Extended Bridge Software Design Pattern

Andrew J. Haigh
Regis University

Follow this and additional works at: http://epublications.regis.edu/theses

Part of the Computer Sciences Commons

Recommended Citation

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact repository@regis.edu.
Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.
EXTENDED BRIDGE SOFTWARE DESIGN PATTERN

A THESIS

SUBMITTED ON 25 OF OCTOBER, 2012

TO THE DEPARTMENT OF INFORMATION SYSTEMS

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF

SCIENCE IN SOFTWARE and INFORMATION SYSTEMS

BY

Andrew J Haigh

APPROVALS

Richard L Blumenthal Ph.D., Thesis Advisor

Ranked Faculty Name

Ranked Faculty Name
Abstract

Software engineers are taught that they should separate form and function. This allows a user interface or front-end to be designed and implemented independently of the engine or back-end.

Using an abstracted interface class provides the opportunity to develop different user interfaces for the same back-end, as each new user interface will all be derived from this agreed upon abstracted interface. Using the Bridge pattern allows the user to also decouple the abstraction from its implementation, providing additional flexibility as they developed multiple user interfaces.

However, user interfaces are typically designed as a single unit, the renderer, even the Bridge pattern refers to it as the ConcreteImplementor. This thesis concentrated on the implementation of the renderer and merged the Bridge pattern with another design pattern, the Presentation Model. The Extended Bridge pattern also includes support for the Single Responsibility principle; a construct that improves upon the Presentation Model by decoupling the logic and the data from the rendering code.

In short the Extended Bridge is an object oriented pattern that provides an abstract interface, an abstract implementation with decoupled logic, data and rendering, that is flexible, maintainable, reusable and highly testable.
Acknowledgements

I would like thank all of the Professors at Regis University for their support, guidance and knowledge. I especially would like to thank Dr. Richard Blumenthal for his patience as I crafted this thesis. My son Daniel deserves a special mention for proof-reading several revisions of this thesis. Finally to my new bride Sheryl and our merged families for letting me find time alone to complete this thesis, at a time when we were getting to know each other.
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Separation of Form and Function</td>
<td>11</td>
</tr>
<tr>
<td>2.2.</td>
<td>UML notation for the Components pattern</td>
<td>12</td>
</tr>
<tr>
<td>2.3.</td>
<td>Model-View-Controller (MVC)</td>
<td>13</td>
</tr>
<tr>
<td>2.4.</td>
<td>UML notation for the Bridge pattern</td>
<td>14</td>
</tr>
<tr>
<td>2.5.</td>
<td>UML notation for the Presentation Model pattern</td>
<td>16</td>
</tr>
<tr>
<td>2.6.</td>
<td>BankAccounts application demonstration user interface</td>
<td>17</td>
</tr>
<tr>
<td>2.7.</td>
<td>BankAccounts application full function user interface</td>
<td>18</td>
</tr>
<tr>
<td>2.8.</td>
<td>Diagram of the BankAccounts user interface design</td>
<td>19</td>
</tr>
<tr>
<td>2.9.</td>
<td>QuizMaster showing the questions</td>
<td>20</td>
</tr>
<tr>
<td>2.10.</td>
<td>QuizMaster showing the test results</td>
<td>20</td>
</tr>
<tr>
<td>2.11.</td>
<td>QuizMaster showing the tests</td>
<td>21</td>
</tr>
<tr>
<td>2.12.</td>
<td>Diagram of the QuizMaster user interface design</td>
<td>21</td>
</tr>
<tr>
<td>3.1.</td>
<td>The Extended Bridge pattern</td>
<td>24</td>
</tr>
<tr>
<td>3.2.</td>
<td>Architecture diagram for the BankAccount application</td>
<td>25</td>
</tr>
<tr>
<td>3.3.</td>
<td>Architecture diagram for the QuizMaster application</td>
<td>26</td>
</tr>
<tr>
<td>3.4.</td>
<td>Instantiation flow for the BankAccount application</td>
<td>27</td>
</tr>
<tr>
<td>3.5.</td>
<td>Instantiation flow for the QuizMaster application</td>
<td>28</td>
</tr>
<tr>
<td>3.6.</td>
<td>Architecture diagram for the BankAccount application</td>
<td>29</td>
</tr>
<tr>
<td>3.7.</td>
<td>Architecture diagram for the QuizMaster application</td>
<td>30</td>
</tr>
<tr>
<td>3.8.</td>
<td>Instantiation flow for the BankAccount application</td>
<td>31</td>
</tr>
<tr>
<td>3.9.</td>
<td>Instantiation flow for the QuizMaster application</td>
<td>32</td>
</tr>
</tbody>
</table>
3.10. Architecture diagram for the BankAccount application .......................... 33
3.11. Architecture diagram for the QuizMaster application .......................... 34
3.12. Instantiation flow for the BankAccount application ............................ 35
3.13. Instantiation flow for the QuizMaster application .............................. 36
3.15. Architecture diagram for the QuizMaster application ........................ 38
3.16. Instantiation flow for the BankAccount application ............................ 39
3.17. Instantiation flow for the QuizMaster application .............................. 40
4.1. Average Complexity per design pattern ............................................. 45
4.2. Distribution of Complexity per design pattern .................................... 46
4.3. Average Number of Executable Lines per Method per design pattern .... 46
4.4. Average Complexity per design pattern ............................................. 47
4.5. Distribution of Complexity per design pattern .................................... 48
4.6. Average Number of Executable Lines per Method per design pattern .... 48
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.</td>
<td>Key metric values for the BankAccounts implementations</td>
<td>45</td>
</tr>
<tr>
<td>4.2.</td>
<td>Key metric values for the QuizMaster implementations</td>
<td>47</td>
</tr>
</tbody>
</table>
Extended Bridge Software Design Pattern

When designing a user interface, is it possible to design it so that it addresses two specific problems? Can it be flexible enough to support different implementations, while ensuring that it maximizes testing, independent of the user interface implementation? When implementing the user interface with object-oriented technology, there is an additional requirement to take the Single Responsibility principle (Martin, 2003) into consideration. The Single Responsibility principle states “every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class”. The Extended Bridge pattern is an object oriented design that has been created to specifically address these problems, including providing support for the Single Responsibility principle.

A typical user interface (UI) consists of windows or forms which contain three distinct elements, the code to render the UI (buttons, lists, text fields, etc...), the logic of the UI (what happens when you click a button) and the data that is rendered by the UI. Without the Single Responsibility principle having these three elements in the same class gives rise to the following problems:

- **Reduced Maintainability** A change to any one element is likely to require changes in the other two to support it, making it harder to implement isolated changes. This in turn reduces the maintainability of the class.

- **Reduced Testability** Whilst it is possible to write the logic and data elements of an application in such a way that each component can be unit tested in isolation, the UI related code is inherently very difficult to unit test. Firstly, it is difficult to test any visualization as it often needs to be judged by a human to see if it ‘looks right’. Secondly user interactions are needed when trying to test UI components, such as buttons. If the
code to render the UI is mixed with the logic and the data, the independent testability of
the logic and data is lost. Automated UI testing solutions are available (Uispec4j, 2012),
but because they tend to ‘mimic’ the user interaction, they are either harder to setup and
maintain than unit tests or they are only useable as integration tests because they need the
entire application to be set up.

- **Reduced Reusability** If the three elements of the user interface are mixed in the same
class, it becomes a lot harder to make that code reusable. For example, if the data is
included in the same class, any changes to the way the data is stored will necessitate
changes in the way it is accessed by the rendering and logic code. Abstracting and
encapsulating the data into an independent class makes it reusable.

Including the Single Responsibility principle in the Extended Bridge pattern requires that
the three UI elements, code, logic and data, be separated. The logic associated with the user
interface components is decoupled from the code and the data. This provides two benefits. Firstly
it allows different styles of user interface to be supported as the implementation now has a
component that is just responsible for rendering the user interface; secondly it allows the logic to
be tested independently of the user interface. Finally the data is decoupled from the
implementation. This eliminates the need for the logic to have an intimate knowledge of the user
interface and how to access the data that would otherwise have been stored there. Including the
Single Responsibility principle provides the Extended Bridge pattern with increased
maintainability, reusability and independent testability of the three UI elements.
Consider the following as a use case: A developer is asked to design a user interface that can be implemented on multiple platforms, with support for a full featured version and a demonstration (light-weight) version on each platform.

Using the Extended Bridge pattern, as described in this thesis, provides the developer with the ability to create a design that is flexible enough to integrate the requested user interfaces, while maximizing testability. Two applications are referenced in this thesis, the first is an application to manipulate a simple bank account record system providing two user interfaces; the second is an application that manages quiz questions providing a user interface that contains interchangeable view components.

The Extended Bridge pattern is based on the result of merging two existing design patterns. The Bridge pattern (Gamma et al., 1995) provides the flexibility, and the Presentation Model (Fowler, 2004) provided some support for the Single Responsibility principle. The Extended Bridge pattern builds on the flexibility of the Bridge pattern, which abstracts not only the functional requirements but also the implementation details, so that the two can be modified independently of each other. It merges this with the Presentation Model which separates the user interface model from the user interface, extending it to the fully support the Single Responsibility principle by separating out the rendering code, the logic and the data into independent classes, providing increased maintainability, testability and reusability.

Many different design approaches exist, specifically in the area of user interfaces. Whilst numerous authors have provided contributions to this broad coverage, almost none of the designs are accompanied by hard metric data supporting their usage. This thesis provides metric data that validate the adoption of the Extended Bridge design. In addition this thesis introduces a new metric, the Distribution of Complexity.
The remaining chapters of this thesis develop the Extended Bridge pattern. Chapter 2 covers the relevant literature reviewed to provide background; it also introduces the two reference applications. Chapter 3 provides a detailed description of the Extended Bridge pattern and discusses how the reference applications have been implemented in several of the design patterns. Chapter 4 provides the results obtained from collecting specific metric data, allowing comparison of the design patterns selected. Chapter 5 provides the conclusion of the thesis with a discussion of the results and offers some future direction.

**Background**

**Review of Literature and Research**

A common thread to the designs chosen for this thesis is that at their foundation is the concept of the separation of form and function. In short, the separation of form and function describes how a developer can separate the way the application is rendered to the user from the function of the application.

**Separation of Form and Function**

An early reference for the separation of form and function is found in the document “DECforms: Guide to Developing Forms” (Digital Equipment Corp., 1990), it shows a figure of the separation of form and function (see Figure 2.1). Prior to this, the code to manage the user interface was embedded in the application program. Using DECforms allowed application developers to abstract the user interface from the application code. This gave them the opportunity to manipulate the user experience independently of the computational code.
The need for a separation of form and function was highlighted when Digital Equipment Corp. introduced DECwindows, an event driven user interface. DECforms was a modal form or panel based interface, where the form code could easily be interwoven with the application logic; whereas with DECwindows multiple windows or panels could be visible and active at the same time, making it impossible to interweave the window code with the application logic\(^1\). The document “DECwindows: Motif Guide to Application Programming” (Digital Equipment Corp., 1994), provided its own definition of separation of form and function.

\[\textbf{1.4.3 Form versus Function}\]

The fundamental concept of programming with the Toolkit is the separation of form and function. Using the Toolkit, you can consider the form your application takes—its user interface—separately from the routines that implement the functions of your application.”

David L. Stone, a vice president of software engineering at Digital Equipment Corp., once stated (company meeting, circa 1990), “if you separate form and function, you can demonstrate the form, get approval for the form and start documenting the form while the backend (the function) is being developed.”

---

\(^1\) The author worked at Digital Equipment Corp. and worked with both DECforms and DECwindows
The following are design patterns that were considered for this thesis.

**Components**

Douglas Young (1992) introduced a new approach with X windows, a pre-cursor to DECwindows, when he introduced reusable modules. Douglas Young, in his book, uses the separation of form and function when he introduces the concept of a component (see Figure 2.2). He describes a component as follows, “A component not only encapsulates a collection of widgets, but defined the behavior of the overall component as well”. What he described were self-contained forms, a pre-cursor to the Presentation Model and the Single Responsibility principle.

![Figure 2.2. UML notation for the Components pattern](image)

**Model view controller**

Model-View-Controller (MVC) (Burbeck, 1987) provides a triad of classes. The first is the Model; this class is the application object and provides the functionality. The second is the View and provides the visual representation of the data held in the Model. The third is Controller that manages the user input and the Model appropriately (see Figure 2.3).
Although this design pattern has merits it was rejected as a reference design pattern for this thesis for the following reasons:

- The View is responsible for parsing the Model (application object) for appropriate data to display; the view has to have intimate knowledge of the Model in order to extract data. Whereas the Extended Bridge pattern uses a data model which contains only the data required by the user interface.

- The user input is designed in MVC to be received by the Controller. This functionality is difficult to separate from the View, when the UI components, such as buttons, text areas and check-boxes are built into it.

Although it is possible to see the Extended Bridge pattern as a derivative of MVC, as it required a fundamental re-architecting to become the Extended Bridge pattern the original MVC pattern has not been considered.
Bridge

The Bridge pattern (Gamma et.al., 1995) (see Figure 2.4), focuses on the structure of the user interface. Prior to this design pattern a user interface that was implemented as components had limited scalability. When multiple platforms needed to be supported, a new platform component variant was created, derived from an abstract interface. This raised issues regarding maintenance, keeping multiple similar variants in synch; it also high-lighted the lack of code reuse. This is a very inefficient approach to designing for multiple platforms. The Bridge pattern, which is described as follows “Decouple an abstraction from its implementation so that the two can vary independently”, addresses this problem. Its approach is to change the relationship between the abstract interface and the implementations. Instead of directly coding to the interface, one half of the component belongs to the interface hierarchy and the other half provides utility routines specific to the platform. Now instead of having N (number of interfaces) * P (number of platforms) classes, you have N + P classes.

Figure 2.4. UML notation for the Bridge pattern
**Separated user interface patterns**

In addition to the Bridge pattern, Martin Fowler (Fowler, 2006a) looks into several constructs that support the separation of form and function. One approach he discusses is the Humble Dialog Box (Feathers, 2002) which strips the user interface of all external knowledge. This approach relies on another class to drive data to the user interface. Although this approach has merits, it does mean that the driver has to have intimate knowledge of how the user interface is implemented. Other approaches explored by Martin Fowler are the Supervising Controller (Fowler, 2006b), the Passive View (Fowler, 2006c) and the Presentation Model (Fowler, 2004) (see Figure 2.5).

The Presentation Model pattern abstracts the presentation state and behavior into the model. As GUI components such as check-boxes and radio-buttons contain state, when they are selected, the user interface has to contain code that supports the changes in state for these components. Additionally, it is difficult to determine the state of these components outside of the user interface, as it involves using the component APIs. This design pattern requires that any change of state is communicated to the model, the model modifies the data associated with other UI components and then requests that the user interface refresh itself. An example could be the enabling of UI components if a check-box or radio-button is selected. The user interface notifies the model, the model changes the ‘visibility flag’ associated with other components, which the user interface uses to modify its rendering of those components.
Three design patterns were chosen, Components, Bridge and Presentation Model. These three patterns will be referred to as ‘Reference Design Patterns’ and will be used to provide a standard (control) for evaluating the Extended Bridge pattern.

The Components pattern was chosen as it provides a baseline for the results. The Bridge pattern and the Presentation Model were chosen as these patterns were merged for the Extended Bridge pattern.

**Reference Applications**

In order to determine whether the new Extended Bridge pattern had any merit, two reference applications were chosen. The first application is a simple bank account monitoring application displaying bank account records; the second is a quiz application displaying questions and allowing the user to answer questions. Each application was implemented in each of the four design patterns (Components, Bridge, Presentation Model and Extended Bridge) and several industry standard metrics were collected using tools available from the internet. The tools used and the metrics collected are described in the results section.

**BankAccounts application**

This application displays two bank accounts, checking and savings. It provides two user interfaces. The first demonstration user interface just shows the account details (see Figure 2.6).
The second full function user interface includes the ability to add and modify entries to either (see Figure 2.7). Both user interfaces provide the user with the ability to ask for a report of transactions starting at a specific date.

**Figure 2.6.** BankAccounts application demonstration user interface
In addition to providing two user interfaces this application was chosen because the layout of its user interface includes a reusable ‘BankAccount’ component. This component is used for the rendering of both the checking and savings account information (see Figure 2.8). Whilst the implementations of the user interface vary across the different design implementations, the functional component of the BankAccounts application is common to all implementations and provides access to the bank account records and data associated with the initial and current state of the reporting options.
QuizMaster application.

Although this application does not provide two user interfaces it was included as it contained interchangeable user interface components with differing functional requirements. This application displays test questions from a database, selected by the user using option buttons at the bottom of the window (see Figure 2.9). The options allow the user to select questions related to programming languages at different test levels. The user also has the ability to view their results for the questions they have already answered (see Figure 2.10). Finally the user can select to answer the questions of any selected test. Although it is indicated to the user if they have already answered a question, they can always try again (see Figure 2.11).
**Figure 2.9.** QuizMaster showing the questions

**Figure 2.10.** QuizMaster showing the test results
The design of the application used three similar panels for the respective views, with only one of them visible at any one time (see Figure 2.12). As with the BankAccounts application, the functional component of the QuizMaster application was common to all implementations; it provided access to the questions and data associated with the initial and current state of the visual options.
Basic Implementation Considerations

The Form of each application was managed by a UserInterface class. This class created the container for the user interface visual elements. The container in these applications was derived from the Java Swing JFrame class. The visual elements were derived from the Java Swing JPanel class.

The Function class was responsible for retrieving the data from the external sources, saving any changes to the data, and for performing tasks in support of the user interface, such as printing the reports for the BankAccounts application. The report printing was not fully implemented, and was provided as dummy functionality as it was outside of the scope of the thesis. The Function class was also responsible for initializing the various option components within each user interface, such as the report options in the BankAccounts application and the initial panel and questions in the QuizMaster application; the user interfaces did not default any of these options, they are all driven from the Function class.

As the Function class held the repository of the bank records, the quiz questions and the state of the user interface options, information about selections and changes had to be communicated back to the Function class. Two options were considered:

- Make the Function class a Singleton. This is a class that can only have one instance, and is accessible from a Class level (static) method. Although this option was the most flexible it was dismissed as it provided no access control to the Function class. It also meant that the user interfaces could not be reused without a Function class being present.

- Pass a reference for the main user interface instance to every component. This instance provided the single point of contact to the application functionality.
Although this was cumbersome, it provided the greatest level of control and reusability, and was the option chosen.
Approach

Extended Bridge Pattern

The Extended Bridge pattern keeps the Abstraction and RefinedAbstraction classes from the Bridge pattern, but augments the other two Bridge pattern classes as it embraces the Presentation Model and the concept of the Single Responsibility Principle. The Implementor class remains, the ConcreteImplementor is divided to support the addition of the Presentation Model based data class, and finally the creation of a logic class to further separate the functionality of the user interface from the rendering code (see Figure 3.1).

![Figure 3.1. The Extended Bridge pattern](image)

Implementing the Reference Design Patterns

Each of the three reference design patterns are described again below. The descriptions include an architecture diagram showing how the design pattern is applied to the reference applications. Also included is a flow diagram showing the order in which the various components are instantiated.
Components

The first reference design pattern considered was component based. This design pattern broke down the user interface into self contained reusable components. Each component was responsible for its own content and associated actions. This design pattern provided a baseline for comparing the other design patterns. The design has the Application (Function) class create an instance of the UserInterface (Form) class, which in turn created instances of the component classes. See Figure 3.2 for the architecture diagram for the BankAccount application and Figure 3.3 for the architecture diagram for the QuizMaster application. The instantiation flow for the BankAccount application is shown in Figure 3.4 and Figure 3.5 for the QuizMaster application.

**Figure 3.2.** Architecture diagram for the BankAccount application
Figure 3.3. Architecture diagram for the QuizMaster application
Figure 3.4. Instantiation flow for the BankAccount application

1) UserInterface created an instance of View.
2) UserInterface created an instance of BankAccount (CheckingAccount).
3) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of BankAccount (SavingAccount).
5) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the ReportPanel.
7) The instance of the ReportPanel was returned to the View so that it can be included into the user interface.
Figure 3.5. Instantiation flow for the QuizMaster application

1) UserInterface created an instance of View.
2) UserInterface created an instance of Questions.
3) The instance of the Questions was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of Results.
5) The instance of the Results was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the Tests.
7) The instance of the Tests was returned to the View so that it can be included into the user interface.
8) UserInstance created an instance of the Options.
9) The instance of the Options was returned to the View so that it can be included into the user interface.

Bridge

In order to apply the second reference design pattern, the Bridge, to the Components pattern, substantial changes had to be made. The single UserInterface class was replaced with UserInterfaceAbstraction and UserInterfaceRefinedAbstraction classes, and the rendering of the container and components were replaced with Implementor and ConcreteImplementor (which for
the reference applications became JavaImplementor). See Figure 3.6 for the architecture diagram for the BankAccount application and Figure 3.7 for the architecture diagram for the QuizMaster application.

As before, the design has the Function class create the user interface, which this time was an instance of UserInterfaceRefinedAbstraction class. The instantiation flow for the BankAccount application is shown in Figure 3.8 and Figure 3.9 for the QuizMaster application.

**Figure 3.6. Architecture diagram for the BankAccount application**

![Architecture diagram for the BankAccount application](image-url)
Figure 3.7. Architecture diagram for the QuizMaster application
Figure 3.8. Instantiation flow for the BankAccount application

1) UserInterface created an instance of View.
2) UserInterface created an instance of BankAccount (CheckingAccount).
3) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of BankAccount (SavingAccount).
5) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the ReportPanel.
7) The instance of the ReportPanel was returned to the View so that it can be included into the user interface.
Figure 3.9. Instantiation flow for the QuizMaster application

1) UserInterface created an instance of View.
2) UserInterface created an instance of Questions.
3) The instance of the Questions was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of Results.
5) The instance of the Results was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the Tests.
7) The instance of the Tests was returned to the View so that it can be included into the user interface
8) UserInstance created an instance of the Options.
9) The instance of the Options was returned to the View so that it can be included into the user interface

Presentation model

The third reference design pattern removes the state and data information from the base Components pattern and adds data classes to each component. See Figure 3.10 for the architecture diagram for the BankAccount application and Figure 3.11 for the architecture diagram for the QuizMaster application. The Function class creates an instance of the
UserInterface class, which creates an instance of the View class. So that the data class was not
cognizant of the component using it, the instance of the data class was created and then passed to
the instance of the component class. The instantiation flow for the BankAccount application is
shown in Figure 3.12 and Figure 3.13 for the QuizMaster application.

**Figure 3.10.** Architecture diagram for the BankAccount application
Figure 3.11. Architecture diagram for the QuizMaster application
Figure 3.12. Instantiation flow for the BankAccount application

1) UserInterface created an instance of View.
2) UserInterface created an instance of BankAccount (CheckingAccount).
3) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of BankAccount (SavingAccount).
5) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the ReportPanel.
7) The instance of the ReportPanel was returned to the View so that it can be included into the user interface.
Figure 3.13. Instantiation flow for the QuizMaster application

1) UserInterface created an instance of View.
2) UserInterface created an instance of Questions.
3) The instance of the Questions was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of Results.
5) The instance of the Results was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the Tests.
7) The instance of the Tests was returned to the View so that it can be included into the user interface.
8) UserInstance created an instance of the Options.
9) The instance of the Options was returned to the View so that it can be included into the user interface.
Implementing the Extended Bridge Pattern

The Extended Bridge pattern merges the Bridge pattern, the Presentation Model and provides support for the Single Responsibility principle. The architecture consists of UserInterfaceAbstraction, UserInterfaceRefinedAbstraction, Implementor, ConcreteImplementor, Data and Logic.

Extended bridge pattern

Not every component in the reference applications required full support from the design pattern. For instance the Questions and Results classes in the QuizMaster application only required the support of a Data class. The classes rendered read-only information and so there was no requirement for a Logic class. See Figure 3.14 for the architecture diagram for the BankAccount application and Figure 3.15 for the architecture diagram for the QuizMaster application. As before, the design has the Function class create user interface, which like the Bridge pattern is an instance of UserInterfaceRefinedAbstraction class, which creates an instance of the View class. So that the data and logic classes are not cognizant of the component using them, the instance of the data class was created and then passed to the instance logic class; they were then both passed to an instance of the component class. The instantiation flow for the BankAccount application is shown in Figure 3.16 and Figure 3.17 for the QuizMaster application.
**Figure 3.14.** Architecture diagram for the BankAccount application

**Figure 3.15.** Architecture diagram for the QuizMaster application
1) UserInterface created an instance of View.
2) UserInterface created an instance of BankAccount (CheckingAccount).
3) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of BankAccount (SavingAccount).
5) The instance of the BankAccount was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the ReportPanel.
7) The instance of the ReportPanel was returned to the View so that it can be included into the user interface.
Figure 3.17. Instantiation flow for the QuizMaster application

1) UserInterface created an instance of View.
2) UserInterface created an instance of Questions.
3) The instance of the Questions was returned to the View so that it could be included into the user interface.
4) UserInterface created an instance of Results.
5) The instance of the Results was returned to the View so that it could be included into the user interface.
6) UserInstance created an instance of the Tests.
7) The instance of the Tests was returned to the View so that it can be included into the user interface.
8) UserInstance created an instance of the Options.
9) The instance of the Options was returned to the View so that it can be included into the user interface.
Results

This chapter details the metrics that were collected for analysis and provide comparison of the Extended Bridge pattern with the three reference design patterns. Some metrics were rejected metrics, these will be discussed. Also discussed are the tools that were used, differences between the tools for similar metrics. Included are the assumptions that were made regarding the implementations of some of the classes. Finally a presentation of the results obtained.

Metrics Collected

The metrics collected were for the most part industry standard metrics:

1. *Number of Classes*, this is a general metric and provides an indication of the complexity of the implementation. It was used for comparison of the implementations.

2. *Number of Methods per Class*, this provides an indication of the complexity of a class. It was used for comparison of the implementations.

3. *McCabe’s Cyclomatic Complexity* (MCC) number is a measurement of the testable paths within a method (McCabe, 1976).

4. *Weighted Methods per Class* (WMC), this provides an indication of the complexity of a class based on the sum of the MCC number for the methods in the class (Chidamber & Kemerer, 1994)

5. *Non-Commenting Source Statements* (NCSS) is a metric that measures the executable lines of code. It does not take in account the lines of declarative code for the class variables (JavaNCSS).

6. *Distribution of Complexity* is a new metric that divides the NCSS value by the WMC. This metric is used to show by comparison the efficiencies of the different
implementations. Additional information for this new metric can be found in the Discussion chapter below.

Rejected Metrics

The following metrics were rejected

- Total Lines Of Code (TLOC) is a measure of the absolute number of lines coded. This metric was rejected as it is not an objective metric. The value obtained for this metric can be adversely affected by coding styles. Simply reformatting the code causes the value of the metric to change.

- Method Lines of Code is a measure of the absolute number of lines coded in a method. The value obtained for this metric can be adversely affected by coding styles. Simply reformatting the code causes the value of the metric to change.

- Lack of Cohesion Of Methods (LCOM) is a measure of how integrated the methods are with the variables of a class. This metric was rejected because in some of the design patterns the data is stored in an independent class and surrounded by getter and setter methods. According to LCOM value obtained the class should be broken into smaller classes. When the data only has a context when taken as a whole, this metric becomes meaningless.

Metric Collection Tools

Several tools were used for the collection of the metrics

Eclipse metrics plug-in.

The Eclipse metrics plug-in provided an extensive range of metrics (Eclipse)(Metrics Plugin). For each of the metrics described above, it provided application, class and method level
values as well as mean and standard deviation values. This tool was integrated into the Eclipse development environment. This tool provides metrics 1, 2, 3 and 4.

**JavaNCSS.**

This is a standalone tool that provided good coverage for the metrics described previously (JavaNCSS). This tool functioned by examining the source code. This tool provides metrics 1, 2, 3, 4 and 5.

**Understand.**

This is a standalone tool that provided good coverage for the metrics described previously (Understand). This tool functioned by examining the source code. This tool provides metrics 1, 2, 3, 4 and 5.

**Tool differences**

JavaNCSS increments its NCSS count for a missing ‘else’ statement. So an ‘if-else’ control structure increments the line count by 2; and a standalone ‘if’ also increments the line count by 2.

Understand has several counters that can be combined to produce NCSS equivalent results. Combining the two metrics ‘CountLineCodeDecl’ (Understand, 2012) and ‘CountLineCodeExe’ (Understand, 2012) gives an NCSS result similar to the result from JavaNCSS, with the exception that it increments the counts for each part of the ‘if-else’ control structure individually. So an ‘if-else’ control structure increments the line count by 2; whereas a standalone ‘if’ only increments the line count by 1. Combining the two metrics ‘CountStmtDecl’ (Understand, 2012) and ‘CountStmtExe’ (Understand, 2012) gives an approximate result that counts the ‘if-else’ control structure as a single statement, in effect ignoring the ‘else’ statement.
For the most part the tools produce similar results. However some methods do generate different results.

**Implementation Assumptions**

For both of the applications there was a MainProgram class, an ATableClass class and a Function class. Whilst there were differences in these classes between the BankAccounts version and the QuizMaster version, they remained constant within the reference application design implementations. The MainProgram class was responsible for instantiating the Function instance. The Function instance was then responsible for instantiating the user interface, initializing any options or settings, preparing the data and instructing it to refresh. The ATableClass was an augmentation class to allow checkboxes to be displayed in a JTable; and to manage whether the JTable could be edited or selected. As the MainProgram, ATableClass and Function classes had been designated as implementation support classes, the assumption was that their metrics should not been recorded.
Results Obtained

**BankAccounts reference design implementations**

The values of three metrics from the BankAccounts implementations were collected (Table 4.1) and used for comparison ratios.

<table>
<thead>
<tr>
<th>Reference design</th>
<th>Number of methods</th>
<th>WMC</th>
<th>NCSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>26</td>
<td>49</td>
<td>266</td>
</tr>
<tr>
<td>Bridge</td>
<td>32</td>
<td>55</td>
<td>276</td>
</tr>
<tr>
<td>Presentation Model</td>
<td>46</td>
<td>72</td>
<td>340</td>
</tr>
<tr>
<td>Extended Bridge</td>
<td>53</td>
<td>78</td>
<td>367</td>
</tr>
</tbody>
</table>

The ratios were as follows:

1) WMC / Number of methods – gives the Average Complexity (Figure 4.1)

2) NCSS / WMC – gives the Distribution of Complexity (Figure 4.2)

3) NCSS / Number of methods – gives the Average Number of Executable Lines per Method (Figure 4.3)

![Figure 4.1. Average Complexity per design pattern](image1)

![Figure 4.2. Distribution of Complexity per design pattern](image2)
QuizMaster reference design implementations

The values of three metrics from the QuizMaster implementation were collected (Table 4.2) and used for comparison ratios.
Table 4.2
Key metric values for the QuizMaster implementations

<table>
<thead>
<tr>
<th>Reference designs</th>
<th>Number of methods</th>
<th>WMC</th>
<th>NCSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>27</td>
<td>50</td>
<td>278</td>
</tr>
<tr>
<td>Bridge</td>
<td>37</td>
<td>60</td>
<td>289</td>
</tr>
<tr>
<td>Presentation Model</td>
<td>56</td>
<td>86</td>
<td>395</td>
</tr>
<tr>
<td>Extended Bridge</td>
<td>70</td>
<td>100</td>
<td>426</td>
</tr>
</tbody>
</table>

The ratios were as follows:

1) WMC / Number of methods – gives the Average Complexity (Figure 4.4)
2) NCSS / WMC – gives the Distribution of Complexity (Figure 4.5)
3) NCSS / Number of methods – gives the Average Number of Executable Lines per Method (Figure 4.6)

Figure 4.4. Average Complexity per design pattern

---

**QuizMaster**

Figure 4.4. Divide WMC by the number of methods
**Figure 4.5.** Distribution of Complexity per design pattern

**Figure 4.6.** Average Number of Executable Lines per Method per design pattern

Figure 4.5. Divide NCSS by WMC

Figure 4.6. Divide NCSS by the number of methods
Discussion

The purpose of this thesis was to introduce the Extended Bridge pattern as a new object-oriented software design pattern that leveraged the best features of existing design patterns and provided support for the Single Responsibility principle. The new design pattern was accompanied by metrics showing how, by comparison, it provided better results than the reference designs.

The existing metrics provided some challenges; either they were fickle and subject to the whims of coding style, such as Lines of Code; or could be manipulated simply by adding methods, most metrics fall into this category. What was needed was a metric that used absolute values. The answer was a new metric called Distribution of Complexity. It took the number of executable lines of code and divided it by the complexity of that code. The metric was used to provide a comparison between the reference design patterns; and was used to show the efficiency of the executable code. For instance, if a developer rewrote a module to use fewer lines of code, with the same complexity that achieved the same outcome, the new metric would result in a lower value; thereby showing that the rewritten code had greater efficiency. Unfortunately the results from both MCC and WMC would remain unchanged, as the complexity had not changed; thereby providing no feedback to the developer that the rewritten code was an improvement over the original code.

The contributing design patterns went from the most simplistic Components pattern, through a fully abstracted interface and implementation in the Bridge design, to the extraction of the data into an independent class in the Presentation Model design.

The reference applications were chosen to exercise the various designs and to provide viable metric values. The BankAccounts application had a single container class, a reusable bank
account panel showing the account details and a report panel. The BankAccounts application also provided a full featured user interface and a demonstration user interface. The QuizMaster application had a single container class, multiple interchangeable panels, of which only one was visible at a time, it also included an options panel.

Although the data for several metrics were collected for the different design implementation for the two reference applications, only three were used for comparison ratios. The ratios were as follows:

1) WMC / Number of methods – gives average complexity per design
2) NCSS / WMC – gives Distribution of Complexity per design
3) NCSS / Number of methods – gives average number of lines per method per design

Ratios 1 & 3 were used to provide absolute numbers, where it can be said that smaller is better. Ratio 2 was used for comparative purposes, comparing overall implementations and individual modules.

During the implementation of the reference applications, it was found that although writing a method containing code that is used only once has little engineering value, it was possible to adversely affect the data and resulting ratios. Creating a method is most effective when it either encapsulates repeated code, or is used to break down complex control structures, thereby reducing the MCC for the method and the associated WMC. As the Distribution of Complexity ratio did not rely on a count of the number of methods written it proved to be the most stable of the metrics when used to compare the different reference implementations.

It can be seen from the results that the individual metric results increase, as the reference implementations are measured for each of the reference design patterns Components, Bridge, Presentation Model and Extended Bridge. However when the results are viewed in the context of
the following ratios: Average Complexity, Distribution of Complexity and Average Number of Lines per Method; the ratio results decline. The associated charts show that the Extended Bridge design pattern returns the best overall ratio results when compared to the three reference design patterns.

**Conclusions**

The thesis introduced the Extended Bridge design pattern. The lineage of the design pattern was fully described, showing how it merged the existing Bridge and Presentation Model design patterns. It was also shown how the Extended Bridge design pattern included to Single Responsibility principle to increase maintainability, increase testability and increase reusability. The data gathered from industry standard metrics and the new metric, the Distribution of Complexity, showed that the Extended Bridge design pattern returned better results when compared alongside the reference design patterns.

The original scope of the thesis did not foresee the introduction of the new software metric, the Distribution of Complexity; although this metric proved invaluable. The new metric allowed code that performed the same function, which had been implemented for different design patterns to be compared in a reliable and consistent manner.

A future area of study is necessary to completely explore the effectiveness of the Extended Bridge design pattern as it is applied to other application user interfaces, for instance kiosk or browser. Another future area of study is necessary to further explore the new Distribution of Complexity metric.
References


Eclipse (Version 3.7.1) [Computer Software]. Eclipse Foundation


JavaNCSS (Version 32.53) [Computer Software]. http://www.kclee.de/clemens/java/javancss/: Clemens Lee


Metrics Plugin (Version 1.3.6) [Computer Software]. http://metrics.sourceforge.net: Frank Sauer


Understand (Version 3.0, build 620) [Computer Software]. St. George, UT: Scientific Toolworks, Inc.


Appendix A

Common Classes

MainProgram.java

```java
public class MainProgram {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private Function function;

    void initialize () {
        function = new Function ();
        function.setupReportOptions ("10/6/2010", Function.checkingFlag, false);
        function.loadData ();
        function.refreshUI ();
        function.displayUI ();
    }

    public static void main (String [] args) {
        MainProgram mainProgram = new MainProgram ();
        mainProgram.initialize ();
    }
}
```

Function.java

```java
public class Function {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    // {Components}  private UserInterface userInterface;
    // {Bridge}      private UserInterfaceAbstraction userInterface;
    // {Presentation} private UserInterface userInterface;
    // {ExtendedBridge} private UserInterfaceAbstraction userInterface;

    public Function () {
        userInterface = new "UserInterface" (this, 2, complexAccountPanel);
    }

    public void setupReportOptions (String startDate, int accountFlag, boolean reconCBFlag) {
        userInterface.setupReportOptions (startDate, accountFlag, reconCBFlag);
    }

    public void loadData () {
        // open the account 'database' and read in the data
    }

    public void refreshUI () {
        userInterface.refreshData ();
    }
}
public void displayUI ()
{
    userInterface.display ();
}

public String [][] getData (int account)
{
    // return the selected 'cache' data
}

public void saveData (String contentName, int type, String saveData[][])
{
    // save data to the 'database'
}

public void addRow (int type, String newDate, String desc, String newAmountStr,
                    boolean recon)
{
    // add a new row to the 'cache'
}

public void modifyRow (int type, int selectedRow, String newDate, String desc,
                       String newAmountStr, boolean recon)
{
    // modify selected row in the 'cache'
}

public void printReport ()
{
    // output data based on report options
}

public void recordReportOptions (String startDate, int accountOptionsFlag,
                                 boolean reconCBSelected)
{
    // update local variables with report options
}
}
Appendix B

Component Model

/bankAccount_Components/BankAccount.java

abstract public class BankAccount extends JPanel
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    public abstract void refreshPanel ();
}

/bankAccount_Components/ComplexBankAccount.java

public class ComplexBankAccount extends BankAccount
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private UserInterface userInterface

    public ComplexBankAccount (UserInterface userInterface, String contentName, int type)
    {
        // create complex account interface
    }

    public void refreshPanel ()
    {
        // removeRow, getData, addRow
    }

    private class ActionButtonListener implements ActionListener
    {
        public void actionPerformed (ActionEvent ae)
        {
            if (button == contentUpdateBtn)
            {
                if (addRB.isSelected ()) // add new row
                    else if (modifyRow.isSelected ()) // modify selected row
            }
            else if (button == contentSaveBtn)
                saveData (...)
            else if (button == addRB)
                // prepare to add a new row
            else if (button == modifyRB)
                // allow rows to be selected
        }
    }

    private class RowListener implements ListSelectionListener
    {
        public void valueChanged (ListSelectionEvent event)
        {
            // move the selected row into the edit area
        }
    }
}
/bankAccount_Components/Container.java

public class Container extends JFrame {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    public Container (int accountPanelType) {
        // initialize title, setup internal JPanel
    }

    public void display () {
        // initialize the JFrame
    }

    public void addBankAccount (BankAccount bankAccount) {
        // add the bankAccount
    }

    public void addReport (Report report) {
        // add the report
    }
}

/bankAccount_Components/Report.java

public class Report extends JPanel {
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private UserInterface userInterface;

    public Report (UserInterface userInterface) {
        // create report interface
    }

    public void setupReportOptions (String startDate, int accountOptionsFlag, boolean reconCBSelected) {
        // initialize the report options
    }

    private class ActionButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent ae) {
            String startDate;
            boolean cFlag, sFlag;
            if (ae.getSource () == reportBtn) {
                // collect the values from the report interface
                userInterface.recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
                userInterface.printReport ();
            }
        }
    }
}
public class SimpleBankAccount extends BankAccount
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private UserInterface userInterface

    public SimpleBankAccount (UserInterface userInterface, String contentName,
                                int type)
    {
        // create simple account interface
    }

    public void refreshPanel ()
    {
        // removeRow, getData, addRow
    }

/publicAccount_Components/UserInterface.java

public class UserInterface
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private Function function;
    private Container container;
    private ArrayList<BankAccount> bankAccounts;
    private Report reportPanel;

    public UserInterface (Function function, int accountCount,
                          int accountPanelType)
    {
        this.function = function;
        container = new Container (accountPanelType);
        bankAccounts = new ArrayList<BankAccount> (accountCount);
        if (accountPanelType == Function.complexAccountPanel)
        {
            setupComplexBankAccountPanel ("checking", Function.checkingAccount);
            setupComplexBankAccountPanel ("saving", Function.savingAccount);
        } else
        {
            setupSimpleBankAccountPanel ("checking", Function.checkingAccount);
            setupSimpleBankAccountPanel ("saving", Function.savingAccount);
        }
        reportPanel = new Report (this);
        container.addReport (reportPanel);
    }

    public void refreshData ()
    {
        // iterate over accounts and refresh them
    }
public void display ()
{
    container.display () ;
}

private void setupComplexBankAccountPanel (String contentName, int accountType)
{
    new ComplexBankAccount (this, contentName, accountType);
}

private void setupSimpleBankAccountPanel (String contentName, int accountType)
{
    new SimpleBankAccount (this, contentName, accountType);
}

public void setupReportOptions (String startDate, int accountOptionsFlag,
                                boolean reconCBSelected)
{
    recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
    reportPanel.setupReportOptions (startDate, accountOptionsFlag,
                                    reconCBSelected);
}

public void recordReportOptions (String startDate, int accountOptionsFlag,
                                 boolean reconCBSelected)
{
    function.recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
}

public String [][] getData (int accountType)
{
    return function.getData (accountType);
}

public void printReport ()
{
    function.printReport ();
}

public void addRow (int accountType, String newDate, String desc,
                    String newAmountStr, boolean recon)
{
    function.addRow (accountType, newDate, desc, newAmountStr, recon);
    refreshData ();
}

public void modifyRow (int accountType, int selectedRow, String newDate,
                       String desc, String newAmountStr, boolean recon)
{
    function.modifyRow (accountType, selectedRow, newDate, desc,
                        newAmountStr, recon);
    refreshData ();
}

public void saveData (String contentName, int accountType, String saveData[][])
{
    function.saveData (contentName, accountType, saveData);
}
Appendix C

Bridge design pattern

/bankAccount_Bridge/BankAccountImplementor.java

```java
public interface BankAccountImplementor {
    public void refreshData();
}
```

/bankAccount_Bridge/BankAccountJavaImplementor.java

```java
abstract public class BankAccountJavaImplementor extends JPanel implements BankAccountImplementor {
    private final String copyright = "Copyright 2012, Andrew Haigh";
    public abstract void refreshData();
}
```

/bankAccount_Bridge/ComplexBankAccountJavaImplementor.java

```java
public class ComplexBankAccountJavaImplementor extends BankAccountJavaImplementor {
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private UserInterfaceAbstraction userInterface;

    public ComplexBankAccount (UserInterfaceAbstraction userInterface,
                                String contentName, int type) {
        // create complex account interface
    }

    public void refreshPanel () {
        // removeRow, getData, addRow
    }

    private class ActionButtonListener implements ActionListener {
        public void actionPerformed (ActionEvent ae) {
            if (button == contentUpdateBtn) {
                if (addRB.isSelected()) // add new row
                    else if (modifyRow.isSelected()) // modify selected row
            }
            else if (button == contentSaveBtn)
                saveData(...)
            else if (button == addRB) // prepare to add a new row
                else if (button == modifyRB) // allow rows to be selected
        }
    }

    private class RowListener implements ListSelectionListener {
        ...
    }
```
public void valueChanged (ListSelectionEvent event)
{
    // move the selected row into the edit area
}

/*bankAccount_Bridge/ContainerImplementor.java*/

public interface ContainerImplementor
{
    public void addBankAccount (BankAccountImplementor bankAccount);
    public void addReport (ReportImplementor report);
    public void display ();
}

/*bankAccount_Bridge/ContainerJavaImplementor.java*/

public class Container extends JFrame
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    public Container (int accountPanelType)
    {
        // initialize title, setup internal JPanel
    }

    public void display ()
    {
        // initialize the JFrame
    }

    public void addBankAccount (BankAccount bankAccount)
    {
        // add the bankAccount
    }

    public void addReport (Report report)
    {
        // add the report
    }
}

/*bankAccount_Bridge/ReportImplementor.java*/

public interface ReportImplementor
{
    public void setupReportOptions (String startDate, int accountOptionsFlag,
        boolean reconCBSelected);
}

/*bankAccount_Bridge/ReportJavaImplementor.java*/

public class Report extends JPanel
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private UserInterfaceAbstraction userInterface;
public Report (UserInterfaceAbstraction userInterface)
{
    // create report interface
}

public void setupReportOptions (String startDate, int accountOptionsFlag,
    boolean reconCBSelected)
{
    // initialize the report options
}

private class ActionButtonListener implements ActionListener
{
    public void actionPerformed (ActionEvent ae)
    {
        String startDate;
        boolean cFlag, sFlag;

        if (ae.getSource () == reportBtn)
        {
            // collect the values from the report interface
            userInterface.recordReportOptions (startDate, accountOptionsFlag,
                reconCBSelected);
            userInterface.printReport ();
        }
    }
}

/bankAccount_Bridge/SimpleBankAccountJavaImplementor.java

public class SimpleBankAccount extends BankAccount
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private UserInterfaceAbstraction userInterface

    public SimpleBankAccount (UserInterfaceAbstraction userInterface,
        String contentName, int type)
    {
        // create simple account interface
    }

    public void refreshPanel ()
    {
        // removeRow, getData, addRow
    }
}

/bankAccount_Bridge/UserInterfaceAbstraction.java

abstract public class UserInterfaceAbstraction
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private Function function;
    protected ContainerImplementor containerImpl = null;
    protected ArrayList<BankAccountImplementor> bankAccounts;
    protected BankAccountImplementor bankAccountPanel;
    protected ReportImplementor reportPanel;
protected UserInterfaceAbstraction (Function function)
{
    this.function = function;
}

public void setupReportOptions (String startDate, int accountOptionsFlag, boolean reconCBSelected)
{
    recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
    reportPanel.setupReportOptions (startDate, accountOptionsFlag, reconCBSelected);
}

public void refreshData ()
{
    // iterate over accounts and refresh them
}

public void display ()
{
    containerImpl.display ();
}

public void recordReportOptions (String startDate, int accountOptionsFlag, boolean reconCBSelected)
{
    function.recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
}

public String [][] getData (int account)
{
    return function.getData (account);
}

public void printReport ()
{
    function.printReport ();
}

public void addRow (int type, String newDate, String desc, String newAmountStr, boolean recon)
{
    function.addRow (type, newDate, desc, newAmountStr, recon);
    refreshData ();
}

public void modifyRow (int type, int selectedRow, String newDate, String desc, String newAmountStr, boolean recon)
{
    function.modifyRow (type, selectedRow, newDate, desc, newAmountStr, recon);
    refreshData ();
}

public void saveData (String contentName, int type, String saveData[][][])
{
    function.saveData (contentName, type, saveData);
}

/bankAccount_Bridge/UserInterfaceRefinedAbstraction.java

public class UserInterfaceRefinedAbstraction extends UserInterfaceAbstraction
private final String copyright = "Copyright 2012, Andrew Haigh";

UserInterfaceRefinedAbstraction (Function function, int accountCount,
int accountPanelType)
{
    super (function);

    BankAccountImplementor tempBAI;
    ReportImplementor tempRI;

    containerImpl = new ContainerJavaImplementor (accountCount, accountPanelType);

    bankAccounts = new ArrayList<BankAccountImplementor> (accountCount);

    tempBAI = setupBankAccountPanel ("checking", Function.checkingAccount,
accountPanelType);
    bankAccounts.add (tempBAI);
    containerImpl.addBankAccount (tempBAI);

    tempBAI = setupBankAccountPanel ("saving", Function.savingAccount, 
accountPanelType);
    bankAccounts.add (tempBAI);
    containerImpl.addBankAccount (tempBAI);

    tempRI = setupReportPanel ();
    containerImpl.addReport (tempRI);
}

private BankAccountImplementor setupBankAccountPanel (String contentName,
int accountType,
int accountPanelType)
{
    if (accountPanelType == Function.complexAccountPanel)
    {
        bankAccountPanel = new ComplexBankAccountJavaImplementor (this,
contentName, accountType);
    }
    else
    {
        bankAccountPanel = new SimpleBankAccountJavaImplementor (this, 
contentName, accountType);
    }
    return bankAccountPanel;
}

private ReportImplementor setupReportPanel ()
{
    reportPanel = new ReportJavaImplementor (this);
    return reportPanel;
}
Appendix D

Presentation Model

/bankAccount_Presentation/BankAccount.java

abstract public class BankAccount extends JPanel
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    public abstract void refreshPanel();
}

/bankAccount_Presentation/BankAccountData.java

public class BankAccountData
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private UserInterface userInterface;
    private String data[][];
    private String contentName;
    private int accountType;

    public BankAccountData (UserInterface userInterface, String contentName, int accountType)
    {
        this.userInterface = userInterface;
        this.contentName = contentName;
        this.accountType = accountType;
    }

    public String [][] getData ()
    {
        return data;
    }

    public void refreshData ()
    {
        data = userInterface.getData (accountType);
    }

    public void addRow (String newDate, String desc, String newAmountStr, boolean recon)
    {
        userInterface.addRow (accountType, newDate, desc, newAmountStr, recon);
        refreshData ();
    }

    public void modifyRow (int selectedRow, String newDa
t, String desc, String newAmountStr, boolean recon)
    {
        userInterface.modifyRow (accountType, selectedRow, newDate, desc, newAmountStr, recon);
        refreshData ();
    }

    public void saveData (String saveData[][])
    {
        userInterface.saveData (contentName, accountType, saveData);
    }
public String getContentName ()
{
    return contentName;
}

public int getAccountType ()
{
    return accountType;
}

/bankAccount_Presentation/ComplexBankAccount.java

public class ComplexBankAccount extends BankAccount
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private BankAccountData bankAccountData;

    public ComplexBankAccount (BankAccountData bankAccountData)
    {
        // get column names from BankAccountData
        // create complex account interface
    }

    public void refreshPanel ()
    {
        // removeRow
        // getData from BankAccountData
        // addRow
    }

    private class ActionButtonListener implements ActionListener
    {
        public void actionPerformed (ActionEvent ae)
        {
            if (button == contentUpdateBtn)
            {
                if (addRB.isSelected ())
                    // add new row to BankAccountData
                else if (modifyRow.isSelected ())
                    // modify selected row in BankAccountData
            }
            else if (button == contentSaveBtn)
                saveData (...) // in BankAccountData
            else if (button == addRB)
                // prepare to add a new row
            else if (button == modifyRB)
                // allow rows to be selected
        }
    }

    private class RowListener implements ListSelectionListener
    {
        public void valueChanged (ListSelectionEvent event)
        {
            // move the selected row into the edit area
        }
    }
}
/bankAccount_Presentation/Container.java

public class Container extends JFrame
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private JPanel inside;
    private int accountPanelType;

    public Container (int accountPanelType)
    {
        // initialize title, setup internal JPanel
    }

    public void display ()
    {
        // initialize the JFrame
    }

    public void addBankAccount (BankAccount bankAccount)
    {
        // add the bankAccount
    }

    public void addReport (Report report)
    {
        // add the report
    }
}

/bankAccount_Presentation/Report.java

public class Report extends JPanel
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private ReportData reportData;

    public Report (ReportData reportData)
    {
        // create report interface
    }

    public void setupReportOptions ()
    {
        // initialize the report options from ReportData
    }

    private class ActionButtonListener implements ActionListener
    {
        public void actionPerformed (ActionEvent ae)
        {
            Object button = ae.getSource ();
            if (button == reportCheckingRB)
            {
                reportData.setAccountOption (Function.checkingFlag);
            }
            else if (button == reportSavingRB)
            {
                reportData.setAccountOption (Function.savingFlag);
            }
        }
    }
else if (button == reportBothRB)
{
    reportData.setAccountOption (Function.bothFlag);
}
else if (button == reportBtn)
{
    reportData.setStartDate (reportDateTF.getText ());
    reportData.setReconCBSelected (reportReconCB.isSelected ());
    reportData.printReport ();
}

/publicAccount_Presentation/ReportData.java

public class ReportData
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private UserInterface userInterface;
    private String startDate;
    private boolean checkingSelected, savingSelected, bothSelected, reconCBSelected;
    private int accountOptionsFlag;

    public ReportData (UserInterface userInterface)
    {
        this.userInterface = userInterface;
    }

    public void setupReportOptions (String startDate, int accountOptionsFlag, boolean reconCBSelected)
    {
        setStartDate (startDate);
        setAccountOption (accountOptionsFlag);
        setReconCBSelected (reconCBSelected);
    }

    public String getStartDate ()
    {
        return startDate;
    }

    public void setStartDate (String startDate)
    {
        this.startDate = startDate;
    }

    public boolean isCheckingSelected ()
    {
        return checkingSelected;
    }

    public boolean isSavingSelected ()
    {
        return savingSelected;
    }

    public boolean isBothSelected ()
    {
        return bothSelected;
    }
public void setAccountOption (int accountOptionsFlag)
{
    // record the report options locally
}

public boolean isReconCBSelected ()
{
    return reconCBSelected;
}

public void setReconCBSelected (boolean reconCBSelected)
{
    this.reconCBSelected = reconCBSelected;
}

public void printReport ()
{
    userInterface.recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
    userInterface.printReport ();
}

/bankAccount_Presentation/SimpleBankAccount.java

public class SimpleBankAccount extends BankAccount
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private BankAccountData bankAccountData;

    public SimpleBankAccount (BankAccountData bankAccountData)
    {
        // get column names from BankAccountData
        // create simple account interface
    }

    public void refreshPanel ()
    {
        // removeRow
        // getData from BankAccountData
        // addRow
    }
}

/bankAccount_Presentation/UserInterface.java

public class UserInterface
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    public UserInterface (Function function, int accountCount, int accountPanelType)
    {
        this.function = function;
        new Container (accountPanelType);

        if (accountPanelType == Function.complexAccountPanel)
        {
            setupComplexBankAccountPanel ("checking", Function.checkingAccount);
            setupComplexBankAccountPanel ("saving", Function.savingAccount);
        }
    }
else {
    setupSimpleBankAccountPanel("checking", Function.checkingAccount);
    setupSimpleBankAccountPanel("saving", Function.savingAccount);
}

new ReportData(this);
new Report(reportData);

public void refreshData () {
    // iterate over accounts and refresh data
    // iterate over accounts and refresh user interface
}

public void display () {
    container.display();
}

private void setupComplexBankAccountPanel(String contentName, int accountType) {
    new BankAccountData(this, contentName, accountType);
    new ComplexBankAccount(bankAccountData);
}

private void setupSimpleBankAccountPanel(String contentName, int accountType) {
    new BankAccountData(this, contentName, accountType);
    new SimpleBankAccount(bankAccountData);
}

public void setupReportOptions(String startDate, int accountOptionsFlag, boolean reconCBSelected) {
    recordReportOptions(startDate, accountOptionsFlag, reconCBSelected);
    // initialize the report data
    // initialize the report user interface
}

public void recordReportOptions(String startDate, int accountOptionsFlag, boolean reconCBSelected) {
    function.recordReportOptions(startDate, accountOptionsFlag, reconCBSelected);
}

public String [][] getData(int accountType) {
    return function.getData(accountType);
}

public void printReport () {
    function.printReport();
}

public void addRow(int accountType, String newDate, String desc, String newAmountStr, boolean recon) {
    function.addRow(accountType, newDate, desc, newAmountStr, recon);
    refreshData();
}
public void modifyRow (int accountType, int selectedRow, String newDate, String desc, String newAmountStr, boolean recon)
{
    function.modifyRow (accountType, selectedRow, newDate, desc, newAmountStr, recon);
    refreshData ();
}

public void saveData (String contentName, int accountType, String saveData[][]) 
{
    function.saveData (contentName, accountType, saveData);
}
Appendix E

Extended Bridge design pattern

/bankAccount_ExtendedBridge/BankAccountData.java

```java
public class BankAccountData {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private UserInterfaceAbstraction userInterfaceAbs;
    private String data[][];
    private String contentName;
    private int accountType;

    public BankAccountData (UserInterfaceAbstraction userInterfaceAbs,
                               String contentName, int accountType) {
        this.userInterfaceAbs = userInterfaceAbs;
        this.contentName = contentName;
        this.accountType = accountType;
    }

    public String [][] getData () {
        return data;
    }

    public void refreshData () {
        data = userInterfaceAbs.getData (accountType);
    }

    public String getContentName () {
        return contentName;
    }

    public int getAccountType () {
        return accountType;
    }
}
```

/bankAccount_ExtendedBridge/BankAccountImplementor.java

```java
public interface BankAccountImplementor {
    public void refreshPanel ();
}
```

/bankAccount_ExtendedBridge/BankAccountJavaImplementor.java

```java
abstract public class BankAccountJavaImplementor extends JPanel implements BankAccountImplementor {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    public abstract void refreshPanel ();
}
```
BankAccountLogic.java

```java
public class BankAccountLogic {
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private UserInterfaceAbstraction userInterface;
    private BankAccountData bankAccountData;

    public BankAccountLogic (UserInterfaceAbstraction userInterface, 
                            BankAccountData bankAccountData)
    {
        this.userInterface = userInterface;
        this.bankAccountData = bankAccountData;
    }

    public void addRow (String newDate, String desc, String newAmountStr, 
                         boolean recon)
    {
        int accountType;
        accountType = bankAccountData.getAccountType ();
        userInterface.addRow (accountType, newDate, desc, newAmountStr, recon);
        userInterface.refreshData ();
    }

    public void modifyRow (int selectedRow, String newDate, String desc, 
                           String newAmountStr, boolean recon)
    {
        int accountType;
        accountType = bankAccountData.getAccountType ();
        userInterface.modifyRow (accountType, selectedRow, newDate, desc, 
                                 newAmountStr, recon);
        userInterface.refreshData ();
    }

    public void saveData (String saveData[][]) 
    {
        String contentName;
        int accountType;

        contentName = bankAccountData.getContentName ();
        accountType = bankAccountData.getAccountType ();
        userInterface.saveData (contentName, accountType, saveData);
    }
}
```

ComplexBankAccountJavaImplementor.java

```java
public class ComplexBankAccountJavaImplementor extends BankAccountJavaImplementor 
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private BankAccountData bankAccountData;
    private BankAccountLogic bankAccountLogic;

    public ComplexBankAccountJavaImplementor (BankAccountData bankAccountData, 
                                               BankAccountLogic bankAccountLogic)
    {
    }
```
// get column names from BankAccountData
// create complex account interface

public void refreshPanel()
{
    // removeRow
    // getData from BankAccountData
    // addRow
}

private class ActionButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        if (button == contentUpdateBtn)
        {
            if (addRB.isSelected())
                // add new row using BankAccountLogic
            else if (modifyRow.isSelected())
                // modify selected row using BankAccountLogic
        }
        else if (button == contentSaveBtn)
            saveData(...); // using BankAccountLogic
        else if (button == addRB)
            // prepare to add a new row
        else if (button == modifyRB)
            // allow rows to be selected
    }
}

private class RowListener implements ListSelectionListener
{
    public void valueChanged(ListSelectionEvent event)
    {
        // move the selected row into the edit area
    }
}

/bankAccount_ExtendedBridge/ContainerImplementor.java

class ContainerImplementor
{
    public void addBankAccount(BankAccountImplementor bankAccount);
    public void addReport(ReportImplementor report);
    public void display();
}

/bankAccount_ExtendedBridge/ContainerJavaImplementor.java

class ContainerJavaImplementor extends JFrame implements ContainerImplementor
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private int accountPanelType;

    public ContainerJavaImplementor(int accountPanelType)
    {
        // initialize title, setup internal JPanel
    }
public void display ()
{
    // initialize the JFrame
}

public void addBankAccount (BankAccount bankAccount)
{
    // add the bankAccount
}

public void addReport (Report report)
{
    // add the report
}

/bankAccount_ExtendedBridge/ReportData.java

public class ReportData
{
    private final String copyright = "Copyright 2012, Andrew Haigh";

    private String startDate;
    private boolean checkingSelected, savingSelected, bothSelected, reconCBSelected;
    private int accountOptionsFlag;

    public void setupReportOptions (String startDate, int accountOptionsFlag,
        boolean reconCBSelected)
    {
        setStartDate (startDate);
        setAccountOption (accountOptionsFlag);
        setReconCBSelected (reconCBSelected);
    }

    public String getStartDate ()
    {
        return startDate;
    }

    public void setStartDate (String startDate)
    {
        this.startDate = startDate;
    }

    public boolean isCheckingSelected ()
    {
        return checkingSelected;
    }

    public boolean isSavingSelected ()
    {
        return savingSelected;
    }

    public boolean isBothSelected ()
    {
        return bothSelected;
    }

    public void setAccountOption (int accountOptionsFlag)
public int getAccountOptionsFlag ()
{
    return accountOptionsFlag;
}

public boolean isReconCBSelected ()
{
    return reconCBSelected;
}

public void setReconCBSelected (boolean reconCBSelected)
{
    this.reconCBSelected = reconCBSelected;
}

public interface ReportImplementor
{
    public void setupReportOptions ();
}

public class Report extends JPanel
{
    private final String copyright = "Copyright 2012, Andrew Haigh";
    private ReportData reportData;

    public Report (ReportData reportData)
    {
        // create report interface
    }

    public void setupReportOptions ()
    {
        // initialize the report options from ReportData
    }

    private class ActionButtonListener implements ActionListener
    {
        public void actionPerformed (ActionEvent ae)
        {
            Object button = ae.getSource ();
            if (button == reportCheckingRB)
            {
                reportData.setAccountOption (Function.checkingFlag);
            }
            else if (button == reportSavingRB)
            {
                reportData.setAccountOption (Function.savingFlag);
            }
            else if (button == reportBothRB)
            {
                reportData.setAccountOption (Function.bothFlag);
            }
        }
    }
else if (button == reportBtn)
{
    reportData.setStartDate (reportDateTF.getText ());
    reportData.setReconCBSelected (reportReconCB.isSelected ());
    reportData.printReport();
}

public class ReportLogic
{
    public ReportLogic (UserInterfaceAbstraction userInterfaceAbs,
                      ReportData reportData)
    {
        this.userInterfaceAbs = userInterfaceAbs;
        this.reportData = reportData;
    }

    public void printReport ()
    {
        // get data from report data
        userInterfaceAbs.recordReportOptions (startDate, accountOptionsFlag,
                                              reconCBSelected);
        userInterfaceAbs.printReport ();
    }
}

public class SimpleBankAccountJavaImplementor extends BankAccountJavaImplementor
{
    public SimpleBankAccountJavaImplementor (BankAccountData bankAccountData)
    {
        // get column names from BankAccountData
        // create simple account interface
    }

    public void refreshPanel ()
    {
        // removeRow
        // getData from BankAccountData
        // addRow
    }
}

abstract public class UserInterfaceAbstraction
{
protected UserInterfaceAbstraction (Function function) {
    this.function = function;
}

public void setupReportOptions (String startDate, int accountOptionsFlag,
    boolean reconCBSelected) {
    recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
    // initialize the report data
    // initialize the report user interface
}

public void refreshData () {
    // iterate over accounts and refresh data
    // iterate over accounts and refresh user interface
}

public void display () {
    container.display ();
}

public void recordReportOptions (String startDate, int accountOptionsFlag,
    boolean reconCBSelected) {
    function.recordReportOptions (startDate, accountOptionsFlag, reconCBSelected);
}

public String [][] getData (int accountType) {
    return function.getData (accountType);
}

public void printReport () {
    function.printReport ();
}

public void addRow (int accountType, String newDate, String desc,
    String newAmountStr, boolean recon) {
    function.addRow (accountType, newDate, desc, newAmountStr, recon);
    refreshData ();
}

public void modifyRow (int accountType, int selectedRow, String newDate,
    String desc, String newAmountStr, boolean recon) {
    function.modifyRow (accountType, selectedRow, newDate, desc, newAmountStr,
        recon);
    refreshData ();
}

public void saveData (String contentName, int accountType, String saveData[][][]) {
    function.saveData (contentName, accountType, saveData);
}
public class UserInterfaceRefinedAbstraction extends UserInterfaceAbstraction {
    private final String copyright = "Copyright 2012, Andrew Haigh";

    UserInterfaceRefinedAbstraction (Function function, int accountCount,
    int accountPanelType)
    {
        super (function);
        ReportLogic rLogic;

        container = new ContainerJavaImplementor (accountPanelType);

        if (accountPanelType == Function.complexAccountPanel)
        {
            setupComplexBankAccountPanel ("checking", Function.checkingAccount);
            setupComplexBankAccountPanel ("saving", Function.savingAccount);
        }
        else
        {
            setupSimpleBankAccountPanel ("checking", Function.checkingAccount);
            setupSimpleBankAccountPanel ("saving", Function.savingAccount);
        }

        new ReportData ();
        new ReportLogic (this, reportData);
        new ReportJavaImplementor (reportData, rLogic);

        container.display ();
    }

    private void setupComplexBankAccountPanel (String contentName, int accountType)
    {
        new BankAccountData (this, contentName, accountType);
        new BankAccountLogic (this, bankAccountData);
        new ComplexBankAccountJavaImplementor (bankAccountData, bankAccountLogic);
    }

    private void setupSimpleBankAccountPanel (String contentName, int accountType)
    {
        new BankAccountData (this, contentName, accountType);
        new SimpleBankAccountJavaImplementor (bankAccountData);
    }
}