A Multi-Touch Explorer Environment for Eclipse

by

Daniel Cope

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Masters of Engineering
in Software Engineering.

Victoria University of Wellington
2013

Abstract

The Multi-Touch Explorer Environment (MTEE) is a tool to aid developers during the production of multi-touch enabled applications. The MTEE tool integrates into the Eclipse IDE and can be used to record and playback user interactions with the program, compare sessions of recorded user interactions and investigate the evolution of the program behaviour. The tool presented in this thesis focuses on the Eclipse IDE and Multi-Touch for Java framework, as they are tools used by both Students and Professional developers. It is demonstrated that the Multi-Touch Explorer Environment can be integrated seamlessly into the Eclipse IDE. It is also demonstrated that the MTEE tool can be used to profile the user's program with little impact on the performance of both the system or the program itself.

Acknowledgments

Firstly I would like to thank Dr. Stuart Marshall, for supervising the completion of my Masters study. He taught me many times in my undergraduate years and supervised the study of my Honours project. I thank him for all of his guidance and support throughout that time.

I thank my fellow peers and the members of the HCI group, for providing feedback and support during my project. I wish them the best with all of their studies.

I would like to thank my proof readers; Fahmi Abdulhamid, Mata Freshwater, Chris Green and Yohan Ng, for providing much needed feedback on the clarity of my work.

Finally, I thank my parents, David and Anne; my sister, Hannah; my girlfriend, Sam; and all of my friends for encouraging me during my Masters study and all throughout my time at Victoria University.

Contents

List of Figures

Chapter 1

Introduction

The Multi-Touch Explorer Environment (MTEE) is a tool to aid developers during the production of multi-touch enabled programs. The tool integrates into the Eclipse IDE to provide developers with a recording and replay feature that allows the developer to record a user's touch interaction during testing of their program and play it back. The thesis focuses on the production and assessment of prototype applications using the Multi-Touch for Java framework. This functionality gives the developers the ability to compare the gestures and interactions during prototyping as the UI evolves. A snapshot system allows developers to compare sessions from earlier versions of their program, long after the behaviour of the program has changed. MTEE collects data about the developers program such as; the structure of components in the display, the gestures each component will respond too, and information about gestures that were performed on a component during a recording. This data is displayed directly in the Eclipse IDE.

The intention of this project is to show that extra tools can be produced to aid a developer when prototyping multi-touch applications. The MTEE tool presented in this thesis chose to focus on the Eclipse IDE and Multi-Touch for Java, as they are tools used by both students and professionals. The broader initial goal was to develop these tools as a use case for how

multi-touch development can be improved with other frameworks and development environments.

This thesis demonstrates that the Multi-Touch Explorer Environment can be integrated seamlessly into the Eclipse IDE. It demonstrates that the MTE tool can be used to profile the user's program with little impact on the performance of both the system or the program itself. The results of the evaluation imply that the project is a success, and that similar tools could be created to interface with different integrated development environments and multi-touch frameworks.

1.1 Contributions

The completion of the Multi-Touch Explorer Environment (MTEE) resulted in three contributions:

- The design of a tool to record and replay user interactions with a multi-touch program. This included the analysis and development of two primary personas related to the target users.

- A proof-of-concept prototype for the MTEE. The prototype integrates the MTEE into the Eclipse IDE and focused on projects using the Multi-Touch for Java framework. The MTEE prototype consists of two parts, the plugin for the Eclipse IDE and the Multi-Touch Explorer Analysis (MTEA) tool.

- An evaluation of the Multi-Touch Explorer Environment. This included a set of performance metrics and test suite to allow continual testing of the non-function requirements. It also included a Cognitive Walkthrough [22] using expert users, resulting in a list of changes and improvements that can be made to the project going forward.

1.2 Motivation

Multi-Touch is a relatively new area of mainstream software and hardware development. While touch capable devices have existed in research laboratories since the 1980's [68], the first consumer level products did not start appearing until the late 2000's with the Apple iPhone [12] and the Microsoft Surface were unveiled [57]. This sudden availability of consumer level hardware quickly made multi-touch an attractive area of software development.

For the completion of my Honours project I used the Java programming language and the Multi-Touch for Java (MT4J) framework, to produce a system for navigating 3D environments using 2D touch interfaces [23]. I created a series of prototype gestures which could be used to move the user around a 3D virtual scene. Due to the added complexity a multi-touch interface introduced, I found it difficult to compare the behaviour of the different sets of gestures, particularly when looking at ways that the user would actually use them. I also had difficultly distinguishing which components related to which gestures.

This led to the idea of creating a dedicated environment for recording and comparing data from multi-touch programs. Such a tool could be applied to any multi-touch development environment, allowing developers to make informed decisions based on data from their application. This would improve the speed at which developers could create and test prototypes of new programs.

1.3 Thesis Structure

This thesis is presented in six chapters; an introduction to the project, the Background and Requirements Analysis, the project design, the project implementation, the project evaluation, and a summary of the project.

The introduction clarifies what the project is and what this thesis as

contributed. The introduction also discusses the motivation for the project.

The background and requirements analysis presents the related works, user profiles, and personas related to the target users. The personas have defined goals and scenarios which are used to draw up both the functional and non-functional requirements of the project.

The design chapter outlines the designs for both the Eclipse plugin and the external tool, called the Multi-Touch Explorer Analysis (MTEA) tool. These make up the core of the prototype design for the MTEE.

The implementation chapter discusses the tools used for the project and how the project was implemented using these tools. This covers the system design of both the Eclipse IDE plugin and the MTEA tool.

The project is evaluated using performance tests and an expert evaluation of the interface. This indicates how the project has met the requirements of the project and how well the interface would perform in front of real users. The summary contains the changes and improvements that will be made to the project going forward, along with recapping the contributions this project has made.

Chapter 2

Background & Requirements Analysis

In this chapter we examine the related works that formed a basis for the project, as while as analyse the project requirements; both functional and non-functional.The functional requirements are informed by the development of a number of personas; a primary, secondary and two negative personas [22]. By researching and defining the background for the personas we were able to accurately measure the user's goals and produce a number of scenarios on how they would use the tool. The non-functional requirements were then developed to meet the system needs of these personas.

We discuss the project's related works, covering other tools and systems that shared functionality in common. We then discuss the target user's profiles in section 2.2. These were used to develop a strong background for each user, which the personas are then be developed from. These finalised personas are presented in section 2.3. Section 2.4 summaries both the functional and non-functional requirements of the system.

2.1 Related Works

The tool presented in this thesis used a number of different techniques for the analysis and display of data. While there have been no projects to bring this collection of techniques together to explore multi-touch applications, other projects have dealt with collecting and presenting data in a similar manner. By examining projects which share these features we could become better informed of the problems and conclusions these projects encountered.

Parts of the functionality of the tool presented in this thesis focuses on providing information to better help new users understand a multi-touch interface and the associate multi-touch framework. Training and educational qualities, such as those, had been considered for this project, however as this topic covers a broad area we deemed it inappropriate to pursue. To properly implement the functionality required to support complete new users into the existing features would have required a large time commitment, which was not suited to the tight time frame associated with completing a Master's thesis. For these reasons using the project as an educational or training tool was deemed out of scope.

2.1.1 User Interaction Recording

One of the key features of this project was the ability to record the users interaction with the program. These tools generally fall into two categories, screen recorders and test automation tools.

Fraps [15] is a commercial piece of software for the Windows platform. It provides both a benchmarking tool, and a screen imaging and recording tool. The benchmarking tool measures frame rate data about the program and has the ability to save this data to disk. The screen capture tool allows both image and video recordings of the running program. The screen captures are saved to disk as a video file and can be replayed in any standard video player. CamStudio [20] is a similar piece of software, but also in-

cludes a tool to edit the recorded video to add captions, custom cursors or picture-in-picture overlays. CamStudio is focused on allowing users to create professional training and support videos.

Xnee is an open-source test automation suite for the X11 windowing system. Xnee allows a user to record and playback a series of input events, including mouse and keyboard interactions. Xnee is designed to not just record interaction on a user interface but can also automate input for command line applications [67]. The Xnee test suite was also used to perform the automated testing during the performance evaluation of this thesis. AutoHotKey is a Windows-only alternative to Xnee. AutoHotKey focuses on the automation of mouse and keyboard macros, including the ability to create a script which can perform a series of interactions. The scripting tool also allows the generation of a script based on recorded user input [52]. AutoHotKey does not focus on test automation but has been used successfully to that effect [71].

Both of these categories generally include external tools that record the entire interaction with the system and not just the program itself. While this is often appropriate for test automation and training, it only provides a coarse level of information about the input performed. By integrating the input recording directly into the application itself, it is possible to gain a much finer view of the user's interaction. Sebatian et al. put forth an example where they captured the user input for a web application as a series of server requests. This allowed them to generate a large number of test cases which mimicked real user interactions [30]. Had they only recorded the mouse and keyboard input input the application it would have made it much more difficult to generalise the data. This is because they would have limited information about the applications state when the actions were performed.

The tool presented in this thesis aimed to provide an input recording system similar to that done by Sebatian et al, including the ability to replay the user's input back into the system. While Sebatian developed for a

web application, they also focused on recording the requests made by the program rather than the location and position of the input. The MTEE project did both, recording the input of the touch point and the requested gesture on a particular component.

2.1.2 User Interface Visualisations

This project required visualisations to help emphasise the components that were being manipulated during the playback of a recording. The goal was to make it clear which components where being used, and where the interaction was coming from. This was made more complex by using multi-touch as multiple input could be interacting with same component.

Software simulation tools such as CamStudio, mentioned above, and Adobe Captivate [7] have a number of features to help with the visualisation of software interaction. These can include; highlighting or changing the cursor to be more visible, text comments and captions to provide explanation, and adding highlighted boxes to indicate buttons or points of interaction. In the context of a training simulation, these techniques help indicate to the user where they need to interact, as well as helping to emphasise a single task within the entire application.

Other projects focus on modifying software to make it easier to use and clearer as to what kind of interactions are occurring. Ripples [78] is a system which enables visualisations around each contact point on a touch display. The system is engineered to be overlaid on top of existing applications, without requiring that the application be modified in any way. The visualisations included; indication of whether a control had been select by a touch point, a cursor trail when a touch point was dragged, a shrinking circle to indicate the exact position that was touched, and tethers to indicate which touch points are attached to which controls. Figure 2.1 demonstrates two of these effects.

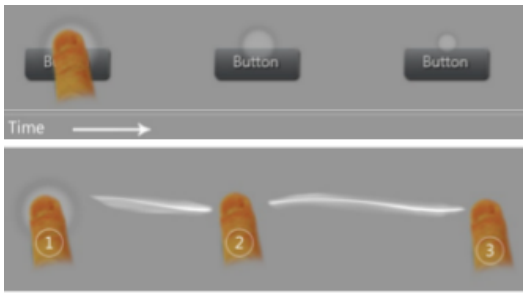


Figure 2.1: Two examples of the visualisation techniques used by the Ripples system.

2.1.3 Software Evolution

This project was not focused on maintaining editable versions of the users code, rather just a working snapshot of their program at a point in time. The program structure and other data associated with each snapshot could be used to compare the program as it evolved over time.

Version control systems such as Subversion [11] or Git [35] use a number of techniques for labelling and controlling versions of software as they are developed. One feature is called tagging which gives the developer the ability to label a particular version of their software. This makes it easier return to that version at a later date. This is often done for stable release versions as it gives the developer a way to check and compare the behaviour of that version against other released versions.

A number of other systems and projects interface with software version data to provide information and visualisations of the software's evolution. Systems such as Evolution Track Table [48], The Evolution Matrix [45] [46] and Evolution Storyboards [16] all provide different kinds of visualisations of this data. None of these tools allow an easy comparison of behaviour between different versions of the software.

2.1.4 Visualisations within a Development Environment

A number of tools provide integration of visualisations to existing IDEs. These can add more information in the form of tables and visualisations or change the entire layout of the display. The common goal is to better the developer's experience when using the tool so they can make easier decisions.

Code Canvas [25] alters the layout of Microsoft's Visual Studio to display all code on a single zoom-able window. Code fragments are instead displayed in forms which can be be moved around the canvas and organised into groups. This allows the user to use spatial recognition to better organise and recall pieces of code. Visualisations can be added in layers either over or under the code layer. Multiple layers can be shown at the same time to better combine and display different sets of data.

Code Bubbles [18] allows the users to places snippets of code into bubbles, similar to Code Canvas. These bubbles would not necessary contain the entire class, but snippets such as functions relating to the current task. Code Bubbles also allows the user to link and tag pieces of code together to provide more context as to how the code is used or to indicate the work flow of the application. A collections of these bubbles, tags and links can be saved into a working set to be recalled later.

Lintern et al [51] speaks of the benefits of integrating with an existing IDE, as they transform their SHriMP visualisations tool into a plugin for the Eclipse IDE. By integrating into Eclipse their project was able to gain easily access to the user's project and repository. This allowed them to focus on the visualisations rather then how they would collect the information about a user's project. Figure 2.2 shows that the SHriMP tool looks once it had been integrated into the Eclipse IDE. They conclude that when integrating a tool into an IDE it is important to decided what level of integration will be required, citing a piece of work by Amsden [8]. The five levels of integration are; none, invocation of external applications, data sharing, API interaction and dynamic user interface integration.

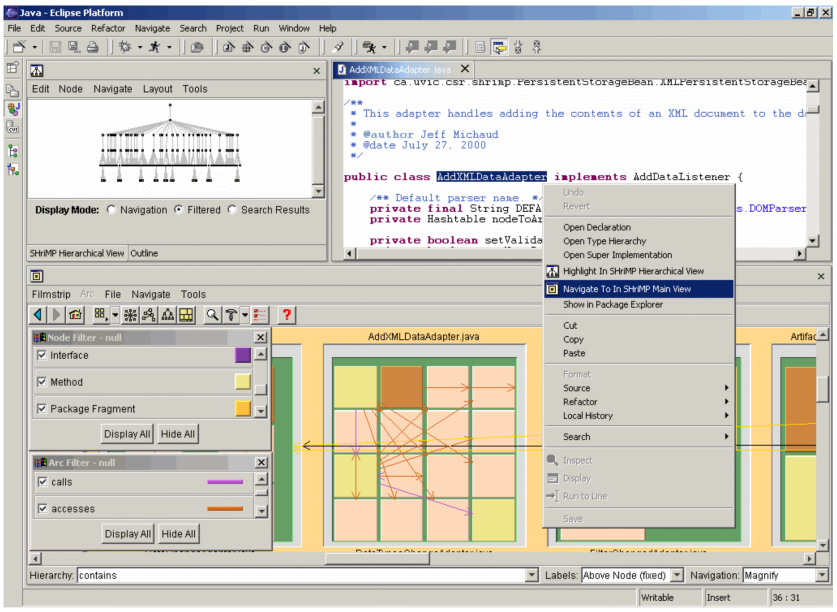


Figure 2.2: The outcome of integrating the Shrimp tool into the Eclipse IDE

2.2 Target Users Profiles

User profiles provide background information about the target users of the project. Information such as; what kind of qualifications a user would have and what their courses would have taught them are very useful for determining the skill set of the user. We can also explore the age, gender and ethnicity of the user and use this to gauge general interest popular with those groups and professions. This information is then used to produce a persona, a sample user against that we validate our design decisions. Personas are discussed more in the following section.

2.2.1 Student Profile

The student profile provided the basic background for a current university student. We look at what courses they were taking and what technology

the courses taught them about. This formed the basis for our primary persona in section 2.3. The student user was the person who must have the required skills to understand and use the tool and was expected to be able to benefit most from the project. The majority of this experience and background was pulled from students at Victoria University as it was the most readily available.

Background The Student User was an example of person who would have experience with computers, but not necessarily with developing for multi-touch input. The students were assumed to have finished up to three years of Computer Science or Software Engineering at a university. As an example, a student from Victoria University may have completed a Bachelor of Science in Computer Science, or a Bachelor of Engineering in Software Engineering. Completing these courses up to the third year of university will have given them a strong background in programming and software development. The focus was to look at how their background can extend their skills to developing with multi-touch interfaces.

Hardware Experience Smart phones and tablets were popular devices among the student population. Studies show that smart phone usage among college students have been on the rise in the last few years and has reached as much as 62% among some student groups [24]. This same study shows that tablet ownership has increased to 15%, with many students owning them along with a smartphone. Other studies have shown that students are evenly split between using Android and iOS devices [17].

While every student might not own a smartphone or other touch screen device, it was highly likely that would have come into contact with one. Because of this we assumed that the majority of students would have an understanding of the differences between touch screen applications and desktop applications. The main difference between them was the increased number of ways to interact with the touch interface when gestures

are avaliable. This also provided some common knowledge as to how multi-touch applications work.

Larger multi-touch devices, such as Microsoft Surface, or home built touch tables, were often not suitable for a whole class of students to work with at once. Teaching development for smaller devices is more common. There were number of courses at different universities offering programming courses which taught Android or iOS development [76] [53] [63]. These courses typically exposed students to learning how to design multi-touch and mobile sized interfaces, and also provided them with some experience about how the multi-touch systems work in each of their respective environments.

Software Experience Students from Victoria University had a very strong background in Java; as it was used to teach the basics of object-orientated program design in their first year. Other Universities used languages such as Python, C/C++ or C# in entry-level courses for the same reasons. If we assume that Universities favour languages that are popular in the software industry, we can see that these languages are all in the top ten most used world wide [73] [44].

At Victoria, students begin by using an introductory teaching environment for Java called BlueJ. BlueJ is designed specifically to teach object-orientation with Java and is popular amount universities [43]. After their first year, students, were encouraged to use the Eclipse IDE for their programming instead. Eclipse is a fully-featured integrated development environment (IDE) that can be extended with plugins. Courses also encouraged the use of plugins which include the JUnit Testing plugin, to unit test code; and the Subclipse plugin, to integrate the Subversion version control system into Eclipse. Students who had courses with Android development, such as those mentioned above, will likely be familiar with the Android Development Toolkit (ADT) plugin for Eclipse. These plugins taught students different ways to interact with a project beyond the

standard debug and editor screens.

Education By the end of their third year, a Victoria University student will have finished either a Computer Science degree, or have completed the core part of their Software Engineering Degree. In both cases the students will be facing their honours year when they continue their study.

The first year of these courses would teach the students Object-Orientated programming; in the case of Victoria University, a strong understanding of Java. The assignments for first year courses consisted of a program with a core algorithm or functionality missing which focused on what was taught in the week prior. While at the first year level, the incomplete part was abstracted from the rest of the program, but still exposes students to work within other peoples code. This helps them develop skills to review code and gain an understanding for how it works.

The second year continues in the style of the first, however students were also introduced to group work and work within IDEs such as Eclipse. The second year also put more emphasis on using tools such as Subversion, JUnit, and their respective Eclipse plugins. The assignments were in the same fashion as the first year however the programs were more complex. The example programs and problems they were faced with for each assignment started to involve computer graphics. This required students to draw and position objects on the screen. The second year also allowed students to start specialisation, some courses offering Android or iOS development experience.

The third year was when students were considered to be competent programmers, in both code and using various tool kits. The student's courses start to specialise, such as user interface design and designing interfaces for multi-touch. The assignments become larger and more complex, where the students are expected to start from the ground up. Students are taught different programming languages, often just to teach other concepts rather than a particular language. They are given the freedom

to use the language and tools of their choice. The User Interface courses are of particular note, as they start to teach the students how to link the behaviour of visible components to how their behaviour is defined on-screen.

Motivation Students that finish their third year of study had the option to do another year to gain honours, and was a requirement for the Bachelors of Engineering. The honours year contained a full year project, which could be on any topic related to computing. Multi-touch development is a particularly modern topic that had started to see more widespread use. This makes it an exciting and attractive area to complete a project in. An engineering honours project will not focus completely on how a system is implemented, rather what the system does. This motivates the students to use any tools available to them that would help them develop their project to its full potential.

Activities It was intended that students use this project to learn about multi-touch and better evaluate components and gesture behaviours. Activities include:

- Learn the difference in interaction between a normal mouse and keyboard interface, and a multi-touch interface.
- Understand how a user interface is structured.
- Understand how gesture behaviour is related to each component in the interface.
- Visualise which gestures are related to which components.
- Visualise data from interactions with the interface.

2.2.2 Professional Profile

The professional profile builds on from the student profile. In context of this project, the professional was someone who was already proficient in development for multi-touch input beyond the scope of what the project taught but would still find use in some of the features.

Background The professional user would have already moved into industry and worked with multi-touch devices. They may not necessarily have a university degree, but would have receive a certificate or diploma in computer programming. The multi-touch development industry had been around for a number of years, first becoming popular with the release of the Apple App Store in March 2008 [13], followed closely by the Google's Android Market in August of the same year [70]. As development for these devices was popular from the beginning, a professional developer working in this area could be assumed to have gained a number of years of experience working with either of these platforms.

Hardware Experience Much like the student profile, a professional would have a lot of experience using touch screen devices in their daily lives. Furthermore this experience would have given them a basic understanding of the differences between touch and mouse-based interfaces, such as the more numerous gestural ways to interact with a touch screen. As the professional user works in the multi-touch industry, it was expected that they would have been exposed to a variety of multi-touch platforms such as Windows Phone, Windows 8, Android, iOS, or other offerings from Blackberry and Nokia. Their experience could extend to larger sized products such as Microsoft Surface or custom multi-touch kits.

Software Experience Any particular company is likely to have a standard Language they use for all their projects. When developing for mobile,

this would mean using ObjectiveC for iOS or Java for Android. Companies tend to use tools that are well supported by the languages. This includes Visual Studio when developing a in a Microsoft Language, XCode when using ObjectiveC or Eclipse when developing with Java. A professional user would have better access to new and different technologies and is more likely to have experience developing for touch-centeric platforms such as iOS, Android or Windows 8; This could include open-source multi-touch frameworks such as MT4J and Kivy or commercial products like the Windows Surface APK.

Education While we do not assume a professional developer would have a degree, it is likely they would have some kind of qualification, such as a diploma or certificate in computing. We assume instead that if professional user has received a formal education it would look much like that of the student use. The added years of industry would strengthen this knowledge by providing more experience working with others and working with commercial programs. A professional user in the multi-touch industry would have more experience working with multi-touch interfaces and understand the differences and complexities top developing gestural behaviour.

Motivation A professional user would be asked to develop a prototype application or gesture behaviours. Users would be motivated to seek any tool which would help them tweak and develop these applications. They would also value tools which allowed them to quickly prototype and demonstrate different behaviour of user interfaces.

Activities A professional user could benefit from the project for use in activities such as:

- Profile a set of gestures, so they can be visualised.

- Record gesture movements so they can be replayed and compared with different types of behaviours.
- Compare different versions of an application to see differences in behaviour.

2.3 Target User Personas

A persona is a design tool used to emulate what a certain group of users would want and expect from an object being designed. They are fictional characters, based on research of real end-users. Personas are used to characterise the goals a group of users might have for the product, and for developing scenarios in which they would use the product. These scenarios consider who a user is and why they would use the tool in that way, based on the users background and experience. The user profiles from the previous section are used as background data to draw up two personas, a primary and secondary. These two personas represents the two user groups the project is targeted at. Two negative personas have also be drawn up to help define the limits of the project.

2.3.1 Primary Persona - Student

Name: Tom
Age: 20 years old
Location: Wellington, New Zealand

Tom is originally from Nelson; now living in Wellington and attending Victoria University full time. He is currently in his fourth year of a Bachelor of Engineering, and majors in Software Engineering. Tom has always enjoyed playing around with computers and likes to work with the latest technologies. He has an iPhone and an iPad and is familiar with multi-touch interfaces but has never tried to develop any applications himself.

Much of his free time is spent playing video games.

Tom has started his honours year, which requires him to complete a full year honours project and six other courses over the two semesters. He has decided that he would like to do a project on visualisations, using multi-touch to interact with them. His previous three years of study have given him a strong background in Java and Python, and he is a competent program with good project management skills. Tom has spent time working in groups, which helped improve his coding standards and experience working with other peoples code. The summer internship he has been involved with through a local company has helped to further these skills, and expose him to different development styles.

As Tom is focused on managing his time for his final year, he is looking for ways to quickly adapt to multi-touch application design. He does not want to be impeded while learning a new language or multi-touch framework, allowing more time to focus on the visualisation side of his project. Such optimisation would ideally allow him to quickly prototype his application and compare different behaviours for each gesture.

Persona Goals

Goal 1 Tom wants to be able to complete an honours year project, while able to manage the rest of his classes. Any tools he use cannot have a steep learning curve or be time consuming to use.

Goal 2 Tom wants to be able to interact with the internals and behaviour of his system without interference from any tools.

Goal 3 Tom wants any tools he uses to collect more more information than he would collect without.

Goal 4 Tom wants to be confident his program will continue to function without needing the tool.

Goal 5 Tom wants to maintain control of how his program is developing.

Context Scenarios

Scenario 1 Having completed a simple prototype of his user interface, Tom can import the profiling tool into his eclipse project. The tool is automatically setup and prompts Tom to switch the plugin view.

Scenario 2 Tom switches to the plugin view, the tool goes through a first run process to build the component tree for his project. Tom can then browse through the tree and see how components are nested and which gestures they are listening for.

Scenario 3 Tom right-clicks a component and clicks 'Preview'; bringing up a window containing a running version of his component, and all of the components children. As Tom interacts with the components, all of the gestures he is performing are logged.

Scenario 4 When Tom closes the window the logged gestures are saved to disk and displayed in a different panel in Eclipse. Tom can now select the saved performance and click 'Replay' to have his gestures with the program replayed on screen for him.

Scenario 5 Tom can also right-click a component and click 'View' to be taken to where that component is created. He repeat this action with a gesture for a component, and can be taken to where the gesture is initialised.

Scenario 6 When Tom selects a performance from the performance panel, the main panel in Eclipse will be filled with a wire-frame of the components, and display the pathing lines the gestures followed. This gives Tom an overview of how the component was used and where the behaviour could be changed.

2.3.2 Secondary Persona - Professional

Name: Harold
Age: 23 years old
Location: Wellington, New Zealand

Harold is a 23 year old software developer who works in Wellington. He is a former Victoria University student, that graduated with a Bachelor of Science, majored in Computer Science. He has been working in industry for 2 years, since he graduated at age 21.

Harold is part of the research and development team for a company that specialises in multi-touch applications for both iPhone and Android devices. He previously worked in one of the core development teams but was recently given the opportunity to change departments, he decided he would like the change of pace. He has been assigned to work on prototype developed for sets of common behaviours and gestures they can use across the companies applications.

Harold is looking for an easier way to prototype applications. While he has experience with iPhone and Android development frameworks, he also has the time to learn a new framework if necessary. Any tools he uses will need to be able to catalogue and record the interactions he makes with his prototypes so that he can analyse the data, and investigate the ease of use for each set of behaviours.

Persona Goals

Goal 1 Harold needs to be able to compare the performance of different types of gestures.

Goal 2 Harold needs to be able to show off the different sets of behaviour.

Goal 3 Harold needs to be feel confident using the tool, as he works in a professional environment.

Context Scenarios

Scenario 1 Once completed a prototype behaviour set in an Eclipse project, Harold can import the profiling tool into his project. The tool is automatically setup and prompts Tom to switch the plugin view.

Scenario 2 Harold can choose one of his components and select 'Preview' to play around with that component. His actions are recorded by the profiling tool.

Scenario 3 Harold is then able to select the recording of the preview he just created and select 'Information' to view statistics about how he performed each gesture.

Scenario 4 Harold's boss wants to see the different sets of behaviour Harold has created prototypes for. Harold selects his performance and clicks 'Replay'. This launches the earlier version of his program, allowing Harold to demonstrate the previous version of his gesture behaviour.

Scenario 5 Harold selects multiple performances, which overlays and colour-codes the wire frame and gesture movements so the gesture performances can be directly compared.

2.3.3 Negative Persona - Power User

Name: John
Age: 27 years old
Location: Auckland, New Zealand

John has been working Auckland for 6 years, for a software company that specialises in touch table technology. John loves his iPhone and iPad,

and frequently develops applications for it. He publishes these applications on the Apple store for free, so that others can get benefit from them.

John is experienced in development with many different multi-touch open source frameworks such as Kivy and MT4J, as well as commercial platforms such as Android and iOS. He is part of a large development team, that works on large scale applications for a touch input tables. The application uses MT4J to provide the multi-touch interface and has been in development for 4 years. John creates new features for the application, and knows all the ins and outs of both Java and MT4J.

John is interested in performance tests and reports on the features of the applications. As there are many features to be retested after each update, he prefers for much of this to be automated.

2.3.4 Negative Persona - Beginner Programmer

Name: Mary
Age: 18 years old
Location: Wellington, New Zealand

Mary has just moved to Wellington and has started a Bachelors of Engineering, and majors in Software Engineering at Victoria University. She has little experience with any programming languages; she has recently learned the basics of HTML and some Javascript in college. Mary is interested in using novel technologies such as multi-touch screens and tables and is eager to start development for them. She intends to use this opportunity to build a foundation of knowledge; Mary hopes to find a tool that helps her add and edit code for her projects as well as furthering her education.

2.4 Project Requirements

By distilling down the persona goals and scenarios we were able to draw up the formal requirements for the project. While the features of the project were still measured against each personas goals, it was important to consider the overlapping requirements. This resulted in the development of the functional and non-functional requirements listed below.

2.4.1 Functional Requirements

These requirements were developed based on the personas, goals and scenarios already put forward by this chapter. The purpose was to summarise the requirements so they could be more easily measured in each section. The square brackets at the end of each requirement indicate which part of the analysis they have been drawn from.

Requirement 1 The tool must not require extensive training to use and the user should be able to pick up and use the tool with little documentation. Usage of the tool can rely on expected experience for the target framework or for any external tools used. **[Tom, Goal 1; Mary]**

Requirement 2 The setup of the tool should be automated where possible, any actions required by the user should not require knowledge of the tools architecture. **[Tom, Scenario 1 & 2; Harold, Scenario 1]**

Requirement 3 The tool should not prevent the user from accessing and using their original project. The user must retain the ability to edit and run their own program even when the tool is installed in their application. **[Tom, Goal 2, 4 & 5; Harold, Goal 3]**

Requirement 4 The tool must have the ability to preview individual components to better focus the users attention. **[Tom, Goal 3, Scenario 3; Harold, Scenario 2]**

Requirement 5 The tool must gather and display information about a user's program. The information can either be static or dynamically generated. The information must be easy to understand by the developer. **[Tom, Goal 3, Scenario 4 & 6; Harold, Goal 1]**

Requirement 6 The tool must have the ability to use recorded information about the program to create a replay of programs behaviour. **[Tom, Scenario 5; Harold, Goal 3, Scenario 4]**

Requirement 7 The tool must have the ability to compare the information between different recordings. **[Harold, Scenario 5]**

2.4.2 Non-functional Requirements

Non-functional requirements are those that can be used to judge the performance of the project, rather than how the program will behave with respect to the user's functional requirements. This section outlines the non-functional requirements below.

Requirement 1 The performance of the user's application cannot be hindered by the tool produced by this project. There should be no noticeable difference for the user running the program with and without the tool.

Requirement 2 The tool should scale to large applications with more complex program structures without suffering from noticeable performance slow-downs.

Requirement 3 The tool needs to ensure that data both imported and exported is validated and will avoid or detect corruption.

Requirement 4 The tool must ensure that the data that it stores does not use an unreasonable amount of disk space, relative to the user's program.

Requirement 5 The tool needs to import and export data and respond to user input in a reasonable manner, such that the user will not be waiting for information to be loaded or processed.

Chapter 3

Design

This chapter discusses the tools required to use the project in section 3.1, the features of the project in section 3.2, design trade-offs in section 3.3, and alternatives designs that were considered in section 3.4. The covered background and personas are used as a foundation for the project design and how it aims to meet the user's goals and requirements.

3.1 Tools and Language Choice

The Multi-Touch Explorer Environment (MTEE) was designed to aid in the development of multi-touch applications. The intention was for the project to become an additional tool in the user's development environment. This required any product to work along side the user's existing tools. A number of languages and tool-kits were considered, however due to the user personas and background analysis, in Chapter 2, it became clear that the Java programming language was the best choice. Both personas, Tom and Harold had experience with Java before, and Tom currently uses it at University. This decision naturally lead to choosing the Eclipse platform as a base for the project. Finally a multi-touch framework was selected; out of the few available for java, MT4J was chosen due to its maturity and due to familiarity on the projects behalf. The alternatives

that we considered for this project are discussed in section 3.4.

The sections that follow consider these choices in more detail; provide background on each of the choices and further justify their selection. It should be clarified that while the user will be developing programs that are multi-touch capable, neither the tools used to create these programs, nor the proposed MTEE will require the use of multi-touch to operate.

3.1.1 Java Language

The Java programming language is a general-purpose, concurrent, class-based, object-oriented, programming language, specifically designed to have as few implementation dependencies as possible. The Java language runs on top of a virtualization environment called the Java Virtual Machine (JVM). This allows the same code to be written for one platform and executed in the same manner on other platforms; provided there exists a JVM implementation for that platform. This makes the language highly portable, as only the JVM has to be ported to new platforms, instead of every individual program. The Java language is related to the C and C++ languages, as they share similar styles and syntax. The focus for Java is to provide high level functions which are useful for commercial development and avoid more 'experimental' features which can cause unavoidable bugs [6]. Java is free to use and currently maintained by Oracle. Both the development kit and the runtime environment for the latest version, Java 7u13, along with previous versions can be downloaded from Oracle's website [62].

Due to both its portability and design, Java is a popular language for teaching object-orientated paradigms at universities and other institutes. Its freedom and support for all major platforms allows users to download and install the language on their personal machines. The language is easy to get started with and can help teach the basics of program design. Victoria University uses the language in the majority of its first and second

year courses. Java is a language Tom and Harold are familiar with, as both have graduated from Victoria University. Java's similarity to other popular languages, such C++ and C#, means that even experienced programmers who may have never used Java are able to pick-up and learn the language quickly.

Java is often criticised for being slower to execute code, as it uses Just-In-Time compiling rather than compiling code ahead of time. These complaints are mostly unfounded with benchmarks showing that the latest version of Java is faster than most modern languages [1] [69] [4]. Choosing a high performance language is important to meeting the performance goals of the non-functional requirements, particularly requirements 1 and 2. For this project believed that Java would be able to meet these requirements.

3.1.2 Tools

Along with the Java programming language, the Multi-Touch Explorer Environment (MTEE) required that the user had two external tools. These were the Eclipse Integrated Developer Environment (IDE) and the Multi-Touch for Java (MT4J) framework. The Eclipse IDE was used as a basis for development with MT4J, and the MTEE integrates into Eclipse through its plugin platform. It should be noted that when 'Eclipse' is referenced, it is the development tool and not the Eclipse Foundation that is being discussed, unless otherwise noted.

Eclipse IDE Eclipse is a widely distributed and used development tool from the Eclipse Foundation. Started by IBM, the open-source Eclipse project has been in development for over 10 years, which has resulted in stable and mature software. It provides a complete development environment, for writing, running and debugging code. While it is often associated purely with Java usage, it has an extensive and mature plugin

Application Programming Interface (API) and is built on the Eclipse Foundation's Remote Applications Platform (RAP). This has allowed the development environment to be expanded across a large group of languages and envelop other development tools; such as unit testing and version control systems [27]. As Eclipse itself is built using the Java programming language, it makes it an ideal companion for Java developers as anywhere their program will run then Eclipse will too.

The maturity and feature set of the software made it an attractive option for universities and other institutions as software to teach their students with. The open-sourced nature of Eclipse meant universities could save on license fees and provide it to their students to use on their personal machines. Victoria University encourages its students to use the software from their second level programming courses. It was also provided on all machines in the Engineering and Computer Science laboratories. This meant Tom, a final year student at Victoria, would have a in depth understanding of how the Eclipse platform worked and was one of the reason for choosing Eclipse.

The after mentioned features make Eclipse an ideal platform for commercial developers. Even developers working with languages other than Java can still use Eclipse as a number of different versions exist for different target users [29]. Free plugins for Eclipse can provide support for other common languages such as Python [5] or C# [49]. Furthermore, the Eclipse Foundation provides instructions on how to add support for custom languages, allowing for companies to develop editors for in-house languages. [14]

Harold, the professional developer, is familiar with the official Android Development Kit (ADK). The ADK provides a version of Eclipse with the Android Developer Tools (ADT) plugin already pre-installed [36]. This would make Harold familiar with using Eclipse and how plugins can change the layout and add additional tools to the Eclipse workspace.

The choice of development environment informs us of two thing; what

features and information could be provided up-front to the user, and how much control over the setup and use of the project the user would be required to do. This decision resulted in splitting the project into two distinct segments, the Eclipse plugin and the core library. These are discussed in detail in 3.2.1.

Multi-Touch for Java Multi-Touch for Java (MT4J) is a Java framework to develop fully multi-touch applications in the Java programming language. Much like the Java programming language MT4J is cross platform and supports a large range of input devices, however as it uses external system libraries it is not guaranteed to work on all platforms. MT4J uses the Processing library to provide visuals and graphics to its program, this supports both 2D and 3D. It comes with a range of pre-programmed standard gestures, such as scaling, rotating and dragging. This makes it ideal for quickly developing multi-touch applications and meshes well with the goals of this project [60] [47].

The MT4J developers encourage the use of the framework with Eclipse by the provision the source code for MT4J including working Eclipse projects [59]. Two projects are provided, MT4J-Core and MT4J-Desktop. MT4J-Core only provides the basic functionality of the framework and is intended to be used as a base for large scale programs. MT4J-Desktop expands on the core project that provides more functionality, as well as examples on how to use them [58]. These cover both basic and advanced usage of the MT4J framework. The user can import these projects into an Eclipse workspace and play around with the examples that are provided.

An MT4J program consists of one or more scenes. A scene consists of groups of nested components, each component can be registered to accept any number of gestures with both standard and custom behaviour. In a scene the components are inherently ordered by what is called the Z-order. This determines the order components are drawn to the screen and thus which component will receive the touch input for a given point. MT4J

fully supports multi-touch, allowing any number of touch points to be feed into an MT4J program at once.

Neither Harold nor Tom were familiar with the MT4J framework. Harold had development experience with multi-touch because he had developed for both Android and iOS; however their prior knowledge is a moot point as neither of them had any experience in development with the alternative Java multi-touch frameworks discussed in section 3.4.2. This means that the choice of framework was better decided by considering how quickly a user could learn it, and the ability to integrate with the framework for collecting data. Learning the framework was perhaps the biggest cost of using the MTEE with Eclipse. we had to ensure that the Functional Requirement 1 was not violated by the choice of framework, along with the tool be mature and stable enough that it could meet Harold's Goal 3. As Tom intended to learn a multi-touch framework for his honours year he was more likely to put the time in, regardless of which framework was chosen.

MT4J was partly chosen because it shared a design goal with the MTEE project, the rapid development of prototypes and applications. This allowed both projects to complement each other; as MT4J provided a fast way to setup a basic program with a multi-touch interface, and MTEE provided a tool to monitor and profile the behaviour of gestures and interaction with that interface. MT4J was also fully written in Java and integrated well with the Eclipse platform. It could be imported into a project as a single file or its project can be included as a dependency. This was both useful for debugging as the source can be referenced, and for development as the built-in examples provided numerous starting points for a project. Finally, I had an existing experience of development with MT4J. This was beneficial when interacting with the internals of an MT4J project, such as accessing and profiling the input and gesture events.

3.2 Project Features

The development of the MTEE project consisted of two parts, the Eclipse plugin and the Multi-Touch Explorer Analysis (MTEA) tool. The Eclipse plugin provided access to the MTEA tool by automating the importation and launch of tool within a user project. The plugin also displayed the information that was gathered by the MTEA tool from the user's program. The MTEA tool itself was the core part of the project; It provided the means to extract the information from the user's program and the ability to replay that information back.

This section investigates the final design features of the project; discusses why they are important and how the came to be selected. Section 3.2.1 discusses the features of the Eclipse plugin, what they do and why the are need. Section 3.2.2 then examines the MTEA tool.

It should be noted that the MTEE (the Multi-Touch Explorer Environment) refers to the project as a whole, where as MTEA (Multi-Touch Explorer Analysis) refers to the tool which collects and processes the information.

3.2.1 Eclipse Plugin

The development of an appropriate project using the MT4J framework and the installation of the MTEE plugin were considered out of scope, as changing these features was entirely out of the control of the project. We assumed that the user had the plugin installed and that they had an active MT4J project in their workspace.

This allowed us to focus on setting up the Eclipse workspace for the MTEE; this included importing the MTEA tool and enabling either the MTEE perspective or associated views.

The layout of the views within Eclipse was important for how the user would first see the tool. Both standard Eclipse views and the two new views needed to be laid out in such a manner that they could all be easily

accessed. Figure 3.1 shows the layout that was planned. The size and position of each of the views was designed to mimic the size of existing views, this was planned to not alter a similar layout to what the user would have been used too and relates to meeting Functional Requirement 1.

MTEE Import Wizard A number of steps had to be planned in order to import and setup the MTEE within an MT4J project. These were handled automatically by a wizard and had been designed to cover everything that is required to run the MTEA tool. Each of these steps are outlined in order below:

1. Creation of the MTEE package within the user project. This was placed in the project source folder and name 'mtexplorer'. The package provided a location to save the start file for the MTEA tool. By creating a separate package it made it easy to remove or ignore the tool so that it would not get in the way of the rest of the users project.

2. Creation of a base folder to store snapshots and recorded data. This was created in the root of the users project and named 'mte'. Like the package folder, this provided a way to hide the data from the user and keep it away from their existing setup.

3. The MTEA library was imported and added to the projects build path. Only a link to the library was imported so it would be updated if the plugin is also updated.

4. The start file for the MTEA tool was created. This was based on the start file for the user's program, ensuring any modifications by the user were carried into the MTEA tool. This also meant the user could run their original start file at any time, without the MTEA tool being active.

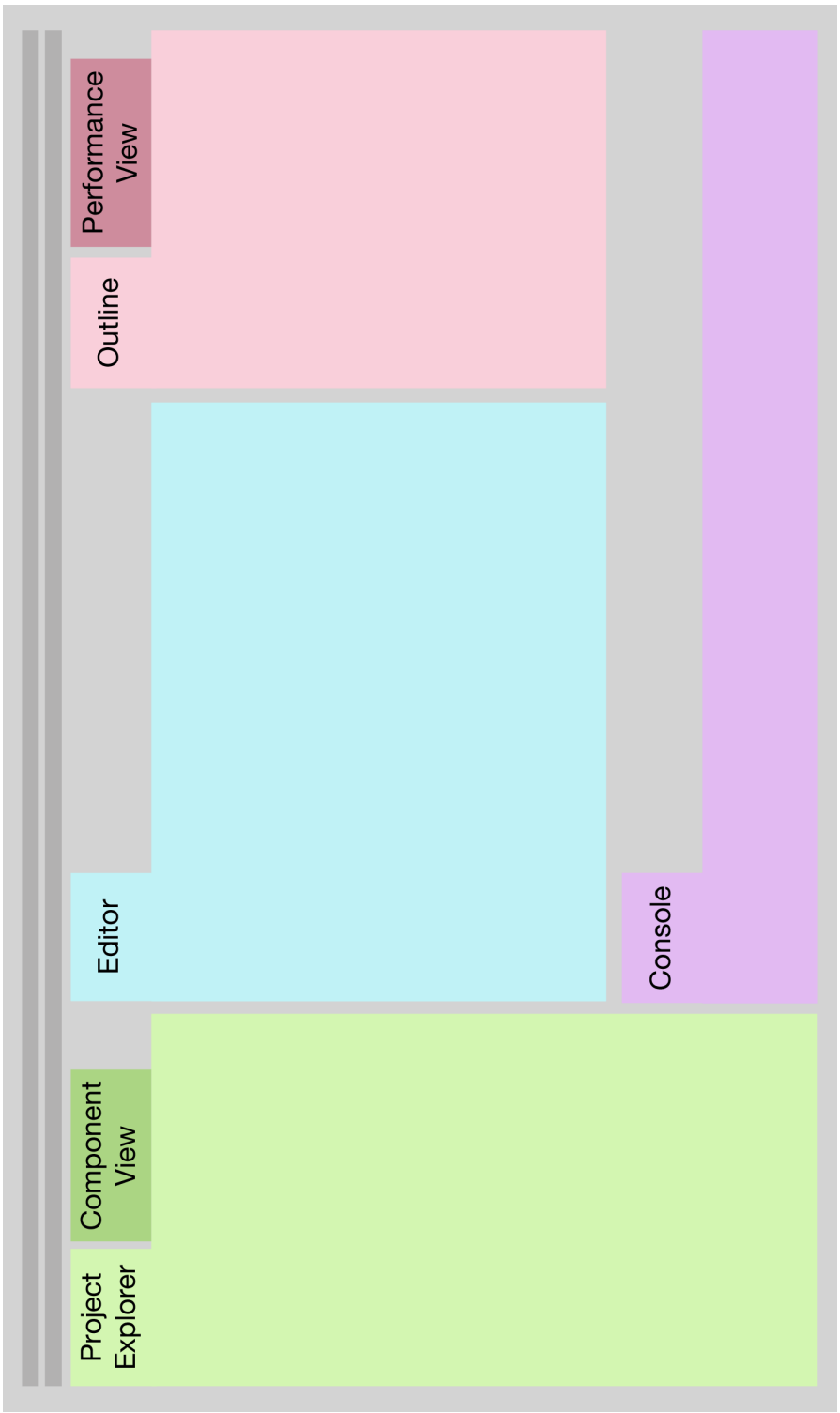


Figure 3.1: The Eclipse workspace layout.

5. The XML schema file for saved data was copied in from the MTEE plugin to the base folder. The schema was used to validate the input and output of the of the XML data from the MTEA tool. The schema was built to the XML specifications [19].

6. A build file was created for the project. This was used to create a JAR file of the current version of the users project. The ANT build file is also saved in the base folder.

7. The build file is executed to create the snapshot of the user's project. This file was named 'latest.jar' and saved in the base folder. It was used by the recording function to save a snapshot of the program at the time of the recording.

8. The new start file for the MTEA tool was run to generate the initial explorer XML file. This was saved into the base 'mte' folder. The file was used to populate the MTE Component View described below. A special argument was used during this run so that the program would exit as soon as the explorer file has been created and exported.

The wizard was designed to handle as much of the setup as automatically as possible. The only input required from the user was to select the correct start file for their MT4J project. The wizard served to lower the barrier of entry to using both the MTEE and MTEA tool and hide unimportant details of the tools design the user shouldn't be concerned about. These were aimed to directly meet Functional Requirements 1 and 2.

Having a separate folder and package to store the data within a users project provided only three points that need to be deleted to remove MTEE from a project. These are the MTE source package, the base 'mte' folder and the imported library. Removing these would return a project back to its original state. This would satisfy Tom's goal 4 and also aid Tom's goal 2 by keeping the data away from the rest of the project. Both Tom and start their process with existing projects in Eclipse. As both favour speed

of development, neither want to have to start a whole new project to gain use of the tool. By allowing them to import the tool straight into their existing projects it saves them both time and hassle.

Multi-Touch Explorer Perspective Eclipse uses a system called perspectives to allow the user to easily switch between different tools and change the usage of the workspace. An example is the standard Java perspective, which contains the project explorer, class outline and the Java editor views. If the user were to switch to the Debugging perspective, these views would change to include a JVM task manager, a list of breakpoints and other debugging controls. The user is free to switch between these perspectives or any others at any time.

The MTEE project provided its own perspective. This was based on the existing Java editor perspective, adding two new views, the MTE Component View and the MTE Performance View. These views were populated by files created by the MTEA tool so would have been blank when the user first switched to the MTEE perspective. This was especially true if they have not yet run the MTEA tool import wizard. It added an option to the existing 'New' context menu called 'MTE Project'. This option started the library import process as described above.

As perspectives are a core part of the work flow of Eclipse, it was felt that both Tom and Harold would have little trouble adding and using the new perspective. By effectively placing the two new views into the original Java perspective, it would have little impact of either of their work flows. Indeed, a user also had the option to import the views separately and place them in to their current setup as they wished. The perspective can also be accessed from the 'Other' context menu, in the browse window it is listed under 'Other' then 'MTExplorer'. Figure 3.2 shows the standard method of accessing different perspectives.

By integrating the tool into Eclipse and making use of common platform functions such as perspectives, we are able to validate Functional

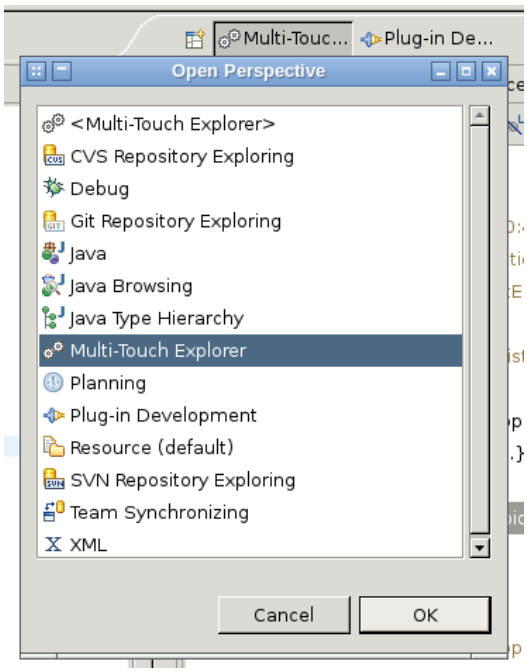


Figure 3.2: This figure displays the perspective selection dialog box in Eclipse, the Multi-Touch Explorer perspective is selected.

Requirement 3. This is because the Eclipse IDE is familiar to both Tom and Harold, and it would allow them to continue to work on there project with out having to run an external tools.

Much like perspectives, views are a core part of the work flow when using Eclipse. The two provided views were designed to display two different sets of information and controls. The views worked like any other piece of the Eclipse workspace, and required little to no setup. The information they display was drawn from data files in the base folder. These are the MTE Component and MTE Performance views.

Component View The component view showed a tree structure of all of the components in each scene. The tree structure consisted of nodes that could be expanded out. It allowed the user to drill down into the structure

of the scenes they had created and get a sense of how each component was nested inside each other. Each component listed its component ID and name, if it was given one, otherwise a default was used. The gestures (such as Drag, Scale or Rotate) that each component were registered to receive could be seen when a component was expanded.

By selecting a component the user was able to access a context menu which gave them the option to launch that component in an isolated mode within their program. This mode was called the 'Preview' mode, as it allowed you preview different components and scenes. This would highlight only that component and disable the rest. If they user instead selected a scene node and then selected the isolate option, they would have been able to launch the entire scene. In both cases the MTEA tool was running in the background with its full functionality available. This functionality was described in section 3.2.2.

Both Tom and Harold benefited from being able to explore and expand a scene tree of the current scene they were working on. It was a good point of reference when thinking about their programs structure, and making sure their program was laid out how they expected. This helped to meet Tom's Goal 3 by providing him with more information that he would have otherwise. It also provided access to the functionality required to meet Functional Requirement 4. As Tom was a not familiar with the development of MT4J project, the component view was able to help him uncover some of the inner workings of each component. The view was also very simple to use, so user do not have to spend much time interacting with it.

This view provided the starting point for the MTEA tool by allowing the user to starting making recordings. The recorded information would then be displayed in the performance view.

Performance View The Performance View was similar to the Component View, in that it used the same tree structure and a context menu for control. The Performance View listed all of the recorded performances

for a project. The performances were named and ordered by a date time stamp. Figure 3.3 shows the layout and positioning of the new MTEE views in Eclipse.

As each performance only worked for a particular version of the original project, the performance could be expanded to display a component tree of the program structure at the time of that performance. This was used to drill down and explore the structure, as with the Component View. This included the same information as the other view, but also provides information for the gestures that were performed on each component during the previously recorded session. Gesture data included information such as how far a component was scaled or rotated.

The context menu provided for this view had three options, preview, replay and delete. Delete was self-explanatory, allowing the user to delete an existing performance. The preview option was similar to the Component View action, however rather than displaying scenes or isolating components from the current program, it instead displayed them from a snapshot of the program taken when the performance was first recorded. The snapshot could then be used by the replay action. The replay action played back the recorded events into the snapshot of the program, providing a playback of the input events as they were originally performed by the user.

Like with the Component View, both Tom and Harold benefited from being able to explore a scene tree of prior versions of their program. This allowed them to compare and contrast the evolution of their program structure over time. Both the data given about the recorded gestures of each component, and the actually ability to playback a prior interaction with the program helped to satisfy Functional Requirement 5 and 6. It also gave Harold the ability to satisfy Functional Requirement 7 by either replaying or previewing two different performances to compare there sets of behaviour.

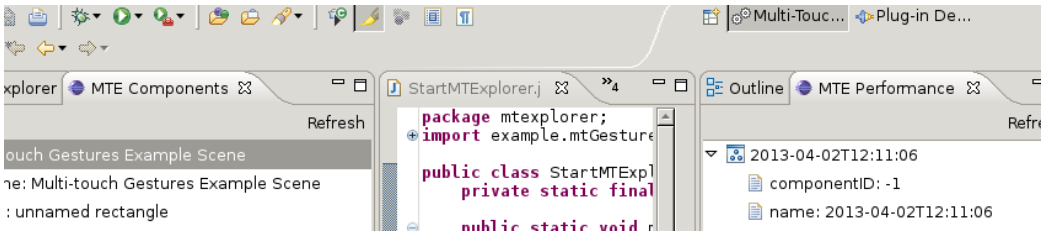


Figure 3.3: This figure shows the layout of the MTE views for the MTE perspective. The MT Components view is on the right with the MTE Performances view on the right.

3.2.2 Multi-Touch Explorer Analysis Tool

The Multi-Touch Explorer Analysis (MTEA) tool provided the main functionality for the project. It was responsible for exporting of the data for both the component and performance views, and for operating the commands that could be executed from these views. As the Multi-Touch Explorer Environment (MTEE) includes both the Eclipse plugin and the MTEA tool, it necessitated a means of communication between the two elements. This required that the MTEA tool included two sets of functionality, the user functionality and the system functionality. These are discussed in sections 3.2.3 and 3.2.4 below.

3.2.3 User Functionality

The user functionality consisted of all the front-facing features that directly addressed the user's goals and requirements. In the case of the MTEA tool this included, Component and Scene Isolation, Performance Isolation, Performance Recording and Performance Playback. All of these features are accessed through the views provided with the MTEE Eclipse plugin, however they were executed by the MTEA tool running on top of the user's program.

Component and Scene Isolation The isolation or 'Preview' mode allowed the user to record the input data they made as they used the tool. This data was then saved as a performance and displayed in the Performance View. This mode was accessed by selecting the 'Preview' option from the context menu from either the Component or Performance View, however performance data could only be exported when running from the Component View. The isolation mode could be run on both a component or scene from the Component View window. The distinction between component and scene changed the way the display was setup by the MTEA tool.

If a scene was selected, then the entire scene was shown unaltered. This was useful for skipping to a particular scene in a project, rather than navigating to it in the application. The input data would still be recorded and could be exported; saving both input data and gesture data for all components that were manipulated.

If instead a component was selected from the Component View then the MTEA library set up the recording function, then did three things. It first disabled all of the input processes for every component expect the isolated one. Secondly it faded out all of the disabled components, they instead appear transparent. Finally, it coloured the isolated component with a transparent overlay. This served to highlight where on the screen the active component was. If a component with child components was selected, then the children will also remain active, however only the parent component will have a colour overlay. Figure 3.4 showed an example of this behaviour, where the isolated component was highlighted red.

The isolation and recording features solved a number of problems. The recording function helps to satisfy Functional Requirement 5; by recording Harold or Tom's input for a certain action, they can then use the Performance Playback or Performance Isolation features to review the behaviour of a gesture and compare it to other recordings he has made of previous gesture behaviour. These features are discussed below.

Figure 3.4: This figure shows an example of an isolated component. The component itself is highlight while the disabled components are faded out.

The isolation feature could be used to help look at the focused behaviour of a particular component and its children. By disabling all but the isolated component, the user it free to interact with just that component. This helps to remove noise from a recording or sort out why a certain behaviour might be occurring for a component. This partly satisfies Tom's goal 3, by providing him an avenue to better debug and test the components he is building. This also helps him resolve issue around problems such as correct z-ordering, as discussed under MT4J in section 3.1.2.

Performance Isolation The component and scene isolation could also be executed from the Performance View. The same distinction between selecting a component and a scene from the view for isolation applied. As the Performance View also contained performance and gesture data, the isolation command would search up or down the tree to find the closest

component or scene in the tree view. While the command was executed in the same manner, it did not run the application from the latest version but instead used the snapshot file which was create at the same time as the performance. This meant the component or scene being isolated may have differed from how the user's current version performed. This was the intended behaviour, allowing the user to record different behaviour for components, and return to then at a later date. When running from the Performance View the user was not be able to save new recordings.

As with the component isolation, performance isolation provides a quick way to review a particular scene or component. By using snapshots of the program over time it allows people like Harold, to experiment and demo a number of different behaviours. This helps to validate Functional Requirement 7.

Performance Recording When an application was run from the Component View, the scene would be setup to record both the input events and the gesture data sent to each individual component. The recording began as soon as the program was started and would continue until the program was closed. The intention was that the user would start the application, perform the desired actions they wanted, then export the data and close the program. The data would only exported when the user selected the save option and would otherwise be discarded. Once a performance has been exported, it would show up under the Performance View.

The input data was used by the Performance Playback feature to replay the series of actions the user performed. The gesture data was provided as feedback information for the user. When a gesture is recorded it was feed through a filter which determined what type of gesture it was and pulled interesting or relevant data from this to export when the recording was saved.

Performance Playback The Performance Playback feature could only be run from the Performance View, as it required the data from an existing performance. As with the Performance Isolation feature, the performance playback used snapshots of the program from the time when the performance was recorded. This ensured that when the input data was feed back into the program it would play out and behave exactly as was originally recorded.

The Performance Playback modified the display of the program to better emphasis what was happening. All components started out at half transparency, then when a component received input it returned to full transparency. A component would also receive a coloured overlay so it could be distinguished from other components that were being used. This was designed to clearly show which components had been used in the replay. Colouring each component a unique colour served to help the users distinguish components from each other.

A cursor trail was also drawn to the screen. This displayed a drawing of a coloured dot at each touch point, showing the path the cursor took. When interacting with a component the dot would be coloured the same as the highlighted component it was interacting with. This provided context to the cursor trail so that the user was able to track the path of components across the screen. Figure 3.5 shows the effect this had on the program display.

The playback data was imported into the MTEA tool when it was first run, but playback would not begin until the user selected it to. As only the input data is feed back into the program the position of the components are not known at any particular time. This limited the tool to only playing from the beginning of the performance and not part way through. This also meant the playback can not be played backwards or from an arbitrary point. To restart the playback the user would have to close the program and relaunch it.

Once the playback was complete the user was free to interact with the

component as they wished. They would gain the benefit of highlighting components as they interact with them, and be able to make new cursor trails which matched the colours of the component. These overlay measures provided a number of benefits in seeing how components received input and handled gestures. The contributes to Functional Requirement 5 by changing the way the information is displayed to the user.

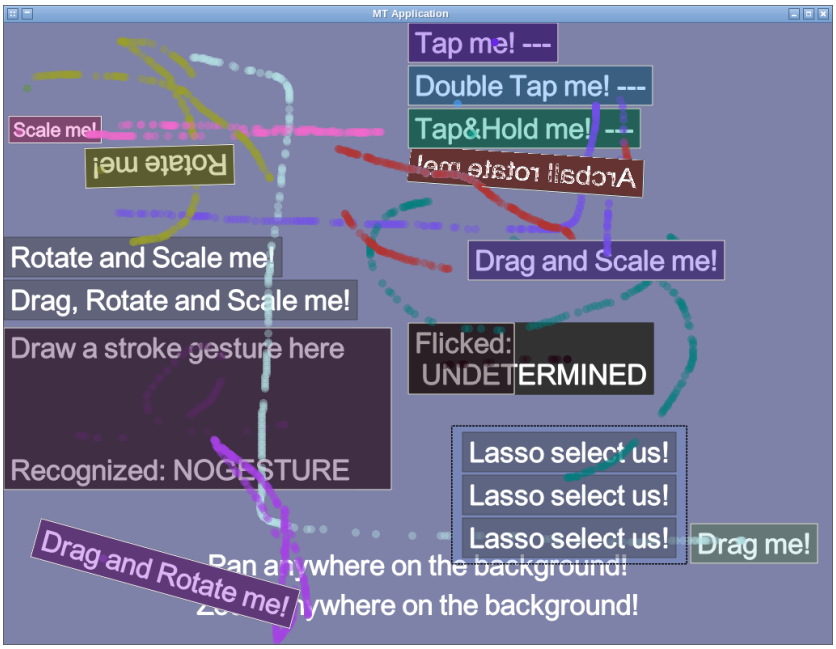


Figure 3.5: This figure shows an example of a performance playback with the MTEA tool. The cursor trails relate to the component which is highlighted the same colour.

3.2.4 System Functionality

Due to the nature of the design, extra system features were required to allow the project to work together. These to not directly address the user's goals, however they were necessary for the success of the project. The system functionality includes; operating the MTEA tool through the use

of runtime arguments, generating and exporting a reliable set of data on component data and input records, and importing previously exported data for use with the performance playback. This section discusses the design of these systems and which internal problems they solve.

Runtime Arguments As with any program that runs through Eclipse, it needs to be run as an external application. This meant Eclipse could not communicate with the MTEA tool directly. This presented a problem as we needed a way for the commands the user selected in the Eclipse plugin to be reflected when running the MTEA tool. Experimental attempts were made to use the Eclipse debugging engine to gain access to the program, but this proved to be tricky and over-complicated. The eventual solution was to setup a number of command line flags to provide basic interaction with a program running the MTE tool. These could be altered before runtime by the Eclipse plugin, and then feed in by Eclipse when it ran the application.

These runtime flags are defined as follows:

-s [Scene Name] This is the name of the scene that should be loaded. This should be specified with the isolate command so that the component is found in the correct scene. If this flag is not set then the currently active scene is used. The current scene is determined by the users program.

-i [Component ID] This command tells the program which component to isolate.

-p [Performance Name] The performance name is the time stamp given to a performance when it is recorded. The purpose of this flag is to specify which performance should be loaded for playback.

-X Runs the program and then exits. The purpose of this flag is to run the component tree export part of the MTEA tool, export the information, then close the file. This is useful for automatically running an

updating the component tree. This flag causes all of the other flags to be ignored.

The design and setup of the command-line flags allowed the MTEA tool to be run with out human interaction to get it started. This helped meet Functional Requirement 2, as running the tool in the correct mode would require knowledge of the projects architecture.

Exporting and Importing Data Much like the communication gap between Eclipse and the MTEA tool, there was also a gap in reverse. Passing information out of the MTEA tool, so that it can be consumed by Eclipse, required a robust method of storing and validating the data. To solve this the XML data format was used, with a schema file for data validation. The schema allowed strict XML data types to be defined that must contain certain attributes and could only contain data of certain types and in certain orders. This allowed the tool to respect the order of the component tree, while adding extra data such as component gesture listeners. The schema was then used both when exporting data and also when importing data for the performance playback. By using the schema we ensured our data was properly validated and would detect corruption as was required by Non-Functional Requirement 3.

The exported data files can then be consumed by Eclipse and displayed as the Component and Performance views.

3.3 Design Trade-Offs

While the design had been initially planned to include Java, Eclipse and MT4J, when the time came to actually implement these features a number of trade-offs had to be made. This was to leverage what was possible in a reasonable amount of time against the proposed features. While the core feature set was been maintained, changes had to made to better meet the

functional and non-functional requirements. This section discusses these changes.

3.3.1 Plugin Environment Setup

The setup of the MTEA tool in Eclipse had originally planned to be fully automated, only requiring the user to activate it. The idea was that the user would install the plugin, and then be asked which MT4J project they would like to import the tool into. Once the import process was completed they would be prompted to switch to the MTEE perspective and could begin using the tool. When implementing this feature it was found that a developer has little control over what happens while a plugin is being installed. This required the user to install the plugin and then activate the MTEE perspective. This was further hindered by the fact that option to import the MTEA tool could only accessed from within the MTEE perspective. While these changes are unavoidable, it was felt that they could be overcome by the users, as they both have strong backgrounds in using Eclipse.

3.3.2 MTEA Tool Interface

When deciding how much control the user would have over the tool, two decisions needed to be made; should a performance be exported each time the user runs the tool in 'Preview' mode and should the performance begin and end automatically in 'Replay' mode. By leaving this decision to the user, an interface between them and the tool had to be created. There were 3 options, overlaying an visual interface on the users existing program, having a separate window with all of the options for the tool, or providing a basic interface through the keyboard.

Having a separate window of controls would give complete control over how the controls were displayed and where they were positioned. It would also provide real-time data to the user. While this was an attrac-

tive option it would cause considerable conflict with the user's program, and require that they remove focus from the program window to interact with the tool. In the case of full screen programs, this would obscure the tool window and require the user to switch back and forth between the two windows. This would have caused problems with programs in different operating system environments that handle full screen windows differently.

By overlaying controls directly onto the screen itself it would have avoided the problem with window focus and full screen displays. It would also give some control over where the controls were positioned and what data could be displayed. The main problem was that without knowing the layout of the user's program, it would be likely some part of it would be obscured by the overlay. It was very undesirable, especially for a tool collecting data on how users interact with that program.

The third option was to provide a keyboard interface, allowing the users to interact with the tool via key combinations. This would not interfere with the users display, but could cause minor functionality problems if the user also uses the same key combination in their program. The keyboard does not provide the user with any feedback that something had occurred when a key was pressed.

It was felt that using keyboard shortcuts was the best choice. As the users of the tool were developers of multi-touch interfaces, they would have avoided keyboard usage in their own programs. If the problem had arisen, using suitably abstract key combinations could help to avoid areas where they clash.

The feedback issue was resolved in two ways. First, by using the Eclipse console. All programs run through Eclipse could output data to the standard system output and it would be displayed in the Eclipse console view. This provided a means to give some feedback via console output, we felt that as the users were using the MTEA tool through Eclipse and were familiar with the Eclipse environment that this would be suit-

able. Secondly, as actions such as replaying a performance are played through the users program this provides suitable feedback that an action had occurred.

3.3.3 Component Code View

During the analysis and feature design phase it was intended for there to be a way to link the components seen in the MTE Component view to where they were created in the code. This would have allowed the user to more easily examine how a component was created and how the behaviour of gestures were coded. Two solutions were investigated, however they all had unsatisfactory problems.

The first solution was to attempt to hook into the Eclipse debugger. This would provide the benefit of on the fly code changes and dynamic access to the running code. Unfortunately this proved easier said than done, while the tool could be run in debug mode there was no method to change the behaviour or manipulate the debugger from within the tool. Short of creating a debugger from scratch it was decided this would not be a viable option.

The second solution was to provide static locations of the components by searching the code for component names. This had two major problems; the biggest being that you cannot rely on developers to actually name each component, making it impossible to distinguish when searching. The other problem is with the way MT4J programs can be structured. Often the location in the code where a component was created would be in a different section to where it had gestures assigned. It could even be created dynamically at runtime.

These problems meant that any solution would be time consuming to create and only provide disjoint and confusing information to the user. For these reasons the feature was scrapped.

3.3.4 Component Information View

An original feature in early design was to provide a separate view for information relating to the currently selected component in the component view. It was found that due to the unnamed component problem, and the amount of data that could be scrapped from a component dynamically, that there was not enough information to justify an entire view. This data included the component ID, the component name and the gestures that the component was listening for. The component view itself inherently provided information about the nesting of the components within each other. Information about recorded gesture data could then be displayed in the scene tree for each performance in the MTE Performance view.

3.3.5 Performance Overview

When deciding how to display the performance data back to the user we considered three options; rendering a mock up of the users interface with performance data in an Eclipse view, recording a video of the users performance with an overlay of the performance data and integrating the performance data and replay directly into the users program.

The first option was to create a view in Eclipse where the selected performance could mock-up the users interface and display the performance data over the top. This would have allowed us great control over what information was displayed, and even allow data from multiple performances to be displayed at the same time. While this would work for simple interfaces, large or dynamic interfaces make the problem far more complex. This would require generating the performance interface as the user plays with their program, along with more intensive data recording, adding overhead to the tool.

The first solution, evolved into the second solution where we considered recording the user's performance with a video. This would have allowed us to play the video back to show their performance. This could

then be coupled with performance data to provide cursor trails and other useful information. It would also make it easier to add time controls to the replay, such time skipping and rewinding. One issue with using a video is it is static information, the user was not able to interact with the program in the state that the recording shows. This could have made it harder to demonstrate behaviour in the system.

As the idea of the performance replay is to show a snapshot of a programs behaviour, a third solution was devised. This involved feeding the event data back into a running instance of the user's program. This created fake input which would perform the actions the user user originally recorded, but on a live version of their program. A snapshot system was used to ensure the performance were always played back from the correct version of the users program. The MTEA tool could then be used to provide extra visualisations on top of the replay, such as cursor trails and component highlighting. This solution wasn't as robust as the others, as it was found to only display the performance data reliably for 2D interfaces.

For the development of the prototype we decided to use the third solution, as it provided the most functionality for the smallest cost. The MTEA tool was already integrated into the user's program and being used to record and manipulate the display. This allowed easy access to reading in recorded data and playing it back as faked user input.

While this solution meant the playback for input data would work for any MT4J program, having the data overlay only work for 2D interfaces was deem acceptable. This was decided after looking at the user personas and investigating what kinds of programs they would be looking to create and test. It was felt that both Tom and Harold would understand the limitations of the tool, but still find it fell within the bounds of what they were trying to achieve.

It should be noted that in particular the second and third solutions are not mutually exclusive. It would be feasible for the project going forward to also include the video recording function, as combining these two solu-

tions could help to alleviate some of the short falls with each.

3.4 Alternative Designs

In this section we discuss the alternative languages that the Multi-Touch Explorer Environment (MTEE) could have been designed for. We also look at what tools were considered to go with these languages and why they were not chosen.

3.4.1 Alternative Tools and Languages

While Java is a popular and modern language, it is not the only one. It was important to consider other languages before making a decision. We found that both Python and C# were very compelling contenders. C++ was also considered due to its popularity, but lack of proper multi-touch framework meant this was not a real option.

Kivy w/ Python Like Java, Python is widely used and free to download. It is supported on Windows, Linux/Unix, Mac OS X and on the Java and .NET virtual machines. Python is maintained by the Python Software Foundation which funds and manages events in the Python community. Unlike Java, Python is a dynamically type language which can act as both a fully object-orientated language or as a simple scripting language. This flexibility made Python a very attractive language and accounted for its popularity today. It was also favoured as a teaching language because of its clear easy to understand syntax [65] [42].

While Python would have been a good choice it was important to consider how it would have worked for multi-touch development. Thankfully Python has the Kivy (formerly PyMT) project, a Python library for the rapid development of applications that make use of innovative user interfaces, such as multi-touch [2] [38]. Kivy is a mature framework that

is still well maintained. Like MT4J it is an open-source project and everything is provided for free. This fitted well with the Python language and made Kivy an attractive option for developing multi-touch.

Using Kivy and Python together would make a great combination and would have been an valid choice for this project. As mentioned earlier Eclipse can also support the Python language, which could provide a front end for the MTEA tool the same as with Java and MT4J. As it was the intention that the MTEE prototype only supported one language, this project chose Java and MT4J on the basis of the personas experience with it. Staying with the Java programming language would also allow both the MTEA tool and the Eclipse plugin to be written in Java.

Surface SDK w/ C# The C# language is developed by Microsoft and fully supports their .NET frameworks. C# is intended to be a simple, modern, general-purpose, object-oriented programming language and is recognised by ECMA as an international standard [39]. It is co-developed by a number of major organisations such as Hewlett-Packard, Intel and Microsoft. The language is designed to be usable by developers familiar with C and C++, however it is not supported on all platforms as the .NET framework is only available on Windows. An open-source version is available on Linux called Mono [3] however it is not as well supported or developed. Basic support for multi-touch in C# is provided by the .NET framework, with more complex support through the Microsoft Surface SDK [55]. The majority of these features are only supported by the Microsoft Surface device.

Microsoft encourages that C# developers use their Visual Studio line of development environments. Visual Studio supports all of the Microsoft Visual languages, with the ability to support almost any language through language services. Visual Studio also provides tools to aid developers, such as debugging integration, word predictions and test support [56]. Visual Studio can be extended via the Visual Studio SDK. This could be

used to add new windows, menu commands and other extensions. This would have allowed Visual Studio to be used to as the interface for the MTEA tool [54].

While C# and Visual Studio are both mature products that would have been suitable for this project, their platform restrictions and price meant they were not chosen. While Microsoft does offer Visual Studio licenses to students through the Academic Alliance program, it could not be assumed that a student would be familiar with these tools. It would also create issues for professional user who don't have a license, effectively barring then from using the tool.

libTISCH w/ C++ C++ is a general purpose programming language built on top of the C programming language. adding object orientated features such as classes along with additional data types, templates, exceptions and other features to the C programming language [41]. C++ can be compiled to run on all major platforms with a number of different compilers available, such as the GNU Compiler Collection or Microsoft's Visual C++ compiler.

While C++ can be made to accept multi-touch input directly, there are few frameworks to help with handling and processing gestures. As the intention of this project is to aid in the rapid development of prototypes, it is not feasible for a user have to integrate touch input and gestures into an applications themselves.

The most portable framework available was the Tangible Interactive Surfaces for the Collaboration between Humans, or TISCH, framework. While this framework is functional and could be used to build multi-touch applications it is no longer maintained, with the last update in September 2010 [26]. For these reasons it was decided the framework would not be suitable for new users to learn with the MTEE project.

3.4.2 Alternative Java Multi-Touch Frameworks

Once we decided to stay with the Java programming language we also considered other multi-touch frameworks for Java aside from MT4J. While MT4J has a slow release cycle, with the latest release on March, 31 2011, it is still actively maintained. We investigated two other multi-touch frameworks for Java, SparshUI and mu3. Unfortunately both of these projects are no longer maintained, SparshUI was last updated in September 2010 and mu3 October 2009 [40] [74]. As with TISCH discussed above, it would be inappropriate to teach a new user either of these frameworks if they are no longer supported by the community or developers.

3.5 Design Requirements Analysis

	FR1	FR2	FR3	FR4	FR5	FR6	FR7	NFR1	NFR2	NFR3	NFR4	NFR5
Java Language	×	×	×	×	×	×	×	✔	✱	✱	×	×
Eclipse IDE	✔	✱	✱	✱	×	×	×	✔	×	×	×	×
Mulit-Touch for Java	✱	×	×	✱	✱	✱	×	✱	✔	×	✱	✱
Import Wizard	✔	✔	✔	×	×	×	×	✱	×	×	×	×
MTE Perspective	✔	✔	✔	×	×	×	×	×	×	×	×	×
Component View	✱	✱	✱	✔	✔	×	✔	×	×	✱	×	×
Performance View	✱	✱	✱	✱	✔	✔	✔	×	×	✱	×	×
Component Isolation	×	×	×	✔	×	×	×	×	✱	×	×	×
Gesture Recording	×	×	×	×	✔	✱	✱	×	×	×	✱	✱
Performance Recording	×	×	×	×	✔	✱	✱	×	✱	×	✱	✱
Performance Playback	×	×	×	×	×	✔	✔	×	×	×	×	×
Runtime Arguments	×	✱	×	×	×	×	×	×	×	×	×	✔
XML w/ Schema	×	×	×	×	×	✱	×	×	×	✔	✔	✔

Figure 3.6: This table indicates if each design decision was expected to meet either a Functional Requirement (FR) or a Non-Functional Requirement (NFR). A ✔ indicates this requirement will be met by the design, a ✱ indicates the requirement will be partially met, and a × indicates that this design decision did not contribute to that particular requirement.

Chapter 4

Implementation

This chapter looks at how the designs from chapter 3 were implemented. Section 4.1 involves critiquing the tools chosen for the design. The project was split into two parts, the Eclipse plugin and MTEA tool, which are discussed separately. The Eclipse plugin is discussed in section 4.2.1 and the MTEA tool in section 4.2.2. Both of these sections discuss how the tools were used to meet the design goals.

4.1 Language and Tools Critique

This section discusses: the chosen tools for this project, what was found useful about them, and how they performed. The chosen tools were the Java Programming Language, the Eclipse development environment and the Multi-Touch 4 Java (MT4J) framework. These are discussed in sections 4.1.1, section 4.1.2 and 4.1.3 respectively.

4.1.1 Java Language

The features of the Java language helped to inform how some of the projects designs were implemented. The Java ARchive (JAR) system for Java programs proved very useful for packaging snapshots of the application for

each recorded performance. The JAR format compresses the Java class files, meta data and other resources into a single portable file. This can then be run on the Java Virtual Machine (JVM). By creating a a JAR file for each performance, the behaviour of the program at that point in time is saved for later replay.

The ability to include both static and dynamic code helped to deal with setting up the MTEA tool at run time. By passing arguments in at run time, the could be assigned statically to the main MTEA tool class. This meant when the class was eventually created by the JVM, it would already be set to run in the correct mode or look for the correct performance data.

The Java programming language was used for all of the software development in the project. The decision to build a plugin for Eclipse meant that that portion of the project had to be developed in Java. To simplify the development and capitalise on my experience, the MTEA tool was also developed in Java. As the Eclipse platform is written in Java itself, it allowed me to better understand how the platform worked and made it easier to observe how Eclipse handled some functions so they could be replicated for the plugin. Other peoples plugins for Eclipse were also written in Java and could be observed for similar reasons.

4.1.2 Eclipse Platform

Eclipse was used both as part of the project itself, and as a development tool for the creation of the project. Eclipse provides a plugin development mode for building Eclipse plugins. The plugins could then be run in a separate instance of Eclipse so they could be used and tested.

The Eclipse Integrated Development Environment (IDE) is built on top of the Eclipse Rich Client Platform (RCP), which is a minimalist set of plugins which are needed for an application built on the RCP platform [28] [61]. This means the Eclipse IDE is very modular, since all of the different views and editors which can be interacted with are all plugins themselves.

Each plugin is made up of additions to the base plugins, called extensions, as they extend or add another piece of functionality to an already existing feature. The places where new extensions can be hooked on to existing functions are called extensions points. An example is an extension point for a menu, which could be used to add extra options to that menu.

A new plugin is defined by a manifest file. The manifest is an XML file containing information about the plugin. This includes information such as which extension points are being used and where the new extensions are defined. A developer is also able to define their own extension points so that other developers may be able to build on top of the first developers plugin in the future. Eclipse reads this manifest file when it loads the plugin, and uses the information to link the new functionality into the correct locations in the Eclipse workspace [9] [10].

It was found that a frequent issue when developing a plugin for Eclipse was the lack of documentation. While some documentation did exist, it was usually out of date and no longer relevant to the current version of Eclipse. This proved to be problematic for a large platform such as Eclipse, as some features were purposefully experimental and were often removed or changed in later versions. Some success was found by using internet forums and other third party channels to gain helpful information. However this was time consuming and less than ideal, since the same out of date information was often being promoted.

Another technique to gather knowledge was to examine the source code for other open sourced Eclipse plugins. This showed how other developers implemented similar features; these solutions were then adapted for this project. It was also possible to use a similar technique with the Eclipse runtime, by using the debugger. The debugger was used to help define what values and data types Eclipse was using in its extensions.

While a good effort was made to implement all the features as they were designed, often it became unfeasible too spend much time trying a particular solution. The result of this was to change how some features

were implemented. Of particular note was changing how the user's program was accessed by the MTEA tool. The original design was to access the program before run time by adding a new Eclipse launch mode. This was redesigned to what is seen in the project, where a separate launch file is created directly in the user's project. This removed the need to extend Eclipse's launch modes.

Overall, Eclipse was an extremely useful tool both as a development environment for the project and as a platform for the user interface of the project. While the plugin development could be confusing at times, the platform was often consistent in how things were expected to work. This allowed solutions to be applied to a number of different problems. In instances where these solutions would not work, it was found best in the interest of time and scope to abandon the manifest file and implement the features directly in code. An example of this was with the implementation of the context menus for the MTE views. These menus were defined in the manifest file with extension points so menu items could be added to them. This attempt proved unsuccessful as the menu items would not register with the context menu extension. Instead the menu items were added directly to the context menu in the code. By using solutions such as this, Eclipse proved to be a feasible tool for providing the project interface and showed that it was a good choice to meet the user's goals.

4.1.3 Multi-Touch 4 Java

MT4J was the most up-to-date and well maintained of the available multi-touch frameworks for Java, which was discussed in section 3.4.2. That section discussed the reasons from the user's perspective about why MT4J was the best choice, however it was also important that MT4J would be accessible to allow the implementation of the desired features. This was found to be true, as the framework provided the ability to traverse through the components in a user's program and attach both gesture and input lis-

teners. These listeners were registered to a scene or component and were executed when the framework detected a gesture had been performed on that component. From there the project was able to record the data about the gesture. A similar technique could be used to feed faked input data back into the same program, by instead registering a new input source developed to generate input data from a file. MT4J would then be able to interpret this input data as if it was a real user interacting with the program.

The way an MT4J project is structured and initialised provided a solution for hooking the MTEA tool into an instance of the user's program. Every MT4J project required a start file which setup the program environment. The start file contained a static main method and a callback method called `startUp`. The main method initialises the environment, returning to the `startUp` method when complete. This method was then the starting point for the user's code. By creating a subclass to a project's start file which overrode this method, the MTEA tool was then able to be run on top of the user's MT4J application. As every MT4J project must use this start file, it provided a common point of access to launch the MTEA tool. By having the separation of the user's start file and the project's start file, it was ensured that the user would always be able to run their project without out the MTEA tool being active. This solution ensured that Functional Requirement 4 could be met.

4.2 Implementation of Design

This section discusses how different parts of the project design has been implemented, for both the MTEE Eclipse plugin and the MTEA tool.

4.2.1 Eclipse Plugin

As discussed in section 4.1.2, Eclipse uses extensions and extension points to allow a user to hook their own functionality into existing Eclipse features. The project made use of these to add some of the plugins functionality, but were also avoided in some instances.

MTE Perspective The MTE perspective defined the positioning and layout for both the MTE views and the standard Java views that were used. While the layout is defined directly in code, the manifest file is used to hook the new perspective into Eclipse so that it can be activated by the user. The manifest file, found in Appendix B, shows how the new perspective was linked into Eclipse on lines 4-12. Figure 4.1 shows the change adding this extension caused in Eclipse. The manifest shows which extension point was specified, in this case org.eclipse.ui.perspective was used; this is the standard point for adding a new perspective. The perspective extension required a class name, where the layout of the workspace was coded, and an ID name to identify the extension by. A name and icon were provided which were used to display the perspective in the perspective list, appearing once the plugin was installed.

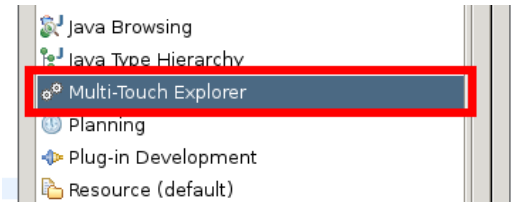


Figure 4.1: This figure shows the output of adding the perspective extension in the Manifest file. The red border shows where the new perspective has been added.

The class referenced in the manifest for perspective extension was required to implement the IPerspectiveFactory interface. This was provided by the Eclipse Plugin library and allowed the class to be recognised as a

perspective. All classes which used this interface needed to implement the `createInitalLayout` method. This method was called when the perspective was enabled and executed the code which setup the views and workspace positions that were used for the perspective. The views which were defined by the perspective were referenced by their ID name, this included both the standard Eclipse views and the new MTE project views.

Component and Performance Views The new MTE views were hooked into Eclipse using the org.eclipse.ui.views extension point. This was put into the manifest file, much like the perspective above, but instead used the view tag name instead of perspective. The new view also required a class, an extension ID name, an icon and name like the perspective. The new views were required to sub class the ViewPart, an Eclipse class which provided the basic functions for a new view. The `createPartControl` and `setFocus` methods were overridden, which allowed the contents and actions associated with the view to be defined.

While the manifest extension and ViewPart class handled making the view available to Eclipse, the actual content of the view needed to be coded in. Eclipse provided a number of predefined 'viewers'. These made handling of the input for the view much simpler. As the MTEE project saved the component and performance data as XML files, it was decided the TreeViewer was a natural choice to display the data. The TreeViewer provided a tree view of the content, allowing the branches of the tree to be expanded or hidden as the user desired. This worked very well with the XML format, as it stores data in a similar tree structure. The data then only needed to be filtered to produce the proper names and information. Figure 4.2 highlights what effects the ViewPart and TreeViewer have in the Eclipse workspace, with the ViewPart encompassing the TreeViewer.

To setup a TreeViewer two extra parts were required, a content provider and a label provider. A sorter could have also been used to order the tree nodes, however this was not used as it was preferred that the data remain

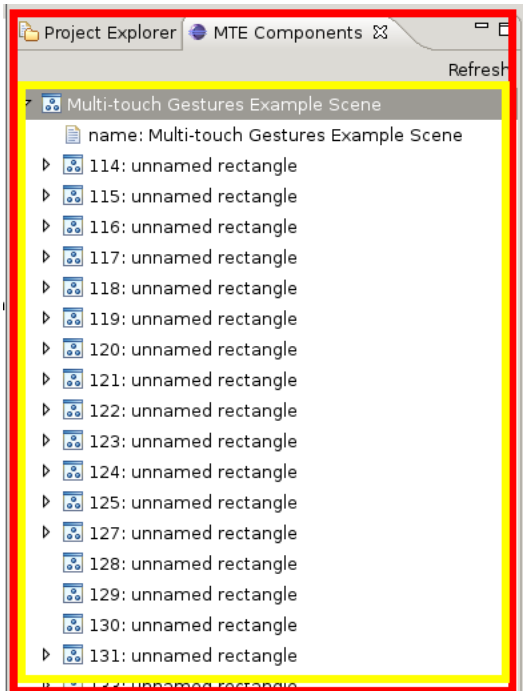


Figure 4.2: A picture of the MTE Component view. The ViewPart is highlight in red and TreeViewer output is highlighted in yellow.

ordered the way it was in the data file.

The content provider loaded in the XML data from a file and turned this into data objects for each element of the file. These data objects were stored by the TreeViewer and were used to retrieve more data when a node of the tree was expanded. The TreeViewer would query the data object and determine if the node had any children, as defined in the content provider. This information would then be displayed in the form of an arrow to the left of the node label, which indicate the entry could be expanded.

The label provider is the other half of the TreeViewer. This would examine the data objects from the content provider and decide what text should be displayed for that node in the tree view. A path to an icon was also specified to help distinguish between different kinds of data objects in the tree view.

Both the MTE Component view and MTE Performance view use the same content and label providers as their data is compatible. This is made possible by using a common XML schema file for both data files. This schema can be found in Appendix A.

View Context Menus Both of the new views required context menus to provide different actions for each view. While an attempt was made to use the Eclipse plugin system to link the menu item actions to the context menu, this was unsuccessful. It was found that while an extension point could be added for a context menu, attempting to link a new action to that menu proved fruitless with the menu item simply never appearing.

Instead these were defined directly in the code, by overriding the `createPartControls` method to achieve this. Implementing the menus in this way proved to allow a finer grain of control over what information the menu actions could receive and when the menu could be displayed. Once the context menu has been created, items were added to it. These items were required to be a sub class of the Eclipse Action class and override the `run` method. The method was then called when the item was selected from the context menu.

Setup Wizard The setup wizard dealt with the importing of the MTEA library and schema files; along with the generation of the MTEE project start file for MT4J and the project build file. It was required that this file be generated for each project as it needed to be adapted to properly import the user MT4J start file which would not have a standard location. The selection of the MT4J start file was handled by the user through the setup wizard. This selection also served to indicate which project the user intended to use with the MTEA tool, so that everything could be imported into the correct place.

To launch the setup wizard the user was required to select the 'MTEE Project' from the new menu in the Package Explorer view in Eclipse. This

would display the setup wizard where the user would select their start file and press the 'Finish' button to initiate the import process. The actual steps the wizard would perform are outlined here:

Step 1 Two new folders were created in the user's project. One was a new source folder to hold the new MTEE MT4J start file that will be generated in a later step. The second was a folder to hold all the MTEA data for the project, this was placed in the root of the users project.

Step 2 The MTEA library and schema files were imported. These files were pulled directly from the MTEE plugin bundle. A plugin bundle in Eclipse is a collection of all the files and classes for any given plugin, providing a link to any files that a developer has placed in their plugin. This gave the correct path to the MTEA files and would not matter where Eclipse was installed on the users system. The MTEA library was then added onto the end of the user's project's class path, where all the libraries and dependencies for the user's project are stored by Eclipse.

Step 3 The three new files required for the MTEA tool were created. First the schema file that was imported in Step 2 was written into the MTEA data folder. This is followed by the build script, which is used to automate the exporting of the user's project into a JAR file. The final file was the new MT4J start file for the MTEA tool. This file is generated from the project's start file, but added extra functionality such as command line arguments to control the MTEA tool and hooking the tool into the user's project, as was explained in section 4.1.3.

Step 4 The build file was then run to generate a snapshot of the user's project at the current state. This was saved as 'latest.jar' and stored in the MTEA data folder.

Step 5 The MTEA start file was run to generate the scene tree data file required for the MTE Component view. A special command line argument was used so that the tool will quit as soon as the scene tree was exported.

Project Launching The project required a number of different tools to be launched from within Eclipse. This was done through modifying the Eclipse launch manager. To set up the launch manager to execute a program numerous information was required. The appropriate launch configuration type needed to be used and decided what type of external or internal program Eclipse would launch. An example of this is shown in Figure 4.3, line 3-5, where the `localJavaApplication` type from the jdt.launching package was used. This type specified that Eclipse is to launch a local project, just like the user would through the Eclipse launch menu.

Once the configuration type was set up, the correct attributes needed to be given. The attributes were specified where the project and main class file were located. When launching an external tool, such as the JAR builder or external JAR file, the application location and work directory were required instead. This also had the option to specify any program arguments that would have been required. It was here that the component ID or replay name were passed into the MTEA tool. The attributes can be seen in Figure 4.3, lines 9-14.

4.2.2 Multi-Touch Explorer Analysis Tool

The Multi-Touch Explorer Analysis (MTEA) tool was designed to hook into any existing MT4J project. This was done by hijacking the start file used to launch an MT4J project and overriding the `startUp` callback method used to create an instance of the MTEA tool. The tool was then able to access all of the data in the MT4J application, including the scenes and components. This allowed the tool to export the scene tree to XML and

```
ILaunchManager manager =
  DebugPlugin.getDefault().getLaunchManager();
ILaunchConfigurationType lct =
  manager.getLaunchConfigurationType(
    "org.eclipse.jdt.launching.localJavaApplication");
ILaunchConfigurationWorkingCopy wc =
 lct.newInstance(null, "PreviewConfig");

wc.setAttribute("org.eclipse.jdt.launching.PROJECT_ATTR",
                project);
wc.setAttribute("org.eclipse.jdt.launching.MAIN_TYPE",
                mainType);
wc.setAttribute("org.eclipse.jdt.launching.PROGRAM_ARGUMENTS",
                programArguments);

wc.launch(ILaunchManager.RUN_MODE, null);
```

Figure 4.3: A code snippet of how launching a program from Eclipse can be done. The example shows how a local project would be setup and launched.

listen for input and gestures on each of the components. Figure 4.4 shows the structure of the MTEA tool and where it sat in relation to the user's program.

The MTEA tool was split into 3 distinct parts; the Explorer part which handled the users application and setup and tear down of changes made to the display, the Processors which provided a means of to capture events and inject new events into MT4J, and the XML part which handled the importing and exporting of data.

Explorer Interface The explorer interface consisted of the AppExplorer and the SceneExplorer. The AppExplorer handled the setup of all the required objects for a particular display mode, such as when replaying a performance or isolating a component. The SceneExplorer class handled the

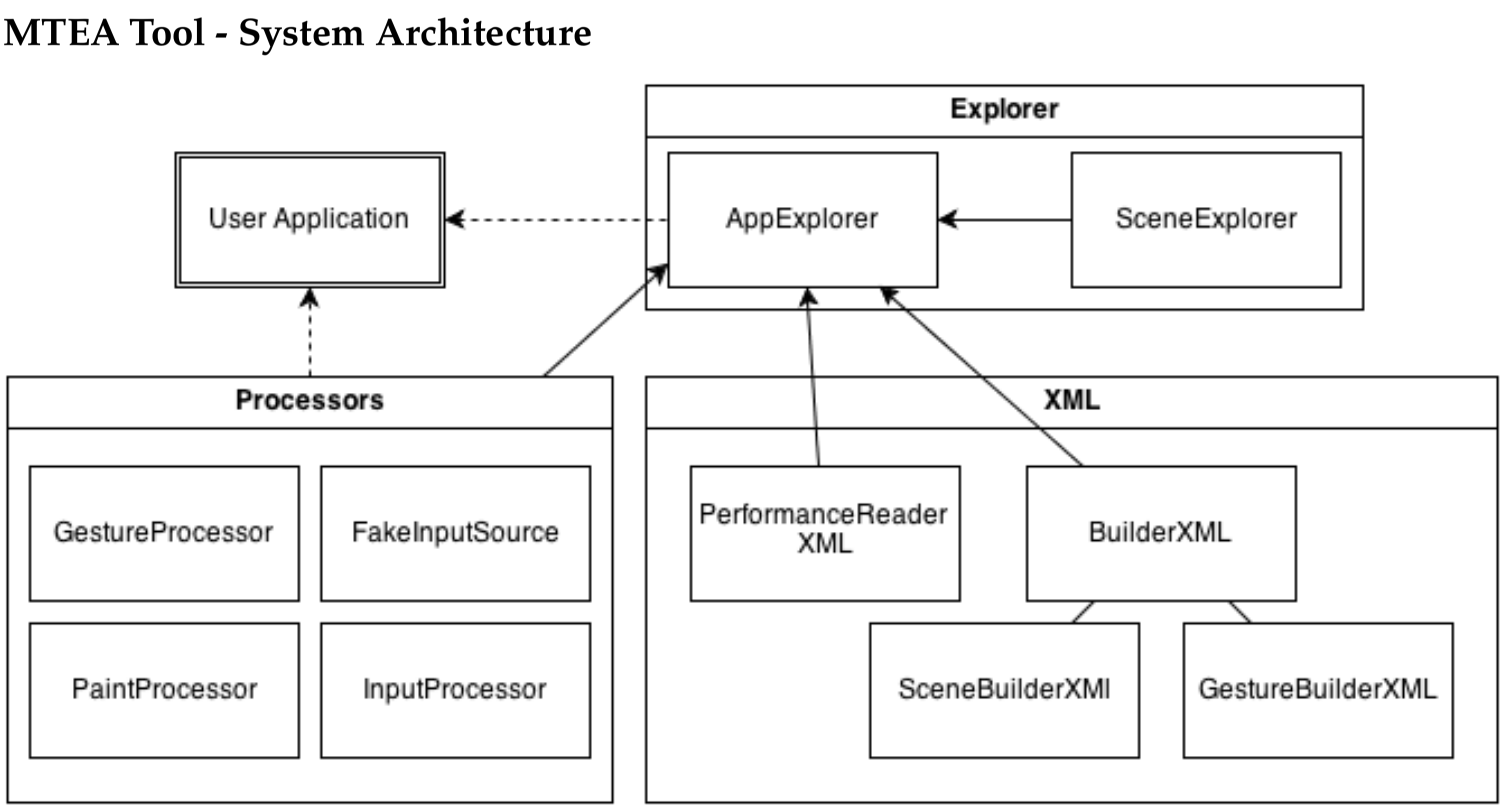


Figure 4.4: The layout of the MTEA tool. The User Application is passed to the AppExplorer class at runtime. The AppExplorer class can then instantiate a number of extra classes, such as the SceneExplorer of XMLBuilder, depending on its needs at runtime.

set up of a scene and its components, so that gestures could be recorded or different components could be highlighted.

The AppExplorer itself had three main functions; handling runtime arguments, setting up the core MTEA tool components, and providing the key bindings for the user. As the runtime arguments were passed in from the main method in the MT4J start file, it was required that these used static methods and variables. Java hands the program arguments to the program as an array of strings. The MTEA tool would parse each of these strings in order looking for recognised strings as detailed in Chapter 3, section 3.2.4.

AppExplorer was managed by three setup methods, `setupSceneExplorer`, `setupPlayback`, and `setupRecording`. As indicated by their names, the playback and recording methods handled the two main modes of the tool. The scene explorer method was used by both modes, where the scene would be setup in different ways depending on which mode was used. The AppExplorer handled the keyboard input, which allowed the user to save and replay a performance. These were implemented by registering the AppExplorer class with MT4J as a key event listener.

The playback setup required creating a new FakeInputSource and registering it into the user's program. The fake input was then built from data read in through the PerformanceReaderXML. It tried to locate and load the correct performance from the performances data file. This was based on the performance name passed in via the command line. The SceneExplorer was used to add the PaintProcessor to each component, which enabled the cursor trail and highlighting of active components. Finally all components in the scene had their alpha faded out, so that active components would be more emphasised when they were used. The recording setup was similar to the playback, but instead would create a new GestureProcessor and InputProcessor. The GestureProcessor was registered to every component in the scene by the SceneExplorer, while the InputProcessor was just reg-

istered to the scene itself. If a component has been selected for isolation, then all but this component and its children were disabled and had their alpha value faded by the SceneExplorer.

The SceneExplorer handled functions for modifying the user's scene and components, as well as traversing the components. These functions revolved around the `traverseComponent` method, which would traverse a scene canvas containing all the components in the scene. This used a depth-first recursion method to find components based on the ID. A number of flags were passed down with the recursion method to decide what features should be enabled or disabled each component. The features included fading the components alpha value, disabling input, adding a gesture processor or highlighting the component.

The key bindings had two important purposes; starting the playback of a performance and exporting a saved performance to disk. The playback of a performance would call the FakeInputSource to begin the playback of events, as described below. The exporting of a performance would save the recorded performance and gesture data to the performances data file, as well as making a copy of the snapshot JAR file that was generated when the MTEA tool was imported. The time taken to create a copy of the JAR is dependent on the size of project the JAR was created from. It was felt that a copy of this would not constitute an unreasonable amount of disk space as a large project would require a larger JAR file. For this reason it is believed that non-functional requirement 4 has not been violated.

XML Handlers The MTEA tool used XML as its data format, with a schema file to provide data validation, shown in Appendix A). This was an ideal format to use as it could imitate the structure of the MT4J scene tree. The MT4J scene tree used a composite design pattern to impose its scene structure and nest components within themselves. The XML data itself was handled by the Document Object Model (DOM) from the standard Java library. This provided both the means for creating documents and

XML nodes, as well as checking for data validation against the schema. This ensured that requirement 3 of the non-functional requirements could be met, as the schema was checked against the data when it was exported and when it was imported.

The XML handlers used a variation of the Builder design pattern to help construct the complex XML structure of the scene tree. This allowed both the explorer data file for the MTE Component view, and the performances file for the MTE Performances view to be constructed from the same collection of classes. The SceneBuilderXML uses depth first recursion to build a model of the component structure, and the GestureBuilderXML turns the different gesture events into a collation of any particular gesture. The BuilderXML deals with and links all of the different scenes and gestures into the correct files, before verifying and saving them to disk.

As the tool recorded data it would build the gestures into the scene tree stored in the BuilderXML class. This data would be stored in memory until the user executed the export function through the AppExplorer. If the user did not save the performance, then the data would be lost when the program was terminated. This was as intended, as the tool would only write data to disk when the user wanted to. This reduced the overhead of waiting on I/O for each gesture and helped to meet requirement 5 of the non-functional requirements.

Event Processors The event processors included both recording and outputting data, and were used to glue different parts of the MTEA tool together. There were four processors, the GestureProcessor, the InputProcessor, the PaintProcessor and the InputSourceProcessor. These processors follow the mediator design pattern, by using the existing Listeners and Processor classes in MT4J as the basis for the mediators. The event processors become the concrete mediators, facilitating the communication between different parts of the MTEA tool.

The GestureProcessor would be registered to a component so that it could receive all of the gestures that were performed on that component. The GestureProcessor implemented the IGestureEventListener interface, which required the `processGestureEvent` method. When a component receives a gesture event, it was sent to this processor and the `processGestureEvent` method would be executed. The method would provide the gesture event object which was then sent to the BuilderXML to be recorded on a per component level. Any other processors listening on that component would also receive the same gesture event. The InputProcessor worked in a similar manner, however instead of being registered to a particular component it is instead registered to the scene itself. This was because all input data was passed into the scene before it was interpreted and turned into a gesture event by the MT4J framework.

The PaintProcessor was a combination of the AbstractGlobalInputProcesser and IGestureEventListener. This gave it the ability to detect when a component was being used so that the SceneExplorer could highlight that component. The PaintProcessor handled painting the cursor trail at every touch point. The colour for the cursor was taken from the highlighted component so that it was easy to match cursor action to a particular component. The creation and deletion of the cursor trail objects were handled by the SceneExplorer. To ensure that the cursor trail would not cause a performance dip, in accordance with requirement 1 of the non-functional requirements, each trail was limited to 150 points. When that limit was reached the oldest point would be removed.

The FakeInputSource used the standard Java Timer class to queue and execute input events. A Timer could queue objects that extended the TimerTask class. Each of these tasks could be queued with a time delay, so that they would be executed at the correct moment. When the events are read in with PerformanceReaderXML the delay time was calculated from the time stamp recorded when the input first occurred. When the delay time was up the `run` method of the TimerTask was executed. The tasks used

in the FakeInputSource used this method to push a new event on to the MT4J event queue. By using the recorded data to mimic touch points being added and removed from the interface it was possible to play back an exact replay of the input events as they were recorded.

By using existing tools, provided by the MT4J framework, for processing and handling input it could be ensured that the best performance was achieved for these each of these functions. This effort was to meet the non-functional requirements of the project, particularly requirement 1 and 2.

4.3 Implementation Requirements Analysis

	FR1	FR2	FR3	FR4	FR5	FR6	FR7	NFR1	NFR2	NFR3	NFR4	NFR5
Java Language	✱	×	×	×	×	✔	×	✔	✱	✱	✔	✔
Eclipse IDE	✔	✔	✔	✱	✱	×	✱	×	×	×	✱	×
Mulit-Touch for Java	✱	×	✱	✱	✱	✔	×	×	×	×	×	×
MTE Perspective	✔	✔	✔	×	×	×	×	×	×	×	×	×
Comp. and Perf. View	✔	✱	✱	✔	✔	✱	✔	×	✱	✱	×	✱
Context Menus	✔	✱	×	✔	×	✱	×	×	×	×	×	×
Setup Wizard	✔	✔	✱	×	×	×	×	✱	×	×	×	×
Project Launchers	✱	✱	✔	✔	×	✔	×	✔	✔	×	✱	✔
Explorer Interface	✔	×	✔	✔	×	✔	✱	×	✔	×	×	×
Xml Handlers	×	×	×	×	✔	✔	×	×	×	×	✔	✔
Event Processors	×	×	×	✱	✔	✔	×	✔	✔	✔	×	✱

Figure 4.5: This table indicates if each implemented feature met either a Functional Requirement (FR) or a Non-Functional Requirement (NFR). A ✔ indicates this requirement was implemented, a ✱ indicates the requirement was partially implemented, and a × indicates that this implementation did not contribute to that particular requirement.

Chapter 5

Evaluation

In order to have properly validated the usefulness and feasibility of the Multi-Touch Explorer Environment (MTEE) a full scale user-study would have needed to be under taken. This study would have involved both senior students and professionals from the software development industry, as these groups would have properly represented both of the personas. When coming to decide how to perform such a study, we were cautioned by the work of Greenberg and Buxton, who put forward that [usability] evaluation can be ineffective and even harmful if naively done 'by rule' rather than 'by thought' [37].

This caused us to rethink what kind of study would be appropriate at this stage of development. It was felt that performing a full scale user study would not have been, as the tool was still in the prototype development stage. A longitudinal user study would require a more complex analysis of what user had learned from using the tool, and need to be performed across multiple sites to better justify the results. Such a study would be restricted by the time available to the completion a Master's Thesis. We instead decided to focus on evaluation tools that were more appropriate for the developmental stage. To evaluate this project two tests where performed; a large scale performance metrics test and a Cognitive Walkthrough [22].

The performance tests allow us to evaluate the project against the non-functional requirements described in section 2.4.2. These performance tests were developed to be an ongoing evaluation tool to continue the assessment of the non-functional requirements against the project. The test were created to be fully automated so that a large sample size could be gathered with ease. The cognitive walkthrough used expert evaluators from the field of user interface development to emulate user groups. This allowed the assessment of a user's ability to explore and operate the tool, along with performing common tasks [72].

The performance metrics are discussed in section 5.1 and the cognitive walkthrough in section 5.2 below.

5.1 Performance Metrics

To determine the impact that using the MTEE would have on the performance of a users project we picked three programs as use cases. Each of these programs was tested over thirty test runs, with each test consisting of three phases. During the test the programs were constantly being monitored by a set of tools to gather the performance metric data. These tools were chosen to best measure the Non-Functional Requirements from section 2.4.

5.1.1 Test Programs

Three programs were used to test the performance of the MTEA tool. The programs were selected to represent different stages of development of a program, and also the underlying complexity of the program. They were selected in such a way so that Non-Functional Requirement 2 could be tested.

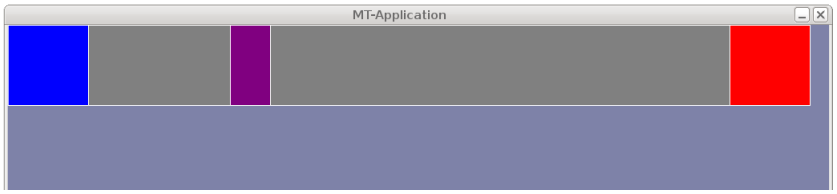


Figure 5.1: A picture showing the interface for the Slider Demo used in the performance evaluation.

Slider Demo The most basic of the programs, it consisted of a slider bar which the user can drag back and forth. The slider also had buttons at each end so the user could move the slider by tapping on those. The demo was designed to be as simple as possible with no complex behaviour. This provides a clean basis for determining the performance of the MTEA tool without any of the programs complexity contributing to the results. Figure 5.1 shows the interface for this program.

Multi-touch Demo The Multi-touch demo represents a simple user interface a developer may have created. It had thirteen different components on the screen which each had a different set of gestures. This was an ideal program for seeing how a user could use the isolation feature of the tool on different components and for performing a different user interactions. This program was taken from the MT4J example set and was found suitable for use in the performance tests. Figure 5.2 shows the interface for this program.

3D Multi-touch Gesture Prototype The 3D Multi-touch Gesture Prototype was a much more complex application where actions on the interface directly affect other objects in the scene. The program contained three different interfaces which were cycled through as the use completed navigation tasks. For the perform test the task order was modified to be constant and the number of tasks were reduced to two per interface style. The interfaces were a mix of control components and pure gestures, providing

the level of complexity required. Figure 5.3 shows the interface for this program.

The program also contained its own logging functions which provided a good means of testing that the MTEA tool did not interfere with other programs output functions. The program was developed by myself for the Honours project [23] completed prior to this Masters thesis. This allowed the program to be modified to better suit the performance tests.

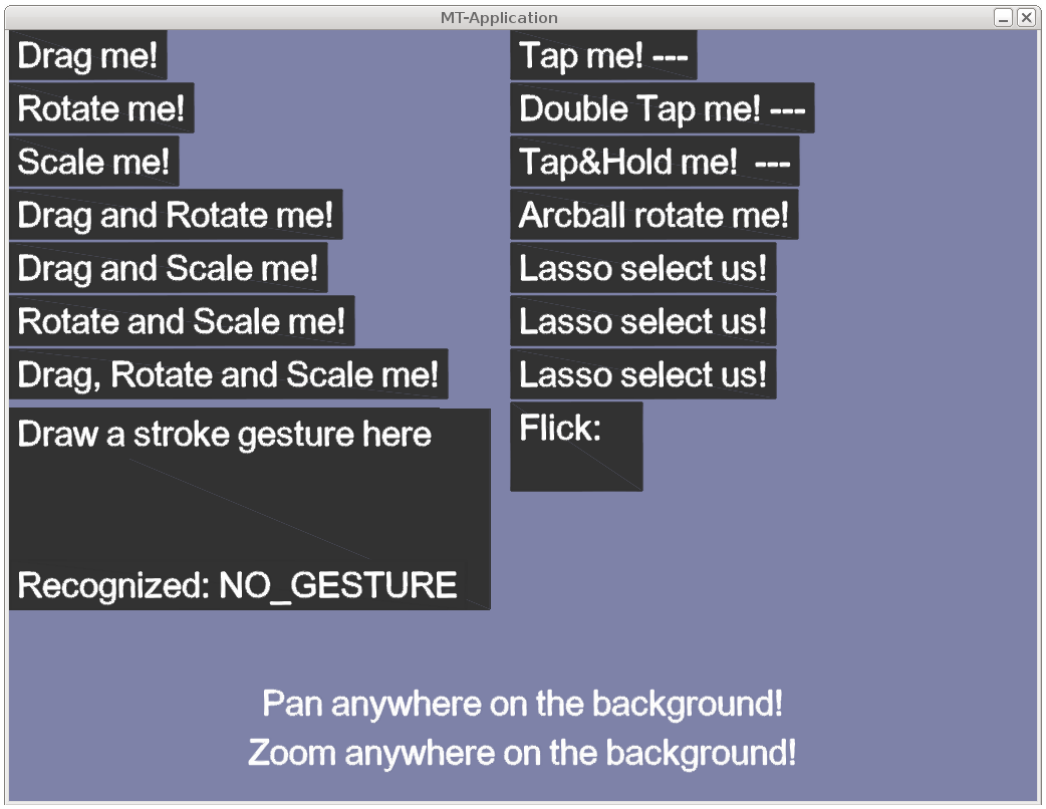


Figure 5.2: A picture showing the interface for the Multi-Touch Demo used in the performance evaluation.

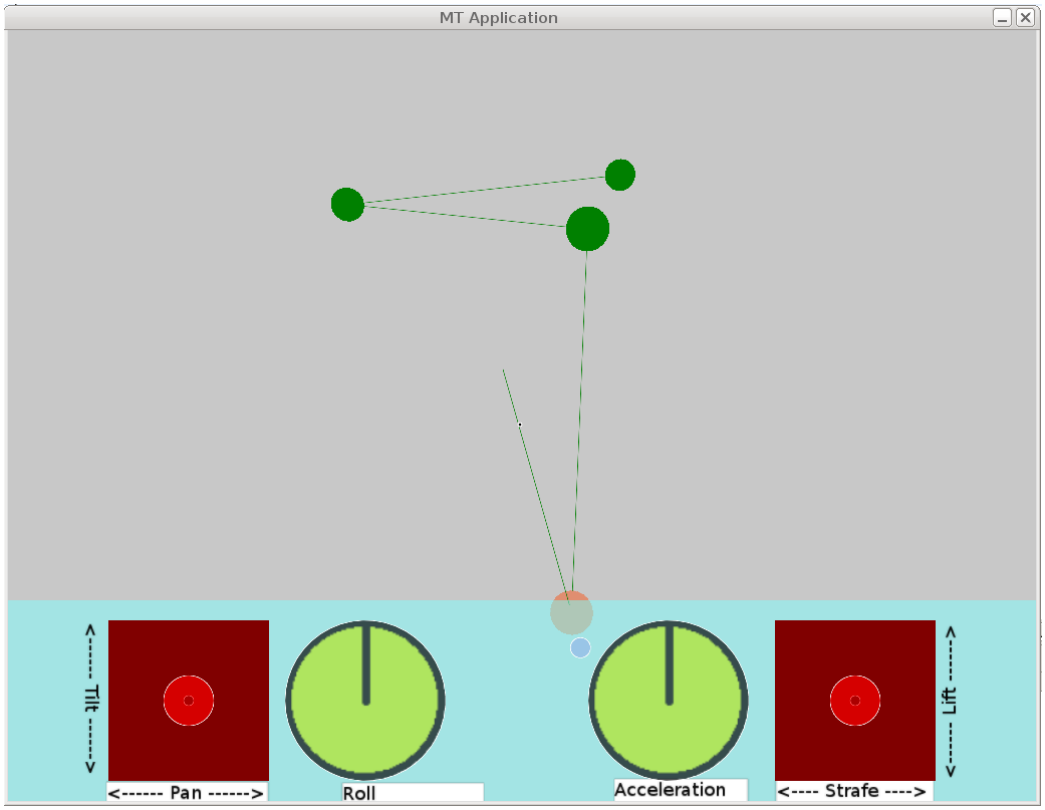


Figure 5.3: A picture showing the interface for the 3D Multi-touch Gesture Prototype used in the performance evaluation.

5.1.2 Test System

These tests were performed on a machine running an x86_64 version of Linux. The hardware specifications of the machine are:

Kernel

```
Linux 3.7.9-1-ARCH #1 SMP PREEMPT Mon Feb 18 02:13:30
EET 2013 x86_64 GNU/Linux
```

System Info

```
CPU Model: AMD Athlon(tm) II X2 255 Processor
No. Cores: 2
CPU Cache Size: 1024 KB
Memory Total: 4052636 kB

Disk Buffered Read: 131.36 MB\/sec
Disk Random Access: 65 seeks\/s, 15.19 ms random
access time
```

5.1.3 Testing Method

These tests were run directly from a command line using pre-compiled JAR files for each program. The JAR files were exported from Eclipse, however Eclipse was otherwise not used during the testing. This was because only Multi-Touch Explorer Analysis (MTEA) tool were being tested, not the Eclipse interface. The interface was instead evaluated by the Cognitive Walkthrough in section 5.2. The JAR files and monitoring tools were launched through a batch script. The tools logged their data to a RAM disk to better prevent interfering with the hard disk I/O being monitored for the test. The tests were fully automated by using a test automation tool, Xnee. This allowed a large sample size to be taken without the tester needing to manually perform each test.

Each test involved operating the programs for two minutes each. For the Multi-Touch 3D Prototype this involved completing the series of gesture tests, to better simulate how a user would interact with that program. For both the Slider Demo and Multi-Touch Demo each of the components were interacted with in turn, until the two minutes was completed. During the playback section of the test, the program was allowed to run slightly over the two minutes to ensure that the playback was fully

completed.

Each test consisted of three phases which each involved running the tool in a different mode. Phase one was running the tool with no MTEA enabled. Phase two was running the tool with MTEA recording all gestures. Phases three was running the tool, only using the playback mode to interact with the interface.

Each phase setup and executed from a single folder. To setup the folder all of the required files were copied into it, then the program was run. This folder was cleared between each phase.

Test Automation To automate the input and simulate a user performing the tests a test automation tool called Xnee [32] [32] was used. Xnee was able to automate input events for the X11 display environment in Linux. This tool was used to first record input for each test phase and test program, which could then be played back to emulate a user interacting with the program. This allowed the tests to be fully automated. The command `cnee --record --time 3 --stop-key x --mouse --keyboard -o <filename>` was used to make a recording and `cnee --replay --stop-key x -f <filename>` was used to play that recording back. [1]

Disk I/O Impact Disk I/O impact was measured using a utility called IOTop [21] [79]. This tool was able to display the disk read and write information for each program. For each test run, IOTop was started using the batch command `iotop -atbP`. This command caused IOTop to output the accumulated disk read and disk write amounts for all user programs. This is then filtered down to include only the java session being used for the test, before being logged to file. This metric was chosen so that Non-Functional Requirement 4 and 5 could be measured.

[1] The program name used in the commands is 'cnee' as the command line interface for Xnee was being used.

Disk Usage Impact This test used the Disk Usage (`du`) tool which was part of the GNU coreutils package [33]. Both before and after a test phase was completed the du tool was used to examine the overall increase in both the test folder size and individual file size. The command `du -ah` was used to list both the folder sizes and the file sizes of each file [34]. This metric was chosen so that Non-Functional Requirement 4 could be directly measured.

Processor and Memory Usage Impact The top tool, from the PROCPS package [64], was used to measure CPU and RAM usage over time. Like with the IOTop tool, top was used in batch mode with the command `top -d1 -bp$PID`, where the $PID represented the process ID of the running Java program. The `-d` flag was used to output a reading every one second. The top tool provided CPU values in percentages which are combined for both CPU cores. This meant a single value could reach as high as %200 [75]. These two metrics were chosen to measure Non-Functional Requirement 1 and 2.

Time Impact The time command [31] was used to give totals for the elapsed time that the program has been running.This included both the user time and the system time. The tool was used by executing it with the program that was being timed. An example `time (java -jar test.jar)`. This would print information to the console output when the Java program terminates. This metric was measured to help ensure the accuracy of performance tests, expecially when operating between different programs.

5.1.4 Limitations

These performance metrics had a number of limitations:

- The system the tests were running on was not a fully controlled environment. As other programs were running in the background, they could have interfered with the test. It was felt that this was acceptable as it replicated the kind of system the user would be using the tool with. This was minimised by automating the tool and manually stopping as many active programs as possible. The large number of test runs also helped to indicate outliers in the data.

- There was limited ability to ensure the same amount of input was put into each program. The amount of activity a user would perform with an application could vary depend on the application. This could affect the amount of processing a program would have to do for input. To help minimise this all test runs were conducted for the same length of time. All test runs of the same program also used the same test automation file to ensure the input was the same.

- There was no way to determine the exact time when a performance stopped and started. This information was only internal to the analysis tool. This was minimised by allowing phase 3 to run longer than the previous phases, ensuring the performance had completed before the program was terminated.

- The JVM itself causes some overhead and interference when monitoring the Java application. While this should no be a big consequence it could cause unexpected results from the test. By using a large test size we can minimise these problems.

5.1.5 Results

After each of the tests were run thirty times, the results were aggregated and averaged. The results of the graphed and tabulated, these have been displayed here. All charts and tables are presented at the end of this section.

Disk I/O Impact Figures 5.4a through 5.4c showed the amount of data each test phase wrote to the hard disk on average. A log scale has been used in the chart to more acurately show the difference in the amount of data being written. Interesting features included the spike in the blue line for each test. This was the recorded input data being exported by the MTEA tool, as well as a copy of the JAR file being made to preserve a snapshot for playback. Figure 5.4a, the Multi-Touch 3D gesture prototype, showed a series of 'steps' of data being written during all three phases. This was where the program was writing its own logging data to disk after each internal test.

Data for the amount of disk read was also collected, however they have been determined to not be accurate. Despite repeated attempts the IOTop tool used to measure the data would not report any amount of data read from disk. A few outlying test runs recorded a large amount of data being read, but the majority of the runs did not register any. It was hypothesised that the files being read were being cached into memory by the operating system, this meant they were not being read from disk each time. The large outlying values were thought to be related to the Java VM performing an unknown function, as there was no function in any of the programs that would require that amount of data. It was expected to at least see the performance file being read in at the beginning of phase three. Due to the inaccuracy of the data, graphs were not created.

Disk Usage Impact The data recorded for disk usage was what was expected. The folder size only increased substantially during the recording

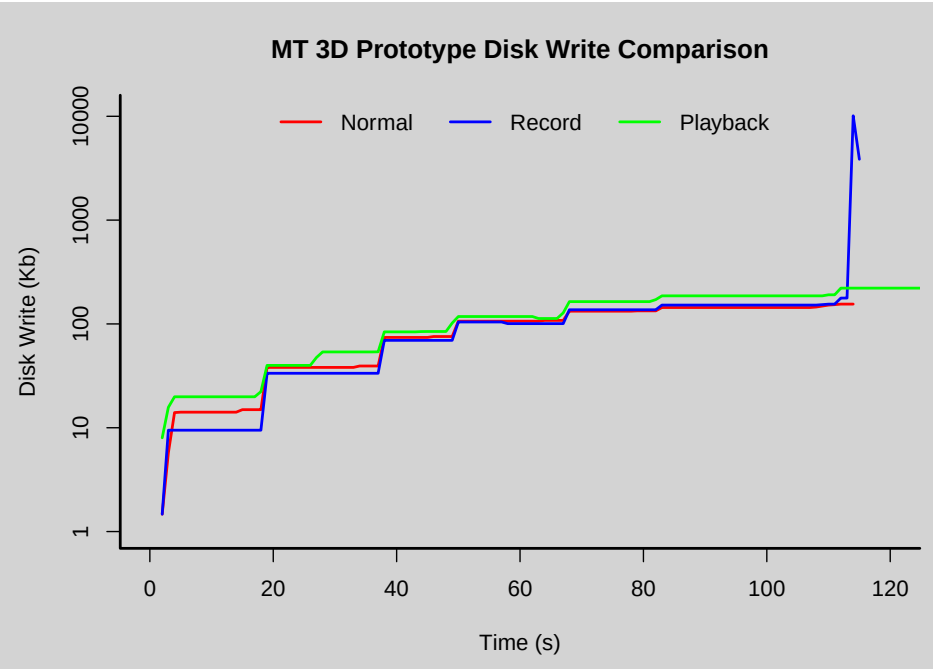
phase, as the performance and snapshot file were written to disk. One interesting point to note is the difference in the size of the performance.xml file for each program. This file was written during the recording phase, and was then read back in during the playback phase to provide the input data. Therefore it can be theorised that when this file was larger it meant that more input had been recorded. This provided a simple indicator of how much interaction had occurred for that performance. The tables can be seen in figures 5.5a, 5.5b and 5.5c.

Processor Usage Impact Processor usage was an important indicator to see how the system was coping with the each programs workload and how much more workload using the tool placed on the CPU. By looking at the general trends of the charts shown in figures 5.6a, 5.6b and 5.6c, we can see that both the normal and record mode tended to have the same amount of CPU utilisation, while the playback mode placed a roughly 10% higher load on the CPU. There were also spikes in the usage at the beginning of the playback mode which were likely when the performance data was first loaded and processed. It was expected that the record mode would place more strain on the CPU, however this was proven wrong. It was also expected that the playback mode would cause a slight increase, it was not expect to be as high as the events are being feed directly into the MT4J event system, skipping the processing of the raw input.

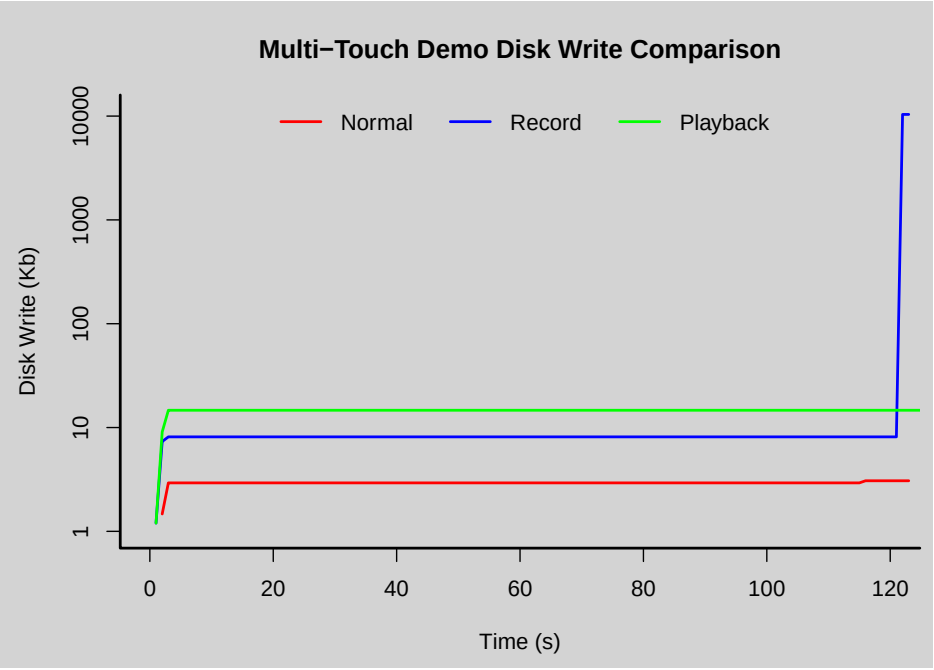
Memory Usage Impact Like with CPU usage the RAM usage was a good indicator of the load a program was putting on the system. This can be hard to gauge for a Java program, as the Java VM can cause a certain amount of over head and accounts for some of the usage seen in the charts. The charts are shown in figures 5.7a, 5.7b and 5.7c. We did observe a larger amount of data being loaded into memory during the initial stage of the playback mode. This was likely the performance data being read in and processed into the Java scheduler as expected.

We can only speculate as to the slow increase of RAM usage over time. One possibility is that it was the data from a large number of component being created for the cursor trail. It was possible that the Java garbage collector had not yet disposed of them properly. More tests would be need to be performed to determine this as RAM usage during this mode is starting to become excessive at over 10% of system memory. This may have indicated the presence of a memory leak in the MTEA tool itself.

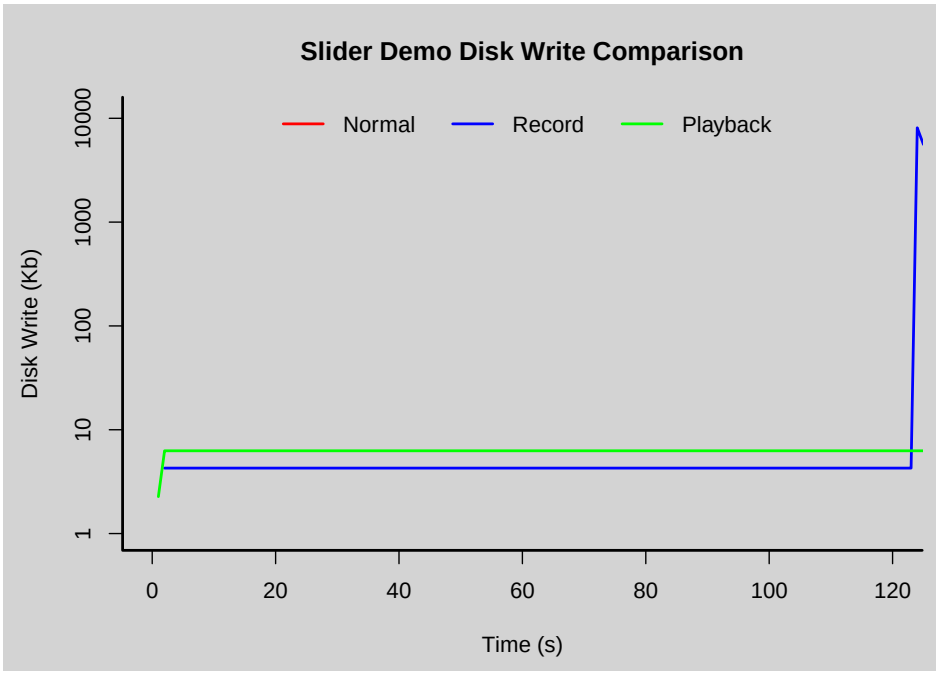
Time Impact The time data was was only very basic but provided information as to how the tests were performed. This is shown in tables figures 5.8a, 5.8b and 5.8c. The playback mode is consistently longer than the other two modes because there was a pause to load the program load before the playback was started and then to ensure that the playback was finished at the end. As a test automation tool was used to run every test it was expected to see consistent times between the normal and record mode, as well as between all of the playback modes. This was because the same test automation file was used for each normal/record test pair, as well a single file for each playback mode. By reusing the same input file we aimed to reduce the anomalies that can be introduced if a user were to perform the same tests.



(a) Multi-Touch 3D Prototype.



(b) Multi-Touch Demo



(c) Slider Demo

Figure 5.4: These charts display the accumulated amount of data written to disk during each test phases. Note that a log scale has been used. The large spike at the end of the record test indicates where the performance snapshot was written to disk.

	Multi-Touch 3D Prototype		
	Normal	Record	Playback
Before	20M	20M	21M
After	20M	30M	20M
mtexplorer.xml	-	4.0K	4.0K
performances.xml	-	608K	608K
timestamped.jar	-	9.4M	9.4M

(a) Multi-Touch 3D Prototype.

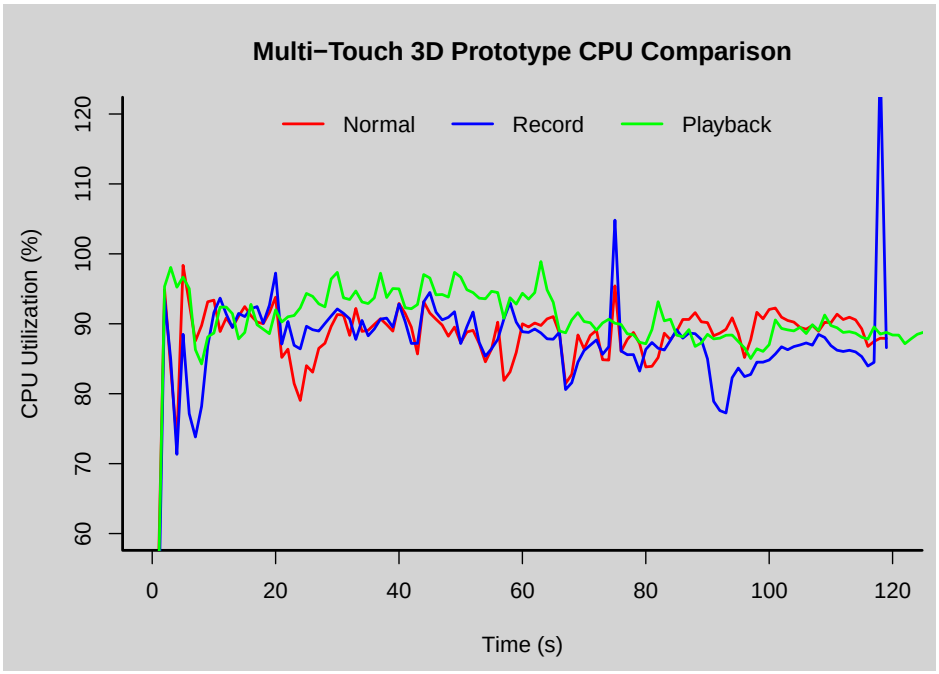
	Multi-Touch Demo		
	Normal	Record	Playback
Before	19M	19M	20M
After	19M	29M	20M
mtexplorer.xml	-	4.0K	4.0K
performances.xml	-	776K	776K
timestamped.jar	-	9.4M	9.4M

(b) Multi-Touch Demo.

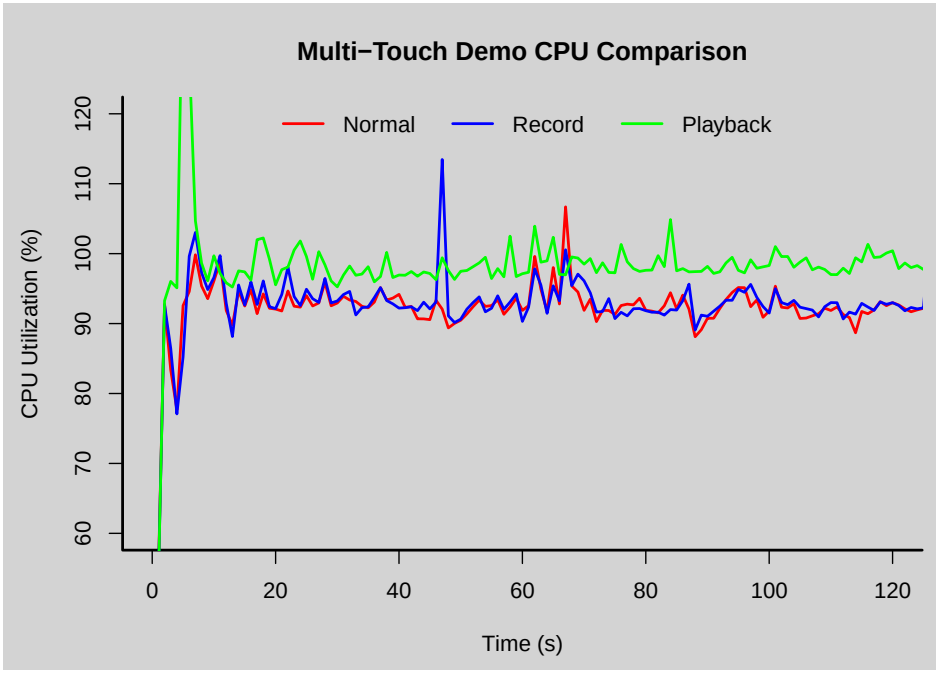
	Slider Demo		
	Normal	Record	Playback
Before	19M	19M	20M
After	19M	29M	20M
mtexplorer.xml	-	4.0K	4.0K
performances.xml	-	844K	844K
timestamped.jar	-	9.4M	9.4M

(c) Slider Demo.

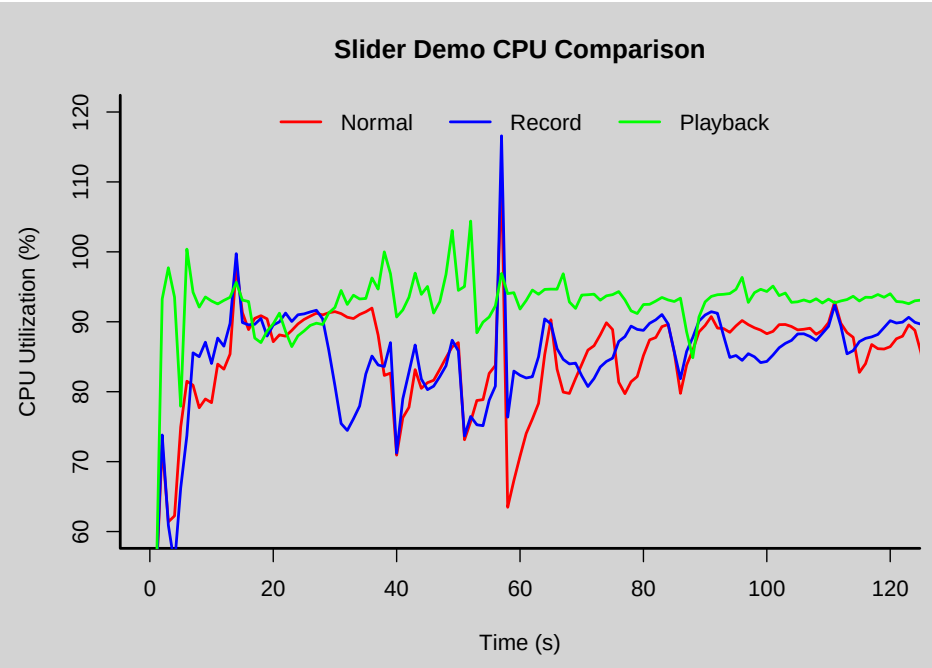
Figure 5.5: These tables show the average folder size before and after each test run, along with sizes of some common file. We can see that all the project increase by around 50% after a the recording test run. This is due to the creation of a new snapshot file and the export of the input data.



(a) Multi-Touch 3D Prototype

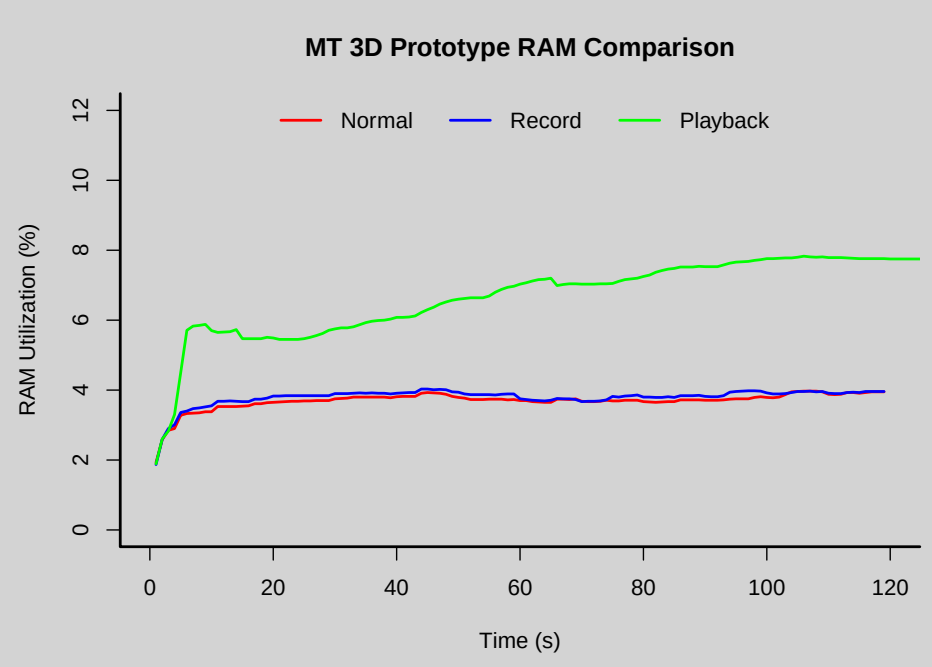


(b) Multi-Touch Demo

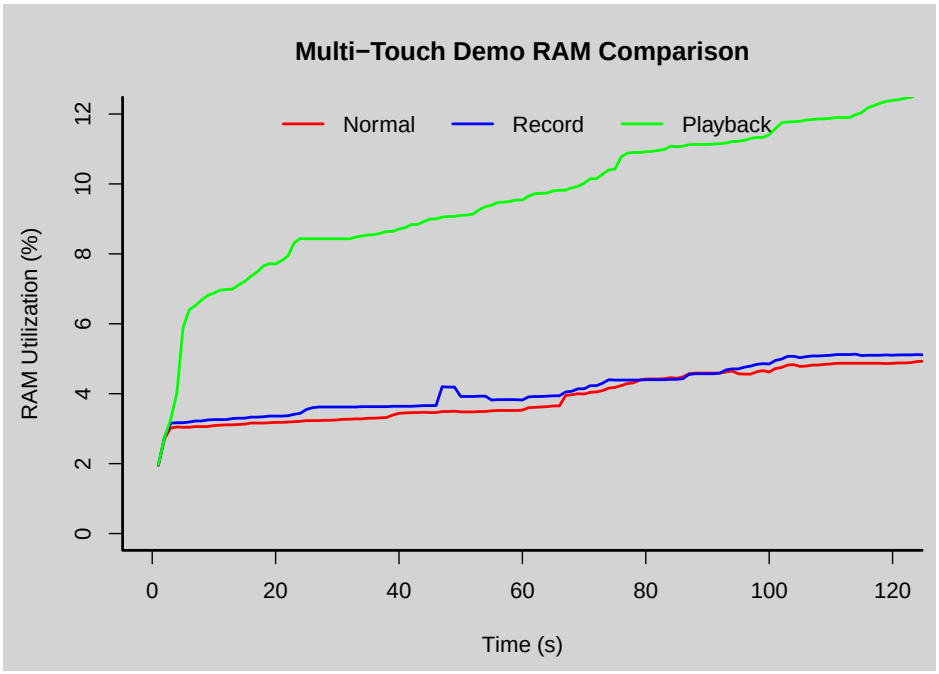


(c) Slider Demo.

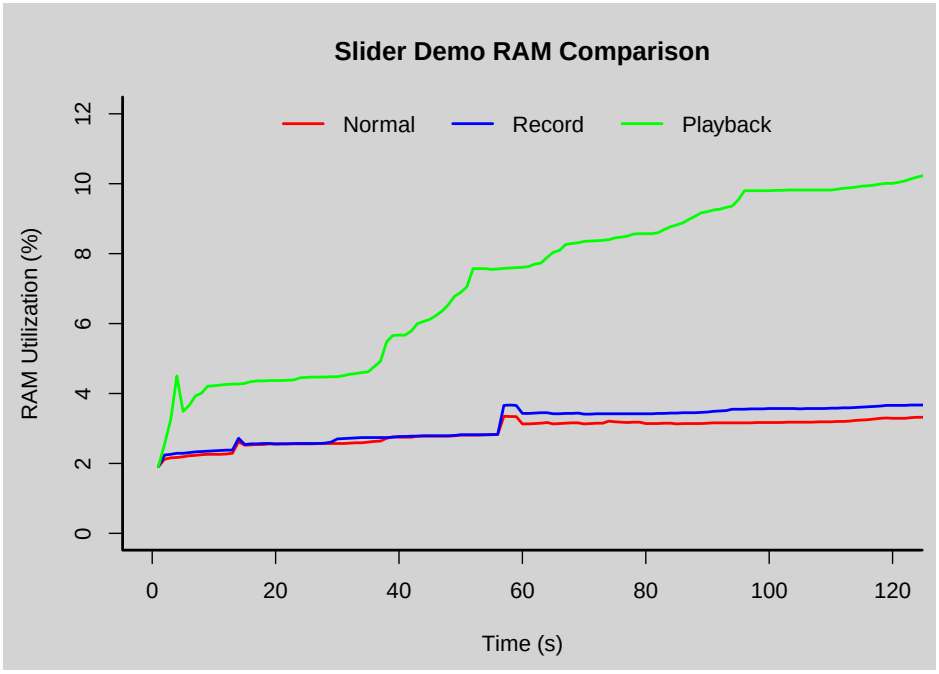
Figure 5.6: These charts shows CPU usage for each of the three test phases. While the information shown is volatile, we can see a clear trend of the Normal and Recording mode having the same usage and the Performance mode being on average 10% above these two.



(a) Multi-Touch 3D Prototype.



(b) Multi-Touch Demo



(c) Slider Demo

Figure 5.7: These charts display the amount of RAM utilisation during each of the test phases. The slow increase of usage over time could be a potential problem for the MTEE tool.

	Multi-Touch 3D Prototype		
	Normal	Record	Playback
User	0m 30.667s	0m 30.253s	0m 39.710s
System	1m 16.333s	1m 15.917s	1m 24.130s
Total	1m 59.935	1m 59.822s	2m 19.246s

(a) Multi-Touch 3D Gesture prototype.

	Multi-Touch Demo		
	Normal	Record	Playback
User	0m 53.460s	0m 53.840s	1m 34.877s
System	1m 5.177s	1m 6.117s	0m 42.230s
Total	2m 8.432s	2m 8.448s	2m 19.206s

(b) Multi-Touch Demo.

	Slider Demo		
	Normal	Record	Playback
User	0m 23.253s	0m 24.471s	0m 50.387s
System	1m 25.913s	1m 26.753s	1m 18.290s
Total	2m 10.106s	2m 10.106s	2m 19.097s

(c) Slider Demo.

Figure 5.8: These tables show the average run time during each phase of the performance test. The time indicate both the time the program spent interacting wiht the user and interacting with the system.

5.1.6 Discussion

By returning to the non-functional requirements from section 2.4.2 we could determine if the requirements could be met by the performance data.

We could see from the CPU and RAM usage charts that system load was not increased during the record mode, but did increase some what during playback mode. This reasonably meets non-functional **Require-**

ment 1, as the user could use record mode with little impact on the program to record their performance. During the playback of the performance the system would handle all of the input so it would not matter as much if the program was less responsive. We saw these same utilisation patterns across all three test applications showing that the tool could scale from small applications to at least medium sized ones, in the case of the Multi-Touch 3D Gesture Prototype. This showed how the MTEE prototype meets **Requirement 2**.

Requirement 4 requires that an reasonable amount of disk space was used by the tool. We could see from the disk usage tables that the JAR file for each of the test phases, along with the total project sizes, are roughly the same. This showed that tool had a small foot print relative to the project size. The one area which could have been improved is with storing a snapshot of the project as a JAR file. This increased most projects by 50% of their disk usage and would only increase as more snapshots are created. An alternative approach would be to look at packaging the users code via a different method. This would remove the overhead present in every JAR file and greatly reduced the size that needs to be stored. While we would lose the portability of the JAR format it would not be an important factor for the project, as it is unlikely the user will be moving the MTEE data and snapshot JAR files around.

Requirement 5 could be gauged by looking at the CPU and RAM usage in the initial stages of the playback mode, along with the same data at the end of the recording mode. These were the two positions where the MTEA tool actually read or wrote data to disk. We could see from the performance data that there were spikes in usage for both, but they were short lived. The written data also showed that the tool was able to write all the data for a performance and snapshot in a very short amount of time, so would barely be noticeable to the user.

5.2 Cognitive Walkthrough

A cognitive walkthrough is a type of formative evaluation [22], where instead of having actual users test the system, the developers and designers perform the tests themselves. The users are pulled from each persona and their thought process represented by a number of questions, which are used to evaluate each action that is performed. This allows each task to be directly scrutinised so problems can be found with the interface and how it works. A cognitive walkthrough is based on the assumption that a user will explore the interface to learn how to use it. It is also intended for first-time users of the system, which was ideal for evaluating this project.

Each task was completed by the evaluators as a series of actions they performed in order. Each action did not represent a specific path, but rather an end result or goal (e.g. Create a new file). This imitated the user exploring the interface for the solution. The questions below were used during each action to evaluate if the user was able to accurately complete the goal for that action. The questions were designed to reflect the process a user would follow when exploring the interface [66] [77].

Each action undertaken by the evaluators used these four questions:

- Will the user be trying to achieve the right effect?
- Will the user know the correct action is available?
- Will the user realise it is the correct action?
- If the correct action is taken, will the user understand the feedback and recognise that progress is being made towards the goal?

The cognitive walkthrough provided a means for the evaluators to imagine the behaviour of entire groups of users. This allowed many more problems to be uncovered than if a single unique user had been used over a single test session.

5.2.1 Task Analysis

In order to perform the cognitive walkthrough a series of task were chosen for each user persona. The task needed to represent something the respective user would have performed, the idea being that the evaluators were to take on the persona of that user and try to behave as that user would. These task were designed based on the context scenarios designed for each persona in chapter 2, section 2.3. [2]

Each task provided an overall goal and the actions required for the user to complete it. A list of optimal steps for each action were also supplied to the evaluators. These steps represented the ideal path the user would discover when completing the goal. The evaluators should also know these steps, as they were not judging their own ability to use the interface. The primary interest was in finding what problems the user would encounter and not how they would attempt to solve them [50].

Five evaluators were used to perform the coginitive walkthrough. All five evaluators were members of a post-graduate Human Computer Interaction research group from Victoria University of Wellington.

Student Persona Task

In this task the user imported the MTEA tool into the mt4j-explorer sample project and used it to create and examine the recorded gestures. This also required activating the MTE perspective within Eclipse.

Action 1 Activate the MTEE perspective.

Step 1 Select the 'Open Perspective' icon in the top right corner, a menu will appear.

Step 2 Select 'Multi-Touch Explorer' and press 'OK'.

[2] It should be noted the some of the tasks involved using the Eclipse interface. For these sections only problems that were controllable by the MTEE project where

Action 2 Import the MTEA tool into the Eclipse project.

- **Step 1** Right-click the project, a menu will appear.
- **Step 2** Enter the 'New' option and select 'MTE Project', the MTEE Project Import Wizard window will appear.
- **Step 3** Select Browse, a file view of available projects will appear.
- **Step 4** Navigate the folder tree to find the correct projects start file.
- **Step 5** Select the start file and select 'OK'.
- **Step 6** Select 'Finish'.

Action 3 Display the scene tree for the current project in the Component View.

- **Step 1** Select the 'MTE Components' tab on left-most panel.
- **Step 2** Expand the 'Multi-Touch Gesture Example Scene' entry.

Action 4 Run a preview of component 120.

- **Step 1** Right-click the entry '120: unnamed rectangle', a menu will appear.
- **Step 2** Select 'Preview', the program will run.

Action 5 Interact with the program then save the recorded gesture information.

- **Step 1** Drag, Rotate and Scale the highlighted component.
- **Step 2** Press the 'S' keyboard key, a success message will appear in the console output.
- **Step 3** Close the program by click the 'x' icon in the top right corner of the window.

Action 6 Examine the gesture data for the performance that was just recorded.

Step 1 Select the 'MTE Performances' tab in the right-most pane.

Step 2 Select the 'Refresh' option.

Step 3 Expand the most recent performance, then 'Multi-Touch Gesture Example Scene' entry, then '120: unnamed rectangle'

Step 4 Expand the Drag, Scale or Rotate processors.

Professional Persona Task

As the student persona task above also covers scenario 1 to 3 of the professional persona these will be skipped. This then assumes the user has import the MTEA tool, opened the MTE perspective and has both the Components and Performances views ready to go. The goal for this task will be for the user to make two different performances for their programs entire scene, then play both of these performances back. It will also use the mt4j-explorer project.

N.B. It is the intention that between the repetition of Action 1 and 2 that the user would modify the behaviour of their code before creating another performance. While the actual editing of the code falls out of the scope of this cognitive walkthrough, this would still require that the user update the data used for the MTE views.

Action 1 Launch a full preview of the project scene - update the project data if necessary.

Step 1 If the program has been modified, launch the StartMT-Explorer file through Eclipse.

Step 2 Looking at the Component View on the left-most panel.

Step 3 Right-click on the scene entry in the list, a menu will appear.

Step 4 Select the 'Preview' option, the program will launch.

Action 2 Save a gesture performance and then repeat from Action 1.

Step 1 Interact with a number of on screen components.

Step 2 Press the 'S' keyboard key, a success message will appear in the console.

Step 3 Close the program by click the 'x' icon in the top right corner of the window.

Step 4 Repeat from Action 1 Step 1 down to this point, then continue with Action 3.

Action 3 Examine the recorded performance in the Performances view.

Step 1 Select the 'MTE Performances' tab in the right-most pane.

Step 2 Select the 'Refresh' option.

Step 3 Expand each of the performances.

Action 4 Replay the earliest performance.

Step 1 Select the performance with the earliest time stamp.

Step 2 Right-click the performance, a menu will appear.

Step 3 Select the 'Replay' option, the program will launch will changes.

Step 4 Press the 'P' keyboard key, the performance will begin playback.

Step 5 When playback is finished, close the program by clicking the 'x' icon in the top right corner of the window.

Action 5 Replay the latest performance.

Step 1 Select the performance with the latest time stamp.

Step 2 Right-click the performance, a menu will appear.

Step 3 Select the 'Replay' option, the program will launch will changes.

Step 4 Press the 'P' keyboard key, the performance will begin playback.

Step 5 When playback is finished, close the program by clicking the 'x' icon in the top right corner of the window.

5.2.2 Walkthrough Results

When the cognitive walkthrough had been completed the problems were collated and a severity level system was created to rank the problems that were uncovered. Solutions to each problem were also proposed. The severity of each problem was then used to determine which solutions could take precedence should there be an overlap in functionality. This create a number of goals for the development of the tool going forward and are presented in the following chapter. The three severity levels have been described here:

Severity 1 This level represented a major problem with how the project was designed and could possibly halt the user's progress. A problem of this level could require a redesign to the way the project functioned.

Severity 2 This level represented a less severe problem which would still hinder a users progress. This type of problem would require a small redesign to a particular function.

Severity 3 This level represented a minor problem which could briefly confuse the user. This type of problem would either be left or only require a minor change.

Student Task Results

The problems presented here were uncovered during the evaluation of the student persona's tasks.

Action 1 Activate the MTEE perspective.

Problem 1 - Severity 3 Once the new perspective had been enabled the layout of the workspace changed. This indicated to the user that they had performed the correct action, but they were not presented with the new views immediately. Instead a number of the standard Eclipse views were displayed by default. The new MTE views could then be accessed through tabs in the workspace display.

This is only a severity 3, as the intended next step is that the user would use the Eclipse Project Explorer view to import the MTEA tool into their project. If the MTE Components view was shown by default then this Project Explorer view would be hidden. Instead the MTE Performances view could be shown, as this was laid out on a separate part of the workspace. Text could be added to the blank performances view to prompt the user of the next step.

Problem 2 - Severity 3 The icon used in the Eclipse perspective list for the MTE perspective was not indicative of multi-touch. A default icon was instead used. A newer icon which signified multi-touch should instead be used. This would help the user identify the correct perspective. This was ranked severity 3 as it would be a minor cosmetic change.

Action 2 Import the MTEA tool into the Eclipse project.

Problem 1 - Severity 2 The task specifically said 'Import the MTEA tool' as this was closest to the technical action being performed. This was not representative of how the user would view the action. The user would would instead be looking to simply use the MTEA tool, and not import or create a new MTEE project. In the evaluated version the correct action was located under the 'New' option in the menu. This menu was accessed by right-clicking on an entry in the Project Explorer view. Neither using the 'New' option or the 'Import...' option was intuitive from a user's point of view.

The best solution would to add a separate entry into the root of the Project Explorer menu. The menu entry would be named to best identify it with the MTEE project. This would provide a obvious choice for the user, and a future extension point for any other features that may be developed. As this change would require a minor redesign of how the menu entry was structured it was given severity of 2.

Problem 2 - Severity 3 As the Student user was very new to using MT4J they were not overly familiar with the MT4J start files. These were require to start up an MT4J application and were required by the MTEA import wizard to generate the MTEE start file. This meant the user had difficultly selecting the correct file. This would cause the wizard to generate the file incorrectly, achieving the wrong effect intended by the user.

More text should be added to the MTEE Import Wizard to indicate what the start file should be. If the wrong file was used then the generated files would fail to run, but not other harm was done to the user's project. The user was able to rerun the wizard with the correct file to resolve the error. Due to these reasons these problem is given a severity of 3.

Problem 3 - Severity 3 After the user pressed 'Finish' on the MTEE Import Wizard a number of things happened in the background. The process was completed when the wizard dialog window closed. A 'Build Successful' message appeared in the Eclipse console view. While the user realised that they had performed the correct action by using the import wizard, there was no other indication of what had been performed or what new actions could be performed. The console output could easily be obscured or hidden entirely if the user had changed the layout of their workspace.

A message should be added after the wizard is finished to tell the user that the process was successful. This message would also prompt the user to explore the MTE Component view. As this is only the addition of more information it has been given a severity of 3. While the import also made changes to the user's project by adding additional files, the user was not required to use these files directly when operating the MTEA tool. This meant the solution did not need to involve drawing the users attention to these changes.

Action 3 Display the scene tree for the current project in the Component View.

Problem 1 - Severity 2 When the user switched to the MTE Component view the scene tree that was displayed was based on which project they had selected in the Project Explorer view. This was not obvious to the user and it made it easy for the user to miss-click and select the wrong project. This would cause the user to become confused when viewing the component tree.

Two possible solutions were developed. The more difficult solution involved switching the project selection to use the cur-

rently active Java file in the editor view. This would make the behaviour consistent with how other views make their selections. This would be sufficient for users that are familiar with Eclipse, such as the target personas Tom and Harold.

A simpler solution would be to switch the position of the MTE Component view and the MTE performance view. This would allow the MTE Component view to be displayed as the user was using the Project Explorer. This would allow the user to discover how the view made its selection for themselves.

As the first solution would require a change to the way the MTE Component view was programmed, this was given a severity of 2. Both solutions could be further appended by adding a label to the MTE Component view which indicated the currently active project.

Action 4 Run a preview of component 120.

Problem 1 - Severity 3 The user may not have known that the correct action was available as it required that they right-click on a specific component. If they were to select the wrong component then they may perform the action incorrectly.

Using the right-click action to display a context menu was a standard operation amount user interfaces. It was felt that the user would not have difficulty finding this option, however functionality could be improved by adding a default action via a double click command. While the implementation of this feature would be non-trivial, it was believed it would not hinder the operation of the tool if it was not included. For that reason it was assigned a severity of 3.

This change would benefit both of the MTE views. The MTE Component default action would be to preview the selected com-

ponent and the MTE Performances default action would be to replay the selected performance.

Problem 2 - Severity 3 When a preview of a component was run, that component would be highlighted in the program display. All of the other components were disabled and made transparent. In the event that the colour used for highlighting was similar to the colour of the background, or to the component itself, it would become difficult for the user to distinguish which component was being previewed.

To help alleviate this the colour of the component would be inverted to better emphasis it. The background and disabled component would be processed through an image filter to add such effects as blurring or desaturation. This would help the user better distinguish each component. As this problem only occurred under specific circumstances it was given a severity of 3.

Action 5 Interact with the program then save the recorded gesture information.

Problem 1 - Severity 2 When the program was running the user did not know they had to press the 'S' keyboard key to save their performance as their was no indication of this. While the use of the 'S' key for saving is not uncommon, the user had no way of discovering this short of trying every key.

This problem did not represent a problem with the function itself as the key used was intuitive. This problem was instead caused by lack of a user manual to explain how to use the project. While the cognitive walkthrough emphasised the user exploring the interface to find the solution, this problem presented the need to also provide documentation. The solution then would be to

provide a smaller user manual to the user, explaining the various functions of the tool. This would aid in reducing confusion for any of the tasks. When a manual had been developed it would be displayed in the editor view when the user first launches the MTE perspective. As this problem uncovered a oversight in the project it has been given a severity of 2.

A further suggestion was to change the key to a combination such as 'Ctrl + S'. It was felt that this was a more standard combination for the save command and would be less prone to accidental execution.

Problem 2 - Severity 2 When the 'S' key was used to save a performance only a small amount of feedback was provided in the Eclipse console view. This would easily be obscured or entirely missed by the user. This would cause undesired results, as the user would attempt to save the performance multiple times or forgo creating a recording entirely.

To solve this a message box would appear when the performance had been saved correctly. This would detail what had been saved and provide the user with information on how to access the performance. Due to the unintended side affects of these problem it has been given a severity of 2.

Problem 3 - Severity 1 As the preview mode disabled all but the selected component, it would end up disabling controls used to close the program. If the user was running in a windowed mode then they were able to close the program with the standard window controls. However, if the user ran the project in full screen mode, they would become locked into the program. This would then require system intervention to stop the program.

To solve this a dedicated quit command would be added to the MTEA tool. This would use the 'Ctrl + Q' key combination to close the program. This problem represented a major oversight into how the user's programs were handled. While the solution would not require a major redesign of the project, the ramifications of not including it give it a severity level of 1.

Action 6 Examine the gesture data for the performance that was just recorded.

Problem 1 - Severity 3 After the user had recorded a performance and switched to the MTE Performances view they were forced to click the refresh button. If the user did not then the view was either be blank, or not up to date with the latest recording. This was a problem as the user would assume that they had failed to save their performance.

A solution to this problem would be to monitor the performance data file for changes and update the view when they occurred. This would require a major change to the functionality of the view to add the file watching functionality. As this is a major change that is not crucial to the operation of the tool, it was felt that it would be unnecessary at this time. Instead it was assumed that having the refresh option available indicated to the user that this was the expected action. It was felt that the user would think to perform this action before panicking. For these reasons this problem was given a severity of 3.

Problem 2 - Severity 3 When the user expanded the data in the performance view, they were required to expand down at least four or five nodes to get to to get to where the data for a particular component was. While the path to locating this data was intuitive, as they had an understanding of how the components

were structured, it was time consuming to access the data. This made the data less enticing.

The best solution would be to have a separate tree for each performance node, that only listed the components that actually recorded data. This would not be nested, as they are in the scene tree, but instead each component would be displayed at the top level. This would only require the user expand the node of the component they wanted to view. As this would only be a change in the way the data was displayed it was assigned a severity of 3.

Professional Task Results

The problems presented here were uncovered during the evaluation of the professional persona's tasks.

Action 1 Launch a full preview of the project scene - after having made changes to the scene.

Problem 1 - Severity 1 The data file used by the MTE Component view is only updated every time the MTEA tool is run. This meant that if a user was to make a change to their project it would make that file obsolete. The user was required to run the project directly from the MTEE start file to regenerate the data. An error would occur when the user attempted to run a preview on a component that no longer existed if they had not performed this step.

This problem was given a severity of 1 as it represented a major oversight to functionality of the MTE Component view. It was never intended for the user to interact with the MTEE start file at any time and this problem would stop the users ability to use the MTE Component view altogether.

To rectify this problem the refresh button on the MTE component view would be changed to include the update functionality. This would run the program to generate the new data file, then refresh the information shown in the view. While this solution would not stop the user from attempting to run their program with an obsolete data file, it would provide them with a reasonable means to update this data. To help remind the user to update their data file, a time stamp of the last update would be displayed. This would allow the user to assess when they last made changes, if those changes would affect the component structure and then update accordingly.

An alternative that was considered would be to monitor the entire project for changes and update the component data file as they occurred. This would be a cumbersome solution, as even a simple change would force a refresh of the data. It was felt that such a solution would reduce the performance of the tool and frustrate the users, so instead the previous solution was deemed best.

Action 4 Replay the earliest performance.

Problem 1 - Severity 2 After they user had launched their program in replay mode the user was required to push the 'P' keyboard key. There was no indication that this key would be required to begin the performance.

This problem fell in line with the problems found during the student task. In particular the indication to use the 'S' key to save a performance. The same solution would be used with this problem, including the suggest to use a key modifier. This would change the replay to require the 'Ctrl + P' key combination. As such it receives the same severity of 2.

Another solution would be to remove the requirement of pressing a key altogether. Instead the performance would be launched as soon as the program was loaded. While this was a valid solution, it was felt that this removed control from the user and there were situations where the user may desire a delay before playback. Such situations could include the playback to two performances at the same time, side-by-side.

Problem 2 - Severity 3 When a playback was underway, there was no timing indicator as to when the playback would be finished. As the playback happened in real time, any pauses in this playback would cause the user to think it had finished and that they should continue with their next action. This would cause the user to miss some parts of the performance.

To solve this, a sound would be played to indicate the beginning and end of the playback. While this would require a minor change to the playback functionality, it has been given a severity of 3 to indicate it's mostly cosmetic nature.

Chapter 6

Summary

Once the evaluation was complete it was possible to assess if the design and implementation of the project had meet the original goals and requirements from section 2.4. The performance metrics were used to determine if the non-functional requirements had been meet, and the feasibility of the whole project. The cognitive walkthrough helped to validate if the persona scenarios were possible and if the designs had been met correctly by the implementation.

6.1 Performance Metrics

By using the performance metric data it was possible to show that 4 of the 5 non-functional requirements had been met by the project. While the status of these requirements could change as the MTEA tool changes, it was possible to evaluate there progress to better ensure they continue to be met in the future. Requirement 3 was not discussed here, as this requirement concerned data validation and could not be evaluated through performance metrics but through implementation instead. Meeting this requirement was discussed in Chapter 3, section 3.2.4.

Requirement 1 This requirement was best measured via the CPU and RAM usage metrics. It was found that using the tool in recording mode did not add any additional strain on either of these resources. This was an improvement from the expectations, where it was assumed that the recording would increase the CPU due to the tool processing every gesture. The RAM usage was also expected to show a larger increase, due to the gesture data being accumulated and stored in memory. Both of these assumptions were proved false, with the CPU usage roughly the same and only a slight increase in memory usage over time. It was believed that this performance improvement related to the proper use of the MT4J listeners and processor, so that the data could be gathered as efficiently as possible.

The playback mode was a different story; while it was also expect for there to be a slight increase in CPU and RAM usage, they were not expected to be as large as shown. The CPU usage was at most 10% higher than when running the program in normal or recording mode. The RAM usage showed that the up to five times the amount of RAM was being used in the playback mode. This was very unexpected and could indicate a memory leak in the design of the playback mode. Possible culprits could the creation of objects to represent the cursor trail or how objects are stored and scheduled using the Java timer. More extensive testing would need to be done to properly determine the cause of this problem and would be a priority for the project going forward.

While it was found that the recording mode had meet the performance requirements, the playback mode was also found to be perfectly responsive from the user point of you. For these reasons it was believe that the requirement had not been violated at that stage. A requirement such as that could never be met entirely, instead the use of the repeated performance evaluation would serve as a means to continually monitor and tweak this during the future of the project.

Requirement 2 This requirement required the evaluation of a number of different applications that represented different complexity levels. To evaluate this the three programs were chosen based on complexity. The slider demo represented the most basic application, a single control built o top of the MT4J framework. The multi-touch demo provided a multitude of possible actions to demonstrate all of the actions, it was chosen to represent a more complex user interface built with MT4J. Finally, the 3D gesture prototype was chosen as it was a real application which include a user interface and the underlying functionality of a real application. By comparing the metric results across these three program its possible to determine that the MTEA tool did scale well across a range of programs. RAM usage for each program was comparable, with normal and recording mode requiring around 3% and he same memory increase during playback mode seen in all programs. CPU usage was particularly interesting as the multi-touch utilised the CPU more on average than the 3D gesture prototype. The remaining metrics did not show a trend in usage that followed the level of complexity in the program. For these reasons it was believed that this requirement had been met.

Requirement 4 Evaluating this requirement relied solely on the disk usage metric as it compared the before and after file sizes. It was interesting to see that the JAR files that was created after the recording mode run is close to the same size for each program. This was because despite the difference in the code based size for each project, the standard libraries that Java includes in all of the JAR files, plus the MT4J framework make up the vast majority of the space. Files such as images and sounds files used by a project had not been included. This meant that even a large project may not see an increase in the size of a JAR.

To further improve on this a method of storing only the compiled class files for a project would be investigated. As the snapshot JAR file are not intended to be portable and distributed by the user it would not require

the internal version of the Java library, or the MT4J framework. These would instead be linked at runtime from the main project itself. This would drastically cut down the size of a snapshot and help to better meet the requirement in the future.

Requirement 5 Requirement 5 was a multi part requirement that was harder to evaluate. The first part involved evaluating that the program was not waiting on disk I/O to import and export data. Due to a problem in the way the data for disk reads was measured it could not be used to evaluate how quickly the program would read in data. This was instead approximated by looking at the file sizes of the data that was known to be read in. In particular the performances data file would be read in as a whole, by both the recording and performances mode. This file was an average of 742 kilobytes across all programs for a single performance. This would increase as more performances were recorded. We knew from the system information that the hard drive can read data at a speed of 131.36 Megabytes a sec, which meant a performance file would need to become substantially large before disk I/O became a problem. Any performances issues are then more likely to occur when processing this data file as it would need to be held in memory.

The correct data for disk writes was able to be measure and it shows that there is no issue with recording a performance to disk and making a copy of the snapshot JAR file. As was shown by the charts, the spike in data written happens over a single data point and would not be noticeable by the user.

6.2 Cognitive Walkthrough

The cognitive walkthrough used experts to evaluate both the user interface for the MTEA tool, and also the functionality of the tool. The tasks used were designed based on the scenarios developed for each persona. This

provided a proxy method of evaluating how a group of users would perform using the tool. The results uncovered a number of severe problems which needed to be addressed with urgency. While solutions were suggested for each individual problem, making the most efficient improvements required these solutions be collated into work plan. This work plan is presented here:

6.2.1 Proposed Changes

Change 1 The functionality of the refresh button on the MTE Component view would be changed to allow the user to properly update the data shown in the view. The new function would rerun the MTEE build file to create and up-to-date snapshot of the project. It would also rerun the project with the '-X' command line flag to run the MTEA tool, generate the updated component data file, then close the application. This change would leave it up to the user to update the data as needed. To help the user better assess when an update would be required a label showing the last update time will be added to the component view.

Change 2 The keyboard keys used to interface with the MTEA tool would be standardised to use the 'Ctrl' key modifier. This would make the combinations more compatible with how other applications handle them. Using a key modifier also helps to bring down the occurrence of miss-pressed keys. This change would include the addition of the 'Ctrl + Q' combination to close the program running the MTEA tool.

Change 3 The position of the menu entry for launching the MTEE import wizard will be moved. A separate MTEE project sub-menu will be added to the root Project Explorer menu. The entry to launch the import wizard will be moved to this sub-menu. The name of the entry will be changed to 'Initialise MTEA Tool'. This will

provide a clear path to the user and and extension point for further functionality to the tool.

Change 4 The layout of the workspace would be changed to display more information to the user and work more efficiently with the work flow of the tool. First the position of the MTE views would be switched to be position the MTE Component view on the right of the workspace, with the MTE performances view on the left. Then the MTE components view would be brought to the front of its position instead of the Eclipse Outline view. Finally, labels would be added to both the project and component view indicating which project is current selected.

Change 5 When the MTE perspective is first activated a new view would fill the editor position on the workspace. The basis for this view would be to contain documentation for how to use the tool. This documentation would cover how to import the tool into an existing project, how to operate the tool the key combinations, and how the data in the each of the views are displayed. Other visual aides such as arrows would be used to indicate positions of the new views which have been added to the workspace. This change would solve a number of problems by helping the user to identify the correct steps when using the tool, and also provide feedback that the tool has been installed correctly.

To better imagine these changes figure 3.1 was modified to show the new layout. This is shown in figure 6.1.

6.3 Future of the Project

Both the performance evaluation and cognitive walkthrough uncovered a number of problems that will need to be solved to move the project forward.

6.3.1 Improvements

While the proposed changes represented the immediate action that the project needed to take, both the feature design and evaluation presented a number of other issues which would eventually be addressed. These take the form of both redesigns to the project as well as future functionality.

- A redesign of the icons and labels used in the project. This will help the user better distinguish information in the MTE views and help maintain consistency with the naming of views and the perspective.
- Further performance analysis to find the cause of the RAM usage in playback mode and to determine its cause. If a memory leak is found then it will be fixed, otherwise solutions for lowering the memory will be explored.
- An alternate method of storing a project snapshot so that only the class files are stored, not the libraries. This would either continue using the JAR archive format or another option of storing and using the snapshot will be explored.
- A method of measuring time when using the playback mode. This will either involve an overlay on the users program or other types of feedback such as sound.
- The ability to use a video with an overlay mode to create a static replay of a performance. The idea would be to allow the user to move backward and forward in time to better assess the behaviour of their program.

6.4 Contributions

The completion of the Multi-Touch Explorer Environment (MTEE) resulted in three contributions:

- The design of a tool to record and replay user interactions with a multi-touch program. This included the analysis and development of two primary personas related to the target users.

- A proof-of-concept prototype for the MTEE. The prototype integrates the MTEE into the Eclipse IDE and focused on projects using the Multi-Touch for Java framework. The MTEE prototype consists of two parts, the plugin for the Eclispe IDE and the Multi-Touch Explorer Analysis (MTEA) tool.

- An evaluation of the Multi-Touch Explorer Environment. This included a set of performance metrics and test suite to allow continual testing of the non-function requirements. It also included a Cognitive Walkthrough [22] using expert users, resulting in a list of changes and improvements that can be made to the project going forward.

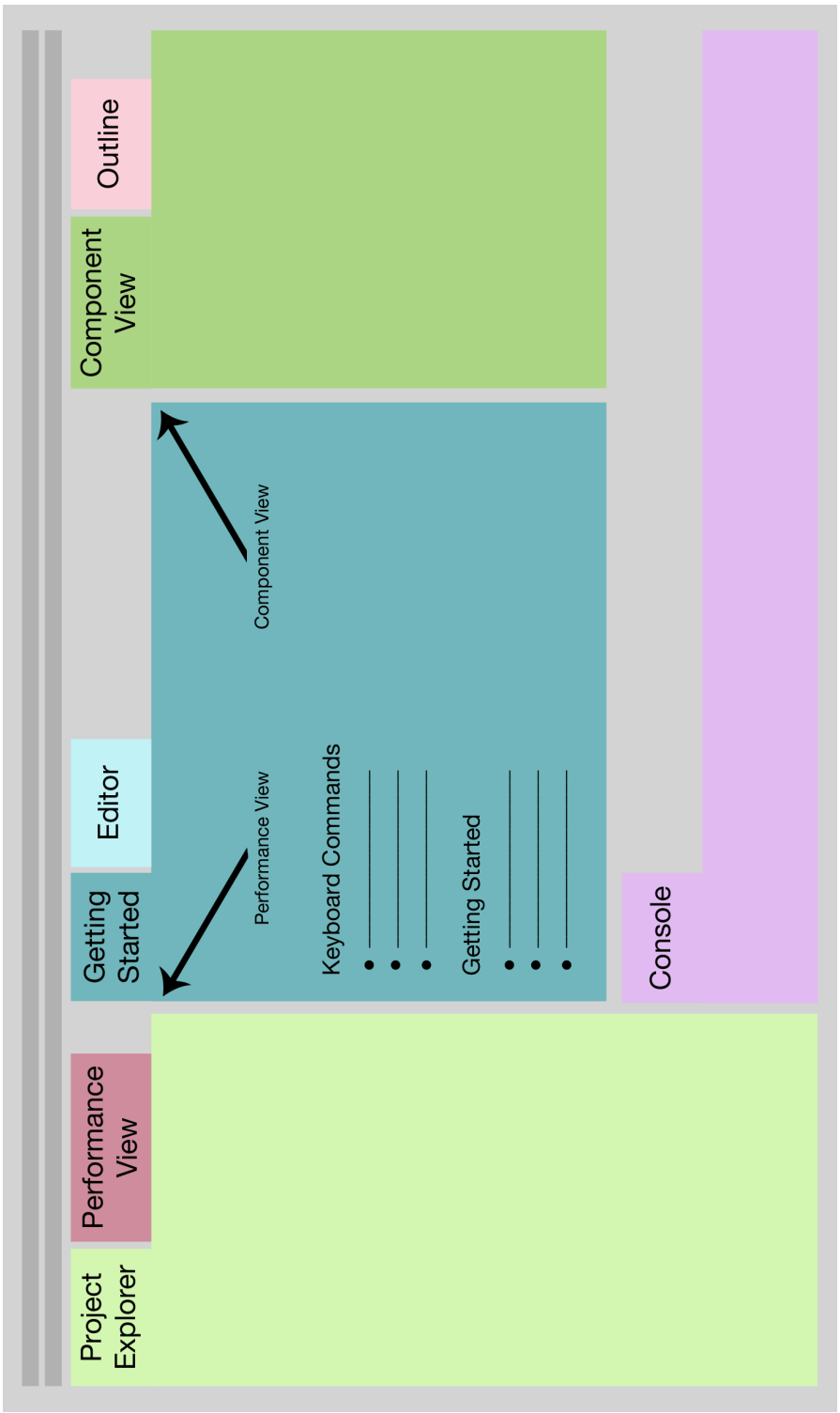


Figure 6.1: The Eclipse workspace layout with changes.

Bibliography

[1] Computer language benchmarks game - which programs are fastest. http://benchmarksgame.alioth.debian.org/u32q/which-programs-are-fastest.php?gcc=on&gpp=on&java=on&csharp=on&python3=on&jruby=on&php=on&perl=on. [Online; accessed 30-March-2013].

[2] Kivy. http://kivy.org/#home. [Online; accessed 31-March-2013].

[3] Mono - cross platform, open source ,net development framework. http://www.mono-project.com/Main_Page. [Online; accessed 31-March-2013].

[4] Algorithmic performance comparison between c, c++, java and c# programming languages. Tech. rep., Cherrystone Software Labs, 2010. http://www.cherrystonesoftware.com/doc/AlgorithmicPerformance.pdf.

[5] Pydev. http://pydev.org/, March 2012. [Online; accessed 30-March-2013].

[6] A., G. J. . J. B. . S. G. . B. G. . B. The java language specification. http://docs.oracle.com/javase/specs/jls/se7/html/index.html, February 2013. [Online; accessed 30-March-2013].

[7] ADOBE SYSTEMS INC. Adobe captivate 6. http://www.adobe.com/nz/products/captivate.html. [Online; accessed 8-April-2013].

[8] AMSDEN, J. Levels of integration - five ways you can integrate with the eclipse platform. http://www.eclipse.org/articles/Article-Levels-Of-Integration/levels-of-integration.html, March 2001. [Online; accessed 8-April-2013].

[9] AMSDEN, J., AND IRVINE, A. Your first plug-in - developing the eclipse hello world plug-in. http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html, January 2003. [Online; accessed 1-April-2013].

[10] ANISZCZYK, C. Plug-in development 101, part 1: The fundamentals. http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/, Febraury 2008. [Online; accessed 1-April-2013].

[11] APACHE SOFTWARE FOUNDATION. Abache subversion. http://subversion.apache.org/. [Online; accessed 8-April-2013].

[12] APPLE INC. Apple reinvents the phone with iphone. http://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html, January 2007. [Online; accessed 5-April-2013].

[13] APPLE INC. Apple announces iphone 2.0 software beta. http://www.apple.com/pr/library/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta.html, March 2008. [Online; accessed 30-March-2013].

[14] ARTHORNE, J., AND LAFFRA, C. Implementing support for your own language. `http://wiki.eclipse.org/The_Official_Eclipse_FAQs#Implementing_Support_for_Your_Own_Language`. [Online; accessed 30-March-2013].

[15] BEEPA PTY LTD. Fraps - real-time video capture & benchmarking. `http://www.fraps.com/`. [Online; accessed 8-April-2013].

[16] BEYER, D., AND HASSAN, A. Animated visualization of software history using evolution storyboards. In *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on* (2006), pp. 199–210.

[17] BOWEN, K., AND PISTILLI, M. D. Student prefences for mobile app usage (research bulletin). `http://www.educause.edu/ecar`.

[18] BRAGDON, A., REISS, S. P., ZELEZNIK, R., KARUMURI, S., CHEUNG, W., KAPLAN, J., COLEMAN, C., ADEPUTRA, F., AND LAVIOLA, JR., J. J. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 455–464.

[19] CAMPBELL, C. E., EISENBERG, A., AND MELTON, J. Xml schema. *SIGMOD Rec. 32*, 2 (June 2003), 96–101.

[20] CAMSTUDIO.ORG. Camstudio - free streaming video software. `http://camstudio.org/`. [Online; accessed 8-April-2013].

[21] CHAZARAIN, G. Iotop. `http://guichaz.free.fr/iotop/`. [Online; accessed 28-March-2013].

[22] COOPER, A., REIMANN, R., AND CRONIN, D. *About Face 3: The Essentials of Interaction Design*, 3rd ed. Wiley, May 2007.

[23] COPE, D. Navigating 3D Worlds via 2D Multi-Touch Interfaces. Tech. rep., Victoria University of Wellington, 2011.

[24] DAHLSTROM, E., DZIUBAN, AND WALKER, J. Ecar study of undergraduate students and information technology. EDUCASE. http://www.educause.edu/ecar.

[25] DELINE, R., AND ROWAN, K. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 207–210.

[26] ECHTLER, F. Tisch - tangible interactive surfaces for collaboration between humans. http://tisch.sourceforge.net/#ack, September 2010. [Online; accessed 31-March-2013].

[27] ECLIPSE FOUNDATION. About the eclipse foundation. http://www.eclipse.org/org/#history. [Online; accessed 30-March-2013].

[28] ECLIPSE FOUNDATION. Rich client platform faq. http://wiki.eclipse.org/RCP_FAQ#What_is_the_Eclipse_Rich_Client_Platform.3F. [Online; accessed 1-April-2013].

[29] ECLIPSE FOUNDATION. Eclipse downloads. , March 2012. [Online; accessed 30-March-2013].

[30] ELBAUM, S., KARRE, S., AND ROTHERMEL, G. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), ICSE '03, IEEE Computer Society, pp. 49–59.

[31] FREE SOFTWARE FOUNDATION, I. Bash reference manual. http://www.gnu.org/software/bash/manual/bashref.html#index-time, 2008. [Online; accessed 28-March-2013].

[32] FREE SOFTWARE FOUNDATION, I. Gnu xnee. http://www.gnu.org/software/xnee/, January 2010. [Online; accessed 28-March-2013].

[33] FREE SOFTWARE FOUNDATION, I. Coreutils - gnu core utilities. `http://www.gnu.org/software/coreutils`, March 2013. [Online; accessed 28-March-2013].

[34] FREE SOFTWARE FOUNDATION, I. du: Estimate file space usage. `http://www.gnu.org/software/coreutils/manual/coreutils.html#du-invocation`, 2013. [Online; accessed 28-March-2013].

[35] GIT. git –distributed-even-if-your-workflow-isnt. `http://git-scm.com/`. [Online; accessed 8-April-2013].

[36] GOOGLE INC. Get the android sdk. `http://developer.android.com/sdk/index.html`. [Online; accessed 30-March-2013].

[37] GREENBERG, S., AND BUXTON, B. Usability evaluation considered harmful (some of the time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2008), CHI '08, ACM, pp. 111–120.

[38] HANSEN, T. E., HOURCADE, J. P., VIRBEL, M., PATALI, S., AND SERRA, T. Pymt: a post-wimp multi-touch user interface toolkit. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces* (New York, NY, USA, 2009), ITS '09, ACM, pp. 17–24.

[39] HEJLSBERG, A., TORGERSEN, M., WILTAMUTH, S., AND GOLDE, P. *C# Programming Language*, 4th ed. Addison-Wesley Professional, 2010.

[40] IOWA STATE UNIVERSITY. Sparsh ui - the power of touch. . [Online; accessed 31-March-2013].

[41] ISO. Information technology – programming languages – c++. ISO 14882-2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[42] KUHLMAN, D. A python book: Beginning python, advanced python, and python exercises. http://cutter.rexx.com/~dkuhlman/python_book_01.html#part-1-beginning-python, April 2002. [Online; accessed 31-March-2013].

[43] LA TROBE UNIVERSITY & UNIVERSITY OF KENT. Bluej - the interactive java environment. http://www.bluej.org. [Online; accessed 30-March-2013].

[44] LANGPOP.COM. Langpop.com - programming language popularity. langpop.com, April 2011. [Online; accessed 30-March-2013].

[45] LANZA, M. The evolution matrix: recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution* (New York, NY, USA, 2001), IWPSE '01, ACM, pp. 37–42.

[46] LANZA, M., AND DUCASSE, S. Understanding software evolution using a combination of software visualization and software metrics. In *In Proceedings of LMO 2002 (Langages et Modles Objets* (2002), Lavoisier, pp. 135–149.

[47] LAUFS, U., RUFF, C., AND ZIBUSCHKA, J. MT4j - A Cross-platform Multi-touch Development Framework. *ArXiv e-prints* (Dec. 2010).

[48] LEE, Y., AND YANG, J. Visualization of software evolution. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)* (2008), pp. 343–348.

[49] LEUNG, C. Black-sun - eclipse plugins. http://black-sun.sourceforge.net/, 2006. [Online; accessed 30-March-2013].

[50] LEWIS, C., RIEMAN, J., AND BLUSTEIN, A. J. Task-Centered User Interface Design: A Practical Introduction, 1993.

[51] LINTERN, R., MICHAUD, J., STOREY, M.-A., AND WU, X. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM symposium on Software visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 47–ff.

[52] MALLET, C. Autohotkey - automation. hotkeys. scripting. `http://www.autohotkey.com/`. [Online; accessed 8-April-2013].

[53] MCCOWN, F., AND FOUST, G. Comp 475 - mobile computing. `https://www.harding.edu/fmccown/classes/comp475-s10/`, 2010. [Online; accessed 28-March-2013].

[54] MICROSOFT. Extend visual studio. `http://msdn.microsoft.com/en-US/vstudio/ff718165.aspx`. [Online; accessed 31-March-2013].

[55] MICROSOFT. Msdn - microsoft surface 2.0 sdk. `http://msdn.microsoft.com/en-us/library/ff727815.aspx`. [Online; accessed 31-March-2013].

[56] MICROSOFT. Msdn - visual studio. `http://msdn.microsoft.com/en-us/library/52f3sw5c(v=vs.100).aspx`. [Online; accessed 31-March-2013].

[57] MICROSOFT. Microsoft launches new product category: Surface computing comes to life in restaurants, hotels, retail locations and casino resorts. `http://www.microsoft.com/en-us/news/press/2007/may07/05-29mssurfacepr.aspx`, May 2007. [Online; accessed 5-April-2013].

[58] NUI GROUP. Downloads - mt4j latest release. `http://www.mt4j.org/mediawiki/index.php/Downloads`. [Online; accessed 30-March-2013].

[59] NUI GROUP. Installation - environment installation. http://www.mt4j.org/mediawiki/index.php/Installation. [Online; accessed 30-March-2013].

[60] NUI GROUP. Mt4j - multitouch for java. http://www.mt4j.org/mediawiki/index.php/Main_Page. [Online; accessed 30-March-2013].

[61] OBJECT TECHNOLOGY INTERNATIONL, INC. Eclipse platform technical overview. Tech. rep., February 2003.

[62] ORACLE. Java se downloads. http://www.oracle.com/technetwork/java/javase/downloads/index.html. [Online; accessed 30-March-2013].

[63] PARLANTE, N. Cs193a android programming. http://www.stanford.edu/class/cs193a/, 2011. [Online; accessed 28-March-2013].

[64] Procps - the /proc file system utilities. http://procps.sourceforge.net/index.html, May 2009. [Online; accessed 28-March-2013].

[65] PYTHON SOFTWARE FOUNDATION. Python - about python. http://www.python.org/about/. [Online; accessed 31-March-2013].

[66] RIEMAN, J., FRANZKE, M., AND REDMILES, D. Usability evaluation with the cognitive walkthrough. In *Conference Companion on Human Factors in Computing Systems* (New York, NY, USA, 1995), CHI '95, ACM, pp. 387–388.

[67] SANDKLEF, H. Testing applications with xnee. *Linux J. 2004*, 117 (Jan. 2004), 5–.

[68] SEARS, A., PLAISANT, C., AND SHNEIDERMAN, B. Advances in human-computer interaction (vol. 3). Ablex Publishing Corp., Norwood, NJ, USA, 1992, ch. A new era for high precision touchscreens, pp. 1–33.

[69] SESTOFT, P. Numeric performance in c, c# and java. `http://www.itu.dk/~sestoft/papers/numericperformance.pdf`, February 2010.

[70] SHU, E. Android market: a user-driven content distribution system. `http://android-developers.blogspot.co.nz/2008/08/android-market-user-driven-content.html`, August 2008. [Online; accessed 30-March-2013].

[71] SLACK, J. M. System testing on the cheap. In *2010 Information Systems Educators Conference Proceedings* (2010), ISECON 2010.

[72] SMITH-ATAKAN, S. *Human-Computer Interaction*. Thomson Learning, April 2007.

[73] TIOBE SOFTWARE. Tiobe programming community index for march 2013. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, March 2013. [Online; accessed 30-March-2013].

[74] UNIVERSITY OF TWENTE. mu3 - java mulit-touch framework. `https://code.google.com/p/mu3/`. [Online; accessed 31-March-2013].

[75] WARNER, J. C. top(1) - linux man page. `http://linux.die.net/man/1/top`. [Online; accessed 28-March-2013].

[76] WELCH, I. Nwen304 - advanced network applications. `http://ecs.victoria.ac.nz/Courses/NWEN304_2013T1/WebHome`, 2013. [Online; accessed 28-March-2013].

[77] WHARTON, C., RIEMAN, J., LEWIS, C., AND POLSON, P. Usability inspection methods. John Wiley & Sons, Inc., New York, NY, USA, 1994, ch. The cognitive walkthrough method: a practitioner's guide, pp. 105–140.

[78] WIGDOR, D., WILLIAMS, S., CRONIN, M., LEVY, R., WHITE, K., MAZEEV, M., AND BENKO, H. Ripples: utilizing per-contact visualizations to improve user interaction with touch displays. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology* (New York, NY, USA, 2009), UIST '09, ACM, pp. 3–12.

[79] WISE, P. iotop(1) - linux man page. `http://linux.die.net/man/1/iotop`. [Online; accessed 28-March-2013].

Appendix A

XML Data Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

  <!--A root element so xml likes me-->
  <xs:element name="sceneroot">
    <xs:complexType>
      <xs:sequence>
          <xs:element ref="scene"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!--A root element so xml likes me-->
  <xs:element name="performanceroot">
    <xs:complexType>
      <xs:sequence>
          <xs:element ref="performance"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

```

```
25  <!--A scene element for an MTApplication-->
26  <xs:element name="scene">
27    <xs:complexType>
28      <xs:sequence>
29          <xs:element name="component" type="mtcomponent"
30                    minOccurs="0" maxOccurs="unbounded"/>
31      </xs:sequence>
32      <xs:attribute name="name" type="xs:string" use="required"/>
33    </xs:complexType>
34  </xs:element>
35
36  <!--A performance element for an MTE recording-->
37  <xs:element name="performance">
38    <xs:complexType>
39      <xs:sequence>
40        <xs:element name="record" type="mtrecord" maxOccurs="1"/>
41        <xs:element ref="scene" maxOccurs="1"/>
42      </xs:sequence>
43      <xs:attribute name="name" type="xs:string" use="required"/>
44      <xs:attribute name="timestamp" type="xs:dateTime" use="required"/>
45      <xs:attribute name="componentID" type="xs:string" use="optional"/>
46    </xs:complexType>
47  </xs:element>
48
49  <!--An individual component inside a scene-->
50  <xs:complexType name="mtcomponent">
51    <xs:sequence>
52      <xs:element name="inputprocessor" type="mtinputprocessor"
53                  minOccurs="0" maxOccurs="unbounded"/>
54      <xs:element name="component" type="mtcomponent"
55                  minOccurs="0" maxOccurs="unbounded"/>
56    </xs:sequence>
57    <xs:attribute name="name" type="xs:string" use="required"/>
58    <xs:attribute name="id" type="xs:string" use="required"/>
59  </xs:complexType>
60
61  <!--An input processor for a particular gesture type in a component-->
62  <xs:complexType name="mtinputprocessor">
```

```
63      <xs:sequence>
64        <xs:element name="gesture" type="mtgesture"
65                    minOccurs="0" maxOccurs="unbounded"/>
66      </xs:sequence>
67      <xs:attribute name="class" type="xs:string" use="required"/>
68      <xs:attribute name="name" type="xs:string" use="required"/>
69    </xs:complexType>
70
71    <!--A gesture that was performed on a component-->
72    <xs:complexType name="mtgesture">
73      <xs:sequence>
74        <xs:element name="event" type="mtevent"
75                    minOccurs="0" maxOccurs="unbounded"/>
76      </xs:sequence>
77      <xs:attribute name="name" type="xs:string" use="required"/>
78    </xs:complexType>
79
80    <!--A particular event inside a gesture-->
81    <xs:complexType name="mtevent">
82      <xs:attribute name="type" type="xs:int" use="required"/>
83      <xs:attribute name="time" type="xs:long" use="required"/>
84    </xs:complexType>
85
86    <!--A record of all the input for a performance-->
87    <xs:complexType name="mtrecord">
88      <xs:sequence>
89        <xs:element name="input" type="mtinput"
90                    minOccurs="0" maxOccurs="unbounded"/>
91      </xs:sequence>
92    </xs:complexType>
93
94    <!--A particular event inside a gesture-->
95    <xs:complexType name="mtinput">
96      <xs:attribute name="source" type="xs:string" use="required"/>
97      <xs:attribute name="cursorid" type="xs:int" use="required"/>
98      <xs:attribute name="type" type="xs:int" use="required"/>
99      <xs:attribute name="positionx" type="xs:float" use="required"/>
100     <xs:attribute name="positiony" type="xs:float" use="required"/>
```

```
101       <xs:attribute name="timestamp" type="xs:long" use="required"/>
102     </xs:complexType>
103   </xs:schema>
```

Appendix B

MTEE Plugin Manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension
      point="org.eclipse.ui.perspectives">
    <perspective
        class="mtexplorer.perspectives.MTEPerspective"
        icon="icons/releng_gears.gif"
        id="mtexplorer.perspectives.mtePerspective"
        name="Multi-Touch Explorer Environment">
    </perspective>
  </extension>

  <extension
      point="org.eclipse.ui.views">
    <category
          id="mtexplorer.views"
          name="MTExplorer">
    </category>
    <view
          category="mtexplorer.views"
          class="mtexplorer.views.MTEComponentsView"
          icon="icons/sample.gif"
          id="mtexplorer.view.mteComponentsView"
```

```
25              name="MTE Components">
26       </view>
27       <view
28           category="mtexplorer.views"
29           class="mtexplorer.views.MTEPerformanceView"
30           icon="icons/sample.gif"
31           id="mtexplorer.view.mtePerformanceView"
32           name="MTE Performances">
33       </view>
34    </extension>
35
36
37    <extension point="org.eclipse.ui.newWizards">
38       <wizard
39           id="mtexplorer.wizards.explorerLauncher"
40           name="MTE Project"
41           class="mtexplorer.wizards.MTExplorerWizard"
42           icon="icons/sample.gif">
43         <description>Import the MTE tool into an existing project.</description>
44         <selection class="org.eclipse.core.resources.IFile" />
45       </wizard>
46    </extension>
47
48    <extension
49        point="org.eclipse.ui.navigator.navigatorContent">
50       <commonWizard
51           type="new"
52           wizardId="mtexplorer.wizards.explorerLauncher">
53         <enablement></enablement>
54       </commonWizard>
55    </extension>
56 </plugin>
```