Another developer concurred with this sentiment, arguing:

Well you want to get away from that whole sort of broken windows thing where the window's broken, you just don't notice it cos it's always broken, and it sort of hides some other problem.

The idea of broken builds hiding other problems was also a common theme. Some noted that if code was committed after a build failure, that new code could conceivably be problematic too, but the confounding factors would make it difficult to determine exactly where the problem was. Another developer went so far as to suggest:

I mean, this would never work right, but you could arguably say "you cannot check code in until the build is fixed". Obviously you could never fix the build because you couldn't check in the code to fix it, but, you know, that's the sort of thing right, like, like, stop people from doing that.

This same developer also stated:

It's almost like you need to somehow force, or make abundantly aware, that people should be stopping what the hell they're doing and trying to fix this, because it is, it's... if people treated it as a bigger deal than people currently do then I think that would be, that would go a long way.

Another practical impact of broken builds was that other developers may have inadvertently obtained copies of the code without realising it was in a broken state. If this occurred, a developer may have begun working on the broken code, committed it, and then caused another broken build even if the initial problem was fixed.

Getting Notified
Once a broken build occurs, a development team must be made aware of this fact in order to rectify the situation. The modalities used to provide notifications depend on the systems and resources available to the team. Within this case study team there were several distinct modalities in use: active monitoring of the build server, email notifications, a dedicated display, and verbal or IM notifications from fellow team members.

In some cases, developers would actively monitor a build as it progressed. One developer noted he did this when committing code that he suspected may be problematic; immediately after committing code he would manually trigger a build and then watch the build logs in real time so he could quickly be made aware of any problems.

After every build, the build server sent emails containing the build results. There was some confusion within the team about who received emails and under which circumstances. Some team members reported receiving an email after every build; others reported receiving nightly build results and only integration build results where they had committed code and the build had failed. Some team members reported that they had set up filters to automatically delete these automated emails as they received so many. In many cases developers also reported they did not have their email client open all day, so by the time they saw these notification emails they were no longer relevant. This highlights the previously noted theme that email is not an integral part of this team's communication system.

Another technique the team used to become aware of build results was a dedicated computer monitor which the team referred to as the 'build status monitor'. The monitor displayed a full-screen web page containing rectangles representing each build plan, including nightly builds and integration builds. The rectangles were coloured green if the latest build in the plan was successful and red if the build had failed.

The screen was located in a corner of the team's work area (Figure 9). The screen was a standard LCD display, and due to the size of the room and the position of the screen it was only visible to a handful of team members. Some team members could not see the screen at all from their workstations; others could see the screen but could not easily distinguish between the individual rectangles representing each build plan. The display was also used by another team situated in the same room, which meant that the individual rectangles were quite small and closely packed, and that each team was only focused on a subset of the items on the screen.

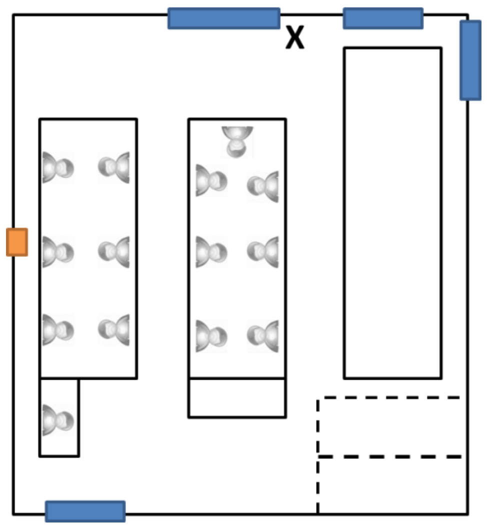


Figure 9. Layout of team workspace (not to scale). Build status monitor was at position marked X.

In addition to the problems with viewing the screen's information, the display was only updated every five minutes. Depending on the timing of the builds for each plan it was possible that the screen would not update for several minutes following a build. As noted previously, the developers commented that a broken build should be prioritised; this type of delay would appear to be incompatible with this goal.

Despite its drawbacks, the screen was seen as a good thing overall, as commented by one developer:

> *It just gives people a feeling that it's really important things when the build's broken. So I think it's just, um, psychological things maybe. Cos if it's just an email, I guess maybe sometimes it gets ignored because people are too busy and things like that. Cos that red colour probably gives people [the idea that it is important].*

Finally, some team members commented that they relied on others to tell them when a problem had occurred. An informal process had emerged in which developers would discover, advertise, and assign responsibility for a broken build within the IM conversation, as described by one developer:

> *Normally if someone notices it they'll put it in the [IM] chat, and someone will say, oh you know, "it's you!" [laughs] and… they'll stop what they're doing and then carry on with what they're doing.*

Some developers commented that they did not actively monitor the email notifications or build status monitor, and assumed that they would be told by a colleague if their check-in had caused a broken build.

Assigning Responsibility

Once a broken build has been discovered, either by the entire team or by an individual developer, responsibility for fixing that broken build was assigned.

Many developers noted that, despite the negative connotations of 'assigning responsibility', they did not intend the term in a pejorative sense; instead, they recognised that the person who made a breaking change was also the person most able to fix it. Yet, some developers expressed frustration with others for not taking this responsibility seriously. Others commented that they were subject to good-natured teasing when they had been responsible for a broken build, including one junior developer who noted:

You feel very embarrassed.

The process of assigning responsibility required information from a number of sources, including shared knowledge about other team members' current and previous work, build server commit logs identifying the developer or developers who had recently committed code, and the unit test results.

Once the developer had been identified, there was an expectation that they would lead the effort to fix it:

Most of the time the person who did the last commit will check it out, and if he can find the fix then he will say, "oh yeah, I didn't break this, it's someone else", and you know, and then the person who made the broken change will try and fix it, if he can. If he cannot then he'll ask someone senior to help him fix it.

In order to fix the build, the developer needed to ascertain the nature of the problem; as noted above, a broken build could occur for a variety of reasons. In many cases the build logs and unit test results were sufficient to diagnose the problem, at which point the developer could take the appropriate corrective action and recommit the code, thereby triggering another build.

Social Dynamics

With ten developers, this was a relatively large team. The team size meant that social dynamics played an important role in their interactions, including their perceptions of and behaviour regarding broken builds.

As previously noted, some team members took responsibility for responding to broken builds while others preferred to be notified when they were required to fix a problem they may have introduced. One developer who fell into the latter category commented that:

There are people who actually pick it up in the team so I tend not to… cos email I tend to just presume that some... one of the more active email watchers is looking at it.

The same developer noted that the size of the team, and the fact that there was no individual directly responsible for the builds, meant that a certain amount of social loafing (Latane, Williams, & Harkins, 1979) occurred:

Potentially [checking build reports regularly is] a valuable... that would be a good thing to do, and I'd probably do it on a smaller team, but when everybody's responsible no-one's responsible [laughs].

The impact of social dynamics was felt beyond simply discovering and notifying others of a broken build. One senior team member noted that a broken build, particularly one which is easily avoidable, has further ramifications for the team as a whole and on team members' perceptions of the developer in question:

If you want to get into that, having some pride that "hey, you broke this, why'd you break that?"... I think the worst thing is if you do something that affects the other guys in the team, so then, you know, they get your code and then it breaks a whole lot of stuff and, you know, then they've gotta screw around for a couple of hours and figure it out, um, that's just bad, right, just kills productivity and then people get a lower opinion of whoever checked the code in. It's just not helpful.

4.3. Summary

This chapter has discussed the initial study components with the case study team: an observational component and a series of comprehensive interviews.

A number of key findings emerged. We found that intra-team communication occurred primarily through electronic means, especially IM, and less so through verbal or face-to-face means. The emphasis that this team placed on IM was largely due to the organisational culture, but team members had become accustomed to this mode of communication and considered it a core part of their daily workflow.

More generally, different stakeholders within and outside the team were concerned with different types of status information as they conducted their everyday work. For the developers within the team, the process of checking in, testing, and building code formed a large part of their work practices. In particular, broken builds were identified as a key area for improvement as there were a number of challenges surrounding the processes by which broken builds were discovered, notified, communicated, and fixed.

The next chapter discusses the design of a system to augment the team's practices surrounding broken builds.

Chapter Five
Design

The observational and interview components of the study uncovered areas of the team's daily work which were problematic or causing issues for the developers on the team. In particular, it was found that the build process had a number of areas for potential improvement. The build process was such an integral part of the team's daily workflow that this area was exclusively focused on for these interventions.

This chapter describes the process that was followed to select a problem and design a suitable set of interventions. Section 5.1 lists a core set of requirements based on a synthesis of the data gathered as well as the related work discussed in previous chapters. Section 5.2 discusses some possible technologies and solutions that could be adopted to begin to address these requirements, and section 5.3 describes a possible evaluation – specifically, a list of testable hypotheses of changes in the team's behaviour following the introduction of the interventions. The chapter concludes in section 5.4 with a brief summary.

5.1. Requirements

The interviews and observations discussed in the previous chapter highlighted the importance of the build process to the case study team's workflow. The build process is directly integrated into the developers' work: every code change they make ultimately gets checked into the repository and built by the build server, and a great deal of software technology and process surrounds this. Other team members and stakeholders are also impacted by the build process and are sensitive to its problems. For example, if a failing integration build is not fixed it may also result in a failing nightly build, which in turn will mean there is no build for the testers to work with the following day. The team leader and other organisational stakeholders also derive metrics that come, directly or indirectly, from the build process. A failing build can be caused by a lack of awareness of the build process, a genuinely accidental code breakage, or a laissez-faire attitude among the developers who may have committed broken code. Broken builds can have widespread ramifications if ignored.

The key problems that team members encountered with the build process have been summarised and translated into a set of core requirements for any interventions which seek to address these problems. The problems and requirements are outlined in Table 12.

Table 12. Key problems and associated requirements for broken build monitoring.

	Problem	Requirement
R1	Some developers did not watch for broken builds themselves, but instead waited to be told by another team member.	Developers should be informed about broken builds automatically.
R2	Even when broken builds were discovered, they were not always prioritised and fixes could take some time. This caused a number of downstream problems such as broken nightly builds or other developers needing to fix a problem unrelated to their own code.	Broken builds should be made to appear important and a priority to fix.
R3	Notifications from the build server were often delayed. Email messages would take some time to be seen, due both to delays in the email system and to the fact that some developers did not have their email applications open constantly. The build status monitor screen was not updated or noticed immediately, except by some vigilant team members.	Notifications of broken builds should arrive as quickly as practicable, and with minimal or no effort required on the part of the developers.
R4	The build status monitor screen was located far away from the team. It was difficult for many team members to discern individual elements (such as the rectangles denoting build plans) on the screen.	The build notifications should be easy to see and require little cognitive effort to perceive and understand.
R5	The process of determining the developer(s) who may have been responsible for a broken build was reasonably manual. Nobody was specifically responsible for this process.	Developers who may have broken a build should be automatically and personally notified.
R6	The large number of developers in the team often meant that developers felt less responsibility for broken builds, even if they were personally responsible.	The entire team should also be notified of a broken build, and a persistent notification should remain until the problem is resolved.
R7	Team members sometimes entirely missed	It should be obvious to the entire team who

	Problem	Requirement
	notifications about a broken build, especially if they had not personally committed to that build. This could result in additional code being committed to a failing build, potentially adding a confounding factor to the diagnosis and resolution of the broken build.	may have broken the build. This will apply implicit social pressure to the breaker(s).
R8	Build results were often lost among the other daily emails and activities of the development team.	Build information should be presented through a separate channel from other information to emphasise its importance.
R9	Developers reported receiving high volumes of email, including many build notifications. If a developer did not check their email for several hours, they may have received a large volume of (now irrelevant) email notifications from the build server. Some developers were not even sure which notification emails they received as they tended to delete them all.	Team members should not be overloaded with multiple build notifications; instead, a single, dynamically updated summary should be provided and should display only the current status

We proposed to address the requirements listed in Table 12 by using ambient device technology. As discussed in Chapter Two, many different types of ambient device technology exist, but the fundamental tenets of ambient devices – constant availability of digital information within a physical environment, dynamic updating, ease of perception, and a separate channel to the user's standard workflow – provide a solid base upon which to construct a build notification system.

5.1.1. Role of Ambient Devices

Two key issues faced during the design phase of the technology were related to the role of ambient devices. First, it was important to assess the complexity of the information to be presented, and consider whether this information would become too cognitively intensive to perceive in a single glance. Second, a decision needed to be made on whether the devices should generate notifications only for exceptional cases – broken builds – or whether they should generate notifications for all builds.

Glanceability

The first concern was strongly related to the 'glanceable' property of ambient devices. This property dictates that the information an ambient device presents should be available effortlessly and with minimal cognitive processing. Some ambient devices have multiple dimensions along which information can be presented, potentially enabling a single device to convey two or more independent channels of information. For example, a single wristwatch may contain multiple displays with the current time, date, day of week, barometric pressure, tides, and so forth. However, taking advantage of these multiple dimensions will require additional cognitive effort to parse. This also becomes a confounding factor in any evaluation of the device, since any effect could be due (at least in part) to the fact that the device carries multiple channels of information rather than due to the information itself.

We decided to avoid these issues by restricting any given ambient device to displaying a single channel of information. Where a device supported the display of multiple dimensions or channels, these could be used to redundantly encode the information, thereby providing multiple cues which could be perceived independently of one another but still carry the same information.

Types of Notifications

The second issue was whether to restrict the system to only notify developers of broken builds, or whether to also generate notifications when a commit had resulted in a successful build. A number of issues factored into this decision. First, the additional level of notification would require the system's behaviour to be at least somewhat more complex. The additional complexity could increase the cognitive effort required to interpret the meaning of the different devices. This would be of particular concern for the case study team as they were working on three separate build plans. Any individual or shared device could seem to be inconsistent with the other devices in the team, depending on the state of each of the build plans and that developer's recent commit history.

Second, if developers are given notifications whenever their own code has successfully been built, they may become less interested in the overall state of the build plans. Instead, they may focus solely on their own notifications and disregard those of the rest of the team. This may be desirable or undesirable depending on the circumstances. It is less likely in a situation where developers only receive failure notifications and are in a 'neutral' state at all other times, since a neutral state does not carry the implication that everything else is fine the way a positive state would.

Third, the addition of feedback for all commits will reduce the extent to which these devices can be considered 'ambient'. Ambient devices are intended to be available within an environment for consultation at will, but not to require overt attention to be paid at all times. If notifications are provided after every commit the devices could become distracting and intrusive.

Finally, we were concerned that a high volume of notifications may result in a diminishing of the value of these notifications over time. If notifications were only provided when builds failed they would be less common, and may therefore be perceived as exceptional cases requiring urgent attention and action.

However, there were also legitimate arguments to add notifications when builds passed. These were based on the principles of positive reinforcement and positive punishment within behavioural psychology, specifically the Law of Effect (Thorndike, 1911). If developers were only to be actively notified when a build failed, that notification could be construed as a form of positive punishment in which a stimulus is delivered in order to reduce the incidence of a behaviour – in this case, discouraging the committing of failing code. By the same logic, if developers were also notified when a build succeeded, they would be receiving positive reinforcement. If the Law of Effect applies to this situation it would suggest that reinforcers and punishers would have a substantial effect on behaviour, and that the combination of both reinforcers and punishers would be stronger than only using punishers.

We ultimately decided to develop multiple prototype systems in order to compare the two distinct behaviours – ambient build failure notifications only, and reinforcement and punishment-based notifications for all build results. This also permitted a comparison between the two conditions against each other and against a baseline condition in which no devices were provided.

5.2. Candidate Technologies

Based on the list of ambient device technologies from Chapter Two (Table 4, p26), a number of possible technologies were considered. Each type of device was evaluated in terms of whether it fulfilled each of the requirements specified in Table 12. A representative sample of that list, as well as the two existing build monitoring technologies the team used, is provided in Table 13.

Table 13. Summary of candidate technologies and the requirements they satisfy. Hollow circles indicate a partial fulfilment of the requirement.

Technology	R1	R2	R3	R4	R5	R6	R7	R8	R9
Email (existing system)	○				●	○			
Dedicated build status monitor (existing system)	●	●	●	○		●		●	●
Software running on developers' workstations	●		○		●				●
Large display	●	●	●	●		●	●	●	●
Ambient Orb	●	●	○	●		●		●	●
Personal LED	●	○	●	●	●			●	●
Nabaztag	●	●	●	●	●		●	●	●
Central USB light	●	●	●	●		●		●	●

A simplistic solution would involve the installation of software components on the developers' workstations. These components could monitor the build server and provide visual notifications when a build had failed. However, such a solution would fail to meet a number of the requirements listed above. The information should be provided through a secondary channel, so as to avoid being integrated into the developers' primary workflow, and this separation of channels would be difficult or impossible if the information was provided on the same physical device as the developer's main coding environment. Additionally, the lack of a shared display or device visible to others would reduce the social pressure to fix a broken build and make it more difficult for developers to determine who may have broken a build.

An added large-screen display would be similar to the existing build status monitor, but could potentially have additional space to display a list of developers who recently committed to broken builds. However, this notification would be dissociated from the developers' personal space – which could reasonably be expected to result in less urgency being conveyed to the developers in question. In addition, depending on the physical location of the display and the size of its contents, it may be difficult to view and interpret from across a room.

Conversely, a small personal LED would provide each developer with a focused channel of information. A red light would convey a sense of urgency and encourage that developer to correct any problems with the build they had committed to. However, such a device would be too small to be seen across a room and therefore limited social pressure would be applied.

Nabaztag Wi-Fi rabbits (Figure 10) are much larger than personal LEDs and can be seen from further away. Their rabbit-like form factor is non-threatening and less industrial than many of the listed alternatives. In addition, they have a number of display parameters to control, such as the position of the ears, audio, and LEDs. These could theoretically be used to convey both the overall build status as well as specific notifications for an individual developer. However, as discussed in section 5.1.1, we opted to avoid using a single device for multiple information channels due to the additional cognitive effort this would require to separate and perceive. Nevertheless the multiple display parameters would allow for redundant visual (and potentially auditory) cues, and therefore would present multiple opportunities for developers to notice the build result even while attending to other tasks.

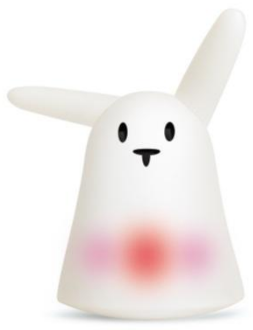


Figure 10. Nabaztag Wi-Fi rabbit.

An Ambient Orb and a central USB light are similar in their general function, but the Ambient Orb uses a proprietary server and communication protocol and notifications through such a device may be delayed or direct control otherwise lost. A central USB-controlled light does not have this problem. Both devices are very visible across a large room, depending on their placement, but are only useful as central shared devices and do not convey information about the identity of the developers who may be responsible for breaking a build.

Based on this analysis, we determined that no single technology we considered could adequately satisfy every requirement. However, a combination of multiple technologies may be suitable; in fact, a number of different permutations could be assembled to fulfil the entire set of requirements. The choice of the specific technologies to use would presumably depend on other factors, such as availability and budget.

For this project, we decided to use two distinct types of technology. A single shared device would be provided to the team in order to provide a global notification of a build failure without isolating the particular failing plan. In addition, individual devices would be provided to each developer to indicate when that developer had committed code to a build that had then failed. The mere presence of these devices in a given developer's work area would convey implicit contextual information about the identity of that developer, and combined with an easily perceptible state conveyed through the use of the device's LEDs or other features, would enable other developers to quickly determine who is responsible for fixing a broken build.

5.3. Evaluation of Technology

Once the build monitoring systems are created and installed, it is important to evaluate whether they are having any measurable impact on the team's work. As noted in Chapter Two, the limited work that has been performed in the area of build status monitoring has not provided any quantifiable evidence of its impact. To address this we developed a set of specific, testable hypotheses designed to evaluate the effects of an ambient awareness build monitoring system compared to a baseline condition with no such monitoring system. These hypotheses are listed in Table 14.

Table 14. Research hypotheses.

	Hypothesis	Discussion
H1	The proportion of builds which fail will decrease.	We expect that developers will aim to reduce the frequency of broken builds to avoid receiving build failure notifications. We particularly expect this to occur when both positive reinforcement and positive punishment are provided.
H2	The overall number of builds per day will increase.	We expect that developers will commit code more regularly, thereby increasing the frequency of the builds. This is based on the expectation that developers will check in more granular changes in order to keep changes to a manageable level, and thereby reduce the likelihood of broken builds.
H3	The number of files checked in per changeset will decrease.	If developers check in more regularly, we expect that the number of files per changeset will decrease as developers check in more granular changes.
H4	The number of commits to a broken build will decrease.	Developers commit to a broken build in order to fix the underlying problem causing the build to break, or to commit unrelated changes (perhaps unaware that the build is breaking at the time of their commit). Since the build status will be easier to ascertain with the monitoring technology added, we expect that developers will be less likely to

	Hypothesis	Discussion
		commit code to a broken build except to fix that build, and will be more likely to either fix a broken build themselves or wait for it to be fixed by another developer before committing additional code to that plan.
H5	The proportion of builds triggered by a manual request from a developer will increase.	We predict that developers will seek to remove build failure notifications from their ambient devices as quickly as possible. They will therefore prefer to manually invoke a build once they have committed code to fix a problem, validating the change they have made, rather than waiting for the build server to automatically perform a build.
H6	The proportion of broken builds due to unit test failures will decrease.	Broken builds can occur for a number of reasons. The three primary reasons are unit test failures, compilation failures, and intermittent problems with infrastructure. Unit test failures are usually avoidable if a developer runs the unit test suite on their development machine before checking in. We predict that developers will do this more often when the ambient build awareness technology is added, thereby decreasing the proportion of broken builds due to unit test failures. Unfortunately, compilation failures and unit test failures are indistinguishable within the build server logs and therefore we cannot test any hypotheses relating to these failure types.
H7	The proportion of failing builds fixed by the breaker will increase.	We expect that developers' personal responsibility for fixing builds will increase as they will be identified more quickly and publicly. Accordingly, we expect that developers will be more likely to fix broken builds that they have committed to personally.
H8	The average duration of broken builds will decrease.	We expect that the amount of time that plans remain in a broken state will decrease: developers will be quicker to address the problem with a persistent and prominent visual cue.
H9	The total proportion of time in which any plan is failing will decrease.	Compared to a baseline condition without any ambient device technology, we expect the total amount of time that any plan is in a failing state will decrease, as the duration of broken builds decreases and the frequency of overall builds increases.

5.4. Summary

This chapter described the high-level design process that was used for determining the requirements of an ambient build monitoring system, and the technologies that could be used to develop such a system. In addition, the chapter outlined a comprehensive set of measurable hypotheses which can be tested as part of the evaluation of such a system.

The next chapter outlines how this high-level design was implemented through hardware and software components. Later chapters then evaluate the system using the hypotheses listed in section 5.3, as well as based on qualitative feedback from the developers in the team.

Chapter Six
Implementation

In order to accurately and quickly display build information within the developers' workspace, a number of hardware and software components were required. This chapter discusses the implementation of the software to monitor the build server and control the hardware devices used in this study. Section 6.1 describes the hardware devices and the protocols used to communicate with them. Section 6.2 discusses the build server in use by the case study team and the extensibility points used by the monitoring system. Section 6.3 then describes the detailed system architecture, including the software components that were constructed to enable the build server to be monitored and the hardware devices to be controlled. We discuss our experiences and challenges in section 6.4, and conclude with a summary in section 6.5.

6.1. Hardware Devices

Based on the design discussed in the previous chapter, two separate types of hardware devices were selected: a central device, which was a simple lamp which could change colour and was visible to the entire team; and individual devices, which were Nabaztag rabbits given to each developer.

6.1.1. Central Device

The central device (Figure 11) was a Delcom RGB USB HID Visual Signal Indicator lamp (Delcom Products Inc., 2009). The lamp consisted of a set of LEDs which could produce red, green, or blue light, easily visible throughout a large room. The lamp was connected to a computer through a standard USB port and used a standard human interface device (HID) controller. Delcom provided a C# library to interface with the device and control the colours displayed.

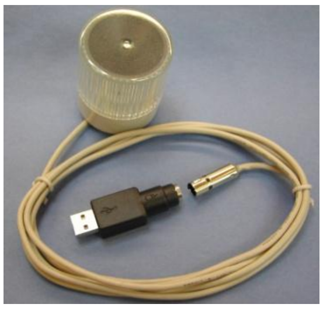


Figure 11. Delcom USB light.

6.1.2. Individual Devices

Each developer was given a Nabaztag:tag Wi-Fi-enabled rabbit (Figure 10, p67). These devices are the second generation of the Nabaztag rabbits (Violet, 2009a).

Nabaztag rabbits use a proprietary, non-documented protocol to communicate and receive instructions (Rajaniemi, 2007). Typically the rabbits communicate with an Internet-based public server provided by the manufacturer. This service allows third-party developers to control any individual rabbit (Violet, 2009b). Communication with the server is through Representational State Transfer (REST), a standardised subset of the HTTP protocol optimised for retrieving and managing data (Fielding & Taylor, 2002). The rabbits must have wireless internet access available if they are to use the public server.

For security reasons the case study company could not provide a suitable wireless network with internet access, nor could the rabbits be added to their corporate network. In order to work around these limitations, we used jNabServer (University of Tampere, 2008), an open-source, Java-based server framework for controlling Nabaztag rabbits on a local network without an Internet connection. The jNabServer framework provides an extensible plug-in infrastructure, allowing third-party developers to customise the rabbits' behaviour programmatically. A jNabServer plug-in was built using this framework; its implementation is described in section 6.3.2.

To allow the rabbits to wirelessly communicate with jNabServer without using the corporate Wi-Fi network, a separate wireless router was used to create a separate network. The server and the rabbits were the only devices which used this secondary network; the server was also connected to the corporate network in order to access the build server. Figure 12 illustrates the network topology used by the system.

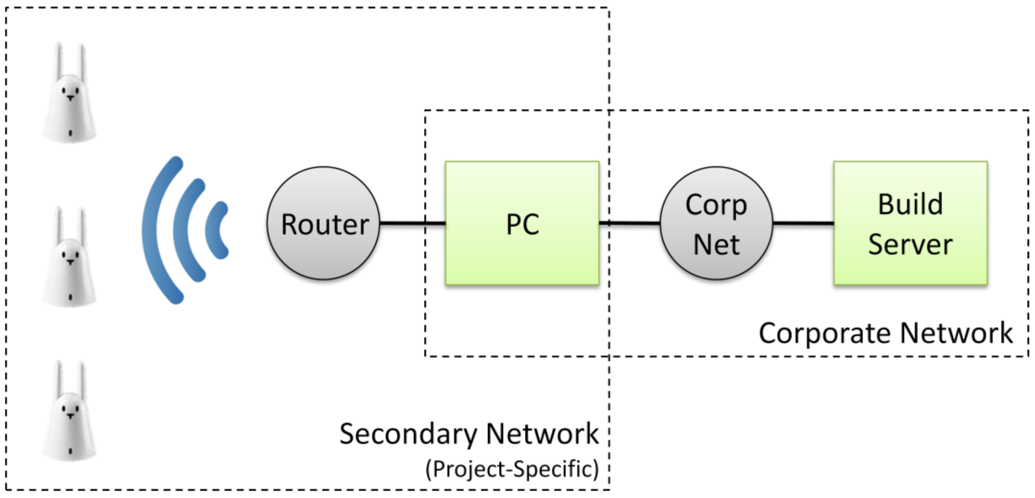


Figure 12. Network topology.

6.2. Integration with Build System

The case study company used Atlassian Bamboo (Atlassian Pty Ltd, 2009c) as their build server. This is a Java-based server which can run on a number of operating systems. For the first part of the study the company used version 2.1.2 of Bamboo; towards the end of the study they upgraded to version 2.4.1.

The primary unit of organisation within Bamboo is a plan. Typically, a plan will correspond to a tree within a source code repository to be built on a defined schedule. Plans are grouped into logically related projects. Figure 13 illustrates the hierarchy of projects, plans, and builds.

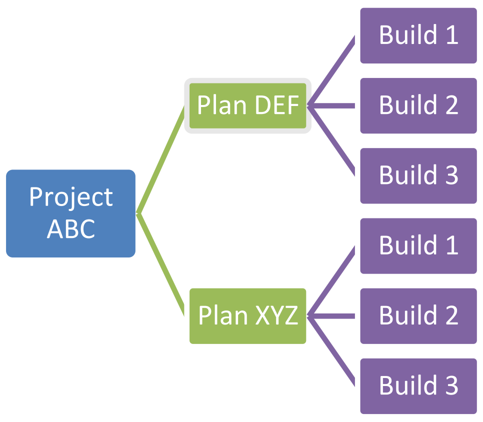


Figure 13. Hierarchy of projects, plans, and builds used within Bamboo.

A plan has settings controlling when and how it should build the code. Integration builds are often set up so that they will poll the source code repository on a regular basis, and if there are changes, Bamboo will begin the build process of checking the latest code out, building it, running tests, and recording the results. Bamboo assigns sequential numeric build identifiers to each build, concatenating the project, plan, and build identifiers together. For example, plan XYZ within project ABC might have 30 builds; the latest build would be identified as ABC-XYZ-30.

Bamboo had a number of extensibility points, including a REST-based API (Atlassian Pty Ltd, 2009a). This API had a number of methods for obtaining build results (Table 15), in addition to other methods for controlling builds that were not used in this project. Unfortunately the build result methods did not provide uniform result sets, so the methods had to be used in combination to obtain more complete build information. In some cases Bamboo did not expose all build information for a particular build due to limitations of the API. In these rare occurrences missing fields could sometimes be inferred from the other information that could be obtained.

Table 15. Methods used from the Atlassian Bamboo REST API.

API Method	Description
Login	Authenticates a user account and provides a time-limited token for use with all other API methods.
getBuildResultsDetails	Retrieves the details for a specified build.
getRecentlyCompletedBuildResultsForBuild	Retrieves a set of recent builds for a specified plan.
getLatestBuildResults	Retrieves a set of recent builds for all projects and plans, including some of the build details. The build information returned by this method is substantially less detailed than the other methods provide.

In order to log and process build information, a software component was constructed to interface with the Bamboo API. This component, described in detail in section 6.3.1, obtained the build information from the Bamboo API and logged the relevant information to a local database. Only a subset of the build information available within Bamboo was required (Table 16).

Table 16. Information retrieved and logged about each build.

Information	Description
Build status	Whether the build passed or failed.
Time and duration of build	The time the build started, and the number of seconds it took to complete. From this, the time the build completed could also be easily determined.
Build identifier	The project, plan, and build identifiers.
Reason for build	The trigger for the build. For integration builds the reason will usually be a code change. For nightly builds it will usually be a schedule. In addition, builds can be manually triggered by a user.
Reason for failure	If the build failed, the reason why it failed - commonly a compilation error or unit test failure.
Commit history	The usernames of all users who committed to this build, as well as the timestamp and number of files in each commit.

6.3. System Architecture

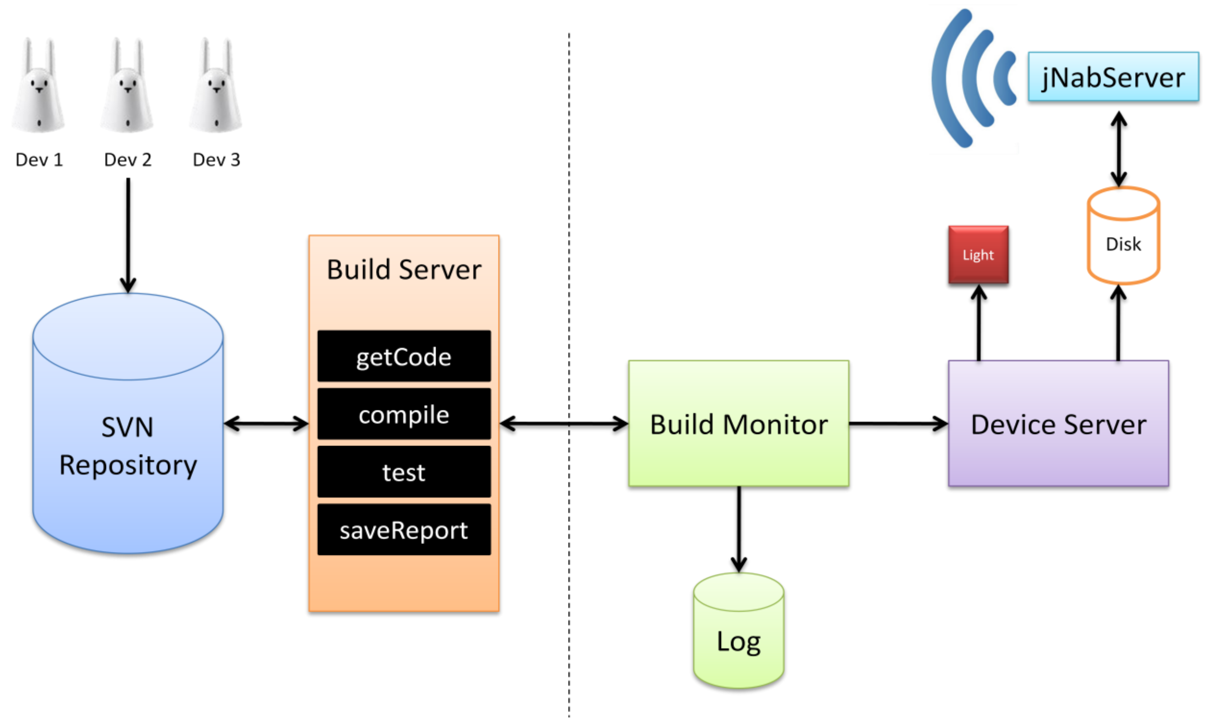


Figure 14. System architecture.

Apart from the jNabServer plug-in, all components were developed using the Microsoft .NET Framework version 3.5 SP1 (Microsoft Corporation, 2009a). The database server was Microsoft SQL Server 2008 Express Edition (Microsoft Corporation, 2009b). Database access code was based on Microsoft LINQ to SQL object-relational mapping technology (Microsoft Corporation, 2007).

For the duration of the case study project, all components were executed on a dedicated PC running on the Microsoft Windows XP operating system.

Three separate prototypes were constructed to enable comparisons to be made between different behaviours. The build monitoring component (discussed in section 6.3.1) and database were not changed between these prototypes. The device server and jNabServer plug-in were modified to enable different behaviour for each prototype, and their implementations are detailed in section 6.3.2.

6.3.1. Build Monitoring Component

The build monitoring component was responsible for communicating with the Bamboo server, monitoring for new build information for the relevant build plans, and logging this information to a database. This component was implemented as a Windows service to ensure it could run satisfactorily for several weeks at a time.

Every sixty seconds, the component retrieved the latest build details from Bamboo and compared these with the build information stored in the database. If any new builds were presented by Bamboo, the component retrieved and parsed the build information, logged it to the database, and notified the device server that new builds were available. The build monitoring component also examined the list of build numbers in each plan during each iteration. If any builds were missing (indicated by non-sequential build numbers), the component used the Bamboo API methods described above to attempt to find and save the build information.

6.3.2. Device Management Components

Two separate pieces of software managed the devices: a device server component and a jNabServer plug-in. These components were modified to enable different behaviour for each prototype.

In order to keep the components loosely coupled, the device server communicated with the jNabServer plug-in by writing marker files to specific folders on the disk. This method of communication was chosen for its simplicity and the fact that a file system is a shared resource which is available in both C# and Java. The jNabServer plug-in could enumerate the marker files and, based on their presence or non-presence, could change its behaviour accordingly.

Prototype 1

The initial prototype was intended to provide the base ambient device functionality for developers. Specifically, this prototype was required to control the central USB light and to notify individual developers when a plan they had just committed to had shifted to a broken state.

Device Server

The device server controlled the ambient awareness hardware devices described in section 6.1. It was fundamentally responsible for transforming input data, in the form of build results, into instructions to the devices. The device server could be instructed to ignore specific build plans: for example, nightly builds could be omitted while integration builds were processed, to ensure the ambient devices were only displaying the most relevant information to team members.

Every thirty seconds, or when triggered by the build monitoring component, the device server examined the database to determine the current state of each plan. If any plans were failing, the central monitoring lamp was turned red, otherwise it turned green.

To manage the developers' rabbits, the device server examined each failing plan, finding the first failing build in the most recent chain of failing builds and determining the list of developers who committed to that build. Each of these developers was marked to receive a build failure notification on their rabbit.

The device server then examined each developer in the system. Developers who had been marked to receive a build failure notification had a marker file placed on the file system in a ‘build failed’ folder for jNabServer to detect. Developers who had been marked not to receive one of these notifications were examined to see if there was already a marker file in the ‘build failed’ folder for that developer from a previous build failure. If there was, the existing ‘build failed’ marker file was deleted, and a new ‘build passed’ marker file was created. This would be detected by jNabServer and would be used to indicate to the developer that the previously failing build had been fixed.

Nabaztag Controller

The Nabaztag rabbits were controlled by jNabServer. This server provided the infrastructure to boot the rabbits, to allow the rabbits to poll for instructions, to receive events from the rabbits, and to play choreographies (sequences of events such as LED changes, ear movements, and sounds) (Rajaniemi, 2007).

The behaviour of the rabbits was managed by a custom plug-in for jNabServer, as well as a number of choreographies. The custom plug-in was written in Java. jNabServer created an instance of the plug-in class for each rabbit that connected to the server, and notified the appropriate plug-in instance when any of the events listed in Table 17 occurred for a specific rabbit.

Table 17. Nabaztag events provided to custom plug-ins by jNabServer.

Event	Description
Ping	When no other events were occurring, the rabbit would ping (poll) the server periodically for instructions. The ping event provided an opportunity to instruct the rabbit to play a choreography, as described below.
Choreography completed	When a choreography finished playing. In our custom plug-in, this caused the same behaviour as a ping event.
Button pressed	When the button on the top of the rabbit was pressed. For the first two prototypes the button pressed events were ignored, but the third prototype did intercept this event.
Ears moved	When the ears were manually moved. This provided the opportunity to move the ears back to their original position, which was achieved by instructing the rabbit to reboot.
RFID tag presented	When the RFID sensor on the rabbit detected a compatible RFID tag within range. For this project the RFID events were ignored.

The primary logic for the plug-in used in prototype 1 was in the ‘ping’ event handler. Approximately every twenty seconds, the rabbit would ping jNabServer. The plug-in simply retrieved the rabbit’s serial number from the event handler context information and then examined the marker folders on disk. Depending on whether an appropriate marker file existed in any of these folders, one of the choreographies described in Table 18 was sent to the rabbit. Additionally, in the case of the BuildPassed choreography, the marker file was deleted to ensure the rabbit would only play the choreography once and then return to the BuildStable choreography.

Table 18. Choreographies for controlling rabbit appearance and behaviour for prototype 1.

Name	Description	Duration	Ear Behaviour	LED Behaviour
BuildStable	Default (neutral) choreography, played when there was no failure or fix to report.	15s (repeating)	Up	None
BuildFailed	Played when the developer breaks a build. Repeated until the plan was fixed.	30s (repeating)	Down	All LEDs red
BuildPassed	Played immediately after the plan was fixed. Played once and then returned to BuildStable choreography.	15s (once)	Up	All LEDs green

If a marker file existed for this rabbit in the 'build failed' folder, the plug-in sent the BuildFailed choreography; if a marker file was present in the 'build passed' folder, the plug-in sent the BuildPassed choreography. If neither file was found, the BuildStable choreography was sent.

Choreography Timing

Each element of a choreography could be timed to occur at a specific offset. For example, a choreography could instruct a rabbit to first move its ears, then turn a set of LEDs on after 10 seconds. During the development of the choreographies listed in Table 18 it became apparent that the timing logic described in the jNabServer source code and the accompanying article (Rajaniemi, 2007) was different to the actual behaviour. Choreography timing was achieved through the manipulation of two parameters: tempo and frequency ratio. We determined that the actual timing formula was:

$$t = \frac{f}{2} \times \frac{t_s}{10}$$

where t was the time offset for the sub-event within the choreography (in seconds), f was the frequency ratio specified, and t_s was the tempo specified.

Prototype 2

Based on feedback from participants, the initial prototype was modified approximately two weeks after the devices were provided to the team. The key changes for this prototype were to alert all users who committed to a failing build, and to avoid the rabbits' nose LEDs flashing every time they pinged the server.

Device Server

The server component remained largely unchanged from the initial prototype. The only modification was to the logic that determined which developers should be notified of a build failure. Instead of alerting just those developers who committed to the first failing build in each failing plan, the new prototype obtained a list of all developers who had committed since the plan began failing and marked each of these developers for an alert.

Nabaztag Controller

The jNabServer plug-in was modified to remove the LED nose flash behaviour when the rabbit was not displaying a red or green notification.

It was found that that the nose flashed when a ping event was handled by the custom plug-in and a choreography was sent to the rabbit. The behaviour could therefore be eliminated by removing the BuildStable choreography. When no notification was required for a particular developer's rabbit, no response was sent to the ping event. This did not have any effect on the behaviour of the system except to eliminate the LED nose flashing when a ping event was handled.

Prototype 3

The third prototype was designed to explore the use of the rabbits as a mechanism for delivery of positive reinforcement and positive punishment, rather than as pure ambient devices, as discussed in Chapter Five. This modification was made approximately one month after the installation of prototype 2.

In order to provide positive reinforcement, developers were to be given notifications whenever a source code commit was received and processed by the build server. As in the first two prototypes, if the build associated with that commit failed, their rabbit was to be instructed to play the BuildFailed choreography. However, unlike the previous prototypes, when a build passed the users who committed to that build were to be reinforced by having their rabbit play the BuildPassed choreography for a set period of time (thirty minutes). Previous prototypes only played the BuildPassed choreography for a short period, and only for developers who had just been played a BuildFailed choreography, in order to alert these developers that the problem had been resolved.

Device Server

A naïve implementation of this strategy would be to evaluate all builds during each pass of the device server logic, examining the last commit for each developer and generating the appropriate marker files. However, this logic would be insufficient for a number of reasons.

First, a given developer may have committed to more than one plan simultaneously or in quick succession: some preliminary analysis of the first two weeks' results showed that developers often committed to multiple plans within seconds or minutes, and any of these could fail. If the system simply used the most recent commit by a given user, the arbitrary order in which the builds were processed by the build server and the device server would determine whether the user received reinforcement or punishment.

Second, an individual developer's last commit may not necessarily have been the last commit to the plan. For example, developer D_A may commit code to plan P, which would result in build B_1 failing. Shortly thereafter, developer D_B may commit code to plan P and (intentionally or inadvertently) fix the problem, thereby causing build B_2 to pass. The naive logic described above would punish D_B despite the code having been corrected and the plan passing.

Third, developers on the team expressed a desire to receive reminders when a plan remained in a failing state for a given period of time. Specifically, developers wished to have a reminder every thirty minutes in the form of the rabbits' ears moving. However multiple plans may fail simultaneously, and only a subset of these may be fixed within the reminder window. Information was required to be kept on each plan's results, as well as the history of these reminders, to ensure that they were generated at appropriate times and only for plans which remained in a failing state for more than thirty minutes.

Fourth, the 'build passed' notification behaviour was changed so that it would remain on the developer's rabbit for thirty minutes, rather than playing once for fifteen seconds and then returning to the 'neutral' state. This was intended to ensure that if a developer temporarily left their desk after committing code, or was not otherwise paying attention to their rabbit, they would still be able to receive this notification. The time that each of these notifications were first sent to the rabbits was tracked so the notifications could be automatically expired after thirty minutes. Developers could also press the button on their rabbits to immediately expire the notification if they wished.

Given these conditions, a considerably more complex algorithm was developed for the device server based on state transitions. A matrix of developers, plans, and statuses was initialised when the device server started and was updated as new builds occurred. This matrix assumed that, for a given plan, a developer was in one of three states (Table 19).

Table 19. States used for each developer-plan combination in prototype 3.

State	Priority	Description
Fail	3 (highest)	The developer recently committed code to this plan, and the plan is currently failing.
Pass	2	The developer recently committed code to this plan, and the plan is currently passing.
Neutral	1 (lowest)	The developer has not recently committed code to this plan, so this plan should not be taken into account when calculating the final result for this developer.

The device server algorithm was executed every thirty seconds, as well as whenever the build monitoring component recorded a new build. This process is described below; pseudocode is included in Appendix D.

The central light was updated in the same way as previous prototypes: if any plans were in a failing state the lamp would turn red; otherwise, it would turn green.

When the device server started, a matrix was created with each cell corresponding to a developer and plan combination. For example, if there were three plans and four developers, there would be a 3x4 table with a total of twelve cells.

Each time the device server algorithm was executed the server evaluated each developer-plan combination and updated the cell based on a set of rules. If the developer had not committed to that plan since the last time it successfully built, the cell was updated to a ‘neutral ‘state. If the developer had recently committed to the plan, and if the plan had changed state (for example, from failing to passing) since the last time the algorithm was executed, the cell was updated accordingly.

A timestamp of the last change was also recorded in each cell to allow reminders and expirations to be processed. If a cell remained in a ‘pass’ state for more than thirty minutes without being updated, the ‘pass’ was changed to a ‘neutral’ state. Similarly, if a cell remained in a ‘fail’ state for more than thirty minutes without being updated, a reminder was triggered. These reminders are discussed in more detail below.

At the conclusion of this updating procedure, an overall status was calculated for each developer. This was achieved by examining that developer’s row within the matrix and sorting it according to the priorities described in Table 19. The highest priority cell's status became that developer's overall status, which was then used to generate the marker files in the same way as the previous prototypes – although with the addition of a new type of marker file to indicate a reminder was required for a still-failing build.

As an illustrative example, consider a case where the device server was started at 12.30pm. For simplicity, we assume that at this point no builds have been recorded in the system. We also assume that the algorithm executes every thirty minutes (instead of every thirty seconds). Initially, a neutral table would be created and all cells would be initialised to a 'neutral' state, and therefore the overall state for each developer will also be 'neutral' (Table 20).

Table 20. Example device server matrix as initialised at 12.30pm.

	Plan A	Plan B	Plan C	Overall
Developer 1	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral
Developer 2	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral
Developer 3	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral
Developer 4	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral

At 1.00pm, the device server algorithm executes. During the preceding thirty minutes, developer 1 and developer 3 have committed code to plan A, which is now failing. Developer 2 and developer 3 have committed to plan B, which is now passing. Developer 4 has not committed code to any plan. The matrix as at 1.00pm is shown in Table 21, including the overall state for each developer as derived from that developer's row. Note that developer 3 receives a 'fail' overall status as this takes precedence over the 'pass' they receive for plan B.

Table 21. Example device server matrix as at 1.00pm.

	Plan A	Plan B	Plan C	Overall
Developer 1	Status: **Fail** Timestamp: 1.00pm Last fail reminder: None	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	**Fail**
Developer 2	Status: Neutral Timestamp: 12.30pm	Status: **Pass** Timestamp: 1.00pm	Status: Neutral Timestamp: 12.30pm	**Pass**
Developer 3	Status: **Fail** Timestamp: 1.00pm Last fail reminder: None	Status: **Pass** Timestamp: 1.00pm	Status: Neutral Timestamp: 12.30pm	**Fail**
Developer 4	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral

At 1.30pm, the device server algorithm executes again. As no new builds have been recorded since the last execution, the algorithm searches for expired 'pass' statuses and 'fail' statuses requiring a reminder. In this example, developers 2 and 3 have previously been given a 'pass' status for plan B, but these now expire and change back to 'neutral'. Developer 1 and 3 have been in a 'fail' state for plan A for more than thirty minutes, so the system marks them as requiring a failure reminder and saves the timestamp that this reminder was generated. Table 22 illustrates the state of the matrix at the conclusion of this execution.

Table 22. Example device server matrix as at 1.30pm.

	Plan A	Plan B	Plan C	Overall
Developer 1	Status: **Fail** Timestamp: 1.00pm Last fail reminder: 1.30pm **Generate reminder marker**	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	**Fail**
Developer 2	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 1.30pm	Status: Neutral Timestamp: 12.30pm	Neutral
Developer 3	Status: **Fail** Timestamp: 1.00pm Last fail reminder: 1.30pm **Generate reminder marker**	Status: Neutral Timestamp: 1.30pm	Status: Neutral Timestamp: 12.30pm	**Fail**
Developer 4	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral

At 2.00pm, the algorithm executes once again. In this case, developer 2 has committed to plan A and has fixed it so that it now successfully builds. All developers who previously received a 'fail' status for this plan, as well as developer 2, are marked to now receive a 'pass' notification for this plan.

Table 23. Example device server matrix as at 2.00pm.

	Plan A	Plan B	Plan C	Overall
Developer 1	Status: Pass Timestamp: 2.00pm	Status: Neutral Timestamp: 12.00pm	Status: Neutral Timestamp: 12.30pm	**Pass**
Developer 2	Status: Pass Timestamp: 2.00pm	Status: Neutral Timestamp: 1.30pm	Status: Neutral Timestamp: 12.30pm	**Pass**
Developer 3	Status: Pass Timestamp: 2.00pm	Status: Neutral Timestamp: 1.30pm	Status: Neutral Timestamp: 12.30pm	**Pass**
Developer 4	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Status: Neutral Timestamp: 12.30pm	Neutral

Nabaztag Controller

The jNabServer plug-in was modified in three ways for this prototype.

First, the green marker files were no longer automatically deleted by the plug-in after the choreography had been delivered to the rabbit. This enabled the choreography to be displayed for thirty minutes.

Second, the rabbit's button click events were intercepted to delete the green marker file when the developer manually requested this to occur. This would switch the developer to the 'neutral' state.

Third, the plug-in checked for the existence of the 'red reminder' marker file; when this marker file was detected the plug-in delivered the new BuildFailedReminder choreography and deleted the 'red reminder' marker file. The new choreography is described in Table 24. After the BuildFailedReminder choreography played once, the rabbit would resume playing the BuildFailed choreography.

Table 24. Choreography added for prototype 3.

Name	Description	Duration	Ear Behaviour	LED Behaviour
BuildFailedReminder	A plan has been failing for more than 30 minutes. (This choreography was sent every 30 minutes until the plan was fixed.)	15s (once)	Complete rotation, ending down	All LEDs red

Unit Testing

Due to the complexity involved in the algorithm developed for prototype 3, a suite of unit tests was constructed to validate the device server functioned correctly. Each of the twenty unit tests corresponded to a test case in which specific builds with a set of commits were provided to the device server in a defined order. The unit tests validated that the algorithm correctly updated the matrix and gave the expected final result for each developer. The set of test cases was designed to exercise as much of the logic as practicable, including boundary cases. Unit tests were written and executed using the Microsoft Visual Studio 2008 Unit Testing Framework (Microsoft Corporation, 2009c).

Comparison of Prototypes

For illustrative purposes, Table 25 lists an example sequence of events and compares the behaviour of the three prototypes. In this example, two developers (D_a and D_b) are working together on a single plan, and a total of five builds (B_1-B_5) are performed. The behaviours of each of the ambient devices (the central light, C, and the developers' rabbits, R_a and R_b) are compared. For the rabbits, 'green' refers to a BuildPassed choreography being played while 'red' refers to a BuildFailed choreography being played.

Table 25. Example scenario describing rabbit behaviour for each prototype.

Time	Event	Behaviour		
		Prototype 1	Prototype 2	Prototype 3
T_1	D_a does not check in code.	C green.	C green.	C green.
	D_b does not check in code.	R_a stable.	R_a stable.	R_a stable.
	Build B_1 occurs.	R_b stable.	R_b stable.	R_b stable.
T_2	D_a checks in good code.	C green.	C green.	C green.
	D_b does not check in code.	R_a stable.	R_a stable.	R_a green.
	Build B_2 occurs.	R_b stable.	R_b stable.	R_b stable.
T_3	D_a checks in faulty code.	C red.	C red.	C red.
	D_b does not check in code.	R_a red.	R_a red.	R_a red.
	Build B_3 occurs.	R_b stable.	R_b stable.	R_b stable.
T_4	D_a does not check in code.	C red.	C red.	C red.
	D_b checks in faulty code.	R_a red.	R_a red.	R_a red.
	Build B_4 occurs.	R_b stable.	R_b red.	R_b red.
T_5	D_a checks in fixes to all problems.	C green.	C green.	C green.
	D_b does not check in code.	R_a green.	R_a green.	R_a green.
	Build B_5 occurs.	R_b green.	R_b green.	R_b green.

6.4. Implementation Experiences

A key requirement of all the prototypes was that they function for extended periods of time without any manual intervention. In order to satisfy this requirement robust error handling was implemented at all levels of the system: exceptions were caught and logged before a retry attempt made; network connectivity was assumed to be unreliable and components were designed to back off and retry when making unsuccessful network requests; components such as the build monitor, device server, and jNabServer were encapsulated in long-running Windows services with policies to automatically restart them if fatal errors occurred; and a comprehensive tracing and diagnostic framework was implemented to log all activity and allow for troubleshooting in case problems did arise. Allowing this level of availability required a considerable amount of thought and code, but ultimately enabled the production of a robust and reliable system.

The software implementation of each prototype presented different challenges. The initial prototype required the most development effort, specifically in architecting the components as well as writing the necessary code and performing unit and functional testing. The modular design of the system made this process substantially less time consuming than it may have otherwise been. The second prototype was relatively straightforward to design and implement, requiring only a small number of changes from the initial prototype as well as some testing.

The third prototype required a great deal of redevelopment within the device server. A new algorithm, involving multidimensional data and state transitions, was required to satisfy the additional requirements for this prototype. Due to the added complexity of the algorithm and the number of boundary cases, the third prototype also required extensive testing and validation.

One of the largest challenges in this project was actually obtaining the Nabaztag rabbits. The manufacturer, Violet, filed for bankruptcy while we were in the process of purchasing the rabbits, and they became extraordinarily difficult to obtain. After spending several weeks attempting to locate the necessary number of rabbits, we ultimately purchased nine devices from three different suppliers in Europe and the United States, and were permitted to borrow two additional devices from the University of Tampere in Finland for the duration of the project.

6.5. Summary

This chapter described the implementation of an ambient awareness system for build notifications based on the requirements and constraints discussed in the previous chapters. A number of individual components were created to log, process, and store build information, transform the information into instructions for the ambient devices, and send those instructions to the hardware. While some of these components were specific to the build server the case study company used, the loosely coupled architecture and general similarities between build servers would allow most of this implementation to be reused for other organisations and other build servers. The next chapter discusses the evaluation of each of these prototypes with the case study team.

Chapter Seven
Evaluation

The previous chapters described the design and implementation of a series of prototypes to provide ambient awareness of build status information to members of a software development team. To evaluate the usefulness and impact of these technologies, an evaluation study was conducted in which both quantitative and qualitative information was collected.

Quantitative information was based on the build information collected during the studies. Section 7.1 describes a number of metrics that were calculated based on the build log data, intended to correspond closely to the research hypotheses described in Chapter Five. Qualitative information was collected through a number of sources: formal interviews with developers as well as informal conversations, emails, and electronic communication with and between the development team. Sections 7.2 and 7.3 describe these qualitative findings.

One member of the case study team worked remotely for the duration of the evaluation study. In addition, other developers inside the organisation, but not on the case study team, would occasionally commit code to the build plans. In these cases their information has been included in the quantitative analysis below, as removing their information from the build logs would have been impossible since they were committing to the same build plans as the rest of the team. However, these developers were not given any ambient devices, nor were they interviewed or otherwise generally included in the qualitative data collected during this study.

7.1. Build Server Metrics

The build monitoring component, described in the previous chapter, compiled a comprehensive dataset of builds and commits as it executed. This information was not only used for real-time control of the ambient devices but also for subsequent post-hoc analysis of the build data. The data collected was sufficient to allow us to evaluate each of the research hypotheses outlined in Chapter Five (Table 14, p68).

Baseline data were collected for a period of approximately one month before the ambient devices were provided to the team. Therefore, there were four study conditions: this baseline and the three prototype interventions described in Chapter Six.

7.1.1. Inferential Statistical Analyses

For the metrics described in this section, no inferential statistics (such as chi squares, t tests, or analyses of variance) have been performed. The decision not to perform such testing was made deliberately and was based on the best practices for the design of case studies and other small sample studies.

Owens (1979) argues that it is critical to choose the right type of analysis for a given study. Any analysis methodology will include a range of assumptions, caveats, and limitations. Inferential testing is predicated on a core assumption that the sample data is representative, either through random selection or through a rigorous attempt to eliminate all reasonable biases (Howell, 2004). If and only if the sample meets this criterion, is sufficiently large in size, has a matched control sample, and meets the other requirements for these tests, it can be subject to inferential analysis and legitimate conclusions can be drawn about the population at large.

Within the field of psychology a number of paradigms (including cognitive science, social psychology, and clinical psychology) meet these criteria and legitimately use inferential analysis as a matter of course. Within other areas of psychology, such as the experimental analysis of behaviour (EAB), a very small number of subjects are typically used for a given study (Barlow & Hersen, 1984). Although inferential analysis can be performed on the raw data generated by these studies, the analysis will be invalid due to the small number of subjects. As such, a variety of other approaches, all using simple descriptive statistics (means, moving means, standard deviations, and other averages) are employed.

Information systems experimental research consists of a range of different approaches, including many case studies (McKay & Marshall, 2000). As discussed in Chapter Five, the case study approach provides a great number of advantages. However, the small sample size necessarily precludes the possibility of inferential analysis, and accordingly does not permit any definitive conclusions about a population to be drawn from the sample. A given observation within a single sample could be a real effect, or it could be random variation due to other factors (Barlow & Hersen, 1984). Case studies and other small-sample designs do not provide a sufficient quantity of data to allow these possibilities to be distinguished. Rather than an indictment upon case study methodologies in general, this is simply a feature of these types of study, and alternatives to inferential testing are available for the analysis of quantitative case study data, such as the use of baseline-intervention designs and the descriptive statistics listed above (Elashoff & Thoresen, 1978).

In this study baseline data were collected before any interventions took place. The same information was collected for each of the three prototype interventions described in Chapter Six. While an ideal study would include a return to baseline, followed by a reintroduction of interventions, time constraints prevented us from performing this type of repeated measures design. Instead, the prototype interventions are compared to the baseline and to each other, thereby using the different study phases as controls for each other.

7.1.2. Excluded Builds

Some builds were excluded from this analysis. Three types of builds were excluded for different reasons: nightly builds; unnecessary builds that may skew the results unfairly; and builds deleted from the build server before the build monitoring component could log them.

- **Nightly builds**. The integration builds were more tightly integrated into the developers' day-to-day workflow, and the nightly builds covered most of the same testing ground as the integration builds, so were excluded from this analysis.
- **Failed builds due to unit test communication failures**. During our discussions with the team we were informed that the architecture of the test lab used by the build process meant that there were cases where builds would fail due to unit test communication errors. The team leader informed us that we could identify these cases relatively easily: if a failing build with source code changes was followed by a manual build with no changes, this indicated that a developer manually retried a build and it passed successfully. The fact that there were no commits on this second build indicated that the initial problem was not with the source code. In these cases, the build data were adjusted: the last failing build (with commits) was considered to have actually passed, and the manual build was excluded from this analysis.
- **Builds deleted from Bamboo.** Some builds were deleted from the build server logs before the build monitoring system could record the information. Builds can be deleted from Bamboo's history for several reasons, including a build being cancelled while in progress (in which case it assigns the next sequential build number when it restarts), a build server failure part-way through a build, or when a team member manually deletes a build. If a build server failure occurred, or a manual deletion occurred before the build monitoring component obtained the build information, the build monitoring system would record it as a deleted build and would not have any further information available. If a manual deletion occurred after the build monitoring component had received the information it would not affect the analysis, since the component only wrote the data to its database once and did

not honour Bamboo's deletion. Only a small proportion of the builds within each condition were deleted in this way.

Analyses are performed on the remaining builds within each condition. Table 26 lists the total original number of integration builds per condition, as well as the number of manual builds excluded from analysis, the number of builds deleted by Bamboo, and the remaining builds that were included within the metrics described below.

Table 26. Total and deleted integration builds by condition.

Condition	Total Integration Builds	Deleted from Bamboo	Manual Builds Excluded	Total Builds Analysed
Baseline	161	10	10	141
Prototype 1	147	6	19	122
Prototype 2	315	10	17	288
Prototype 3	257	7	18	232
Total	**880**	**33**	**64**	**783**

7.1.3. Results

The following sections outline the metrics calculated, including a description of the measurement process for each metric and the actual results. Charts with means and proportions are provided in this chapter. Where applicable, standard deviations are presented as error bars. A complete set of numeric results is provided in Appendix E.

Broken Builds per Day

This metric considered the number of broken builds per day and the percentage of that day's total that they comprised. For the purposes of this metric only workdays were considered (i.e. Monday through Friday, and weekend days were only considered when builds were observed on those days). The baseline condition lasted for 24 workdays, prototype 1 lasted for 10 workdays, prototype 2 lasted for 20 workdays, and prototype 3 lasted for 16 workdays.

As the study conditions were of variable lengths, the percentages are more useful as a comparison than the absolute values. Figure 15 illustrates the mean percentage of broken builds per day as a function of the study condition.

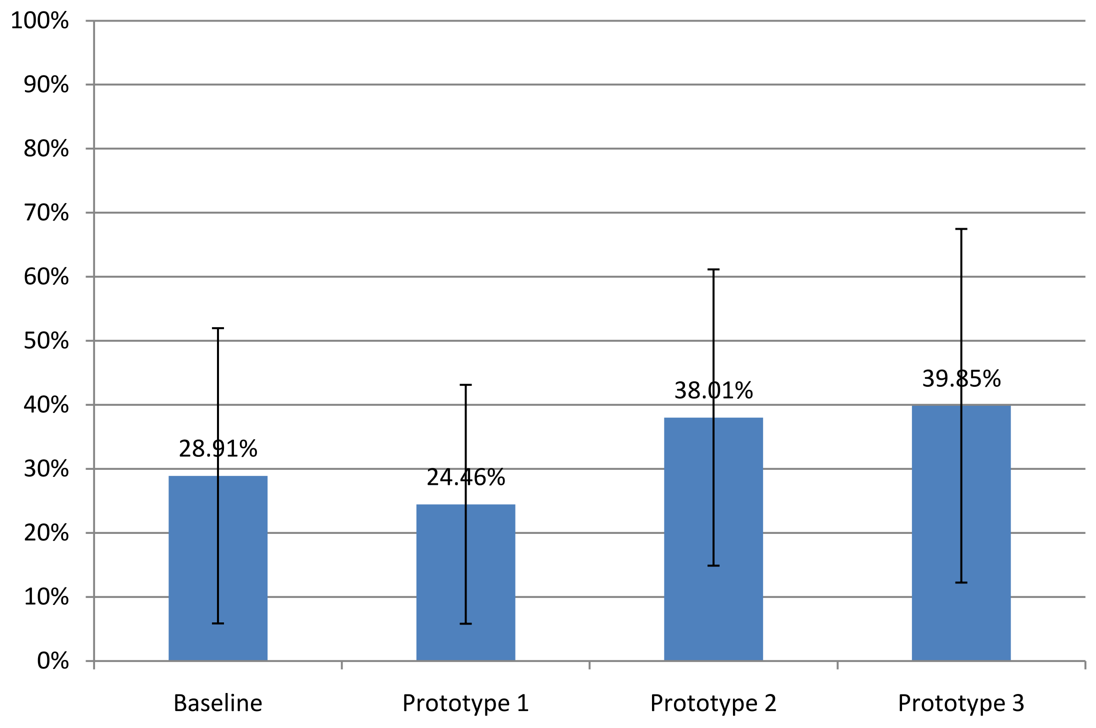


Figure 15. Mean percentage of broken builds per day by condition.

Additionally, Figure 16 illustrates the percentage of broken builds per day over the course of the entire study, including a moving mean.

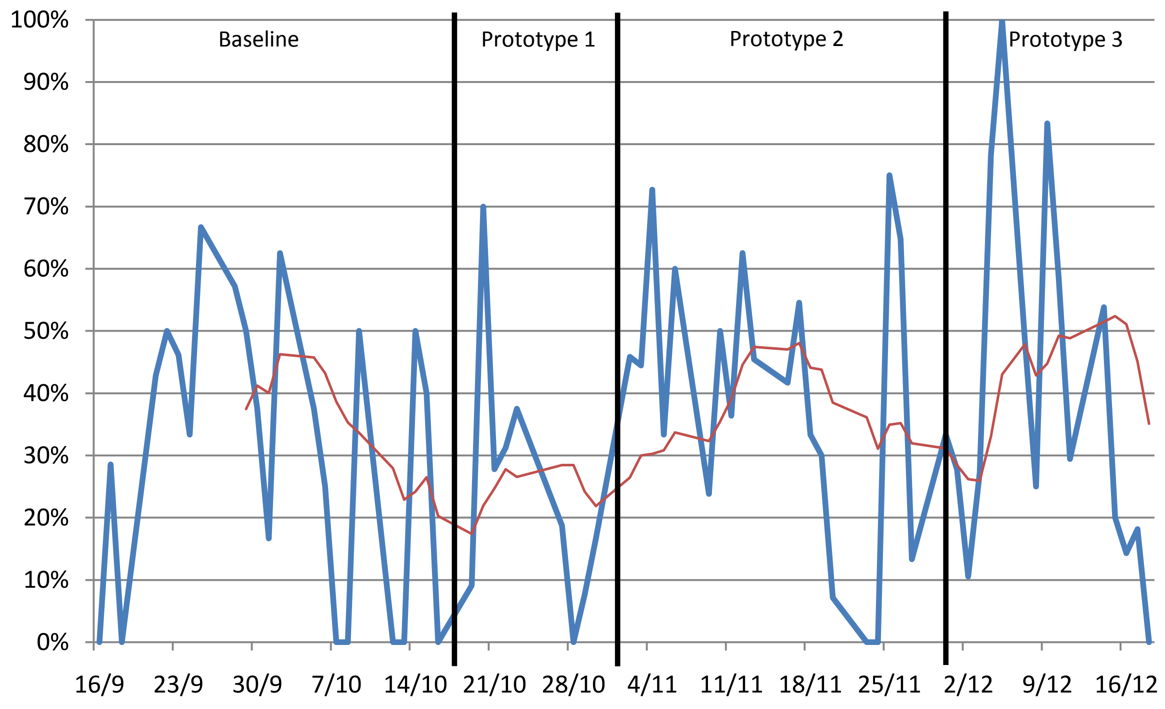


Figure 16. Percentage of total builds classified as broken per study day, including a 10 day moving mean.

Total Builds per Day

Another metric was the mean number of total builds per day. This metric did not consider whether the build passed or failed, but was simply a measure of the total number of integration builds that occurred each day. As with the previous metric, only workdays were considered.

Figure 17 shows the mean number of builds per day by condition.

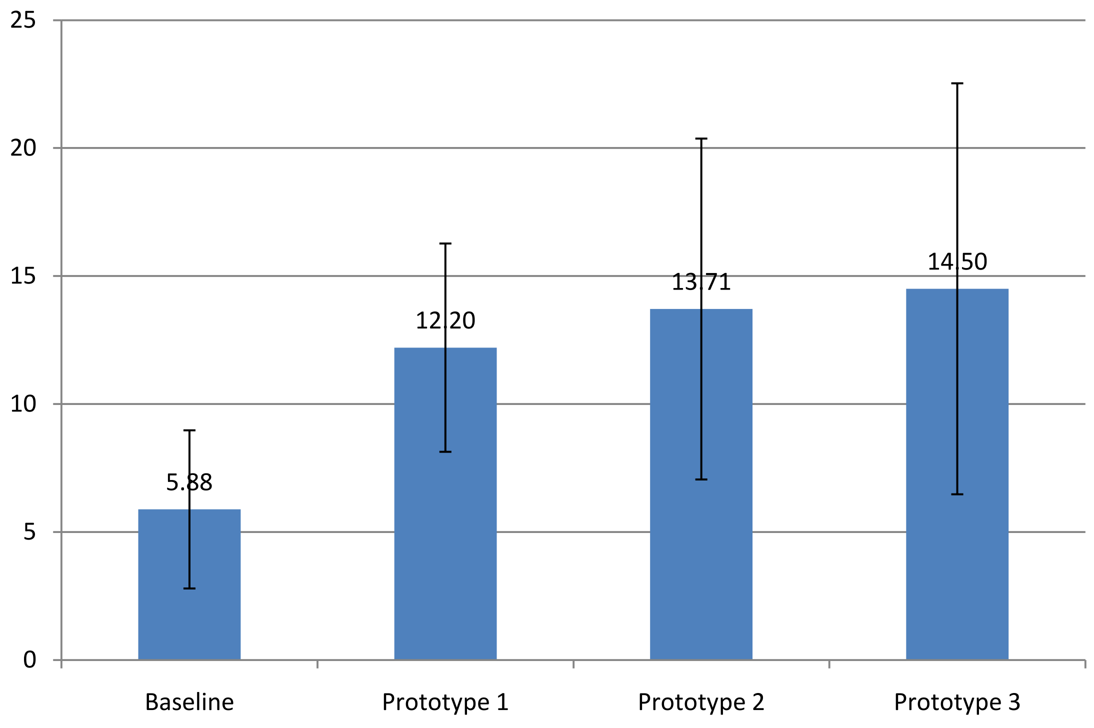


Figure 17. Mean number of builds per day by condition.

Additionally, Figure 18 illustrates the total builds per day over the course of the entire study with a moving mean.

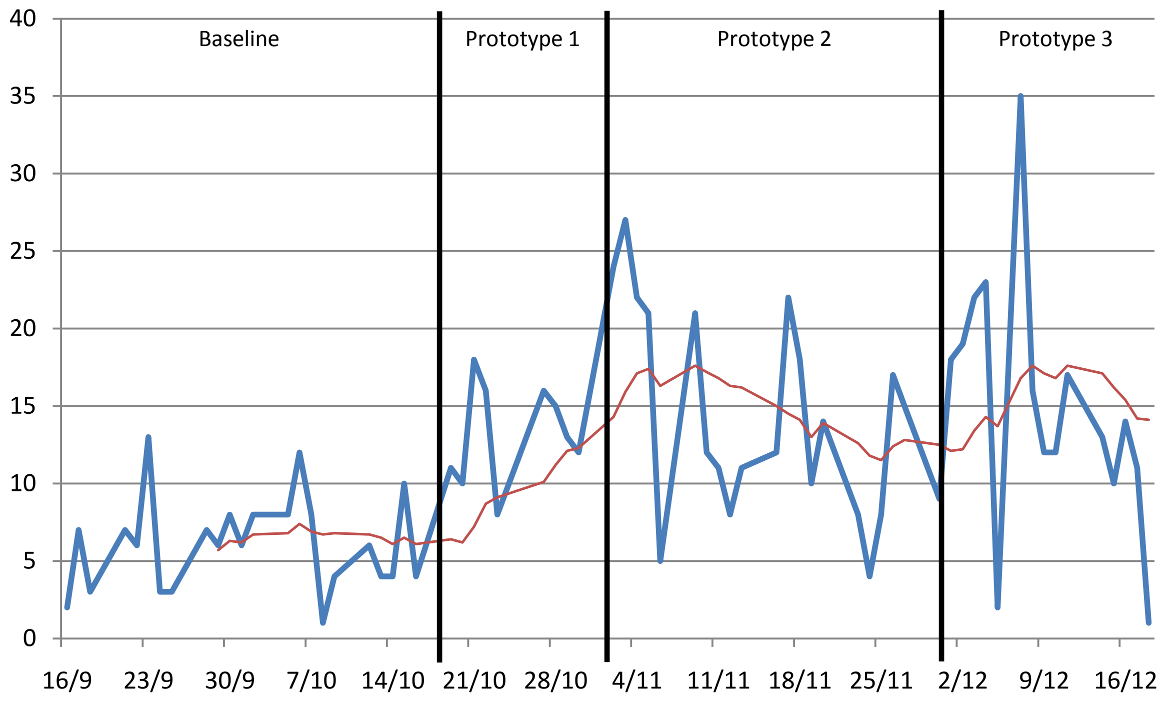


Figure 18. Total builds per day for each study day, including a 10 day moving mean.

Files per Changeset

Every build has an associated changeset of zero or more commits. These commits correspond to developers' commits to the source code repository, and may involve one or more files. Figure 19 illustrates an example build with a changeset of three commits, each containing a number of source code files. In total, seven files were modified in this example changeset.

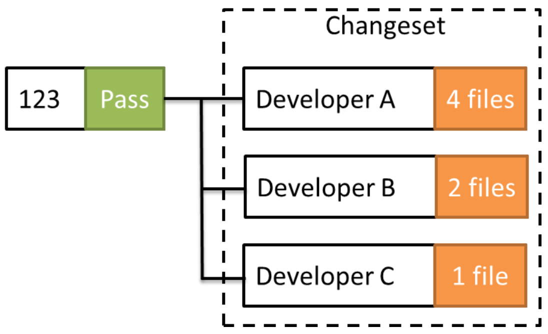


Figure 19. Example build with a changeset comprising three commits and seven files.

Any given build may have been triggered manually, in which case it will likely not be associated with any commits and will therefore have an empty changeset.

This metric compared the mean number of files per changeset by condition, excluding empty changesets. The results are illustrated in Figure 20.

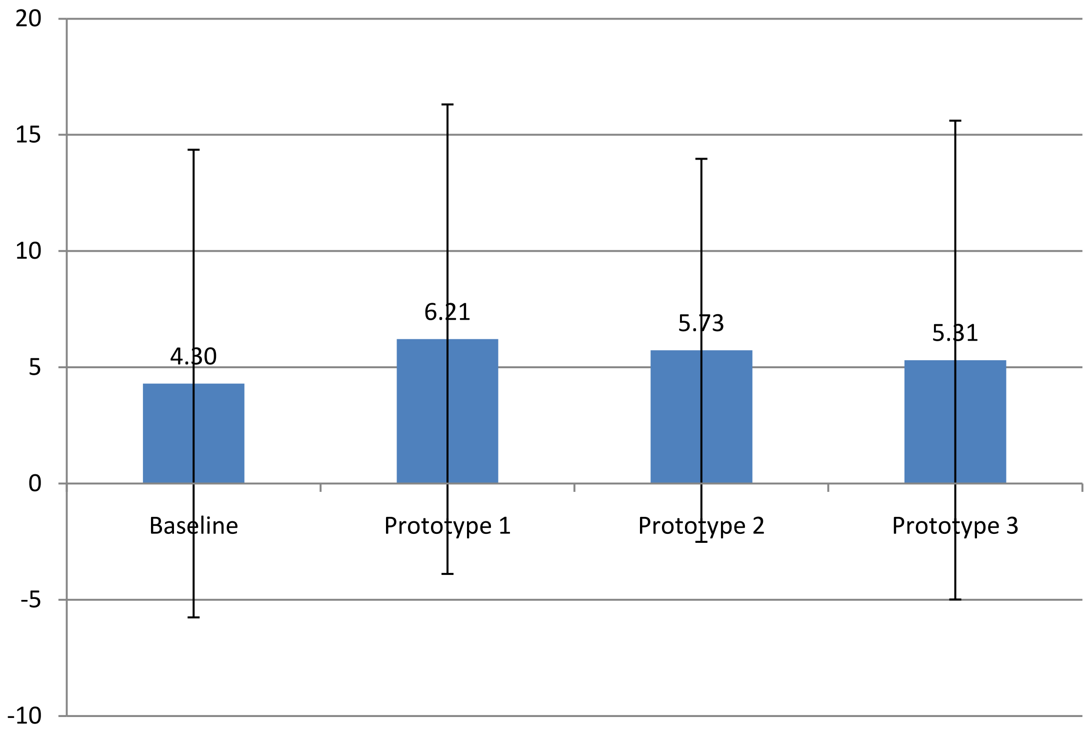


Figure 20. Mean files per changeset by condition.

Commits During Broken Builds

This metric compared the number of commits that occurred during broken builds. In order to determine this, each build plan was examined and a list of broken build chains was compiled for each plan. Broken build chains were defined as one or more sequential failing builds and were terminated by the next passing build. An example set of two broken build chains is illustrated in Figure 21.

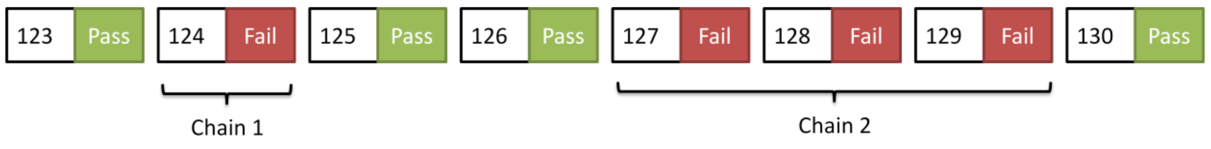


Figure 21. Broken build chains.

Unlike the previous metric, rather than examining the number of files changed, this metric focused on the number of commits to the source code repository for a build. For example, the situation illustrated in Figure 19 (p95) was considered to have three commits, each containing a different number of files.

For this metric all broken build chains were examined within each plan, and the total number of commits that were recorded within each chain was calculated. The initial build within each chain was not included (for example, builds 124 and 127 in Figure 21 were not considered) because commits to these builds occurred before the plan began failing.

Figure 22 illustrates the mean number of commits to broken build chains for each study condition.

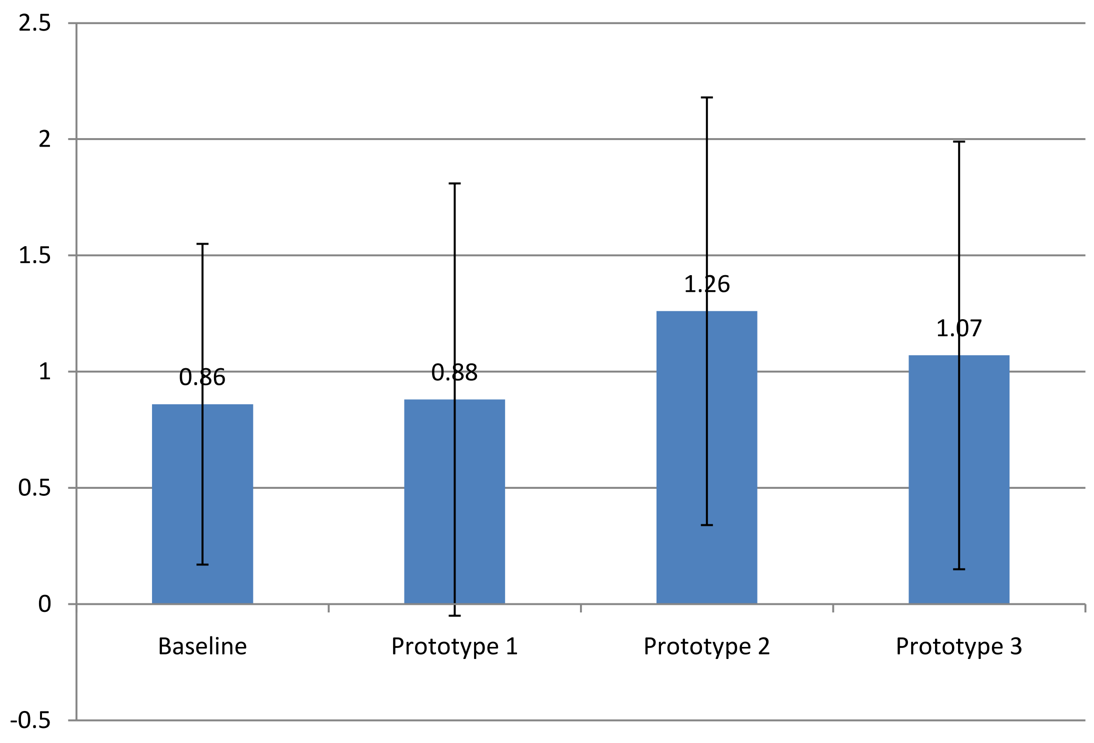


Figure 22. Mean commits to broken build chains by condition.

Reasons for Builds

Integration builds could occur for a number of reasons. The build server automatically performed an initial build whenever a build plan was created or recreated; builds were triggered when source code was changed (after a timeout period, to allow multiple developers' changes to be integrated into a single build); and developers could instruct the build server to perform a manual build at any time. This metric quantified the number and proportion of builds occurring for each of these reasons within each condition.

As the conditions were not equal in length or number of builds, it is more meaningful to compare the proportions of build occurring for each reason between each condition. Figure 23 illustrates the proportion of builds as a function of the trigger reason for each study condition.

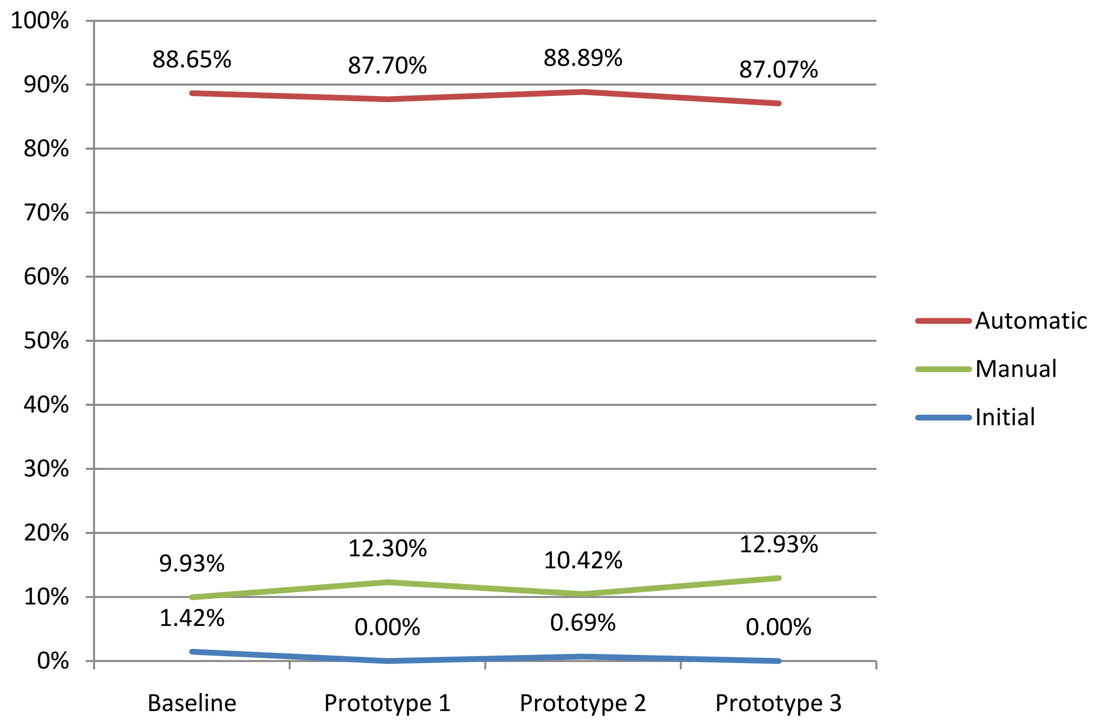


Figure 23. Proportion of builds as a function of trigger reason, by condition.

Cause of Broken Builds

The build server recorded limited information describing the reason a build failed. The only reason that could reliably be derived from the build information was 'test failure', indicating that one or more unit tests had failed. Other problems included compilation errors, network outages, and other infrastructure issues. Unfortunately these causes could not be distinguished, so they were grouped together as 'other' causes of build failures.

As with some other metrics, it is more meaningful to compare the proportions of broken builds by cause and condition. Figure 24 illustrates this information.

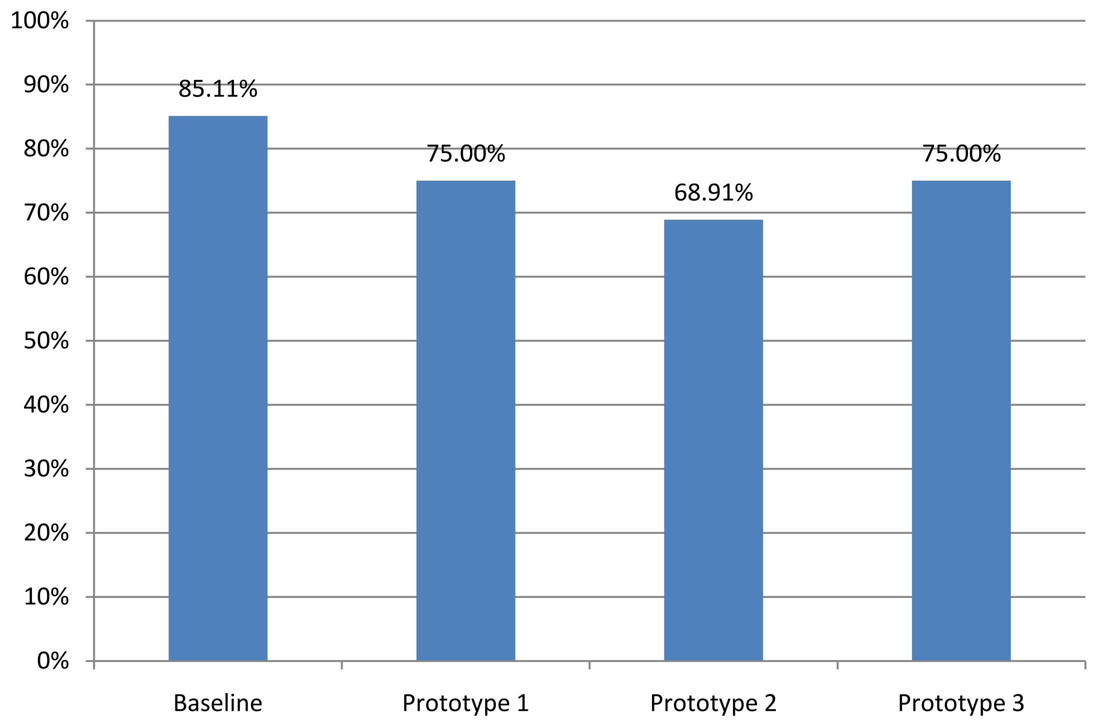


Figure 24. Percentage of broken builds caused by unit test failures, by condition.

Fixing Own Builds

This metric was intended to find cases where developers fixed a failing build that they had committed to. To calculate this metric, a list was constructed for each broken build chain containing the developers who committed to any build within that chain. This list was compared with the developers who committed to the next passing build. If the next passing build contained a committer from the list, the broken build chain was considered to have been fixed by a breaker; otherwise, it was not. This was based on the assumption that a developer who committed to a passing build following a broken build chain was likely to have been responsible for fixing the breakage.

For example, consider builds 127 through 130 in Figure 21. If developer D committed to build 127, and also committed to build 130, this was considered a broken build chain fixed by a breaker.

Figure 25 illustrates the percentages of broken build chains fixed by a breaker by each study condition.

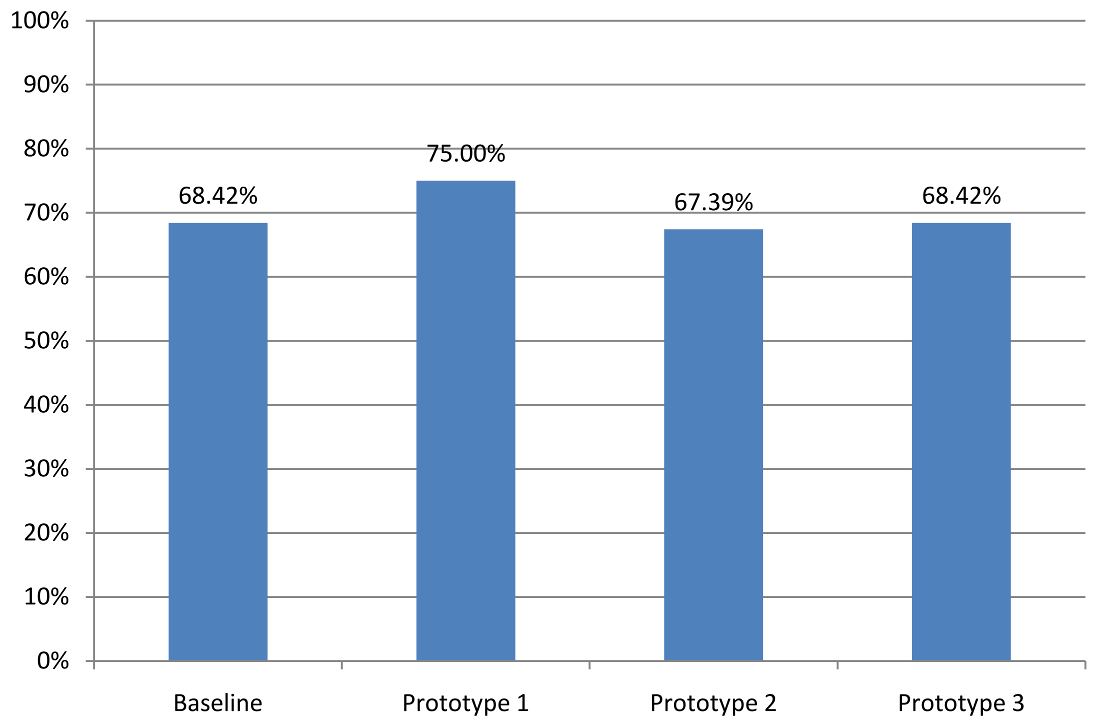


Figure 25. Percentage of broken build chains fixed by breaker by condition.

Duration of Broken Builds

This metric examined the time in which a build remained in a broken state. The duration of a broken build was defined as the time between when the first build within the broken build chain ended to when the next passing build began (Figure 26).

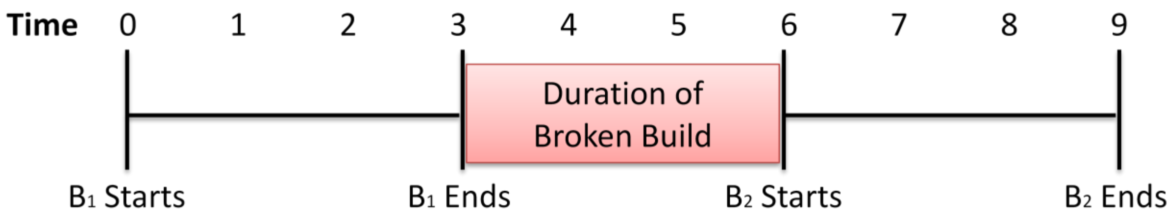


Figure 26. Example builds illustrating the calculation of the duration of broken builds for a build chain of length 1.

This timing was chosen to ensure that the time the builds took to complete was not included in the duration: these could be subject to a number of environmental and transient issues unrelated to the development itself and would add a confounding factor to the analysis. In cases where multiple builds broke in succession, however, the time of intermediary builds could not be meaningfully factored out. Figure 27 illustrates this situation using an example where builds B_1 through B_4 failed and B_5 passed, thereby terminating the broken build chain.

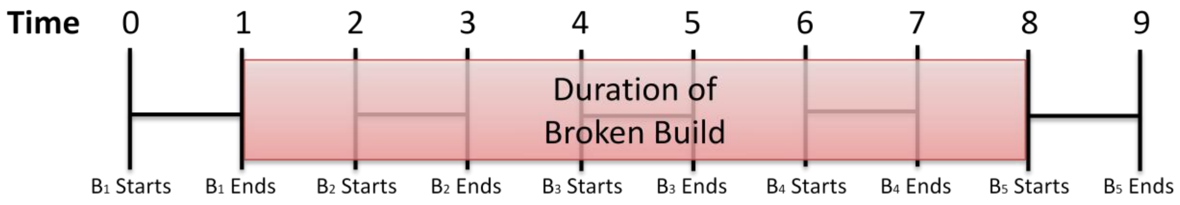


Figure 27. Example builds illustrating the calculation of the duration of broken builds for a build chain of length 4.

An initial analysis of these results showed that, on average, the durations were substantially longer than expected. An investigation found that some broken build chains were fixed over a period of two or more days, thereby increasing the average duration as a whole.

To account for this, the broken build chains were split into two groups: those which were broken and fixed within a single day, and those which were broken and fixed over multiple days. Based on this separation of the broken build chains, three separate sub-metrics were calculated.

Figure 28 illustrates the proportion of broken build chains that were fixed within a single day. An increase in this metric indicates that broken build chains are generally being fixed more quickly.

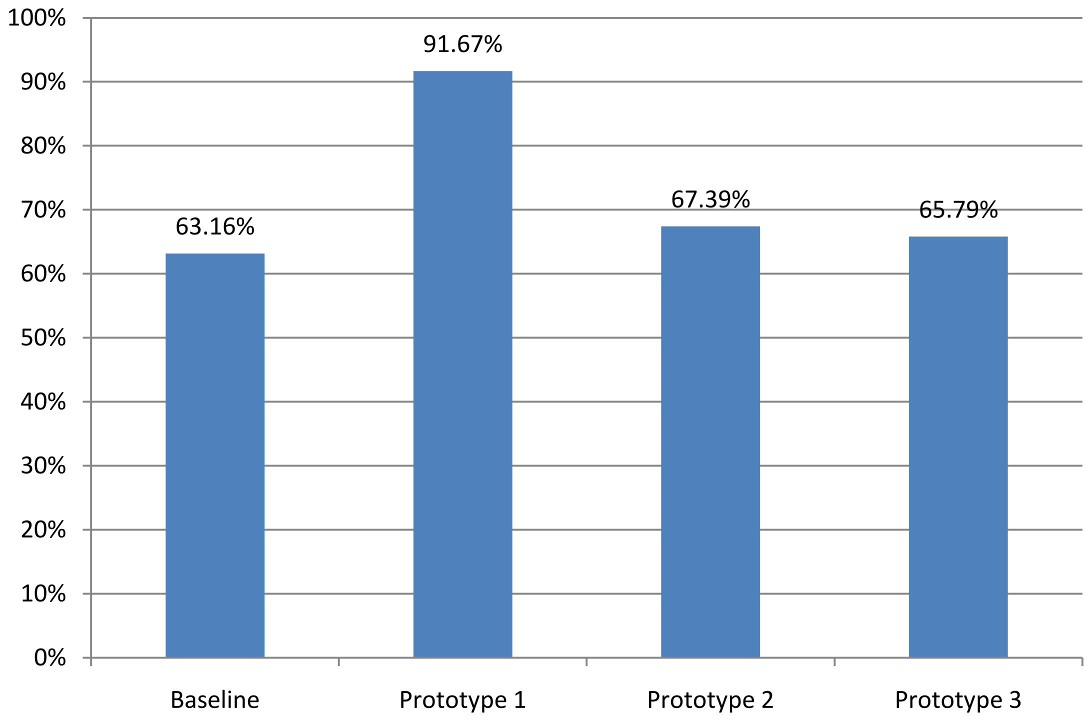


Figure 28. Proportion of builds fixed the same day they were broken.

Figure 29 illustrates the mean duration of broken build chains for chains that were fixed within a single day.

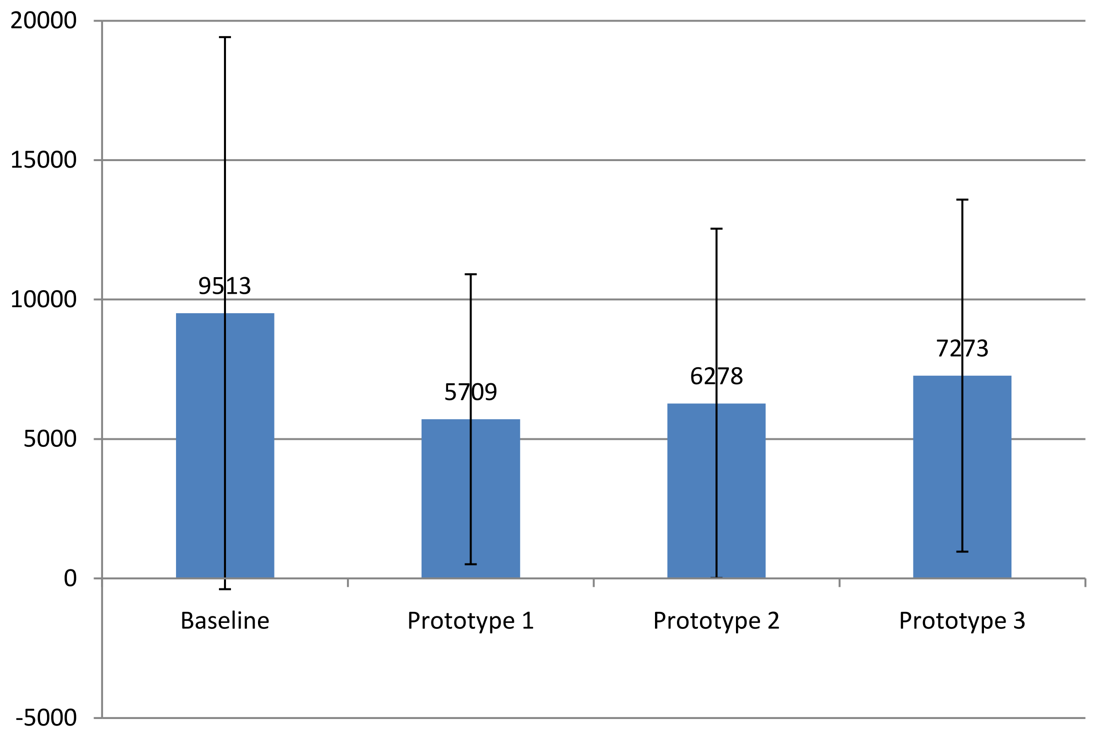


Figure 29. Mean fix duration (in seconds) for builds fixed the same day as they were broken.

When calculating the duration of broken build chains that were fixed over multiple days, non-working time was subtracted from these durations. Working time was defined as 7am to 6pm, Monday to Friday. This time period was selected based on the initial observational component, interviews with developers, and on an analysis of the standard times that builds occurred from the build server logs. While a very small number of builds did occur outside of these hours, we could not reliably ascertain when team members were working for each day. However, during the initial observational component (described in Chapter Four), we found that most team members worked between 7am and 6pm, and so we considered this to be a reasonable timeframe for this analysis. Figure 30 illustrates the mean duration of broken build chains that were fixed over multiple days.

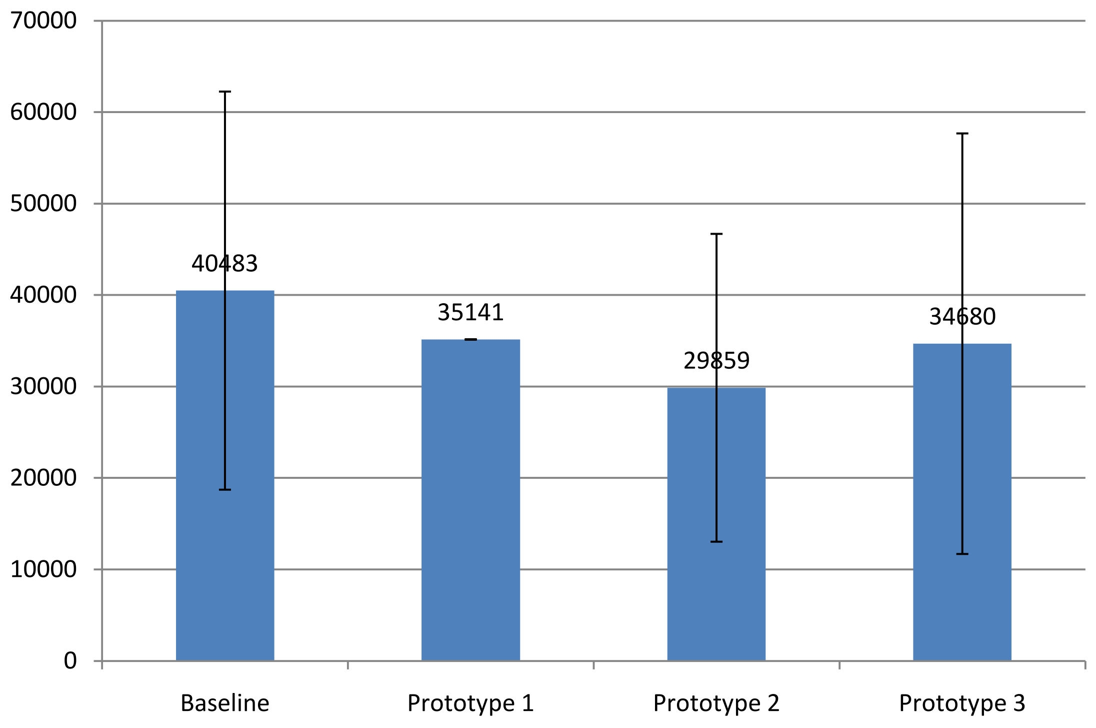


Figure 30. Mean fix duration for builds fixed on different days to when they were broken.

Time in Red vs. Time in Green

A final metric was the comparison of the total amount of time that the central light was red (indicating one or more integration plans were failing) with the total amount of time that it was green (indicating that all integration plans were passing).

This metric was calculated for all conditions, including the baseline condition. Even though the central light was not provided to the team during the baseline condition, the time the light would have displayed each colour if it had been present was calculated, and this information is included for comparison.

As in the previous metric, this was calculated based on work hours of 7am to 6pm, Monday through Friday. Figure 31 shows the proportion of work time the central light showed red for each study condition.

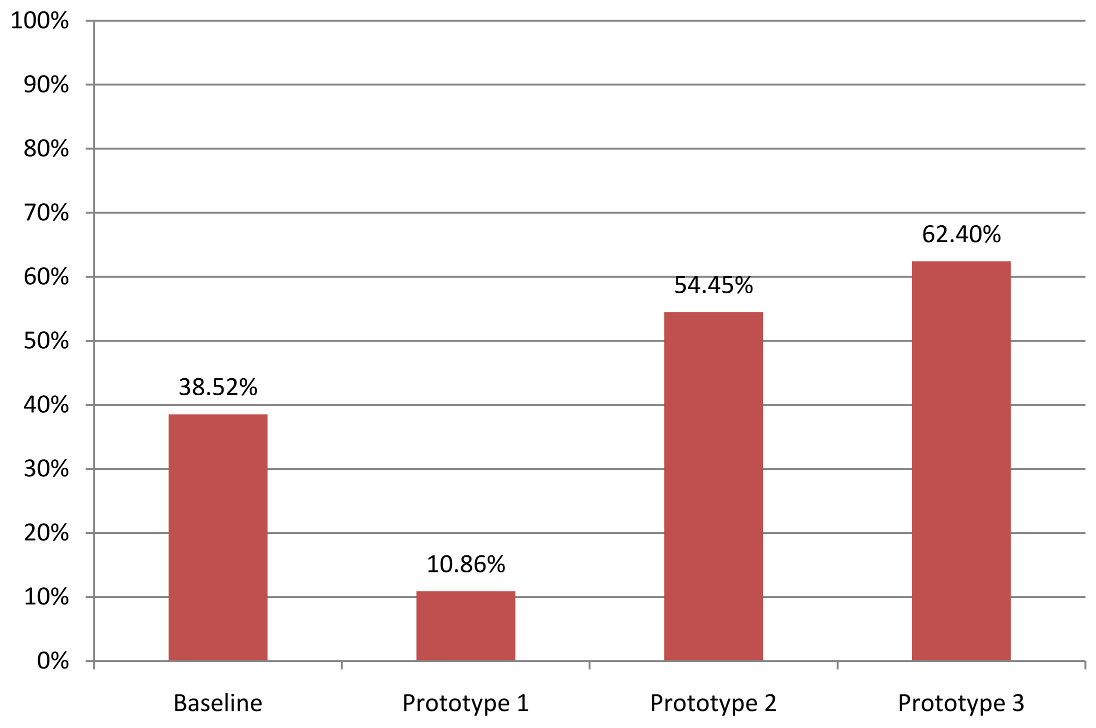


Figure 31. Percentage of work time the central light showed red, by condition.

Summary

This section described a range of quantitative metrics based on the build logs obtained from the build server. Chapter Eight will discuss these results with reference to the hypotheses outlined in Chapter Five (Table 14, p68).

7.2. Developer Engagement

One source of qualitative feedback was the level of engagement the developers exhibited with the project and with the devices. During our visit to install the devices we observed some interesting behaviour from team members. Even before the devices were installed some team members began referring to broken builds as "*sad bunnies*". Immediately after the system was installed a build failed, and one team member commented:

This is the first time I've noticed a broken build!

In the days following, team members engaged strongly with the project, making suggestions and giving feedback. In particular, one developer – who had previously been considered (by himself and others) as having the least interest in broken builds – was the most active in making comments and suggestions.

Team members requested a shared document (reproduced in Appendix F) be made available for them to make suggestions and have a conversation about the project. These comments ultimately led to the design and development of prototypes 2 and 3, described in Chapter Six. In addition, the organisation set up an internal wiki page for discussion of the project, although this simply contained an explanation of the project and provided a forum for humorous comments about the devices.

Approximately one week after the installation of the devices, the team requested an extra Nabaztag be set up as a surrogate for the developer who worked on the team's code from a remote location. We considered this an indication of the team's acceptance of the technology, and a validation that team members found the system useful. The developer seated next to the surrogate rabbit reported he would send instant messages to the remote worker whenever the surrogate rabbit indicated a build failure.

We believe that the level of engagement exhibited by the developers throughout the project was indicative of the level of enthusiasm they had for the technology, the importance they placed on broken builds, and their desire to resolve the issues they faced with the build process.

7.3. Interviews

In order to explore the team's thoughts on the ambient build monitoring systems more comprehensively, short interviews were conducted with team members. A list of the themes explored in these interviews is provided in Appendix G.

Some team members had been reassigned since the initial interviews, but seven of the eight remaining developers on the team were interviewed. The eighth developer chose not to participate in these interviews, although he did participate in the rest of the study.

Due to time constraints these interviews were performed shortly after the third prototype was installed, and therefore did not include questions relating to the positive reinforcement behaviour added in prototype 3. Interviews were recorded, had an average duration of approximately 15 minutes, and were analysed informally by replaying the interview and noting core themes, opinions, and suggestions. A formal analysis such as that conducted for the initial interviews (described in Chapter Four) was not performed for these interviews. As with the initial interviews, all interviews were conducted by the author.

Overall, every developer who was interviewed was very positive about the ambient devices, and especially about the Nabaztag devices. One team member commented that:

> *I think it solved pretty much all the problems with the old email system. I liked knowing quickly when I'd broken the build, I liked being able to point the finger when someone else broke the build.*

The following sections describe the common themes which emerged from our analysis of these interviews.

7.3.1. Nabaztag Devices

The team members interviewed had unanimously positive reactions to the Nabaztag devices and their use as build monitoring devices. Many of the developers noted that the movement of the ears was helpful for drawing attention, and since the ears typically only moved when a build had been broken, this interruption was considered both desirable and necessary.

The software components were generally regarded as accurate at assessing who was responsible for broken builds, although some developers suggested enhancements that would further restrict the set of candidate developers based on additional information from the build server. (Unfortunately, much of this information was unavailable through the build server's API.)

The physical nature of the devices had a surprisingly large effect on the developers. Many commented that they would frequently scan the room to see which rabbits appeared 'sad', and would use that information to evaluate the overall build status and potentially decide whether to commit changes or wait. The rabbit form factor was generally regarded as a positive feature of the devices; one team member commented:

> *This one has kind of some personality, a rabbit, that everyone thinks that... it kind of just brings some fun into the team environment I suppose, so it's been a good thing.*

The form factor was also seen to be less threatening than alternatives such as emails, the central light, and the other build monitoring systems. Rather than being purely functional and sterile, the devices were seen to be whimsical and to have some sort of personality. This had some noticeable effects; for example, one senior team member noted that:

> *When it's a device doing something like that then it's not a personal thing, "oh you've done it again", versus "your rabbit ears have gone sad again"... yeah, it's more indirect, and also... it gives you a good way to say "oh, your bunny's sad" rather than "you've done something stupid", so I think the bunnies have been very good from that point of view.*

7.3.2. Central Light

Perceptions and opinions of the central light were less consistent. Some developers argued that it was redundant since the build status monitoring screen presented essentially the same information, albeit with less visibility. The position of the central light was not ideal, with some developers facing away from the light. However these developers were close to the build status monitor, meaning that all developers on the team had either the build status monitor or the central light within their line of sight.

Many of the developers noted that they used the central light most when they were walking in and out of the room. In a comment typical of the sentiment of these developers, one team member said:

> *I pay attention to the builds now... it's nice to walk into the room and have something that potentially changes what you were going to work on. So if you come in in the morning, you might be thinking about what you're gonna do, but instead of that you actually have a little scan of the light, and if it's green you keep on going and if it's red you've got something else to think about. I guess the build monitor is there but [the central light is] just a bit more prominent and a bit nicer.*

7.3.3. Separate Channel

When the devices were first installed, some developers openly questioned the necessity of using separate devices rather than notifications displayed on the developers' computer screens. During the interviews, however, several team members specifically commented that having the build status information presented within the separate channel created by a distinct physical device meant the device appropriated a unique meaning and context. One developer commented:

> *You get email from all sorts of things throughout the day, whereas this is a unique thing... you know straight away [what it means].*

The same developer also said of the Nabaztag rabbits:

> *I think this is a really ideal use for that technology, it's really good.*

7.3.4. Expected Effects

Each team member was asked to predict the sorts of effects that might be observed from the build status metrics, as well as describe general changes in the team or other effects of the devices. A number of effects were cited by team members.

The most frequently mentioned effect was that the devices increased the level of awareness of the build status – in the team in general, as well as in individual developers who had traditionally not paid attention to breaking builds even when they were directly responsible. One developer who fell into this category (and who had noted "*this is the first time I've noticed a broken build!*" when the devices were installed) commented:

> *It's definitely been a change for me in terms of the way I've looked at [broken builds].*

A second effect was the reduced need for team members to 'nag' each other about broken builds. Previously the more vigilant developers would be unsure if the team was aware of broken builds, and would therefore take responsibility for notifying and reminding developers when they had broken a build. Over the course of the study the individual devices largely came to assume this responsibility. Similarly, since the devices stayed red until the problem was fixed, broken builds became perceived as more of a priority. The fact that the devices were public in nature, visible to other people inside and outside the team, meant that the developers felt social pressure to fix builds. Developers would often be gently teased if their rabbit stayed red for longer than expected, which they appreciated.

Six of the seven developers interviewed said they believed the time to fix broken builds was reduced when the build monitoring systems were introduced. In addition, three of the developers said they believed there had been a higher frequency of builds due to the technology. Unfortunately, due to confounding factors involving the project they were working on at the time, the team expected the number of builds to increase anyway, making an interpretation of this analysis difficult. Even with this caveat some developers said they felt that the devices had been directly responsible for at least some of the increase in builds per day.

7.3.5. Comparison to Existing Systems

The two primary build notification mechanisms the team used before the devices were deployed were the build status monitor screen and email notifications. The build status monitor was only visible to a few team members due to its location and was designed to simply indicate whether each build plan was passing or failing. Email notifications were sent to team members after builds, but the high volume of the emails and the effort it took to parse their contents meant they were ignored by many team members. Several team members commented that the ambient build awareness devices solved all of the problems they had previously faced with the email notifications.

7.3.6. Suggestions for Improvement

Suggestions were solicited for both improving the build monitoring system and for other applications for the technologies. The most frequent suggestion was to enable the Nabaztag devices to indicate when a build was in progress, either to minimise the likelihood of code being checked in while a build was in progress or just for general build awareness:

> *When you kick off a build... you could slowly raise the ears, and it passes and the ears are at the top, or it fails and the ears drop back down. That would just show that your code is in the process of being evaluated by [the build server].*

Other suggestions included adding notifications when builds following a code commit passed (which was the behaviour added in prototype 3), displaying the overall build status on the rabbits somehow (essentially incorporating the functionality of the central light into the rabbits), and integrating the technology with other systems such as the issue tracking system.

7.4. Summary

This chapter has described the evaluation study conducted with the case study team to determine the usefulness and effectiveness of the build monitoring systems described in earlier chapters. A number of quantitative metrics were calculated based on the hypotheses described in Chapter Five. In addition a large amount of qualitative feedback was obtained through team emails, a shared suggestion document, and interviews and conversations with team members.

Chapter Eight discusses these findings with reference to the hypotheses made at the beginning of the study, and identifies the overarching themes that emerged throughout the study.

Chapter Eight
Discussion

Chapter Seven provided the results from the evaluation study conducted with the build monitoring system. This chapter provides commentaries and explanations about the effects as observed in the build metrics and as articulated by team members in interviews, emails, and other conversations. In section 8.1, the primary research hypotheses are evaluated based on the build metrics. The remaining sections describe the implications of these findings, as well as the other themes which emerged during the course of the study as a whole.

It is important to note that this project involved a single case study team. As such, the extent to which these results can be generalised is difficult to ascertain. However, this chapter discusses the results that were observed and, where appropriate, makes comparisons and distinctions in areas where other teams are likely to differ in their approaches.

8.1. Support for Research Hypotheses

The metrics calculated in Chapter Seven provide varying degrees of support for some hypotheses. Table 27 lists the hypotheses used for this project and summarises the degree of support based on the evidence from the evaluation study. In total, three hypotheses were supported by the metrics, three were not supported, and three were ambiguous.

Table 27. Evidentiary support for research hypotheses based on evaluation study results.

	Hypothesis	Evidence from Evaluation
H1	The proportion of builds which fail will decrease.	**Ambiguous.** The proportion of failing builds decreased when prototype 1 was installed, but then increased beyond baseline levels during prototypes 2 and 3.
H2	The overall number of builds per day will increase.	**Supported.** The mean number of builds per day increased each condition.
H3	The number of files checked in per changeset will decrease.	**Not supported.** The mean number of files per changeset increased each condition.
H4	The number of commits to a	**Not supported.**

	Hypothesis	Evidence from Evaluation
	broken build will decrease.	The mean number of commits per changeset increased over the study period relative to the baseline.
H5	The proportion of builds triggered by a manual request from a developer will increase.	**Ambiguous.** The proportion of manually triggered builds increased during prototype 1 relative to baseline, returned to baseline in prototype 2, and increased again in prototype 3.
H6	The proportion of broken builds due to unit test failures will decrease.	**Supported.** The proportion of builds breaking due to unit test failures decreased over the study period relative to the baseline.
H7	The proportion of failing builds fixed by the breaker will increase.	**Ambiguous.** The proportion of builds fixed by breaker increased when prototype 1 was installed, but then decreased to approximately baseline levels.
H8	The average duration of broken builds will decrease.	**Supported.** The proportion of builds fixed the day they were broken increased substantially when prototype 1 was installed, and then decreased somewhat but remained slightly above baseline levels for the duration of the study. Of the builds fixed the day they were broken, the mean fix duration decreased substantially and remained lower than the baseline, although did increase as the study progressed. Of the builds fixed on separate days, the mean fix duration was consistently lower than the baseline level.
H9	The total proportion of time in which any plan is failing will decrease.	**Not supported.** While the proportion of time the light showed red decreased substantially in prototype 1 relative to the baseline condition, the proportion then increased dramatically in prototypes 2 and 3, well beyond the baseline levels.

In general, the introduction of the build monitoring system did have an effect on the team's work and on their workflow for building software. Team members performed more builds, fixed broken builds more quickly, and may have been more productive as a consequence. Additionally, every team member interviewed was enthusiastic about the technology and expressed positive opinions. Based on this, we can reasonably state that the project was a success.

However, a number of fundamental questions remain: exactly what effects did the system have on the team, and why? Which components of the system worked well and which did not? Were the effects consistent or were they haphazard? Did they relate to individual developers' behaviour, or the team's behaviour as a whole? The following sections discuss these concerns with reference to the metrics from Chapter Seven, the research hypotheses, interviews with team members, and informal conversations with developers throughout the study.

8.2. Novelty

In many case studies involving the application of interventions, a Hawthorne effect can often be observed. This effect is simply a response to the novelty of the introduction of the intervention, and will typically disappear within a relatively short period (Clark & Sugrue, 1996; Mayo, 1949).

In the case of this study, several of the metrics listed above demonstrated a characteristic Hawthorne pattern of an initial effect followed by a return to baseline levels. Four metrics in particular – H1 (proportion of broken builds), H7 (builds fixed by the breaker), H8 (duration of broken builds), and H9 (time the central light showed red vs. green) – showed at least some degree of a noticeable behavioural shift in the early stages of the study which subsequently disappeared. These four metrics share a common theme in that they are all related to broken builds – namely, how often they occurred, how long they lasted, and who fixed them.

Over time, the proportion of broken builds did not decrease, but the durations of broken build chains decreased, and the total number of builds increased. These findings are consistent with each other: as builds occurred and the proportion of failing builds increased, these were being fixed more quickly, and therefore more builds were occurring. In addition, the proportion of broken build chains fixed by a breaker did not show a consistent increase.

Some of this Hawthorne effect is likely due to the project itself. The process of developing commercial software is difficult and complex. Team members often noted that it is simply not reasonable to expect that every build will pass, or that every broken build will be fixed immediately. In particular, the nature of the project that the team was working on in the later parts of this study had an effect on the team's expectations around the number and quality of builds. However, these metrics do also suggest that there may have been at least some initial impact on the team's treatment of broken builds which was initially high and then decreased.

Two possible reasons for this type of effect are that the effect would have been consistent across all prototypes but quickly diminished as team members adjusted to the devices, or that the changes that were introduced as the study progressed – prototypes 2 and 3 – were less successful at impacting these metrics.

Both of these possible explanations appear plausible, and both have some degree of empirical support. The overall pattern of metrics, and the number of metrics exhibiting some degree of a Hawthorne effect, suggests that the initial effects of the system wore off as the devices became a part of the team's work environment. However, because of the additional alerts added in the second and third prototypes (namely, build failure notifications being given to all developers who committed to a failing plan, even if they committed working code to an already-failing plan), it became more difficult for each developer to establish exactly who was responsible for a given build failure and to then take individual responsibility for a given broken build. In some cases, breakages persisted for hours or even days, and as more developers committed to these plans over time the pool of potential breakers increased. This, in turn, diffused the responsibility over multiple team members, and reduced the usefulness of individually notifying potential breakers.

Despite the transient nature of some of these effects, it is also important to note that a number of effects that are indirectly associated with behaviours relating to a reduction in broken builds – such as H5 (builds manually triggered by a developer) and H6 (broken builds due to unit test failures) – did show a positive effect, and this effect was consistent for the entire study. Even though these effects were comparatively small and difficult to isolate, it would seem that some positive habits became ingrained.

8.3. Awareness and Responsibility

A core principle upon which these types of devices are predicated is that an increase in team members' awareness into the build process will, either directly or indirectly, increase the entire team's overall sense of responsibility for the build process. This is thought not only to discourage broken builds, but also to encourage team members to quickly fix those builds which do break. The interplay between awareness and responsibility implies a measurable causal relationship between the introduction of a heightened level of awareness and a corresponding reduction in broken builds.

The metrics calculated for this study were not specifically designed to distinguish between the effects of awareness and responsibility. However, on consideration, it became apparent that the team's general awareness of build status would be represented through hypotheses H1 (that the proportion of broken builds would decrease) and – particularly – H8 (that the duration of broken builds would decrease as they were getting fixed more quickly). H1 had limited support from the data collected, but H8 did show a consistent decrease in the duration of broken build chains.

Similarly, the construct of individual responsibility can be inferred based on hypothesis H7 (the proportion of broken builds fixed by a breaker will increase). The results showed a Hawthorne effect for H7 which disappeared by the introduction of the second prototype.

Overall, this pattern of results suggests that awareness of build failures does not necessarily lead to an increased sense of responsibility, even when one or a very few developers are individually alerted. Instead, these results indicate that a distinction should be drawn between awareness and responsibility.

Awareness, in this context, is simply being made aware of the situation and of broken builds. This does seem to lead to an overall effect of fixing the broken builds more quickly as a team. Additionally, the simple fact that team members can easily scan the room for 'sad bunnies' means that they could become aware of broken build plans fairly easily, and with the added contextual information of knowing who was working on which components, could more easily work together to resolve these issues. These effects persisted across the study, were not affected by the specific prototype behaviours, and were also reflected in the comments made by team members.

Individual responsibility for fixing broken builds is more complex. The results suggest that the introduction of the devices had mixed effects on team members' sense of responsibility. While developers did initially fix their own broken builds more often, this effect was eliminated by the time the second prototype was installed. As noted above, the behavioural characteristics of the second and third prototypes did mean that more developers were potentially given notifications for any given broken build.

Despite these complexities, and the limited quantitative support for the idea that individual responsibility was increased by these devices, a number of team members reported that they did feel an increased sense of responsibility when they caused a broken build, largely because they were cognisant of their red rabbit advertising this fact to the team at large. However, they also noted that they were not always able to fix these problems immediately due to the complexity of the task.

Accordingly, it is important to distinguish between two types of individual responsibility. An external (or exogenous) sense of responsibility is due largely to peer pressure applied through mechanisms such as friendly teasing or commenting about a 'sad bunny'. This form of responsibility wore off as the novelty wore off. An internal (or endogenous) sense of responsibility, as indicated by the feeling of responsibility or embarrassment, appeared consistent throughout the study, based on statements from team members, but did not seem to have a noticeable impact on the quantitative metrics.

Further study would be needed to isolate these types of responsibility and determine which interventions might result in lasting increases in either or both.

In summary, the results from this study indicate that awareness and responsibility are not as tightly linked as we initially expected. The team's overall awareness of broken builds was certainly increased, based on the developers' comments and on the consistently lower duration of broken build chains. However, the degree of individual responsibility for fixing these broken build chains is substantially more difficult to isolate. The results suggested that while developers did feel a consistently stronger sense of internal responsibility from having these devices, this did not translate into any long-term measurable effects. External responsibility, due to team mates' commenting or teasing, was demonstrated but did not persist in the long term. Accordingly, we can conclude that while individual responsibility for broken build fixes does require awareness of the build status, this awareness does not necessarily lead to responsibility.

8.4. Changes in Individuals' Behaviour

The most striking example of a behavioural change was the developer who, having previously been known for ignoring failing builds, became enthusiastic about the project and later commented that he was now actually paying attention to builds. Whether this change resulted in a measurable difference in the developer's work performance was not something that could be ascertained from our data. However, because software development is a team exercise, and given the other team members' stance that broken builds disrupt the development process and frustrate other team members, it is reasonable to expect that their opinions of this developer's work would have improved given the change in his behaviour.

More broadly, a number of other hypotheses were related to changes in behaviour surrounding the build process. The proportion of failing builds that were due to unit tests (H6) decreased from a baseline of 85% to between 69% and 75% for the subsequent prototypes. This level of difference suggests that developers were either paying closer attention to unit tests as they wrote code, or even running the unit test suite before checking in to verify that the tests would pass.

This type of behaviour is beginning to be encouraged and even required in other teams. For example, Microsoft Visual Studio Team System 2010 (Microsoft Corporation, 2010) introduces a 'gated check-in' feature, requiring changesets to meet a minimum set of quality criteria (usually including all unit tests passing) before allowing the source code to be committed. A further extension of such behaviour would be to require the code to be built locally to validate the correctness of the code before sending it to the repository and build server.

8.4.1. Productivity

While this study was intended to evaluate the changes in build-related behaviours as a result of the introduction of the ambient build monitoring devices, we also recognise that such interventions can have broader implications. In the case of this case study, we observed a higher number of builds, but also an increasing number of files per changeset. This pattern of results was somewhat unexpected, as our initial expectation was that an increasing frequency of builds would lead to a corresponding decrease in the size of each build's changeset.

There are several possible reasons for this incongruity. The most plausible is related to the way in which the changeset size metric was calculated: specifically, it only considered the number of source code files added or changed, and did not consider the size of each code file change. Accordingly, it is probable that while more files were being modified for each build, the totality of the size of the modifications (perhaps measured in terms of lines of code changed) remained roughly constant.

A second possibility which is also consistent with these findings is that more code was actually being written and committed. Such a difference could be considered an increase in productivity: developers wrote more code and did so more quickly. The reason for such a shift in productivity could be due to a number of factors, including the nature of the project that the developers were working on at different points during the study. It is also a theoretical possibility that such a difference could be due to the interventions that were performed.

The probability that an increase in productivity is directly and exclusively due to the introduction of ambient build monitoring devices is low. This theory has minimal prior plausibility and little empirical justification beyond the correlation between these two metrics. However, it does serve to emphasise the point that the effects of the build process can be widespread within a team, and that little is known about the effects of the build process on other aspects of team-based software development. Other metrics, such as the rate at which work items are closed, may be more accurate measures of this construct.

8.5. Device Characteristics

Chapter Five outlined the process of selecting appropriate technologies and behaviours for the ambient build monitoring devices. An important aspect of this project was the evaluation of these decisions.

8.5.1. Individual and Central Devices

Other teams who report using ambient build monitoring devices (see Table 5, p30) typically use a single, shared device for the entire team. In this project, we evaluated such a shared device as well as individual devices for each team member. Having the two device types proved to be invaluable, with team members holding very different opinions and impressions of each device type.

The individual Nabaztag rabbits were extremely popular, even after several months. Every team member commented that they had positive opinions of the rabbits, considered them fun, and that they suited the personalities of the team. Of course, the extent to which these opinions may be generalised to other organisations and teams will depend in large part on the personalities of those team members.

By the time the team members were interviewed towards the end of the study, there were several indications that the rabbits had become ingrained in the team's daily lives. First, many team members were resistant to the idea of taking the rabbits away. While this may have been partly due to the fact that participation in this study made them unique among their colleagues, team members also noted that they found the rabbits so useful that they would like to keep them indefinitely. Second, a senior team member noted that the rabbits gave him a level of indirection when commenting that a particular team member had broken a build – instead of using a personal and direct attribution, such as "you broke the build", he could instead say "your rabbit's sad". Finally, the term 'sad bunny' appeared to have entered the team's vernacular and had developed into a euphemistic synonym for 'broken build'.

The central light provoked more varied opinions. Based on an analysis of the interviews and metrics, it appears that the concept of providing a central build monitoring device is generally sound, but that the usefulness will depend on the team's work practices. In this case, multiple build plans were aggregated on a single light, and this level of aggregation was too high to be useful. When evaluating hypothesis H9 (the time the central light showed red vs. green) it became apparent that the proportion of time that the central light showed red was increasing dramatically over the study. Further investigation revealed that while the duration of broken build chains was decreasing (H8), the increase in the number of builds (H2) and proportion of broken builds (H1), and the fact that there were multiple build plans being simultaneously triggered, meant that broken build chains were frequently overlapping. Even though any given broken build chain was shorter following the introduction of the interventions, there were more of them. Because of the high granularity of the information displayed on the light there was little opportunity for team members to determine the exact status of the overall build, and they had to turn to other sources – their individual rabbits, the build status monitor, their email, and so forth – to determine exactly what had broken.

This highlights the importance of taking a team's workflow into account when designing these types of interventions. If a team has multiple codebases or projects being built simultaneously, a single light is less likely to be useful. Individual devices, specific to a given developer, are more helpful, particularly when combined with the shared knowledge of which developers are working on different parts of the system. In the case of smaller teams, or teams only working on a single codebase, a shared aggregate device may be more useful than it was here.

Despite this caveat, some team members did find the central light valuable. A number said they used it to get a general impression of the team's status as they were walking in and out of the open plan work area. One noted that he would make a point of investigating if he saw the light turn red – even if it was not his own problem, he wanted to be aware of the state of the build plans.

8.5.2. Separation of Channel

A core tenet of ambient devices, and ambient awareness more broadly, is that information should be presented through a distinct channel to avoid it becoming confounded with other sources of information or attention. In this case, each of the devices presented a number of potential channels of information that were independent of the developers' main workflow.

At the beginning of the study, some developers argued that a build monitoring system could simply consist of software components that ran on each developer's workstation, perhaps integrated into their IDE in some way. When the ambient devices were being installed in the team's workspace some organisational members openly expressed scepticism that such devices would add any value. However, the comments from the developers as they proceeded through the study were very different.

The effects of this separation of channel could be seen in a number of ways. The devices – and particularly the Nabaztag rabbits – became synonymous with the build process and began to represent the build process to the team. This was evident not only through the team's integration of the rabbits into their workflow but also in their language – as previously noted, on several occasions we observed developers referring to broken builds as 'sad bunnies'. The degree of this association was emphasised during the interviews with developers, a number of whom commented that the devices removed the ambiguity associated with build awareness. A developer may have previously missed a notification email or message, but they would immediately see – and understand the meaning of – a change in their rabbit's appearance.

Despite these effects, the specialised device form factor was seen as luxurious and indulgent by some team members. One commented that he found the devices somewhat frustrating as they were single-purpose, which went against his nature as a developer of general-purpose software components. He also commented that the same overall effect could likely be obtained from having an LED light on a stick, rather than a rabbit. However, even with these comments, this developer recognised the value of having a specialised delivery mechanism for build notifications given the importance of build notifications in the team's work.

8.5.3. Redundant Encoding

Another characteristic of the rabbits was the multiple dimensions along which information could be presented. The rabbits contained audio capabilities, multiple LEDs, and moveable ears. None of the prototypes used audio since this would likely be considered socially inappropriate in an open-plan team environment. However, in all prototypes, the information presented on the rabbits – build success and failure notifications – was presented both through the coloured LEDs and through movement of the rabbits' ears.

This redundant encoding was useful for several reasons. First, some team members found they noticed different aspects of the devices at different times. For some, the coloured lights were more noticeable than the ear positions, while for others, they noticed the movement of the ears more readily. Removing one or the other may have made the devices substantially less useful and more difficult to comprehend. Second, in the last prototype, a build failure reminder was given by having the rabbits' ears rotate every thirty minutes when a build plan remained in a failing state. In some cases, developers could not fix a build plan within that timeframe and began to find the movement irritating. In these cases the developers simply detached the ears from their rabbits temporarily, and could still receive notifications through the LED colours. Accordingly, this property of the devices had both a conceptual and a practical function.

8.5.4. Simplicity

The information obtained from the build server was reasonably complex, as it corresponded to multiple build plans, for multiple developers, and was based on an aggregation of several data sources. A principle upon which the devices' behaviour was designed was simplicity: as with other ambient devices, we argued that this type of system would be most useful if the devices presented information in a straightforward and easily perceptible manner.

This decision was validated throughout the study with developers noting that they could easily understand when a build was broken by looking at one or more of the devices, and could scan the room to quickly get an impression of which developers were likely to be busy working on fixes. However, during the interviews near the end of the study, it also became apparent that a small number of developers had formed mental models of the system which did not correspond to the system’s actual behaviour: one developer had assumed that the process by which build logs were transformed into the device output was substantially more complicated than it actually was, while another carefully explained that he wanted the devices to exhibit a particular behaviour, although that behaviour had been added several weeks previously.

These misunderstandings both validated the importance of the principle of simplicity, and also suggested that even the behaviour we had implemented (and which we had considered unambiguous) had the potential for further improvement. In particular, with this type of device – where a visual channel corresponds to a specific piece of dynamic information – the more manipulation that is done on that information, the more complex the cognitive processes required to decode it.

Several team members suggested adding additional behaviour to the devices, and particularly to the rabbits. A common suggestion was to use their multiple LEDs to indicate different types of information – for example, if a given rabbit’s top left LED displayed a particular colour it would indicate that the developer had committed a build to an already failing plan, but the bottom right LED would indicate that the failure was likely due to infrastructure failures. A similar suggestion was to use the ears as a basis for semaphore-like communication, with different ear positions representing different information. We resisted these suggestions on the basis that the simplicity of the device would be compromised and the amount of cognitive effort required to decipher the messages would be too great. This decision appears to have been validated by the fact that even the relatively simple behaviour that was implemented was misconstrued by some team members.

8.5.5. Individual Device Prototypes

Three prototype behaviours were trialled. Each prototype only changed the behaviour of the rabbits and did not affect the central light. The first prototype alerted the committers to the first build of a failing build chain, and when a build chain was fixed, would turn the first committers’ rabbits green for 15 seconds and then return to a neutral state. The second prototype was similar but turned rabbits red for all developers who had committed to a failing build chain, even if that build was not the first in the failing build chain. The third prototype behaviour turned rabbits green when developers committed code that was successfully built, and also rotated the ears every 30 minutes when a build plan remained in a failing state. (For a more complete description of the prototype behaviours, please refer to section 6.3.2 in Chapter Six, p76.)

The change of behaviour in the second prototype was requested by the team for two reasons. First, it functioned as a warning when a developer committed to a failing build: this was not considered a good idea, and the team wanted to ensure that developers who did this were made aware of the fact that their code would not be fully validated and may have confounded the fix to the original failing build. Second, it was intended to expand the pool of developers who were potentially responsible for fixing the underlying problems. As other developers continued working on work items and committing these to a failing build, these developers would be implicitly recruited into the effort to fix the problem.

Developers had mixed reactions to this change. While many thought it was an improvement and brought the system in line with the team's standard policies, others saw the behaviour as holding them responsible for something they did not do – i.e. breaking the initial build. This aversion to being held responsible for a broken build reinforces the notion that developers did pay attention – at least to some extent – to the negative connotations associated with a broken build and a 'sad bunny'.

The third prototype's behaviour was intended to be explored as part of the study, but was also suggested by some of the team members. The prototype was primarily intended to explore the extent to which these devices were seen as providing reinforcement and punishment. If the behavioural psychological Law of Effect held for this situation, it would be expected that the first two prototypes – which primarily notified developers who broke a build – would be focused on punishment. According to the principles of behavioural psychology, a stronger effect could be expected when combining this punishment with positive reinforcement.

Unfortunately, due to time constraints, we could not discuss the third prototype during the follow-up interviews with team members. However, some team members did respond to email requests for their thoughts on this behaviour. In general the developers found that notifications for successful builds were useful as well; however, they saw these simply as notifications and not as reinforcers. There was no comment that suggested that the introduction of reinforcement encouraged the developers to take appropriate steps to mitigate build failures. Additionally, some of the metrics which might show such an effect – such as H6 (proportion of builds due to unit test failures) – did not show an effect of reinforcement.

A number of reasons could be given for this. First, the metrics were not intended to specifically examine the effect of reinforcement and punishment. In order to fully explore these issues a more targeted and rigorous experimental design would be required. A second reason is more closely related to the principles of behavioural psychology. Because of the transient infrastructure failures which resulted in some of the broken builds, builds would sometimes fail randomly and through no fault of the developers. Under the behavioural paradigm this could be considered as an operation of a random reinforcement and punishment schedule: while many notifications were directly tied to the actual build results, these cases of random failures would dilute the effect of these reinforcers and punishers. A practical mechanism by which this could occur would be if other developers on the team began to disregard 'sad bunnies' based on their expectation that it could be a random occurrence. A third reason is that punishers only have an effect if they are truly aversive, and reinforcers only have an effect if they are truly rewarding. While social pressure and social reinforcement may be sufficient, as the novelty of the devices wore off the value of these may also have diminished, and by the third prototype, may have all but disappeared.

In summary, the extent to which these devices operated within a behavioural paradigm requires further study. It is possible that the devices function purely as notification mechanisms, with limited power to act as delivery mechanisms for reinforcement or punishment. Alternatively, it is possible that the devices do work within a behavioural paradigm, but for various reasons these effects did not last.

8.6. Satisfaction of Requirements

A final measurement of the success of these interventions is to evaluate them with respect to the design requirements outlined in Chapter Five (Table 12, p62). Table 28 evaluates and briefly discusses each of the requirements.

Table 28. Evaluation of design requirements.

	Requirement	Comments
R1	Developers should be informed about broken builds automatically.	**Satisfied**. Notifications were sent to the ambient devices automatically and rapidly after a build occurred.
R2	Broken builds should be made to appear important and a priority to fix.	**Satisfied**. The duration of broken build chains was decreased. Sometimes, a broken build could not be satisfied for various reasons (e.g. a developer may be waiting on someone else; or starting another build may cause problems for the team). However, this meant that when broken builds did occur, they were always subject to

	Requirement	Comments
		consideration and a conscious decision was made to fix or not to fix depending on the context.
R3	Notifications of broken builds should arrive as quickly as practicable, and with minimal or no effort required on the part of the developers.	**Satisfied**. Team members reported the devices were rapidly updated and that they noticed breakages substantially more quickly.
R4	The build notifications should be easy to see and require little cognitive effort to perceive and understand.	**Satisfied**. Each developer had a personal device on their desk. Also, all developers could either see the build status monitor or the central light from their desk. Even though the central light was not sufficiently detailed to be useful, a change from green to red would prompt them to seek more information. Team members reported that the meaning of the devices was obvious from the selection of colours and the types of movement.
R5	Developers who may have broken a build should be automatically and personally notified.	**Satisfied**. Assuming that the builds were mostly passing, a failing build on a given device would be rapidly noticed by that developer. If there was a large pool of committers, some work was still required to narrow down exactly who should fix it.
R6	The entire team should also be notified of a broken build, and a persistent notification should remain until the problem is resolved.	**Partially satisfied**. There was an increased consciousness of broken builds. However, this did not directly translate into people fixing their own breakages more frequently, as discussed above.
R7	It should be obvious to the entire team who may have broken the build. This will apply implicit social pressure to the breaker(s).	**Satisfied**. The developer(s) who were most likely responsible for breaking the build had an immediate and obvious signal in the form of their rabbit. Nothing in the system would stop developers from

	Requirement	Comments
		committing to a failing build chain, but if they did, they would receive a notification and would then have to consider their change more carefully.
R8	Build information should be presented through a separate channel from other information to emphasise its importance.	**Satisfied**. Having a separate channel was a substantial advantage, and was cited by team members (even those initially sceptical of the need for a specialised device).
R9	Team members should not be overloaded with multiple build notifications; instead, a single, dynamically updated summary should be provided and should display only the current status.	**Satisfied**. The devices only showed the most recent and up-to-date information. When necessary, historical information was taken into account when determining the information to display (e.g. in the second and third prototypes), but this was only a small factor – and regardless, only the most recent information was relevant and was displayed.

8.7. Implications and Advice to Practitioners

Because of the extent to which these devices are beginning to be used within industry, it is important to consider the implications of these findings for other teams. This section provides general advice for practitioners who may wish to implement ambient build monitoring systems in other organisations.

First, it is of critical importance that any devices intended to support ambient awareness fit within the team's practices and workflow. Different teams will emphasise different aspects of their development methodology, of the information they generate and obtain, and of their practices and processes. In this case study team a great deal of emphasis was placed on the build process, but this was tempered by a level of practicality and an acknowledgment that broken builds could not always be avoided.

Second, the nature of the devices needs a great deal of consideration. While many types of displays and devices can present what is fundamentally the same information, the subtle differences between each type of device can have profound effects on the perceptions of that device and of the team's willingness to integrate it into their work. In this case, the form factor of the Nabaztag rabbits was very popular within the team; devices with similar overall output but a different form factor, such as a simple LED, would probably not have engendered such a positive response, and accordingly would likely not have been accepted by team members as readily.

Third, consideration must be given to whether a central device or a set of individual devices are most appropriate. This decision will be based on factors including the nature of the projects the team are working on, the size of the team, and the available resources. In this case study individual devices were more appropriate and useful due to the large number of developers and the multiple build plans. Aggregating these multiple data points onto the central device necessarily involved a loss of information, and accordingly diluted the value of the system too much. A smaller team, working on a single codebase, may not have this same problem.

Fourth, strong consideration must be given to the simplicity and glanceability of the information presented on any such device. An ambient device will only be successful if the information it presents is easily perceivable. When dealing with general-purpose devices such as Nabaztag rabbits, there is a strong inclination to integrate multiple channels of information and to use the full set of device features to provide as much information as possible. However, such a practice will make the device difficult to comprehend. In this case study, even the relatively simple behaviour of the devices was occasionally misinterpreted.

Finally, objective criteria should be established for measuring the success and impact of such a system. The hypotheses listed in Table 14 (p68) provide a useful starting point. Ultimately, the designer of such a system should be prepared to make alterations to its behaviour depending on the results of such an evaluation. Some effects will likely be short-term in nature due to the Hawthorne effect. A focus on long-term quality and performance improvements is likely to require a long-term trial.

8.8. Summary

This chapter has outlined the overall findings from this study, and has evaluated and discussed the research hypotheses constructed in previous chapters. Overall, the introduction of the ambient build monitoring devices did affect the team's behaviour in a multitude of ways.

A number of effects were observed through the build log metrics and from discussions with team members. While some of our hypotheses were supported – such as an increase in the overall number of builds, and a decrease in the duration of broken builds – others were more ambiguous or presented incongruous results. Additionally, some effects demonstrated an initial Hawthorne effect which diminished over time.

The notion that an increase in the team's awareness of broken builds will result in a heightened sense of individual responsibility for correcting and avoiding these problems received mixed support. However, the design characteristics of the devices – separation of channel, redundancy, and simplicity – were validated as appropriate for this type of device, and the overall design goals and requirements were met.

Chapter Nine
Conclusions and Future Work

This chapter summarises the research presented in this thesis. Section 9.1 reviews the project in its entirety. Section 9.2 concludes this thesis with suggestions for areas of future research.

9.1. Conclusions

This research has explored the use of ambient devices as build monitoring devices for software development teams. Using a linear action research methodology, the focus of this research has been on eliciting the key requirements for such a system, then implementing, refining, and evaluating that system.

A review of related research found that software development teams generate and use large volumes of information as part of their everyday work. The build process is the most common point at which source code from multiple developers is integrated, compiled, tested, and evaluated. In continuous integration methodologies this process may happen many times each day, and following each build the team's developers need to be informed of its results.

To stay aware of this type of dynamic information, a number of principles and technologies can be employed. One such principle is the distinction between purposeful and incidental awareness. The former requires that users be consciously and actively seeking out information in order to maintain their awareness of a system or phenomenon. The latter, however, describes situations where a user passively becomes aware of information while conducting other tasks. Ambient devices build on the principle of incidental awareness and enable digital information to be unobtrusively presented within a physical environment, so as to avoid requiring constant perceptual and cognitive effort to acquire the meaning of the information.

Software builds are an example of a dynamic and complex process to which ambient devices can be applied. However, despite several industry teams reporting the introduction of ambient build monitoring systems into their work environment, no formal research had been conducted to evaluate the effects or success of such systems.

Chapter Three described the selection of an appropriate methodology within which to evaluate the use of ambient build awareness technologies. While a number of different methodologies are used to explore the effect of interventions in software development teams, the linear action research paradigm provided the most suitable framework in which this study could operate. This framework took the limited study time into account while still promoting a systematic set of procedures to gather and assess the requirements, plan and implement concrete actions, evaluate these actions, and then specify the learning which had been achieved and the lessons drawn.

Chapter Four discussed the initial exploration in detail. Following an observational component, a series of interviews was conducted with every member of the case study team. From these sources of information it was possible to find a number of themes relating to the team's daily work, their priorities and concerns, and the ways in which they communicated. In particular, the build process was an area where a number of technical and social barriers existed. When broken builds occurred, team members would often be unclear on who was responsible for fixing the build, and a lack of social pressure meant that these builds would often stay broken for longer than they should.

In Chapters Five and Six, a system to address these issues was discussed. First, a set of requirements was created, drawing on the information obtained from the interviews, observations, and related literature. The role of ambient awareness was then evaluated in terms of the properties of the display and the suitability of different types of notifications to the application at hand. Next, a series of ambient device technologies were considered; ultimately, a decision was made to provide the team with a central shared lamp as well as multiple individual devices in the form of Nabaztag Wi-Fi rabbits. A set of hypotheses was constructed to allow a formal quantitative evaluation of the system.

The implementation of the system required the construction of a number of software components to control the ambient devices and to interface with the team's build server. A custom device server translated the build results into hardware instructions. An initial prototype was provided to the team, and after some discussion with team members, was modified so it more closely fit with the team's work practices. A third prototype was constructed to evaluate the effect that more explicit positive reinforcement would have on the team members' behaviour.

Although this project was conducted as a single case study, the longitudinal nature did allow for a number of effects to be observed. The evaluation of the prototypes, discussed in Chapter Seven, was conducted over a period of several months and was preceded by a baseline phase in which build logs were recorded but no devices were provided. Based on the hypotheses developed in Chapter Five, a series of quantitative metrics was calculated to measure such constructs as the number of builds per day, the proportion of broken builds, and the duration of broken builds. Additionally, follow-up interviews were conducted with team members to gain a qualitative insight into their opinions of the system and its effects.

In Chapter Eight, these findings were summarised and discussed in more detail. In particular, the hypotheses outlined in Chapter Five were evaluated. Of the nine hypotheses, three were supported by the evidence, three were ambiguous, and three were not supported. Further investigation and consideration of the metrics, and of the information given by team members, indicated that the introduction of the ambient build monitoring systems did have an effect on the team's work, but that the nature of these effects, and the causal relationships that may control them, were difficult to explicate given the amount of variability that a single case study presents.

The strongest effects were seen at the beginning of the study, when a number of the hypothesised effects were observed. Over the course of the study these effects reduced (and, in some cases, disappeared). This pattern of results is consistent with a Hawthorne effect, suggesting that the novelty of the devices had a substantial impact on the team's work. Only some of these effects persisted beyond the first prototype phase.

One effect which did persist was a substantial decrease in the length of broken builds: not only were more broken builds being fixed on the day they were broken, but the durations of these broken build chains also decreased. This suggested that the team's overall awareness of broken builds was heightened. Additionally, team members reported they felt more aware of, and more responsible for, broken builds due to the public nature of the devices. However, there was no overall effect on whether any given broken build was fixed by a breaker. As such, a distinction should be drawn between the internal sense of responsibility a developer may feel and report, and the external responsibility that may be applied by social pressure. Awareness and these different forms of responsibility became central concepts, and our results indicate that the concepts are linked but should not be conflated.

While the sense of responsibility felt by individual developers was not directly linked to a quantifiable change in the observed metrics, other changes were observed which suggested individuals were in fact changing their overall behaviour. Several developers reported that they felt a stronger sense of awareness of the build status, including one developer who had previously been known for ignoring broken builds. Additionally, while the proportion of broken builds did increase over the study, the total number of builds also increased, as did the number of files per changeset. The reason for this may have been due to the specific development work being conducted by the team at the time of the evaluation, or the results may have been due to a larger change in the behaviour of team members.

Ambient devices have a number of important characteristics, and we were careful to follow the established conventions for the design of such devices. The distinct channel of information they provided meant that developers came to associate the devices with the build process. Similarly, the redundancy of the information encoded on the Nabaztag rabbits allowed multiple opportunities for developers to become aware of changes in the build status – through the observation of colour changes, ear movements, or by scanning the room and getting a general idea of who is currently working on broken builds. The principle of simplicity, in which devices presented a small amount of information in an easily perceivable manner, was highly successful with respect to the rabbits, but the level of aggregation this required for the central light made this device less useful to the team.

The key contributions of this research are:

- The introduction of a taxonomical framework to evaluate and compare ambient devices, and an initial comparison of a representative sample of known ambient devices.
- The introduction of a real-world case study of a moderately large development team, and a set of requirements that a build monitoring system should meet for that case study team.
- The explication of a set of nine specific and testable hypotheses based on the information gathered in the requirements gathering phase, which can reasonably be expected to be fulfilled with the introduction of a build monitoring and awareness system.
- The design and implementation of a prototype build monitoring and awareness system based upon these principles, as well as on suggestions and recommendations made during the prototyping phase.
- A set of data from the case study which provides support for and against the hypotheses above.
- A framework for constructing and evaluating build monitoring systems for individual teams.
- An improvement in the understanding of the effects of build monitoring systems, and of ambient awareness and ambient devices in general.

9.2. Future Work

This work has suggested several potential future directions for research into the area of ambient build status monitoring.

9.2.1. Further Case Studies

An obvious initial starting point for future work is to expand the pool of case studies using the framework outlined in this thesis. This project adopted a linear action research methodology because of the limited timeframe available. However, it would be relatively straightforward to expand this to a full cyclical action research methodology by introducing additional case study teams. Subsequent case studies could, depending on the circumstances, either be conducted simply by using the same prototypes and evaluation criteria, or could involve a more comprehensive analysis of those teams' practices and procedures with regards to broken builds.

Such an expansion would have the primary effect of improving the ability to generalise these results. A single- or small-N study such as this can be relied upon as a data point in a broader set of studies (Owens, 1979). This case study can provide a starting point for further exploration into this area, and after a series of replication studies sufficient data may be collected to allow more definitive conclusions to be drawn.

Similarly, additional case studies may adopt a 'return to baseline' design when providing the interventions. Such a design would improve the reliability and validity of the results by reducing the probability that observed effects are due to the timing of the study or coincident activities and changes within the team.

A longitudinal study may also present effects which have not been observed in the data described in this thesis. We are currently working with the case study team to continue this study for a much longer time period.

9.2.2. Expansion of Case Study

Even within this single case study, a number of findings emerged which were of interest but could not be fully explored due to time constraints. Further studies could broaden their scope of inquiry, or alter their focus, in order to examine some of these issues.

One such finding was the team's use of instant messaging as their main communication modality, described in Chapter Four (section 4.2.3, p49). The team had infrequent verbal and face-to-face interactions but used IM as the main substitute. A possible reason was to do with the physical layout of the team's workspace, although the team moved to a separate area during the study and still did not appear to alter this focus on IM. A number of study designs could be employed to examine this phenomenon, either within this organisation or more broadly.

Another incidental finding was the team's daily work practices, described in section 4.2.2 (p46). Developers held a large volume of shared knowledge about the ways in which they should perform their work, from assigning work items through to committing changes and waiting for the appropriate feedback from the build server. Similarly, the team had no formal process for assigning work items to individual developers, instead relying on shared knowledge and convention. The nature and extent of this shared knowledge, and the degree to which other agile teams use similarly informal practices, would be an important and interesting adjunct to this study.

9.2.3. More Complex Build Result Analysis

Within the arena of build status monitoring more specifically, there is scope for improving the analysis of the build server's results to improve the accuracy of the notifications. In particular, the process by which the build monitoring system assigns responsibility to individual developers could be improved through the inclusion of information beyond the commit logs – for example, the issue tracking system, granular unit test results, and so forth. Such an improvement must, however, be balanced against the need for transparency and visibility into the system. If complex manipulation or data mining is done on the data before it is presented to users, our findings suggest that it will become difficult for those users to trust the devices to give useful and consistent information.

For the purposes of evaluating such systems, more metrics could also be developed, and the existing metrics improved. The degree to which this is possible will be largely dependent on the build server and the logging information it provides. Atlassian Bamboo provides relatively little information, making such an analysis difficult and subject to inference and some degree of uncertainty. By coalescing results and data directly from the information sources – code compilation, unit tests, smoke tests, and the other metrics described in Chapter Two (section 2.2, p13) – more granular results could be obtained and more reliable correlations could be drawn.

9.2.4. Analysis of Other Build Status Monitoring Devices

Various types of ambient build monitoring devices have been adopted by teams within the software development industry. Little follow-up work has been reported in which these teams' devices are evaluated for actual (or even perceived) effects. To rectify this, a survey of these teams could be conducted to determine exactly what effects these teams expected to find, and what they observed following the introduction of these devices. Such a survey would also provide more concrete information on the prevalence of these systems; while some comments from industry members (e.g. Savoia, 2004) suggest that ambient build monitoring devices are becoming common, this has not actually been quantified. Furthermore, it would be a worthwhile exercise to correlate the prevalence of these devices with the various development methodologies that teams use.

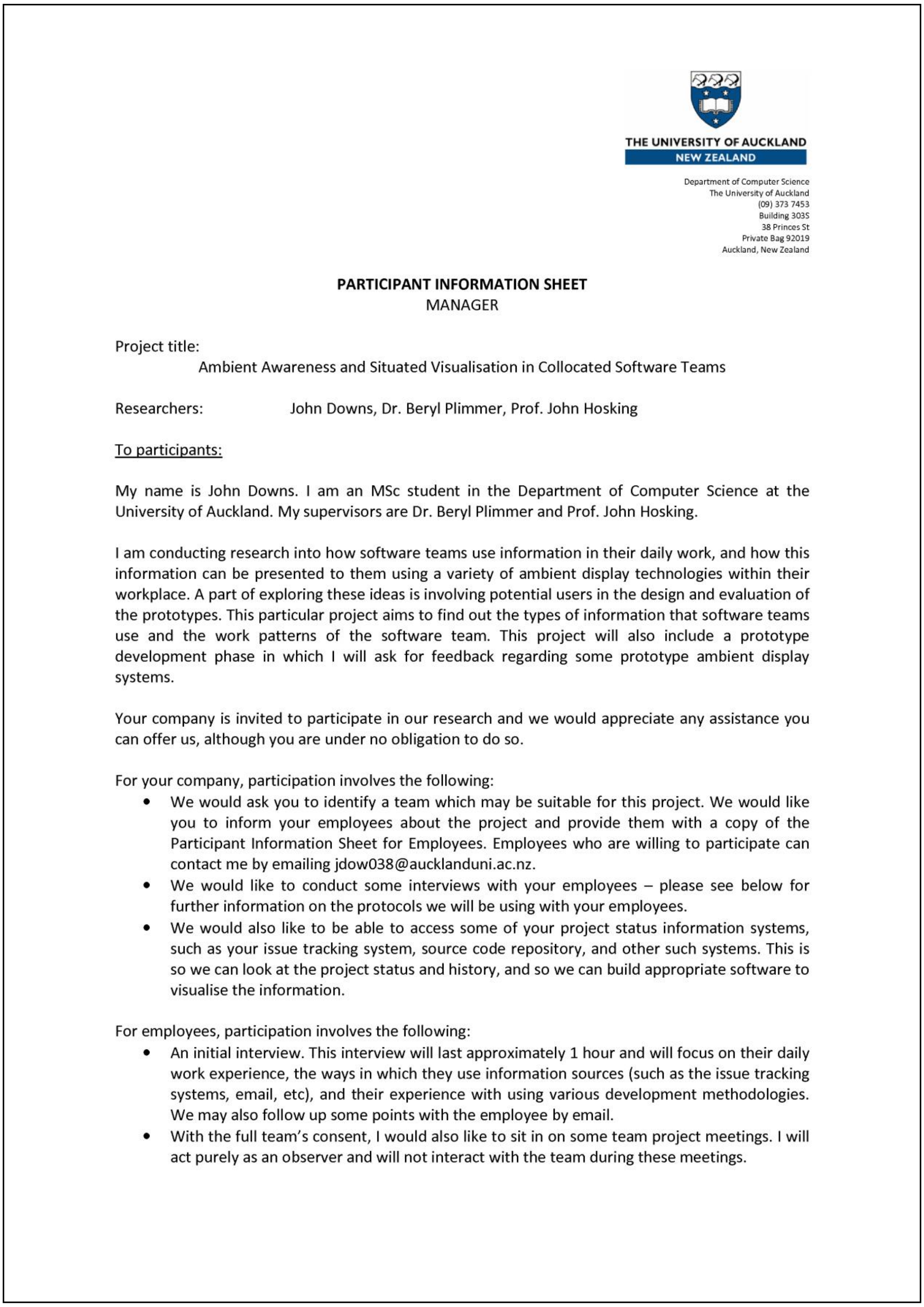
9.2.5. Examine Behavioural Implications

Finally, the extent to which ambient devices can be viewed strictly as sources of build failure notifications versus providing reinforcement and punishment is an area that requires further study.

A wide literature exists on the effects of reinforcement and punishment, and suggests that pairing reinforcement and punishment together will have the greatest effect.

This question has implications broader than purely within build notifications, with ambient devices becoming more common in myriad other application areas including homes, hospitals, offices, and even roads. As ambient devices become seen as a legitimate option for presenting complex and dynamic information in an contextually appropriate and cognitively minimalistic manner, the exploration of their meanings and properties, and of the design characteristics associated with their success, will gain increasing importance.

Appendix A: Participant Ethics Information



THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

Department of Computer Science
The University of Auckland
(09) 373 7453
Building 303S
38 Princes St
Private Bag 92019
Auckland, New Zealand

PARTICIPANT INFORMATION SHEET
MANAGER

Project title:
Ambient Awareness and Situated Visualisation in Collocated Software Teams

Researchers: John Downs, Dr. Beryl Plimmer, Prof. John Hosking

To participants:

My name is John Downs. I am an MSc student in the Department of Computer Science at the University of Auckland. My supervisors are Dr. Beryl Plimmer and Prof. John Hosking.

I am conducting research into how software teams use information in their daily work, and how this information can be presented to them using a variety of ambient display technologies within their workplace. A part of exploring these ideas is involving potential users in the design and evaluation of the prototypes. This particular project aims to find out the types of information that software teams use and the work patterns of the software team. This project will also include a prototype development phase in which I will ask for feedback regarding some prototype ambient display systems.

Your company is invited to participate in our research and we would appreciate any assistance you can offer us, although you are under no obligation to do so.

For your company, participation involves the following:

- We would ask you to identify a team which may be suitable for this project. We would like you to inform your employees about the project and provide them with a copy of the Participant Information Sheet for Employees. Employees who are willing to participate can contact me by emailing jdow038@aucklanduni.ac.nz.
- We would like to conduct some interviews with your employees – please see below for further information on the protocols we will be using with your employees.
- We would also like to be able to access some of your project status information systems, such as your issue tracking system, source code repository, and other such systems. This is so we can look at the project status and history, and so we can build appropriate software to visualise the information.

For employees, participation involves the following:

- An initial interview. This interview will last approximately 1 hour and will focus on their daily work experience, the ways in which they use information sources (such as the issue tracking systems, email, etc), and their experience with using various development methodologies. We may also follow up some points with the employee by email.
- With the full team's consent, I would also like to sit in on some team project meetings. I will act purely as an observer and will not interact with the team during these meetings.

- With the full team's consent, I would also like to work in the same physical environment for approximately 1 week. This will help me understand what a regular daily workflow is like for your team. During this time I will not interrupt the team's work or disturb any team member in any way.
- Next, I will develop a system with some ambient display technology. This is likely to be an iterative process, and I may ask employees for feedback at various stages of the process.
- Finally, I will conduct a concluding interview with each employee.

In total, I expect each employee's participation to take 2-3 hours of work time.

Employees' decisions to participate or not to participate will not be able to be disclosed to you, or to any third party. We seek assurance that employees' participation or non-participation will not have any effect on their employment.

With the employees' consent, we would like to record our interviews. This will only be used as an aid to us while analysing the information. Recordings will be stored electronically. These recordings will not be provided to any person outside the research team. You can elect to stop recording at any time during the interview.

Employees also have the right to withdraw from participation at any time, without giving a reason. Employees can elect to have their data destroyed for up to 1 week after providing the data. Data will be retained indefinitely unless the employee or you request it to be destroyed.

Data for this project will be used as part of my MSc thesis, and may also be used for research reports. We will either remove or change all personally identifiable information, and all information which identifies your company, when using information your employees provide during any of our interviews or conversations.

You will have the right to withdraw your company from participation at any time, without giving a reason. You may also withdraw any data you may have provided by contacting us within one week of providing that data.

Funding for this project has been provided to John Downs in the form of a Masters scholarship from the FRST RFI funded Software Process and Product Improvement project that John Hosking (co-supervisor) is an Objective leader on.

Our contact details are:

Researcher	**Supervisor**
John Downs MSc Student Department of Computer Science Ph 021 399 972 Email jdow038@aucklanduni.ac.nz	Dr. Beryl Plimmer Senior Lecturer Department of Computer Science Ph 3737599 x 82285 Email beryl@cs.auckland.ac.nz
Supervisor	**Head of Department**
Prof. John Hosking Professor Department of Computer Science Ph 3737599 x 88297 Email john@cs.auckland.ac.nz	Associate Professor Robert Amor Head of Department Department of Computer Science Ph 373 7599 x 83068 trebor@cs.auckland.ac.nz

For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Office of the Vice Chancellor, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 83711.

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 13 MAY 2009 FOR (3) YEARS REFERENCE NUMBER 2009/141

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

Department of Computer Science
The University of Auckland
(09) 373 7453
Building 303S
38 Princes St
Private Bag 92019
Auckland, New Zealand

CONSENT FORM
MANAGER

THIS FORM WILL BE HELD FOR A PERIOD OF 6 YEARS

Project title:
Ambient Awareness and Situated Visualisation in Collocated Software Teams

Researchers: John Downs, Dr. Beryl Plimmer, Prof. John Hosking

I have read the Participant Information Sheet, have understood the nature of the research and why I have been selected. I have had the opportunity to ask questions and have them answered to my satisfaction.

- I agree for my company to take part in this research.
- I agree that I will provide the research team with access to the project team's workplace, and to relevant sources of project information (such as the issue tracking system and source code repository), as agreed with the research team.
- I understand that employees' participation will involve approximately 2-3 hours' time each, including approximately 1 hour for an initial interview between the employee and a member of the research team. I understand that this time will be within work hours.
- I understand that employees are free to withdraw participation at any time without giving a reason, and to withdraw any data traceable to that employee up to one week after the interview.
- I understand that employees' identities will not be revealed to any third party, including myself. No third party will have any knowledge of whether a specific employee is participating in this research.
- I agree that no employment relationships will be affected in any way by an employee's participation or non-participation in this research.
- I understand that all information provided by employees during interviews, in other conversations with the research team, and through other means, will not be given to any third party, including myself, except for the purpose of the thesis or for publication of results. All information which identifies the company or the employee will be removed or changed prior to inclusion in the thesis or publication.
- I understand that I will receive a copy of the thesis and all publications resulting from this research.

- I understand that data will be kept indefinitely, unless I request its destruction.

Name ____________________________

Signature ____________________________ Date __________________

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 13 MAY 2009 FOR (3) YEARS REFERENCE NUMBER 2009/141

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

Department of Computer Science
The University of Auckland
(09) 373 7453
Building 303S
38 Princes St
Private Bag 92019
Auckland, New Zealand

PARTICIPANT INFORMATION SHEET
EMPLOYEE

Project title:
Ambient Awareness and Situated Visualisation in Collocated Software Teams

Researchers: John Downs, Dr. Beryl Plimmer, Prof. John Hosking

To participants:

My name is John Downs. I am an MSc student in the Department of Computer Science at the University of Auckland. My supervisors are Dr. Beryl Plimmer and Prof. John Hosking.

I am conducting research into how software teams use information in their daily work, and how this information can be presented to them using a variety of ambient display technologies within their workplace. A part of exploring these ideas is involving potential users in the design and evaluation of the prototypes. In this project we aim to find out the types of information that software teams use and the work patterns of the software team. This project will also include a prototype development phase in which I will ask for your feedback regarding some prototype ambient display systems.

You are invited to participate in our research and we would appreciate any assistance you can offer us, although you are under no obligation to do so. We have received approval from [redacted] to conduct this project, and your development team was identified as a possible team for us to work with.

Participation involves the following:

- An initial interview. This interview will last approximately 1 hour and will focus on your daily work experience, the ways in which you use information sources (such as the issue tracking systems, email, etc), and your experience with using various development methodologies. We may also follow up some points with you by email.
- With the team's consent, I would also like to sit in on some team project meetings. I will act purely as an observer and will not interact with the team during these meetings.
- With the team's consent, I would also like to work in the same physical environment for approximately 1 week. This will help me understand what a regular daily workflow is like for your team. During this time I will not interrupt your work or disturb you in any way.
- Next, I will develop a system with some ambient display technology. This is likely to be an iterative process, and I may ask you for feedback at various stages of the process.
- Finally, I will conduct a concluding interview.

In total, I expect your participation to take 2-3 hours of time.

Every effort will be made by the researchers to ensure that your decision to participate or not to participate will not be disclosed to your employer, or to any third party. However, given the small team size, complete secrecy may not be possible.

Your participation or non-participation will not affect your employment in any way.

With your consent, we would like to record our interviews with you. This will only be used as an aid to us while analysing the information. Recordings will be stored electronically. These recordings will not be provided to any person outside the research team. You can elect to stop recording at any time during the interview.

You can elect to have your data destroyed for up to 1 week after you provide the data. Data will be retained indefinitely unless you request it to be destroyed.

Data for this project will be used as part of my MSc thesis, and may also be used for research reports. We will either remove or change all personally identifiable information when using information you provide during any of our interviews or conversations.

You will have the right to withdraw from participation at any time, without giving a reason. You may also withdraw any data you may have provided by contacting us within one week of providing that data.

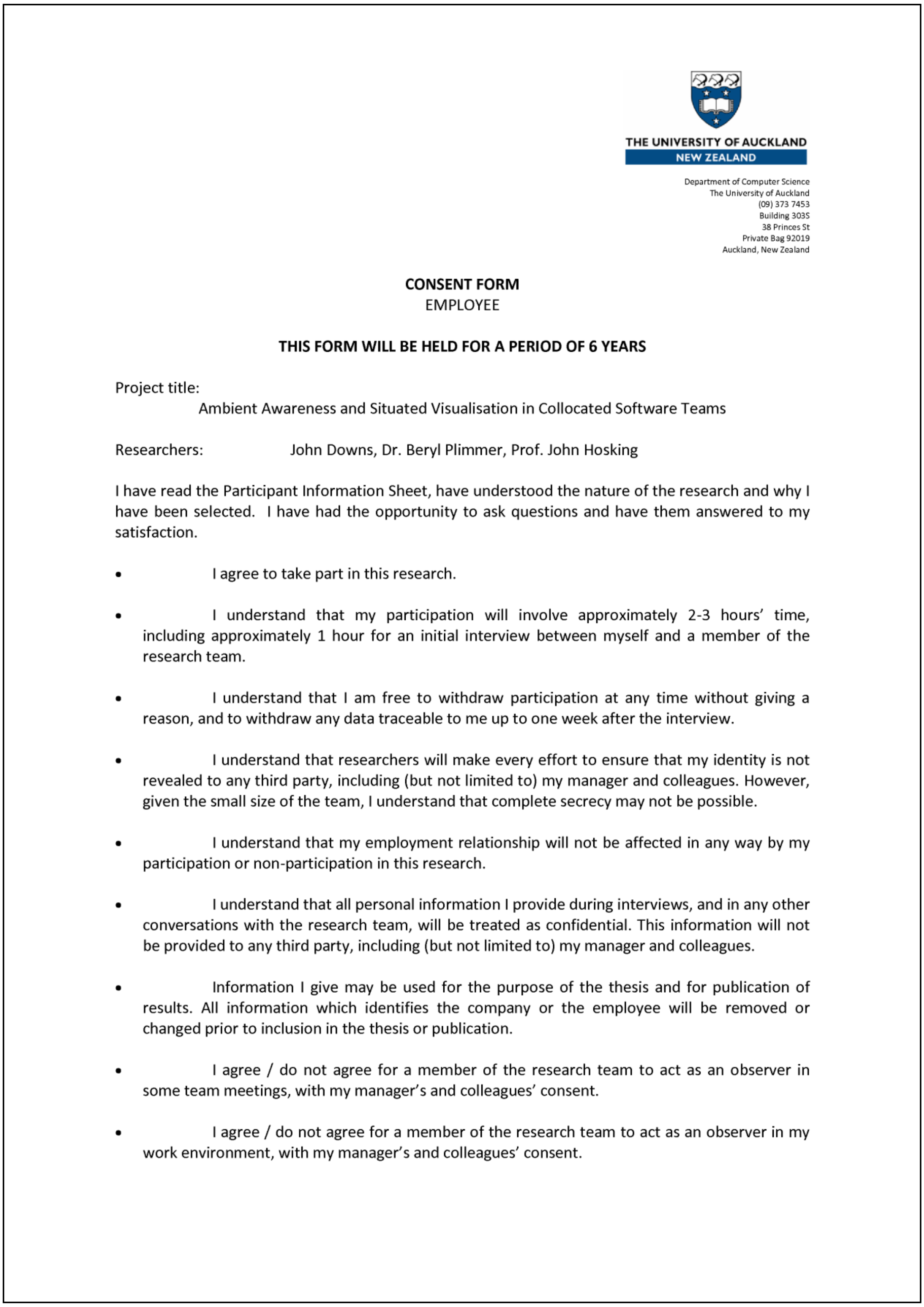
Funding for this project has been provided to John Downs in the form of a Masters scholarship from the FRST RFI funded Software Process and Product Improvement project that John Hosking (co-supervisor) is an Objective leader on.

Our contact details are:

Researcher	**Supervisor**
John Downs MSc Student Department of Computer Science Ph 021 399 972 Email jdow038@aucklanduni.ac.nz	Dr. Beryl Plimmer Senior Lecturer Department of Computer Science Ph 3737599 x 82285 Email beryl@cs.auckland.ac.nz
Supervisor	**Head of Department**
Prof. John Hosking Professor Department of Computer Science Ph 3737599 x 88297 Email john@cs.auckland.ac.nz	Associate Professor Robert Amor Head of Department Department of Computer Science Ph 373 7599 x 83068 trebor@cs.auckland.ac.nz

For any queries regarding ethical concerns you may contact the Chair, The University of Auckland Human Participants Ethics Committee, The University of Auckland, Office of the Vice Chancellor, Private Bag 92019, Auckland 1142. Telephone 09 373-7599 extn. 83711.

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 13 MAY 2009 FOR (3) YEARS REFERENCE NUMBER 2009/141



THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

Department of Computer Science
The University of Auckland
(09) 373 7453
Building 303S
38 Princes St
Private Bag 92019
Auckland, New Zealand

CONSENT FORM
EMPLOYEE

THIS FORM WILL BE HELD FOR A PERIOD OF 6 YEARS

Project title:
Ambient Awareness and Situated Visualisation in Collocated Software Teams

Researchers: John Downs, Dr. Beryl Plimmer, Prof. John Hosking

I have read the Participant Information Sheet, have understood the nature of the research and why I have been selected. I have had the opportunity to ask questions and have them answered to my satisfaction.

- I agree to take part in this research.
- I understand that my participation will involve approximately 2-3 hours' time, including approximately 1 hour for an initial interview between myself and a member of the research team.
- I understand that I am free to withdraw participation at any time without giving a reason, and to withdraw any data traceable to me up to one week after the interview.
- I understand that researchers will make every effort to ensure that my identity is not revealed to any third party, including (but not limited to) my manager and colleagues. However, given the small size of the team, I understand that complete secrecy may not be possible.
- I understand that my employment relationship will not be affected in any way by my participation or non-participation in this research.
- I understand that all personal information I provide during interviews, and in any other conversations with the research team, will be treated as confidential. This information will not be provided to any third party, including (but not limited to) my manager and colleagues.
- Information I give may be used for the purpose of the thesis and for publication of results. All information which identifies the company or the employee will be removed or changed prior to inclusion in the thesis or publication.
- I agree / do not agree for a member of the research team to act as an observer in some team meetings, with my manager's and colleagues' consent.
- I agree / do not agree for a member of the research team to act as an observer in my work environment, with my manager's and colleagues' consent.

- I agree / do not agree for my interview to be recorded.

- I wish / do not wish to receive the summary of findings.
- I understand that data will be kept indefinitely, unless I request its destruction.

Name ____________________________

Signature ____________________________ Date __________________

APPROVED BY THE UNIVERSITY OF AUCKLAND HUMAN PARTICIPANTS ETHICS COMMITTEE ON 13 MAY 2009 FOR (3) YEARS REFERENCE NUMBER 2009/141

Appendix B: Observational Study Results

This appendix lists the main results from the observational component described in Chapter Four.

Table 29. Total observations by behaviour type and day.

Day	Observation Time	Number of Instances of Behaviour			
		Talking	Calling to Person	Meeting	Total
Monday	6h 45m	43	12	1	56
Tuesday	5h 31m	33	41	0	74
Wednesday	3h 44m	16	19	0	35
Thursday	5h 59m	39	10	3	52
Friday	4h 38m	18	19	2	39
Total	**26h 37m**	**149**	**101**	**6**	**256**

Table 30. Average observations by behaviour type and day.

Day	Average Instances of Behaviour Per Hour			
	Talking	Calling to Person	Meeting	Total
Monday	6.37	1.78	0.15	8.30
Tuesday	5.98	7.43	0.00	13.41
Wednesday	4.29	5.09	0.00	9.38
Thursday	6.52	1.67	0.50	8.70
Friday	3.89	4.10	0.43	8.42
Total	**5.60**	**3.79**	**0.19**	**9.62**

Table 31. Mean number of participants and duration by behaviour type.

Behaviour Type	Mean Number of Participants (SD)	Mean Duration (mins) (SD)[1]
Talking	2.11 (0.39)	5.38 (19.76)[2]
Calling to Person	2.17 (0.52)	1.50 (1.53)
Meeting	11.67 (3.30)	42.67 (35.77)
Total	**2.36 (1.59)**	**4.75 (17.21)[3]**

Notes for Table 31:

1. Durations were not recorded for 16 communication instances (12 in the 'Talking' behaviour, 4 in the 'Calling to Person' behaviour). These have been excluded from these statistics.
2. One code review took 225 minutes in total, which skewed this mean and standard deviation. If these statistics are recalculated when this instance is excluded, the mean duration of Talking behaviour is 3.77 minutes with a standard deviation of 5.99 minutes.
3. As with 2, the 225 minute conversation skewed this mean and standard deviation. When excluding this instance the total mean duration is 3.83 minutes with a standard deviation of 9.67 minutes.

Appendix C: Initial Interview Schedule

This appendix includes the interview themes used for the initial interviews conducted with team members. These interviews are discussed in Chapter Four.

Interviews proceeded in a semi-structured manner, and although they loosely guided around these themes, interviewees frequently discussed other matters of interest or relevance.

- Job title and description
- Daily workflow – what is a typical day like
- The project they are working on – an overview and description
- The development methodologies they have used, and their thoughts on these
- Use of test-driven development and/or unit testing
- What they do while working (e.g. email, listen to music, surfing the web)
- Main software programs used each day
- Use of knowledge management tools – e.g. wikis, bug tracking systems, etc
- Types of information they use each day, and how – e.g. build reports
- What value is the burn down chart, and are there other similar charts they use
- Frequency and reason for checking in code
- Build process
- How they keep track of their part of the project and current tasks
- How they monitor the entire project as a whole
- A summary of their perception of the current project state
- General problems they encounter (with keeping aware of status and in general)

Appendix D: Device Server Pseudocode for Prototype 3

The pseudocode below accompanies the description of the third prototype's device server behaviour as described in Chapter Six, section 6.3.2 (page 76).

```
for each plan which is not marked as ignored {
   get most recent build
}

if all builds obtained above are passing {
   set central monitoring lamp to green
}
else {
   set central monitoring lamp to red
}

get the list of plans and their current status (passing or failing)

find the recent committers to each plan

refresh the matrix of results for each developer-plan combination

collapse the matrix into a single column with the final result for each developer

generate the marker files based on this matrix
```

Figure 32. Pseudocode for the overall Device Server logic used in prototype 3.

```
for each plan that should be included in the system {
   find the latest build in the plan
   find the latest successful build in the plan

   if the latest build was successful {
      get the last successful build
   }
   else {
      get a list of all builds since, but not including, the last successful build
   }

   for each build in the list obtained above {
      for each commit for this build {
         save the committer to the list of recent committers to this plan
      }
   }
}
```

Figure 33. Pseudocode for obtaining the list of committers to each plan, used in prototype 3.

```
if matrix does not exist yet {
   initialise matrix with a cell for each developer and plan
   set each cell to an initial status of ‘Neutral’
}

for each cell in the matrix {
   determine if the plan is currently passing

   set the currentResultForDeveloper variable to the default value of 'Neutral'

   if this plan is currently passing {
      if this developer is a recent committer to this plan {
         set the currentResultForDeveloper variable to 'Pass'
      }
      if this developer was given a Fail for this build last time {
         set the currentResultForDeveloper variable to 'Pass'
      }
   }
   else {
      if this developer is a recent committer to this plan {
         set the currentResultForDeveloper variable to 'Fail'
      }
   }

   if the current plan has had a new build since the last pass of this algorithm {
      update the cell with the new currentResultForDeveloper
   }
   else {
      if the current result is Pass and was given was more than 30 minutes ago {
         put the developer back to the Neutral state for this plan
         update the cell with the build number of the latest build in the plan
      }

      if the current result is Fail and the time that the last failure
         reminder was more than 30 minutes ago {
            mark the developer as requiring a failure notification
      }
   }
}
```
Figure 34. Pseudocode for updating the matrix of developer-plan results, used in prototype 3.

```
create a list to store the final results for each developer
initialise each result to null

for each cell in the matrix {
   if the current result for this developer is null {
      update the current result for this developer to the result in the cell
   }

   if this result has a higher priority than the current result for this
      developer {
         update the current result for this developer to the result in this cell
   }
}
```
Figure 35. Pseudocode for collapsing the matrix of developer-plan results into a final result for each developer, as used in prototype 3.

Appendix E: Full Evaluation Study Results

This appendix includes the complete set of results for the evaluation study metrics outlined in Chapter Seven.

Table 32. Broken builds per day by condition.

Condition	Total Broken Builds	Percentage of Total	Mean Broken Builds Per Day (sd)	Mean Percentage Broken Builds Per Day (sd)
Baseline	47	33.33%	1.96 (1.74)	28.91% (23.06%)
Prototype 1	28	22.95%	2.80 (2.14)	24.46% (18.66%)
Prototype 2	119	41.32%	5.67 (4.29)	38.01% (23.14%)
Prototype 3	92	40.00%	5.75 (5.08)	39.85% (27.62%)

Table 33. Builds per day by condition.

Condition	Duration	Total Builds	Mean Builds Per Day (sd)
Baseline	24 workdays	141	5.88 (3.09)
Prototype 1	10 workdays	122	12.20 (4.07)
Prototype 2	20 workdays	288	13.71 (6.66)
Prototype 3	16 workdays	232	14.50 (8.03)

Table 34. Mean number of files per changeset by condition.

Condition	Mean Number of Files Per Changeset (sd)
Baseline	4.30 (10.06)
Prototype 1	6.21 (10.10)
Prototype 2	5.73 (8.24)
Prototype 3	5.31 (10.30)

Table 35. Mean number of commits made while plan is broken, by condition.

Condition	Mean Commits to Broken Build Chains (sd)
Baseline	0.86 (0.69)
Prototype 1	0.88 (0.93)
Prototype 2	1.26 (0.92)
Prototype 3	1.07 (0.92)

Table 36. Reasons for builds occurring by condition.

Condition	Build Reason		
	Initial	Automatic	Manual
Baseline	2	125	14
Prototype 1	0	107	15
Prototype 2	2	256	30
Prototype 3	0	202	30

Table 37. Proportions of build reasons by condition.

Condition	Build Reason		
	Initial	Automatic	Manual
Baseline	1.42%	88.65%	9.93%
Prototype 1	0.00%	87.70%	12.30%
Prototype 2	0.69%	88.89%	10.42%
Prototype 3	0.00%	87.07%	12.93%

Table 38. Causes of broken builds by condition.

Condition	Cause of Failure	
	Test Failure	Other
Baseline	40	7
Prototype 1	21	7
Prototype 2	82	37
Prototype 3	69	23

Table 39. Proportions of broken builds for each cause by condition.

Condition	Cause of Failure	
	Test Failure	Other
Baseline	85.11%	14.89%
Prototype 1	75.00%	25.00%
Prototype 2	68.91%	31.09%
Prototype 3	75.00%	25.00%

Table 40. Builds fixed by developers who committed to failing plan by condition.

Condition	Total Broken Build Chains	Builds Fixed by Breaker	Percentage of Total
Baseline	19	13	68.42%
Prototype 1	12	9	75.00%
Prototype 2	46	31	67.39%
Prototype 3	38	26	68.42%

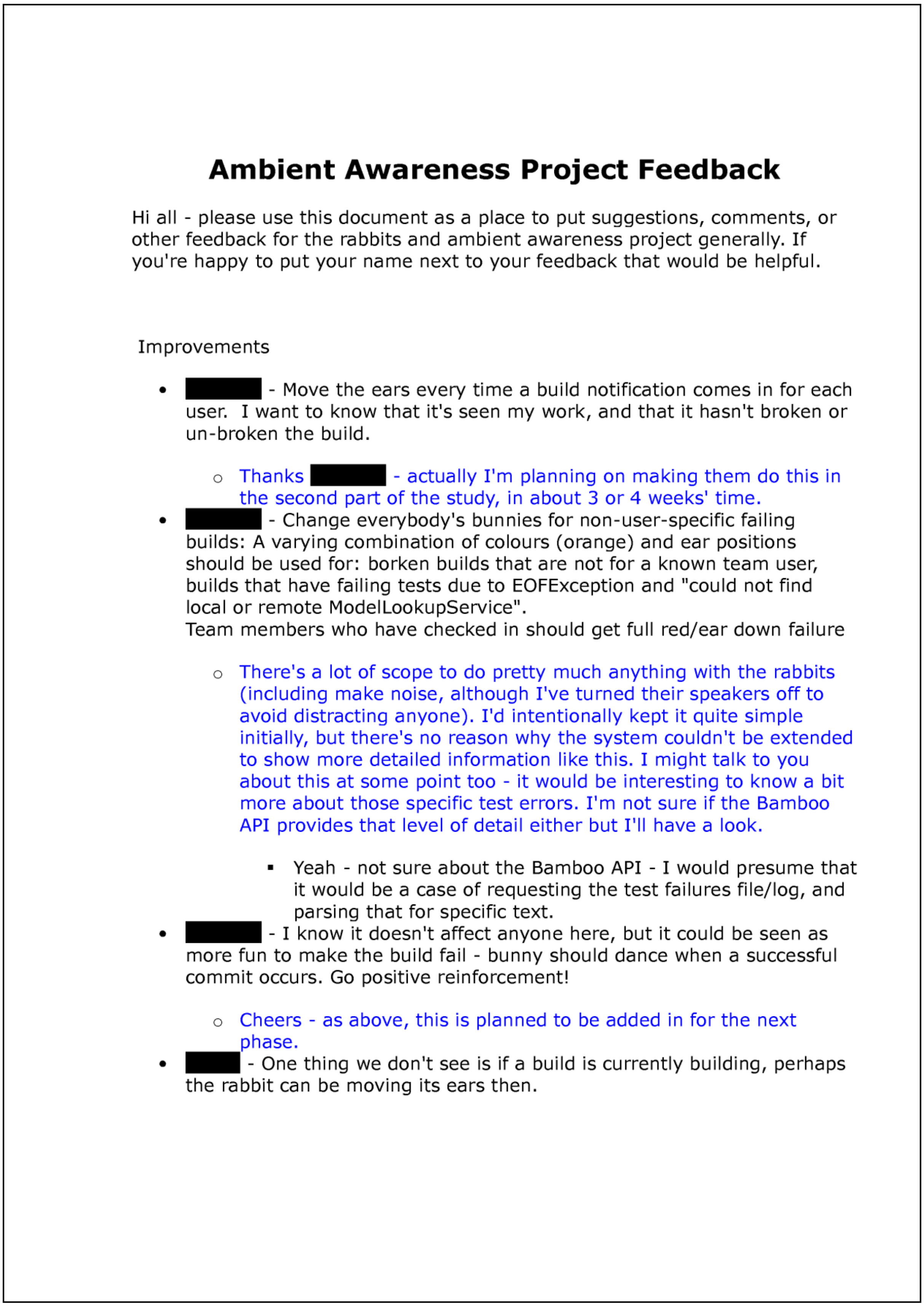
Table 41. Mean duration (in seconds) of broken build chains by condition.

Condition	Fixed Same Day	Fixed Same Day – Mean Duration (sd)	Fixed Multiple Days – Mean Duration (sd)
Baseline	63.16%	9513 (9897)	40483 (21765)
Prototype 1	91.67%	5709 (5199)	35141 (0)
Prototype 2	67.39%	6278 (6264)	29859 (16829)
Prototype 3	65.79%	7273 (6311)	34680 (22993)

Table 42. Time central light showed red vs. green (in seconds) by condition.

Condition	Total Work Time	Time Showing Red	Percentage of Time in Red
Baseline	937800	359927	38.52%
Prototype 1	397800	84740	10.86%
Prototype 2	793800	432223	54.45%
Prototype 3	547200	341435	62.40%

Appendix F: Project Feedback from Team



Ambient Awareness Project Feedback

Hi all - please use this document as a place to put suggestions, comments, or other feedback for the rabbits and ambient awareness project generally. If you're happy to put your name next to your feedback that would be helpful.

Improvements

- [redacted] - Move the ears every time a build notification comes in for each user. I want to know that it's seen my work, and that it hasn't broken or un-broken the build.
    - Thanks [redacted] - actually I'm planning on making them do this in the second part of the study, in about 3 or 4 weeks' time.
- [redacted] - Change everybody's bunnies for non-user-specific failing builds: A varying combination of colours (orange) and ear positions should be used for: borken builds that are not for a known team user, builds that have failing tests due to EOFException and "could not find local or remote ModelLookupService".
  Team members who have checked in should get full red/ear down failure
    - There's a lot of scope to do pretty much anything with the rabbits (including make noise, although I've turned their speakers off to avoid distracting anyone). I'd intentionally kept it quite simple initially, but there's no reason why the system couldn't be extended to show more detailed information like this. I might talk to you about this at some point too - it would be interesting to know a bit more about those specific test errors. I'm not sure if the Bamboo API provides that level of detail either but I'll have a look.
        - Yeah - not sure about the Bamboo API - I would presume that it would be a case of requesting the test failures file/log, and parsing that for specific text.
- [redacted] - I know it doesn't affect anyone here, but it could be seen as more fun to make the build fail - bunny should dance when a successful commit occurs. Go positive reinforcement!
    - Cheers - as above, this is planned to be added in for the next phase.
- [redacted] - One thing we don't see is if a build is currently building, perhaps the rabbit can be moving its ears then.

- [redacted] - Perhaps a little tweak at the start; else [redacted]'s rabbit will permanently wear a hat or something. Alternatively, one of the lights could flash occasionally.
- Good point. Unfortunately the Bamboo API doesn't provide a way to find out when a build is in progress. This is something I'll look at when you upgrade to Bamboo 2.4 though - there may be something new in there to enable this.

Bugs

- New committers to a failing build don't get sad bunnies.
    - This is actually by design. If multiple builds fail in succession it only looks at the first failing build in the chain. This was based on the assumption (admittedly not perfect) that the initial build was the one that is mostly likely to have been broken somehow, and than subsequent builds weren't likely to have fixed that problem but introduced others. However, I could change it so that anyone who commits to a failing build gets a sad rabbit, if you think this would be better?
        - [redacted] - yes, probably better if anyone who commits to a failing build gets a sad rabbit. If nothing else, it gets more people aware of the fact that build is still broken and needs to be fixed promptly.
            - OK great. I'll make this change over the next few days and will update the software. Thanks!
            - Will be changed on Monday 2nd November.
- The flashing red light on the rabbits' noses will be removed on Monday 2nd November.

Appendix G: Final Interview Schedule

This appendix includes the interview themes used for the final interviews conducted with team members following the main evaluation study. These interviews are discussed in Chapter Seven. Note that at the time of the interviews, the prototype 3 behaviour had only been provided to the team for one day and therefore was not generally discussed in the interviews.

Interviews proceeded in a semi-structured manner, and although they loosely guided around these questions, interviewees frequently discussed other matters of interest or relevance.

- What did you think about the central light?
    - Did you find that you looked at it often?
- What did you think of the rabbits?
- Did you think either device made much of a difference to anything?
- What happened when a build broke? Was this different to what used to happen?
- How urgent is it to fix broken builds?
- Would you want to keep any of the devices indefinitely? (If so, which would you want to keep more?)
- Was there anything that annoyed you, or that would discourage you from using a similar device in the future?
- Thinking about the different behaviours (i.e. prototypes 1 and 2) - which was most useful?
- Which would you like to keep?
- Have you heard about or used other build monitoring systems – e.g. lava lamps, etc?
    - If so, what do you think of these vs. the rabbits and light?
- Do you have any other suggestions?

References

Ambient Devices, Inc. (2009). Ambient Orb. Retrieved March 19, 2009, from http://www.ambientdevices.com/cat/orb/orborder.html

Apache Software Foundation. (2009). Apache Maven. Retrieved December 30, 2009, from http://maven.apache.org/

Atlassian Pty Ltd. (2009a). Bamboo Remote API. Retrieved July 20, 2009, from http://confluence.atlassian.com/display/BAMBOO/Bamboo+Remote+API

Atlassian Pty Ltd. (2009b). Bug Tracking, Issue Tracking & Project Management Software - JIRA. Retrieved July 20, 2009, from http://www.atlassian.com/software/jira/

Atlassian Pty Ltd. (2009c). Continuous Integration Tool - Bamboo. Retrieved July 20, 2009, from http://www.atlassian.com/software/bamboo/

Atwood, J. (2005). Automated Continuous Integration and the BetaBrite LED Sign. Retrieved April 8, 2009, from http://www.codinghorror.com/blog/archives/000238.html

Barlow, D. H., & Hersen, M. (1984). *Single Case Experimental Designs: Strategies for Studying Behavior Change*. New York: Pergamon Press Inc.

Baskerville, R. L., & Wood-Harper, A. T. (1996). A critical perspective on action research as a method for information systems research. *Journal of Information Technology, 11*(3), 235-246.

Baskerville, R. L., & Wood-Harper, A. T. (1998). Diversity in information systems action research methods. *European Journal of Information Systems, 7*, 90-107.

Beck, K. (2002). *Test Driven Development: By Example*. Boston, MA, US: Addison-Wesley Longman Publishing Co., Inc.

Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change* (2nd ed.). Boston, MA: Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al. (2001). Manifesto for Agile Software Development. Retrieved December 14, 2009, from http://agilemanifesto.org/

Beedle, M., Devos, M., Sharon, Y., Schwaber, K., & Sutherland, J. (1999). SCRUM: An extension pattern language for hyperproductive software development. *Pattern Languages of Program Design, 4*, 637-651.

Biehl, J. T., Czerwinski, M., Smith, G., & Robertson, G. G. (2007). FASTDash: A visual dashboard for fostering awareness in software teams. *Proceedings of SIGCHI Conference on Human Factors in Computer Systems*, San Jose, CA, pp. 1313-1322.

Bowyer, J., & Hughes, J. (2006). Assessing undergraduate experience of continuous integration and test-driven development. *Proceedings of 28th International Conference on Software Engineering*, Shanghai, China, pp. 691-694.

Boyatzis, R. E. (1998). *Transforming Qualitative Information*. Thousand Oaks, CA, US: Sage Publications, Inc.

Cadiz, J. J., Fussell, S. R., Kraut, R. E., Lerch, F. J., & Scherlis, W. L. (1998). The Awareness Monitor: A coordination tool for asynchronous, distributed work teams. Unpublished Manuscript.

Ceschi, M., Sillitti, A., Succi, G., & De Panfilis, S. (2005). Project management in plan-based and agile companies. *IEEE Software, 22*(3), 21-27.

Chumby Industries. (2006). Chumby. Retrieved December 10, 2009, from http://www.chumby.com/

Clark, M. (2004). *Pragmatic Project Automation*. Raleigh, NC: The Pragmatic Programmers, LLC.

Clark, R. E., & Sugrue, B. M. (1996). Research on instructional media, 1978-1988. *Instructional Technology: Past, Present and Future*, 336-346.

Clarke, K., Hughes, J., Rouncefield, M., & Hemmings, T. (2003). When a bed is not a bed: The situated display of knowledge on a hospital ward. In K. O'Hara, M. Perry, E. Churchill & D. Russell (Eds.), *Public and Situated Displays: Social and Interactional Aspects of Shared Display Technologies*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

CollabNet. (2009). SVN. Retrieved July 3, 2009, from http://subversion.tigris.org

Cunningham, W. (2007, October 12). Framework for Integrated Test. Retrieved May 19, 2009, from http://fit.c2.com/

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM, 31*(11), 1268-1287.

da Silva, I., Alvim, M., Ripley, R., Sarma, A., Werner, C., & van der Hoek, A. (2007). Designing software cockpits for coordinating distributed software development. *Proceedings of First Workshop on Measurement-Based Cockpits for Distributed Software and Systems Engineering Projects (SOFTPIT 2007)*, Munich, Germany, pp. 14-18.

Damian, D., Izquierdo, L., Singer, J., & Kwan, I. (2007). Awareness in the wild: Why communication breakdowns occur. *Proceedings of International Conference on Global Software Engineering*, Munich, Germany, pp. 81-90.

de Morlhon, J.-L. (2009). Nabaztag Scala Library. Retrieved November 15, 2009, from http://morlhon.net/blog/2009/11/09/nabaztag-scala-library/

Delcom Products Inc. (2009, Jan 29, 2009). Delcom Products Inc. Web Site. Retrieved September 1, 2009, from http://www.delcomproducts.com/

Dourish, P., & Bellotti, V. (1992). Awareness and coordination in shared workspaces. *Proceedings of 1992 ACM Conference on Computer-Supported Cooperative Work*, Toronto, Ontario, Canada, pp. 107-114.

Downs, J., & Plimmer, B. (2008, June). The use of digital photo frames as situated messaging appliances. *Proceedings of 1st International Conference on Computer-Mediated Social Networking*, Dunedin, New Zealand, pp. 100-105.

Downs, J., & Plimmer, B. (2009, April). Evaluating non-interactive situated SMS messaging. *Proceedings of 27th International Conference on Human Factors in Computing Systems*, Boston, MA, pp. 3709-3714.

Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston, MA: Pearson Education, Inc.

Ebert, C., Parro, C. H., Suttels, R., & Kolarczyk, H. (2001). Improving validation activities in a global software development. *Proceedings of 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, pp. 545-554.

Eclipse Foundation. (2009). Eclipse. Retrieved December 13, 2009, from http://www.eclipse.org/

Edgewall Software. (2009). The Trac Project. Retrieved December 9, 2009, from http://trac.edgewall.org/

Eick, S. G., & Steffen, J. L. (1992). Visualizing code profiling line oriented statistics. *Proceedings of 3rd Conference on Visualization (VIS '92)*, Boston, MA, pp. 210-217.

Elashoff, J. D., & Thoresen, C. E. (1978). Choosing a statistical method for analysis of an intensive experiment. In T. R. Kratochwill (Ed.), *Single Subject Research: Strategies for Evaluating Change*. New York: Academic Press.

Elliot, K., & Greenberg, S. (2004). Building flexible displays for awareness and interaction, *Video Proceedings and Proceedings Supplement of the UBICOMP 2004 Conference, (Sept 7-10, Nottingham, England). 6 minute video and 2-page summary.*

Elliot, K., Watson, M., Neustaedter, C., & Greenberg, S. (2007). *Location-dependent information appliances for the home*. Paper presented at the Graphics Interface 2007.

Few, S. (2006). *Information Dashboard Design: The Effective Visual Communication of Data*. Sebastopol, CA, US: O'Reilly Media, Inc.

Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology, 2*(2), 115-150.

Flanagan, J. C. (1954). The critical incident technique. *Psychological Bulletin, 54*(4).

Fowler, M. (2006). Continuous integration. Retrieved April 21, 2009, from http://www.martinfowler.com/articles/continuousIntegration.html

Greenberg, S. (1999). Designing computers as public artifacts. *International Journal of Design Computing: International Journal of Design Computing: Special Issue on Design Computing on the Net (DCNet'99)*.

Gutwin, C., & Greenberg, S. (2001). *The importance of awareness for team cognition in distributed collaboration*: Department of Computer Science, University of Calgary, Alberta, Canada.

Gutwin, C., Penner, R., & Schneider, K. (2004). Group awareness in distributed software development. *Proceedings of ACM Conference on Computer Supported Cooperative Work*, Chicago, IL, pp. 72-81.

Harrison, S. (2007). Putting the SnowMan to work: Cruise Control .Net Build Status Monitor. Retrieved March 16, 2008, from http://www.analysisuk.com/blog/archives/33-Putting-the-SnowMan-to-work-Cruise-Control-.Net-Build-Status-Monitor..html

Holck, J., & Jorgensen, N. (2004). Continuous integration and quality assurance: A case study of two open source projects. *Australian Journal of Information Systems, 11*, 40-53.

Howell, D. C. (2004). *Fundamental Statistics for the Behavioral Sciences* (5th ed.). Belmont, CA, USA: Thomson Learning.

Jeffries, R., Anderson, A., & Hendrickson, C. (2001). *Extreme Programming Installed*. Boston: Addison-Wesley.

Karlsson, E.-A., Andersson, L.-G., & Leion, P. (2000). Daily build and feature development in large distributed projects. *Proceedings of 22nd International Conference on Software Engineering*, Limerick, Ireland, pp. 649-658.

Keil, M., & Robey, D. (2001). Blowing the whistle on troubled software projects. *Communications of the ACM, 44*(4), 87-93.

Kim, S., Park, S., Yun, J., & Lee, Y. (2008). Automated continuous integration of component-based software: An industrial experience. *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, pp. 423-426.

Latane, B., Williams, K., & Harkins, S. (1979). Many hands make light the work: The causes and consequences of social loafing. *Journal of Personality and Social Psychology, 37*(6), 822-832.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: A study of developer work habits. *Proceedings of 28th International Conference on Software Engineering*, Shanghai, China, pp. 492-501.

Mayo, E. (1949). *Hawthorne and the Western Electric Company: The Social Problems of an Industrial Civilisation*: Routledge.

McConnell, S. (1996). Daily Build and Smoke Test. *IEEE Software, 13*(4), 143-144.

McKay, J., & Marshall, P. (2000). *Quality and rigour of action research in information systems*. Paper presented at the 8th European Conference on Information Systems.

Meier, J. D., Vasireddy, S., Babbar, A., & Mackman, A. (2004). Use CLR Profiler: Improving .NET Application Performance and Scalability. *Microsoft Patterns & Practices Developer Center* Retrieved May 20, 2009, from http://msdn.microsoft.com/en-us/library/ms979205.aspx

Microsoft Corporation. (2007). LINQ to SQL: .NET Language-Integrated Query for Relational Data. Retrieved October 20, 2009, from http://msdn.microsoft.com/en-us/library/bb425822.aspx

Microsoft Corporation. (2009a). Microsoft .NET Framework. Retrieved October 20, 2009, from http://www.microsoft.com/net/

Microsoft Corporation. (2009b). SQL Server 2008 Express. Retrieved October 20, 2009, from http://www.microsoft.com/express/sql/

Microsoft Corporation. (2009c). Visual Studio Team System: Unit Testing Framework. Retrieved November 9, 2009, from http://msdn.microsoft.com/en-us/library/ms243147.aspx

Microsoft Corporation. (2010). Define a Build to Validate Changes Before Check-in. Retrieved January 21, 2010, from http://msdn.microsoft.com/en-us/library/dd787631(VS.100).aspx

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008). Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering, 13*, 289-302.

NCover. (2004). NCover: Code Coverage for .NET. Retrieved Feb 7, 2005, from http://www.ncover.com

NUnit.org. (2009). NUnit. Retrieved May 19, 2009, from http://www.nunit.org/

Owens, R. G. (1979). Do psychologists need statistics? *Bulletin of the British Psychological Scoiety, 32*, 103-106.

Peled, D. (2001). *Software Reliability Methods*. New York: Springer.

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*. New York: McGraw Hill.

QSR International. (2009). NVivo 8 research software for analysis and insight. Retrieved September 24, 2009, from http://www.qsrinternational.com/products_nvivo.aspx

Quibell, S. (2006). Continuous Integration with CC.Net and Dell XPS LED "Ambient" Lights. Retrieved December 14, 2009, from http://blogs.quibell.net/blogs/scott/archive/2006/07/23/10.aspx

Rajaniemi, J.-P. (2007). Java Framework for WiFi-Based Nabaztag Device. Retrieved September 7, 2009, from http://www.cs.uta.fi/reports/dsarja/D-2007-11.pdf

Rogers, O. (2008, August 14). Continuous monitoring tutorial at Agile 2008. *Exortech.com Blog* Retrieved March 3, 2009, from http://exortech.com/blog/2008/08/14/continuous-monitoring-tutorial-at-agile-2008/

Ross, P., & Keyson, D. V. (2007). The case of sculpting atmospheres: Towards design principles for expressive tangible interaction in control of ambient systems. *Personal and Ubiquitous Computing, 11*(2), 69-79.

Saff, D., & Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis* Boston, MA, pp. 76-85.

Sarma, A., Noroozi, Z., & van der Hoek, A. (2003). Palantír: Raising awareness among configuration management workspaces. *Proceedings of 25th International Conference on Software Engineering*, Portland, Oregon, pp. 444-454.

Savoia, A. (2004). On Java Lava Lamps and Other eXtreme Feedback Devices. Retrieved October 3, 2009, from http://www.artima.com/weblogs/viewpost.jsp?thread=67492

Schwaber, K. (1995). *SCRUM development process*. Paper presented at the OOPSLA'95 Workshop on Business Object Design and Implementation.

Sellen, A., Eardley, R., Izadi, S., & Harper, R. (2006). The Whereabouts Clock: Early testing of a situated awareness device. *Proceedings of Conference on Human Factors in Computing Systems*, Montreal, Quebec, Canada, pp. 1307-1312.

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. *Proceedings of IEEE Symposium on Visual Languages*, Los Alamitos, CA, USA, pp. 336.

Sillito, J., Murphy, G. C., & Volder, K. D. (2006). Questions programmers ask during software evolution tasks. *Proceedings of 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, OR, pp. 23-34.

Sommerville, I. (2007). *Software Engineering* (8th ed.). New York: Addison-Wesley.

Sutherland, J. (2001). Agile can scale: Inventing and reinventing SCRUM in five companies. *Cutter IT Journal, 14*(12), 5-11.

Swanson, M. (2004). Automated continuous integration and the Ambient Orb. Retrieved April 21, 2009, from http://blogs.msdn.com/mswanson/articles/169058.aspx

Thorndike, E. L. (1911). *Animal Intelligence*. New York: Macmillan.

ThoughtWorks, Inc. (2009). Cruise: Continuous integration and release management. Retrieved April 21, 2009, from http://studios.thoughtworks.com/cruise-continuous-integration

University of Tampere. (2008). jNabServer - Control your bunny. Retrieved July 27, 2009, from http://www.cs.uta.fi/hci/spi/jnabserver/

Violet. (2009a). The first smart rabbit - Nabaztag. Retrieved July 15, 2009, from http://www.nabaztag.com

Violet. (2009b). Nabaztag's API. Retrieved July 25, 2009, from http://help.nabaztag.com/fiche.php?fiche=29

White, S. (2008). *Interaction and presentation techniques for situated visualization*. Paper presented at the ACM Symposium on User Interface Software and Technology.

Wolf, T., Schroter, A., Damian, D., & Nguyen, T. (2009). Predicting build failures using social network analysis on developer communication. *Proceedings of 31st International Conference on Software Engineering*, Vancouver, BC, Canada, pp. 1-11.

Woodward, M. (2007). Team Build 2008 API Example: The Build Wallboard. Retrieved October 3, 2009, from http://www.woodwardweb.com/vsts/000395.html

Woodward, M. (2008a). Behind Brian the Build Bunny. Retrieved October 3, 2009, from http://www.woodwardweb.com/vsts/behind_brian_th.html

Woodward, M. (2008b). Brian the Build Bunny. Retrieved October 3, 2009, from http://www.woodwardweb.com/gadgets/000434.html