UI-Driven Test-First Development of Interactive Systems

Judy Bowen
Department of Computer Science
The University of Waikato
Hamilton
New Zealand
jbowen@cs.waikato.ac.nz

Steve Reeves
Department of Computer Science
The University of Waikato
Hamilton
New Zealand
stever@cs.waikato.ac.nz

ABSTRACT
Test-driven development (TDD) is a software development approach, which has grown out of the Extreme Programming and Agile movements, whereby tests are written prior to the implementation code which is then developed and refactored so that it passes the tests. Test-first development (TFD) takes a similar approach, but rather than relying on the testers to infer the correct tests from the requirements (often expressed via use cases) they use models of the requirements as the basis for the tests (and as such have a more formal approach). One of the problems with both TDD and TFD is that is has proven hard to adapt it for interactive systems as it is not always clear how to develop tests to also support user interfaces (UIs). In this paper we propose a method which uses both formal models of informal UI design artefacts and formal specifications to derive abstract tests which then form the basis of a test-first development process.

Author Keywords
Interactive system design, test-first development, formal specification.

ACM Classification Keywords D.2. Software Engineering: D.2.2. Design Tools and Techniques: User Interfaces

General Terms
Design, Reliability

INTRODUCTION
The test-driven development cycle follows a pattern of short iterations: write a test which is failed; write just enough code to pass the test; refactor the code. This is repeated until the implementation is complete [2]. Once the code is finished the tests then act as a regression test suite and if there are new requirements for subsequent versions tests can be added to the test suite which drive the new code in the same way. Tests are developed from the requirements of the code which may be expressed as use cases or scenarios, and as such rely on the skill of the developer to translate these into comprehensive tests.

EICS'11, June 13-16, 2011, Pisa, Italy.

TDD assumes no pre-defined plan for constructing the code beyond the requirements (which drive the tests). Test-first development takes a similar approach but instead of assuming that the tests can be directly inferred from the requirements with no other development steps, it instead uses models developed from requirements as the basis for the tests.

Test-driven development can be problematic for interactive systems as the development of the UI is more usefully built in conjunction with users and with a full understanding of their needs beyond the functional requirements. Kent Beck (often credited with the re-invention and popularization of TDD) says of TDD:
"*I'm not submitting this as something everyone should do all the time. I have trouble writing GUIs test-first, for example*" [3].

Because TFD allows for pre-development modelling and uses these models as the basis for the tests, it appears that this might be more suitable for interactive system development. For example, it might enable user-centred design (UCD) artefacts, such as prototypes (which as an abstraction of the intended design are themselves a type of model) to be used to help drive test creation. However, there is still no immediately obvious way to use these artefacts in this way.

In previous work we have shown how abstract tests can be derived from formal models of user interfaces and system specifications. Our work is based on the premise that a UCD approach has been used in conjunction with formal system development (by way of specification and/or refinement) [4]. We have subsequently described how such tests can be instantiated for post-implementation model-based testing as well as used as the basis for user evaluations of the UI [5].

In this paper we discuss how the same automatically generated abstract tests can be used as the basis for a TFD approach for interactive software. We again place this work in the context of a formal development process incorporating UCD techniques for the UI, and show how TFD can ensure consistency throughout the development process as well as ensuring that subsequent refactoring (due perhaps to changes in requirements or feedback from users) maintains the consistency between UI and functional core.

We present a methodology of our approach and details enabling others to replicate our work, but do not attempt to present an evaluation in this paper, rather we leave that for future work.

The contributions this paper proposes are a method for combining the benefits of TFD and functional TDD for interactive systems. This method is based upon existing sound principles of user-centred design and functional specification. These support the development of robust, usable systems and incorporate models of design artefacts, which are easy to produce and maintain, which provides both a formal and user-centred description of interactive systems.

BACKGROUND AND RELATED WORK

UI testing is an essential, yet complex, requirement of software development. Not only must we ensure that the interface is usable and understandable by the target group of users, but also that it provides all of the necessary required functionality to the user by interacting correctly with the functional core of the system. As such it can be a time-consuming and expensive process involving both human-based testing (by way of user evaluations and usability testing) as well as extensions to traditional functional testing which often do not include (or which use methods which are not appropriate for) UI testing. One of the challenges is dealing with the complexity of the UI, which may present the user with a large number of possible interactions at any given time, so the state space of possible interaction paths is often unmanageable in terms of both modelling and testing.

One of the first issues we face when testing UIs is finding the right tool. UIs are designed for human interaction and as such do not lend themselves naturally to automated testing. Record and playback testing is a technique often used to test UIs. A user (usually one of the developers or testers) interacts with the software to exploit particular functionality and the sequence of interactions is recorded and can subsequently be played back automatically to repeat the test. This approach can, however, be time-consuming and prone to needing constant revision, in that as soon as a change is made to production code the test needs to be re-recorded and then re-run. These tools also require most, if not all, of the UI to be complete, and so are most useful in post-implementation testing. Another problem is that the tests are generated in a runtime environment and then stored as scripts which are not easy to read or interpret.

Abbott [6], a JUnit extension for Java Swing applications, partially solves some of these problems by recording the tests and defining them at a more abstract level (so they are not coupled to co-ordinates of widgets as is common in record and playback tools) making the mechanism more robust to changes. It also allows tests to be defined using an XML script editor so that they can be used for TDD. An extension to such tools are the simulated interaction testing tools which interact with UIs of a system in the manner of a user (*i.e.* by way of mouse clicks and widget activation) based on scripted or coded commands. For example, the FEST tool, which we have previously used for model-based testing [4], is an extension to Abbott which allows developers to write unit tests for UIs written in Java based on simulated interaction. Similar tools, such as Ranorex [14], exist for the Microsoft .NET platform. The simulation tools can have similar problems to the record and playback methods, as the tests are often tightly coupled to the code, and both methods still rely heavily on expertise to create the correct tests to ensure full coverage.

These problems of UI testing are equally true in a TFD process, with the added difficulty that there is a requirement to consider not only *how* to test the interactive parts of the system (what sort of method and tools to choose) but also what sort of tests should be included. While the functional specification may lend itself well to the derivation of tests, the UI design artefacts do not.

TFD has been adopted by the Agile community, along with many other practices such as prototyping from the UCD community. In [8] Hellmann *et. al*. take a similar approach to ours to trying to solve the difficulties encountered using TFD for interactive systems. Their work uses prototypes – or rather scenarios of interaction sequences derived from prototypes – as the basis for tests for simple systems with UIs consisting of limited widgets. As such they have moved the burden of the work of creating the tests into that of creating the scenarios of interaction sequences, but with no consideration of correctness or coverage, which is not a focus of their work in the same way that it is in ours.

In order to consider what sort of tests we might use, and how these can be derived, we can look at model-based approaches to post-implementation testing. Memon et al. [11][12] use reverse-engineering techniques to create models of interactive systems and from these automatically generate tests which (try to) exercise all paths throughout the UI and can be used for regression testing. Similarly, Gimblett and Thimbleby [7] have developed an algorithm for discovering models of UIs, producing a finite directed graph which can be analysed to discover design defects or structural usability concerns. While both of these approaches are post-implementation, the methods for using models derived by UI inspection to develop tests may suggest useful ways forward for TFD. In particular, Gimblett and Thimbleby's method of using the model discovery and analysis as a lightweight approach, which is part of a collection of methods to try and increase formality and correctness, sits well with our own approach to lightweight modelling.

We have shown in previous works how simple abstract models of a user interface (which may be based on a design such as a prototype or an actual implementation) can capture required information about UI designs and behaviours while avoiding state explosion via abstraction. We have also shown how we can use these models to derive

abstract tests which can then be used as the basis for model-based testing. By automatically generating tests from models we can ensure the coverage of the UI functionality, the consistency of the testing process and reduce the burden of test derivation.

Functional test driven development (FTDD) [1] tries to improve UI code and its inclusion within the testing process by delaying the UI development until later (after much of the functional core has been developed using standard unit tests) and then exercising UI tests. In addition, the tests are used both for development and to act as a requirement specification. There is an additional requirement that the tests be specified in such a way that they are unambiguous (and compatible with the chosen testing framework) but at the same time can be understood by non-technical subject matter experts as well as by multiple development teams who inherit both the code and test-defined specification as the project moves through development phases. These are challenging aims, and this area of research currently suffers from lack of tool support.

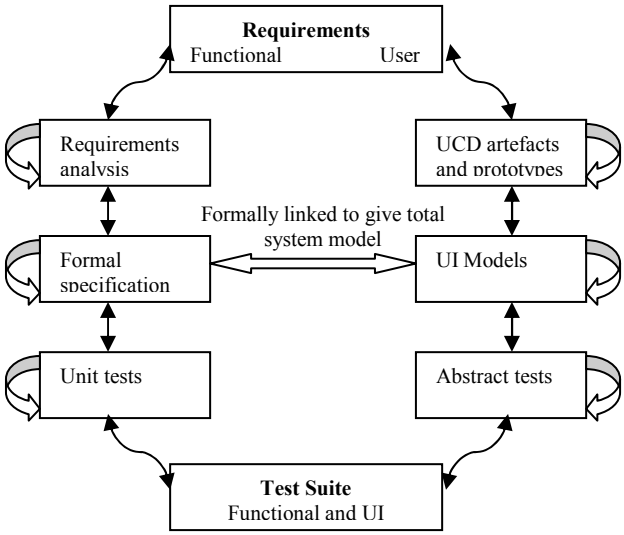
Our intention is that by keeping the tests tied to the UCD artefacts, we achieve simplicity and unambiguity by already having a specification which can be understood by users and developers alike. Our UI models are lightweight and easily updated as new requirements are added, and there is an automated process for producing abstract tests ensuring consistency. In addition, the UI models are linked to the existing formal system specification which forms part of the basis for the instantiated tests, and so we do not rely solely on the tests themselves to provide such a specification.

THE TEST-FIRST PROCESS – OUR APPROACH

In this paper we use the UISpec4J testing framework [15] which is a simulated interaction testing tool for Java Swing applications which works alongside JUnit [10]. While this has the usual disadvantage of simulated interaction tools of requiring tests which are tightly coupled to application code (in order to simulate interaction on a widget the test must refer to the type and one of the properties of the widget) this is less of a problem with TFD as we already have models which give us information about the type of the widgets. Also, the UISpec4J framework supports tests that are easy to write (often considered an important goal of TFD) by providing wrapper classes with interaction methods (such as click and double click for Buttons) for Swing component classes, enabling a higher level of abstraction than is possible when directly interacting with the Java Robot class. Another advantage of UISpec4J (and similar tools) is that we can use it within standard IDEs, such as Eclipse, which then provides all of the usual support for the test code that we expect from such tools.

As the tests are written first we can use the naming conventions from the model also and continue this in the implementation. As in any testing process having the correct tests (in terms of both coverage and behaviour) is essential to ensure a correct end result. It is not enough to develop a system which satisfies a set of tests if the tests are missing crucial parts of the functionality or UI, or are incorrectly defined.

Starting with the requirements (both user and functional) we develop a set of models which consist of a functional specification and UI models derived from the UCD process (including user requirements, task analysis, prototyping *etc.*) The models are created when we are happy we have satisfied all of the requirements within our functional and user requirement descriptions. From the UI models we then automatically derive a set of abstract tests using the PIMed tool [13]. We use these tests in conjunction with the specification and the prototypes as the basis for the test-first process. Figure 1 illustrates the process.



1: Process Flow

Unlike a test-driven approach where we define the tests as we go along and can satisfy them in any way that is appropriate, we have additional constraints given by the prototypes and UI models which act as a guide. It is not enough to test that a particular function is accessible from the UI, it must also meet the design given in the prototype by providing a correctly typed widget, and when we come to the layout of the UI (during the code refactoring stage) we use the prototype to inform how this should be done. The actual development steps are then the same as in a test-driven process:

- Write a test
- Check that the test is failed
- Write enough code to pass the test
- Refactor

The abstract tests derived from our models determine what the next test is (we describe this process later) and as part of the refactoring we must also ensure that we continue to satisfy the layout and designs created as part of the UCD process. Once we have reached a certain point in the

development (which might be defined to when a specific set of states of the model have been implemented or a given amount of the development finished, whichever is most appropriate) we can extend the testing process to include users, who can begin to interact with the software to ensure that usability is also being satisfied, and if necessary refactor designs to maintain this. The automated tests show that we are following the agreed designs and specification but by adding humans into the equation of testing we may find additional issues that were not obvious from the paper prototypes.

While the UISpec4J tests can still be difficult to read and understand for those unfamiliar with the framework, we include the abstract test within the comments and documentation for each concrete test, making it easier to relate the test code to the model-generated abstract tests and thereby easier to understand the test suite.

The tests we derive allow us to examine the expected behaviours of the UI in terms of the resulting functionality. We are not interested in making sure that default component behaviour works, such as does a button click occur when the cursor is positioned over it and the mouse button clicked (we assume the programming language will take care of this), but rather what happens to both the UI and system state when such an action occurs.

In the next section we describe the Simple Calendar application which is used as an example throughout the rest of this paper.

SIMPLE CALENDAR EXAMPLE

The example application we use in this paper is that of a simple calendar which can be used to record events associated with particular dates. This example is based on a real-world system, but has been simplified to make the modelling and explanations required suitable for description in this research publication.

Requirements and Specification

The functional requirements for SimpleCalendar are as follows:

The application is a calendar which incorporates a diary allowing events to be assigned to days, which can then be viewed in the calendar overview. As well as a month-by-month view, the application provides the ability to view the events of a single day and to add, edit and remove events for any visible day. The calendar view can be changed to move forward and backward between months.

The application should provide the following functionality to users:

- A visual display of a single month where any events assigned to days in that month are visible
- The ability to view the previous month
- The ability to view the next month
- The ability to view the events of a single day
- The ability to add new events to any day
- The ability to edit any existing events in the calendar
- The ability to remove any existing events in the calendar

Based upon these requirements a functional specification was created using Z [9]. In fact any formal specification language can be used, we use Z for convenience. The user requirements were broken down into tasks and subtasks, and prototypes were developed of the possible UI. The specification and prototypes were validated and refined until we were satisfied we had correctly described all of the requirements and that the UI was satisfactory for the intended users of the system. Standard UCD processes were followed whereby the users validated the prototypes (we did not, of course, expect them to understand or comment on the formal specification) and the formal experts validated the specification.

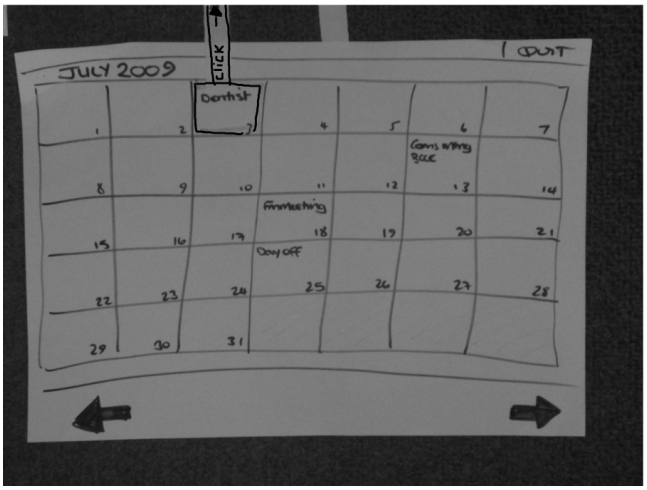
UI Design Prototypes and Models

UI prototypes for each of the states of the UI had been developed and these were then modelled as presentation models and presentation interaction models (PIMs). Presentation models describe the narrative of a prototype by describing each state of the UI in terms of its component widgets, their type, and their associated behaviour. The prototype for the main opening UI for simple calendar is shown in figure 2. The presentation model for this part of the UI is:

MainView is

(QuitButton, ActionControl, (Quit)),
(PrevArrow, ActionControl, (S_PrevMonth)),
(NextArrow, ActionControl, (S_NextMonth)),
(DayDisplay, ActionControl, (I_DayView)),
(DayDisplay, Responder, (S_PrevMonth, S_NextMonth, S_AddEvent, S_RemoveEvent, S_EditEvent)),
(ShowMonth,SValResponder (S_PrevMonth, S_NextMonth)),
(ShowYear, SValResponder, (S_PrevMonth, S_NextMonth))

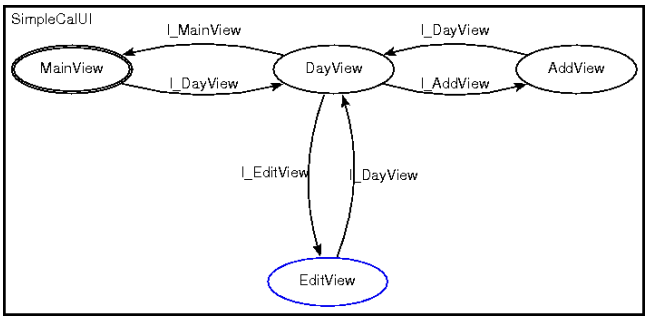
The behaviours associated with widgets are defined either as S-Behaviours (denoted by prefixing the name with S_) which are behaviours which relate to underlying functionality of the system, or I-Behaviours (denoted by prefixing the name with I_) which relate to interactive behaviours which change the state of the UI. Widgets may have more than one behaviour associated with them and either generate these behaviours (and as such are categorised as an Action Control or a sub-type of Action Control) or they respond to behaviours (and are categorised



2: Prototype for opening screen of simple calendar

as a Responder or a sub-type of Responder). Widgets which both generate and respond to behaviours are described twice within the model.

The second part of the model is the PIM which is a state transition diagram showing how the UI moves between states as a result of the I-Behaviours. The PIM for Simple Calendar is given in figure 3.



3: **PIM for simple calendar**

The final part of the model is the PMR which shows how the S-Behaviours defined in the presentation model relate to operations in the functional specification (and corresponds to the horizontal double arrow in figure 1). The PMR for Simple Calendar is:

{ S_PrevMonth ↦ ShowPreviousMonth,

S_NextMonth ↦ ShowNextMonth,

S_RemoveEvent ↦ DeleteEvent,

S_UpdateEvent ↦ EditEvent,

S_AddEvent ↦ AddEvent,

S_ShowEvent ↦ SelectSingleEvent ,

S_ShowSingleDay ↦ SelectSingleDay }

Each of the operations listed on the right of the relation are defined within the Z specification, for example the ShowPreviousMonth operation is:

ShowPreviousMonth

$\Delta Calendar$

$allevents' = allevents$

$currentMonth > 1 \Rightarrow currentMonth' = currentMonth - 1 \land$

$currentYear' = currentYear$

$currentMonth = 1 \Rightarrow currentMonth' = months \land$

$currentYear' = currentYear - 1$

$visibleDates' = allDates \triangleright \{currentMonth'\}$

Abstract Tests

Once the models were complete we used the PIMed tool to generate the abstract tests. These tests describe the conditions of the presentation model in first order logic where all of the conditions on each of the UI states are defined. That is, each state is described by a series of predicates which give the requirements for the widgets of that state. The *MainView* state, for example, contains (among others) the following tests.

$State(MainView) \Rightarrow Visible(DayDisplay) \land Active(DayDisplay) \land$

$hasBehaviour(DayDisplay, I\_DayView)$

$State(MainView) \Rightarrow Visible(PrevArrow) \land Active(PrevArrow) \land$

$hasBehaviour(PrevArrow, S\_PrevMonth)$

Each test describes the conditions on one of the widgets of one of the states of the UI. The *Visible* and *Active* predicates have the (expected) meaning that in this state of the UI the named widget must be both visible and available for interaction (which is not always the case, sometimes particular widgets might not be enabled in particular states) and the *hasBehaviour* predicate gives the designated behaviours for the widget. Widgets which respond to behaviours rather then generate them result in a test predicate called *resBehaviour*.

Using the Tests in a TFD Process

With the complete set of abstract tests derived from the models we now move on to development. The abstract tests must be instantiated into concrete tests in a way which supports the TFD process. The tests can be divided into categories as follows:

- Test of state
- Test of widget instantiations
- Test of I-Behaviours

- Test of S-Behaviours

The first category defines what it means to be in a particular state of the UI. Each of the abstract tests relates to a particular state, so for example the test:

$State(MainView) \Rightarrow Visible(DayDisplay) \wedge Active(DayDisplay) \wedge hasBehaviour(DayDisplay, I\_DayView)$

is only of interest when we are in the *MainView* state. Our first set of concrete tests defines each of the states so we can determine which we are in. From the PIM we also determine the start state of the system. The states of Simple Calendar are:

- MainView
- DayView
- AddView
- EditView

and the start state is *MainView*. We can now write our first test which is to ensure that when Simple Calendar is started a window which can be identified as *MainView* is visible.

Within our test class we create a reference to an instance of the SimpleCalendar class (although this does not exist of course as the test is the first thing written) which we call *sysEx* and which will be used as the basis for all of the tests (UI and functional). We also create a reference to the UISpec4J class Window which handles all of the UI testing for an identified window and which we call *calExUI*.

UISpec4J follows the JUnit convention of providing setup and teardown methods, so we instantiate the references we have declared within the setup method which ensures we use the same instances throughout the testing process. The first test we write looks for an active window once the application is started (we will subsequently test that this window matches the start state, once we have defined the state tests).

In order to run the test we must start writing the application code, and in keeping with the TFD process aim to write the smallest amount required to satisfy the test. In order to make the test compile and run we need to satisfy the variable declarations by creating a class template for both *SimpleCalendar* and *MainView*. This gives us a failing test, and we then pass it by extending the description of *MainView* so that it creates a visible window which is instantiated when *SimpleCalendar* starts up. Of course if we strictly follow the mantra of "write the smallest amount of code possible to pass the test" we could just define the *SimpleCalendar* class and have it contain a reference to a Java Window and then subsequently refactor this into two separate classes, but as is usual in TFD (and TDD) we balance the process with some informed design knowledge which enables us to make steps that are larger than the bare minimum and give a more realistic development process (while remaining mindful of not over-constraining ourselves by making big decisions too early).

Now for each of the states of the PIM we develop a test to determine whether we are correctly in that state or not. To be correctly in a state means that the UI state has all of the described widgets which are both visible and active as per the abstract tests. So for example from each of the abstract tests which refer to the state *MainView* we determine that in this state the following widgets should be visible and active (*i.e.* available to the user):

- QuitButton
- PrevArrow
- NextArrow
- DisplayMonth
- DisplayMonth
- DisplayYear
- DayDisplay

which we capture with the following test:

```
public void testMainViewState(){
 calExUI.getButton("Quit").isEnabled();
 calExUI.getButton("Quit").isVisible();
 calExUI.getButton("Prev").isEnabled();
 calExUI.getButton("Prev").isVisible();
 calExUI.getButton("Next").isEnabled();
 calExUI.getButton("Next").isVisible();
 calExUI.getSwingComponents(cont.getClass(),
    "DisplayMonth")[0].isVisible();
 calExUI.getSwingComponents(cont.getClass(),
    "DisplayYear")[0].isVisible();
 calExUI.getSwingComponents(cont.getClass(),
    "DayDisplay")[0].isVisible();
 calExUI.getSwingComponents(cont.getClass(),
    "DayDisplay")[0].isEnabled();
 }
```

When this test is passed we are certain that we have a correctly defined state, in terms of the component widgets (we consider the behaviour of these widgets later). The state tests also have the effect of ensuring all non-behavioural widgets (such as text entry fields) are correctly defined and available for use. As we will also want to reuse this test (and the other state tests) later when we write the tests to ensure correctness of behaviours, we generalise the state tests so that each of them receives a UISpec4J Window as a parameter and then checks the given window for the relevant widgets. We can then write a single test called "TestUIStates" which calls all of the state tests, and then we can subsequently call any of the state tests individually as required within other tests. We are not concerned at this stage as to the *type* of the widgets (in order to pass the test we can define each of them as a JButton, or any other type of Java Swing widget) but as part of the refactoring we refer to the models to ensure that we make a correct choice. This

is what we mean by using the models to support the tests within the development process.

For some of the widgets we may not yet have considered their concrete type. For example, while many widgets described as *ActionControls* in the presentation model may be instantiated as JButtons, this is not necessarily the case. In the example above for `testMainViewState` we have written a test which expects some of the widgets to be JButtons, whereas others we just abstractly describe as SwingComponents. However, this has implications as testing progresses as it affects the methods we can invoke on them within the test. The UISpec4J Button class, for example, has a click() method allowing us to simulate user interaction, whereas the SwingComponent class does not have such a method.

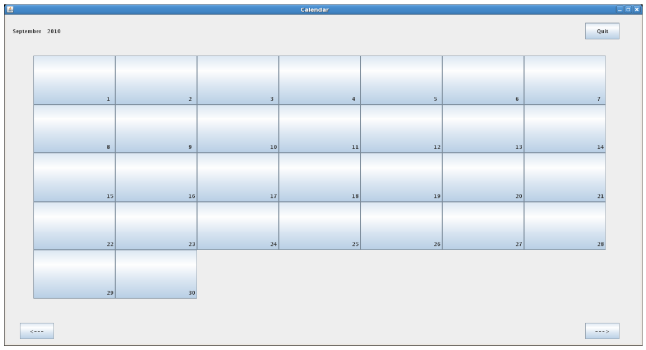
Those widgets we have defined as Buttons pass the test because this is how we have instantiated them in the first version of the code. However, if we subsequently refine them to a different type of widget we would then get a failing test, not because of problems with the code, but due to the way we have defined the test. There is, therefore, a trade-off between being specific enough early on to make interaction simulation easy or being more general and having to write more detailed test code to interact with the widgets.

We can, of course, refactor the tests to match more specific widgets once the code has been refactored. This may not seem like an ideal solution as changing the tests to match the code seems to go against the principles of TFD. However, as we will show later, the tight coupling of the UI code with the tests means that sometimes this is a necessity.

There is one other element of state that needs to be tested and that is modality. The presentation model and PIM inform us which of the windows and dialogues in the UI are modal (by showing the availability, or not, of their behaviours in conjunction with other states). From the Simple Calendar models we know that *DayView*, *AddView* and *EditView* are all modal and so we add this condition to each of the relevant state tests and update our code in order to pass the tests.

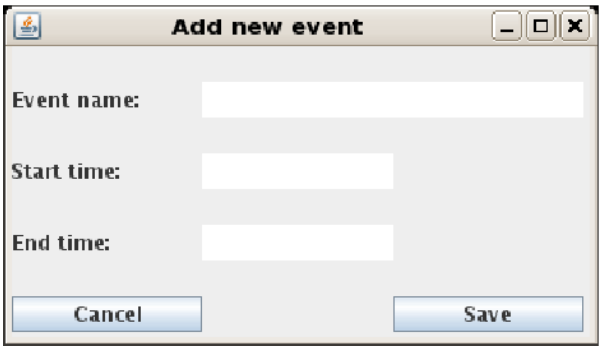
Once we have created tests for each of the states, and then the necessary classes and code to ensure that the tests are passed, we begin refactoring to tidy up the code, remove duplication and set the UI layout in accordance with the design prototypes. As such, defining the UI layout becomes part of the refactoring process and is informed by the design models.

After the state tests have all been passed and the first stage of refactoring has taken place, we have UIs for each state of the system. Some examples of these are shown in figures 4 and 5.



4: Main View

One of the things that has happened during refactoring is that a new class has been created which combines the code for *AddView* and *EditView*. In the first version of the code each was defined in its own class, but as the code for these two classes was almost identical refactoring led to them being combined. We have added a Boolean value which sets the state of the UI to either add or edit and which will be used to control the elements which are different. The next step is to write the tests for the I-Behaviours. From the set of abstract tests we identify those relating to I-Behaviours and for those tests where widgets have multiple behaviours we split them into smaller tests with single behaviours.



5: Add View

For example, the test:

$$State(AddView) \Rightarrow Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(saveButton, I\_DayView, S\_AddEvent)$$

is split into two new tests:

$$State(AddView) \Rightarrow Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(SaveButton, I\_DayView)$$

and

$$State(AddView) \Rightarrow Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(SaveButton, S\_AddEvent)$$

For each state we identify the required widgets and test that activating the widget changes the state as defined by the I-Behaviour. For example, in *MainView* we test that a widget called *dayDisplay* changes the UI state from *MainView* to *DayView*. We already know that the widget exists from the earlier state test and to ensure we have reached the correct state we capture the window after the behaviour has occurred and call the relevant state test with this window as the parameter.

When we test the *I_DayView* behaviour from the UI we are initially interested in only one instance of the *DayDisplay* widget and so just arbitrarily use the first available instance which performs the required command and ensures that the user can get from here to the *DayView* state. When we add the functional tests we will need to extend this to ensure that each visible day of the month has the correct behaviour *i.e.*goes to the correct day view.

As there is a requirement for windows to be modal we use a specific window interceptor in UISpec4J which handles modal windows. A consequence of this is that you must specify within the test the widget which closes the modal window (to return control to the test suite) which means each test of a state change to a modal window ends up testing two I-Behaviours: the behaviour which moves to the modal state as well as the behaviour which takes the user out of that state. Where there is more than one way to exit a modal window (as in the case of *AddView* where we could use either the *Save* button or the *Cancel* button) then we must, of course, ensure both are tested.

After we complete each I-Behaviour test and implement the code to ensure they are passed, we again refactor. At this stage we now have a working UI (albeit with no functionality) and this is a good time to extend the testing to user evaluation. We can repeat the automated tests by asking users to move between different states of the UI (or to just explore the UI) and check that their experience is satisfactory and there are no human-centred errors which could not be captured by the automated testing. In our example there were no changes to be made, but if there are then we must amend the models as necessary, reproduce the abstract tests and extend the test suite as required.

Next we move on to test the non-behavioural widgets. Having already ensured that they exist in the correct states, and are visible, we must rely on the widget type information given in the presentation model to understand what other properties we must test. In Simple Calendar all of the non-behavioural widgets have the type 'Entry', *i.e*. they are used to provide a way for users to enter information. Our tests then are to ensure that it is possible to enter information into each of the entry widgets. We do this by using UISpec4J windows and identifying the relevant widgets within those windows and using the `setText()` method to enter pre-defined strings. Note that the UISpec4J `setText()` method attempts to put text into the field using simulated keystrokes and does not use the underlying Java Swing `setText()` method of the widget. We then use assertions to ensure that the contents of the entry widgets match the pre-defined strings. Extending the code to pass these tests and subsequent refactoring did not lead to any changes in the appearance of the UI, but just added functionality.

The final sets of tests relate to the S-Behaviours. These are the behaviours linked to underlying system functionality. As such the meanings of these behaviours are given by the functional specification. The tests we write for these behaviours relate both to the underlying system state and the UI. S-Behaviour tests fall into two categories. There are widgets which generate S-Behaviours (typically defined as *ActionControls* like JButtons) where a user interaction causes a function to be activated, and those which respond to S-Behaviours (typically widgets which display dynamic data) where system functionality (which may or may not be user initiated) causes a widget to change its state. Just as we did with the I-Behaviour tests we first simplify the abstract tests by splitting them into single behaviour tests. We write the tests for S-Behaviour generating widgets first, and will complete the process with the S-Behaviour responding widgets. We do this for convenience, as generally the responding widgets rely on some of the code which we will write to satisfy the generating tests.

For each S-Behaviour we write a test that both invokes the widget which has the behaviour we are testing and performs the functionality testing based on the defined operation in the specification. As an example, consider the following abstract test:

$$State(AddView) \Rightarrow Visible(SaveButton) \wedge Active(SaveButton) \wedge hasBehaviour(SaveButton, S\_AddEvent)$$

Our test consists of the UISpec4J code to interact with the *SaveButton* widget in *AddView* and a unit test to ensure the correct functional behaviour occurs following the interaction. We find the definition of *S_AddEvent* by its related operation in the Z specification, which is *AddEvent*, and which is defined as follows:

AddEvent

$$\Delta Calendar$$
$$i?: EVENT$$
$$selectedDay?:Day$$

$$(i?, currentMonth \mapsto selectedDay?) \notin allevents$$
$$allevents' = allevents \cup \{(i?, currentMonth \mapsto selectedDay?)\}$$
$$currentMonth' = currentMonth$$
$$currentYear' = currentYear$$
$$visibleDates' = visibleDates$$

This gives us the requirements for the test. We are interested in the observation of the system called *allevents* and input values called *i?* of type *EVENT* and *selectedDay?* of type *DAY*. If the *i? selectedDay* pair does not exist in the set *allevents*, then after the operation it will have been added (denoted in the Z schema by the primed observation *allevents'*). The *currentMonth*, *currentYear* and *visibleDates* observations remain unchanged by the operation.

For this test we are only interested in the behaviour of the system state and not in any UI displays which may related to the display of events – these are considered later when we write the S-Behaviour responder tests. We start with a test that checks the number of events that currently exist in the system state, interacts with the UI to add an event and then uses an assertion to check that the number of events has increased by one. We then extend the test to ensure that the event which has been added is the correct one (*i.e.* that it matches the entered information), and finally we try and add the same event to the set and check that the size of the event set does not change.

When we write the S-Behaviour tests, we order them for convenience. That is, we write the test for *S_AddEvent* before the test for *S_DeleteEvent* as in order to test delete we first need to be able to add an event.

The final set of tests for the UI functionality are the S-Behaviour responder tests. We leave these until last as they rely on S-Behaviours which will have been (mostly) coded as part of the previous set of tests. For some of these tests we are only interested in part of the related operation. For example, when we test the responders for *S_ShowSingleDay* (which are the displayDate and displayMonth widgets) we are only interested in the parts of the operation relating to the date. The rest of the operation has already been tested as part of the S-Behaviour generating widget tests (*e.g.* the dayButton widget in *MainView*). We could, of course, take the approach of always testing complete operations and combining generating and responding widget tests as necessary to accommodate this. However, the approach we have taken enables us to make the tests smaller (and therefore easier to develop from) and allows us to rely on the structure of the abstract tests rather than grouping things together based on the functional specification.

Once the abstract tests have all been concretised the final step is to ensure that there are no parts of the formal specification which are not related to the UI functionality which still need to be implemented. For example, system functionality which is not available to users (perhaps interacting with operating system calls or network capabilities) will not be included in the abstract tests which have been generated from the UI models. We identify these as any operations in the specification not related to S-Behaviours in the PMR. In the Simple Calendar example there is an operation called *Init* which meets this criterion. *Init* is described as follows:

Init

$$
\begin{array}{|l}
\hline
Calendar \\
actualMonth? : MONTH \\
actualYear? : \mathbb{N} \\
\hline
allevents = \emptyset \\
visibleDates = allDates \rhd \{currentMonth\} \\
currentMonth = actualMonth? \\
currentYear = actualYear? \\
\hline
\end{array}
$$

This leads to a test that ensures that on start up the set of events is empty and the *currentMonth* and *currentYear* observations are set to the input values *actualMonth?* And *actualYear?*. For Simple Calendar these inputs are obtained from the operating system. The *visibleDates* observation is the set of all dates constrained by the *currentMonth*. Once the code has been written to ensure these tests are passed the implementation is complete.

Conclusions

We set out to find a suitable approach for supporting TFD for interactive systems by using abstract tests which can be automatically generated from UI models as the basis for the test development. Our intention was to find a suitable tool for running the tests (which could integrate both system and UI functionality) as well as providing a re-usable method for determining what tests should be written, with the aim of ensuring full coverage of all parts of the system to be implemented.

The work we have presented here provides a repeatable process which can be used for any interactive system and consists of the following steps:

- Designing the UI following a UCD process
- Formally modelling the system specification
- Developing models of the UI designs
- Automatically generating abstract tests
- Concretising the abstract tests
- Creating any required additional functional tests

This method enables TFD to be used for interactive systems in an integrated manner, *i.e.* both system and UI can be developed using the TFD approach with tests for all parts of the system being created, and run, using the same tool within a standard IDE.

We chose to start development from the UI behaviour tests and UI state tests, which meant we constructed the UI first and added system functionality last. However, we could just as easily have taken the reverse approach with very little difference, although the advantage of the first approach is that we quickly have a semi-functioning user interface which means we can begin user testing earlier on in the development.

What we have presented in this paper is the detail of *how* we propose TFD for interactive systems may be achieved.

Evaluating this to determine its ease of use, success, scalability *etc*. is part of ongoing work.

The method we have described here could be used with any simulated interaction testing tool and is not tied to the UISpec4J framework. However, while UISpec4J is not without its limitations, we did find that in most cases it was suitable for our requirements. In the cases where its inbuilt classes could not handle behaviours we wanted to assign to widgets (such as double clicking on a listbox item) it was fairly straightforward to extend its functionality to handle this (using the Mouse class for example). This did mean that there were times when we had to write tests in a more long-winded way to compensate for this, but we were able to replicate all of the interaction we needed.

By taking a TFD approach rather than TDD, we have been able to incorporate UI models developed as part of a UCD process and use these as the foundation for tests and development. This assists with supporting interactive systems with well-designed UIs; it is a recognised problem that this is often not the case in TFD and TDD. We have shown in this paper that TFD can be successfully applied to interactive systems and that by using formal models as the basis for the tests we can automatically generate abstract tests. This provides a structured and easy to use framework, which does not require input from a test expert, which can be used by anyone wishing to develop interactive systems in this way.

For the small example application used in this paper we have assumed we require 100% test coverage, and the size of the system means that this was not overly onerous in terms of test generation. However, it still remains to be seen how this will scale to larger systems as well as defining how decisions should be made regarding levels of test coverage with respect to which areas should be considered critical and which optional.

In summary, we have investigated the lack of support for (integrated) TFD for interactive systems and propose a solution based on developing abstract tests from UI models and using these as the basis for the test development. The benefits of this approach are: the ability to automatically generate the abstract tests, ensuring that coverage remains consistent without the need to rely on experienced test writers; tests which are easy to understand and relate back to initial specifications and models; a set of integrated tests which can subsequently be used for regression testing for all parts of the system; a lightweight approach with an underlying sound theory based on the specification and models.

REFERENCES

1. Andrea, J. Envisioning the Next Generation of Functional Testing Tools. *IEEE Software 24*, 3, 58-66, May/June 2007.
2. Beck, K. *Test-Driven Development by Example*. The Addison-Wesley Signature Series, 2003.
3. Beck, K. Aim, Fire. *IEEE Software 18*, 5, 87-89, September 2001.
4. Bowen, J. and Reeves, S. UI-Design Driven Model-Based Testing, in *Electronic Communications of the EASST 22 : Formal Methods for Interactive Systems*, 2009.
5. Bowen, J. and Reeves, S. Developing usability studies via formal models of UIs, in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive Computing Systems* (Berlin, Germany, June 19 - 23, 2010). EICS '10. ACM, New York, NY, 175-180.
6. Dutta, S. Abbot - A Friendly JUnit Extension for GUI Testing, *Java Developer Journal*, pp 8-12, April 2003. (http://abbot.sourceforge.net)
7. Gimblett, A. and Thimbleby, H. User interface model discovery: towards a generic approach, in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive Computing Systems* (Berlin, Germany, June 19 - 23, 2010). EICS '10. ACM, New York, NY, 145-154.
8. Hellman, T., Hosseini-Khayat, A. and Maurer, F. Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design, in , *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation Workshops*. IEEE 2010.
9. ISO/IEC 13568. *Information Technology - Z Formal Specification Notation - Syntax, Type System and Semantics*. First edition. Prentice-Hall International series in computer science. ISO/IEC 2002.
10. JUnit. http://junit.sourceforge.net.
11. Memon, A., Banerjee, I. and Nagarajan, A. GUI Ripping: Reverse engineering of graphical user interfaces for testing, in, *Working Conference on Reverse Engineering*, 260, 10th Working Conference on Reverse Engineering (WCRE 2003), 2003.
12. Memon, A, Nagarajan, A. and Xie, Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice 17*, 1, 27-64, Jan/Feb 2005.
13. PIMed. An editor for presentation models and presentation interaction models. Available from: http://www.cs.waikato.ac.nz/Research/fm/PIMed.html
14. Ranorex. http://www.ranorex.com
15. UISpec4J. http://www.uispec4j.org