Virtual Frameworks for Source Migration

by

Jack S. Chi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada

Abstract

Virtual Frameworks for source migration is a methodology to extract classes and interfaces from one or more frameworks used by an application. After migration, a new set of frameworks called virtual frameworks can replace the original frameworks used.

The classes and interfaces extracted are used to create a proxy layer for these new frameworks. The application then depends on this proxy layer, and through it the new frameworks, rather than on the original frameworks. A combination of three patterns: Bridge, Adapter, and Proxy are used in these new frameworks. By doing so the changes made to the application source code are minimized during migration.

Acknowledgements

Thanks to all members of the SWAG group at the University of Waterloo for their thoughts, encouragement and good times while this thesis was in preparation.

Contents

List of Figures

Chapter 1

Introduction

Software requirements can change quickly over time. Changes in the deployed environment can force frequent changes to the software running in it. Such changes often require source level modifications.

We have focused our research on one area of source migration which is particularly interesting and troublesome. In this chapter, we introduce our definition of source migration and describe this commonly encountered problem. We will propose an approach for solving this problem, and discuss it in greater detail in Chapter 3.

1.1 Source Migration

In *The Migration Barbell*[23], Malton defines the process of source migration as: "the reengineering task of deploying existing software in a new environment, by modification of the source code."

We expand on this by considering source migration to be the act of producing a minimally modified version of a software artifact, such that functionality is preserved in a new environment. This task can be set in the larger context of software maintenance. Software maintenance activities are classified as corrective, preventative, perfective or adaptive[22]. According to this classification scheme, source migration is considered an adaptive task which incorporates software into a previously unsupported execution environment.

The common goals of source migration, as outlined in Malton's paper, are:

- Make efficient use of the target environment
- Preserve functionality
- Make keeping source code maintainable a priority

The first two of these goals are the most relevant and *must* be achieved. If one of these can not be accomplished, the migration process will result in a product that does not run properly on the target platform. However, the goal of maintainable source may or may not be achieved through the efforts involved in learning about the target platforms and performing the migration. Achieving this third goal is typically difficult and time-consuming.

We will address this last goal using another approach to source migration, which we will describe in Chapter 3.

Before introducing our new method and discussing how it accomplishes these goals, let us review several types of source migrations commonly encountered.

1.2 Types of Source Migration

Malton categorizes source migration into three distinct forms[23], which are listed in order of increasing risk and difficulty:

- Dialect conversion
- API migration
- Language migration

Dialect conversion usually happen when the supporting compiler technology or the programming language specifications change or evolve. Examples of this type include migrating from Borland C to GCC or C++, or from Java to Java 2.

API migration occurs when an API call needs to be replaced by a new ones. This can occur either due to the new API offering improved quality or performance or even when the current API is deprecated.

Language migration is considered the most difficult type of source migration. When the decision has been made to move to a new language, every line of the original source must be rewritten in the new language. The difficulties of doing so are summarized below:

- Rewrite every instruction from the original source in the new language
- Differences in constructs between the two languages demands higher-level changes
- Software architecture and functionality needs to be preserved.

One example of this kind of migration is a C program being converted into a Java program. The study and experiences of one such a conversion have been provided in an experience report: *C to Java Migration Experience*[24].

The problem addressed in this work is that of an API migration. For such a task, we need to remove the dependency on one API by providing another API. We explain the difficulties of such a replacement in the following section.

1.3 Framework Migration

Many programs are constructed in a way such that they are dependent on the underlying operating system, specifically the call level libraries. Software construction extends this concept and introduces intermediate layers of libraries that can be used as building blocks. Modern applications are commonly built many of these building blocks.

These building blocks are called "Application Frameworks" or simply "Frameworks". A framework[18] is defined as a software foundation around which other modules and functions can be built. This definition and framework properties will be discussed in further detail in Chapter 2.

Many software vendors have adopted an object-oriented framework model to help programmers build software. They provide reusable components, package them as frameworks and perform a thorough testing o before these are shipped to customers. This means programmers do not have to start coding from scratch. They only need to integrate off-the-shelf components from frameworks into the application being developed. A significant amount of time is thereby saved and the software being built is more robust because the off-the-shelf components have been rigourously tested. Software vendors partition software functionality and factor these functions into different frameworks. Some of these frameworks are even built on top of other frameworks. A tree-of-frameworks structure is thus formed, which can be called the framework hierarchy. NetBean IDE's windowing framework is a good example of such a structure, which is based on the Java Swing framework.

Object-oriented frameworks provide features as outlined above, and can be used in many of the areas where software is created. We feel that areas where it might not be appropriate include situations of extremely limited memory and computational power, such as embedded devices, or real-time systems where performance issues are vitally important.

One of the benefits of using a framework-based development approach is that the amount of effort spent is greatly reduced by *software reuse*[16]. In fact, the claim could be made that software reuse is the main advantage gained by using these object-oriented frameworks. These frameworks also provide general solutions to common problems encountered in a specific application domain. Thus, the complexity and amount of work for a newly developed component is greatly reduced by using frameworks for development.

As software evolves, a framework can be inserted into, or removed from, the framework hierarchy. As mentioned above, a framework hierarchy is built by having one framework depend on another. A graphical editor framework is a good example of a framework hierarchy, as the editor framework will usually extend an existing GUI framework. The dependency relations between frameworks is an important consideration when parts of the software are reengineered into newly introduced frameworks.

Migrating software at the source level involves tasks such as the modification of statements, removing dependencies on the source frameworks, and adding statements which call the target frameworks. As a whole, these tasks can be categorized as API migration[23]. We refer to this type of code change as framework migration because a framework is more than just a simple API.

However, there are potential problems when the newly introduced frameworks are not compatible with existing frameworks. A landscape editor is a good example to illustrate this problem. *lsedit* is one such a tool[1]. This landscape editor is a graphical editor used for visualizing software architecture components. It is a Java application with a GUI front end implemented using the Java Swing framework. As the system evolved, it became necessary to reengineer *lsedit* it for various environments and to re-deploy it using a different GUI frameworks. The need for such a reengineering became evident when it was desired to integrate *lsedit* with Eclipse, but such an integration was hinder because Swing is incompatible with Eclipse. We found that *lsedit* was heavily dependent on Java's Swing framework and this dependency needed to be carefully dealt with during migration.

We propose a solution to integrate and encapsulate conflicting frameworks by the introduction of a new, higher level framework on top of the incompatible frameworks. We will examine the details of this approach in Chapter 3.

We develop a set of classes that match the interfaces of classes extracted from the original framework. By mimicking these interfaces, newly created classes can be used as proxies. These proxy classes will in turn control instantiation of underlying framework classes.

Such an approach requires additional examination of the issues relating to the protocols that the proxy will use to communicate with the framework. The framework in a particular

[1]We will use the terms landscape editor and *lsedit* interchangeably to refer to this tool in the rest of this work

environment will need to cooperate with other frameworks and the proxy classes help make this cooperation work.

We have introduced the idea of maintainable source at the center of our research in order to reduce the amount of maintenance required in the future. It is hoped that this will reduce the maintenance costs associated with framework migration when an execution environment requirement changes.

Chapter 2

Related Work

Our work proposes a low-impact technique for migrating framework-based code to a new environment. To evaluate and use our technique requires knowledge of object-oriented design concepts, specifically frameworks and their application in modern Java development environments. In this chapter we give an overview of these topics and present some examples of object-oriented frameworks commonly used in various Java development environments.

2.1 Toolkit and framework

In this section, we give definitions for both toolkits and frameworks. It is our belief that toolkits are often integrated into frameworks and thus are relevant to this work.

Toolkit

A toolkit is a set of related and reusable classes, implemented to provide a general purpose functionality that can be incorporated into many applications[18]. Toolkits do not impose design decisions; they instead provide functionality that can be used by a variety of applications.

Toolkits are designed to be useful in different application domains. Hence, assumptions and dependencies on a particular environment need to be avoided. While it would be useful to have a general purpose toolkit, appropriate for all environments, it is difficult to design such a toolkit since it is impossible to know the scenarios in which it will be used.

What typically makes toolkits valuable is that they enable code reuse. Object-oriented toolkits include classes that can either be used directly or extended. Toolkits can be considered the object-oriented equivalent of procedural language libraries.

There are many examples of toolkits. One examples is Motif[19], a graphical user interface toolkit that is used by C and C++ programmers to develop applications targeting heterogeneous, networked computing environments. The Motif toolkit is available on many UNIX platforms, including Linux. By using Motif, programmers can write portable code for the GUI portion of their software.

Framework

A framework[17] is a set of reusable classes that express a design for a particular application domain. Developers can create an application by subclassing the abstract classes from the framework or by using the concrete classes provided by the framework directly.

A framework predefines some of the architectural components for an application. This allows programmers to concentrate more on the specifics of the application that is being built, rather than high level structure.

Reuse at this level leads to an inversion of control between the application and the frameworks on which it is built[18]. When a programmer writes an application using a toolkit, he writes the program's main body and calls the procedures that he wants to use. Conversely, when a programmer uses a framework, he takes the skeleton provided and writes the code that the framework body will use.

"An object-oriented abstract design, also called a *framework*, consists of an abstract class for each major component"[18]. The programmer can either use these components directly or extend their abstract classes. This allows faster development of applications and ensures applications developed with a common framework have a similar structure. Such applications are easier to maintain, because the framework designers have made design decisions for the programmers. Instead of having to make possibly poor design decision in a well understood domain, programmers can instead follow design suggestions expressed as frameworks. A further examination of the inversion of control, with respect to frameworks, will be presented in Section 3.2. Examples of object-oriented frameworks will be given in Section 2.4.

White-box vs. black-box frameworks

One important characteristic of a framework is that application methods are often called from within it. The framework is an active entity which controls the execution of the application.

A programmer uses some frameworks by extending them, that is by defining subclasses of the framework's base classes. The semantics of these framework classes are required knowledge, and therefore developers need to understand the internals of the framework before it can be used to guide application behavior. These frameworks are called white-box frameworks [18] because the framework details are known to the programmers.

There are two problems with using white-box frameworks[18]. The first is the number of newly extended subclasses required to run the application. The number of classes makes the applications hard to change. The second problem is that since frameworks are usually huge, a white-box framework will be time consuming to learn. These two problems need to be considered before a white-box framework is integrated into a project.

Another method of customizing a framework is to provide a set of components. These components supply the application functionality. A protocol is defined such that components can coordinate, and all components implement their interfaces in terms of the given protocol. Components respond to messages sent from other components. Such a topology keeps the internal details of components isolated from other components. Encapsulation is preserved in this type of framework.

To build applications using this kind of framework, programmers only need to understand the protocols of the framework components. A detailed understanding of the internals of the frameworks is not necessary to use them. These kinds of frameworks are called black-box frameworks.

In general, a black-box framework is more reusable than a white-box one [18]. White-box frameworks use inheritance relations to customize the applications while black-box frameworks instead have a plug-in architecture.

In practice, we find that many frameworks are hybrids of white-box and black-box frameworks. Thus, a partial understanding of the framework being used is still essential, but such an understanding is more easily attained. The hybrid approach is often popular as it is the most flexible. It offers the opportunity for programmers to explore the framework in as much detail as they desire, but doesn't demand complete understanding.

The approach we propose for performing framework migration is to address the two concerns of white-box frameworks. This approach will be detailed in Chapter 3 after an example of framework migration is provided.

2.2 Objects Collaborations and Software Reuse

Before providing examples of frameworks that are currently deployed, we feel it would be worthwhile to explore the important properties of frameworks and the advantages they provide.

2.2.1 Object Collaborations

If software use classes from different frameworks, these frameworks must collaborate to support the software's purpose. Class instances from different frameworks need to interact and communicate with one another. Unfortunately, such a harmony typically does not exist. We must therefore consider the difficulties of object collaborations before attempting a framework migration.

In this section, we use UML notation to illustrate collaboration diagrams of important interactions between objects created in an application. These objects work together to provide the functionality of an application.

Interaction diagrams[21] are a generalized form of two more specialized UML diagrams.

Both diagrams are used to illustrate message interactions. These two diagrams are:

- collaboration diagrams
- sequence diagrams

For the purpose of illustrating such inter-framework cooperation, we use collaboration diagrams. These are diagrams that express interactions between objects. Such diagrams are a simplified version of sequence diagrams, which shows the relationship between objects. An example collaboration diagram is shown in Figure 2.1.

In this diagram, ClassA, ClassB and ClassC are instantiated by an application. ClassA is a user-defined class while ClassB and ClassC are associated with framework F1. These three instances will collaborate by sending messages to each other and by reacting to messages received. Arrows indicate the direction of messages. The numbers preceding the message names indicate the sequence of the messages. A message with two numbers separated by a period, indicates another message was sent in response to the initial message. In the scenario illustrate, the instance of ClassA will receive a message, msg1(), from an entity not specified in this diagram. In turn, ClassA will send messages msg2(), msg3() and msg4() sequentially to the instance of ClassB. Message msg4() will trigger a nested message msg5() which will be sent from the instance of ClassB to the instance of ClassC.

Figure 2.1 gives an example of objects interacting by sending and receiving messages which are understood by all object involved. Messages can not be sent from one framework

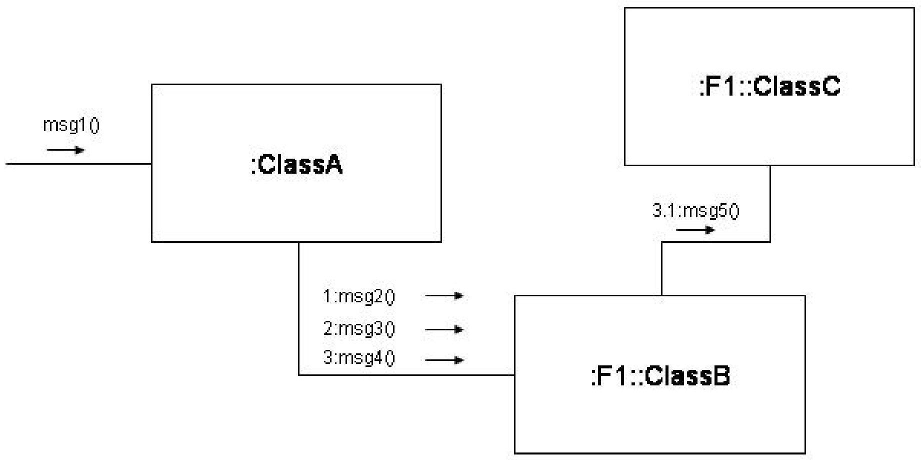


Figure 2.1: UML collaboration diagram using Framework F1

to objects in another framework as they do not use the same message format. Figure 2.2 is another collaboration diagram. In this diagram, ClassB and ClassC are imported from another framework, F2. ClassB and ClassC in these two distinct frameworks are similar in their functionality. However, their message specification could be completely different. Therefore, the messages sent by ClassA must be modified so that ClassB and ClassC will receive an understandable message and react accordingly. These collaborations must be altered in order to allow ClassA to send messages in a format that is appropriate for Framework2.

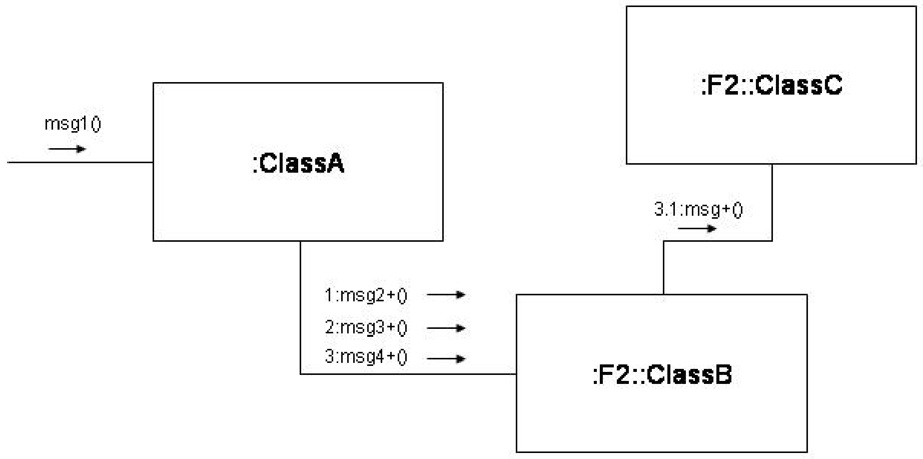


Figure 2.2: UML collaboration diagram using Framework F2

As mentioned previously, a framework is the object-oriented approach of managing collections of related libraries. Objects, therefore, are at the center of the frameworks universe. They play an essential role in this setting. As a group of objects interact and collaborate by sending messages to each other, the collaborative behaviors of the framework objects must be predefined. UML is useful for understanding this collaborative behavior.

Within each framework, class definitions provide the message specifications that can be sent to an instances of a class. This is referred to as the application programming interface (API). Some industrial practitioners alternatively refer to API as a protocol. This interface consists of a message name and a message parameter list, which is composed of various numbers of message parameters of any data type. The data type can be simple or complex.

Collaboration is more than just the interface however. Although the interface defines what messages can be sent to the objects, some messages are required to be received before others. Order is important, making sequence constraints a vital collaboration consideration.

2.2.2 Software Reuse

Another important factor in the success of framework migration is software reuse. If a programmer switches from one framework to another, reuse relationships must be removed from the source framework and recreated for the target framework.

One of the reasons that object-oriented programming has become so popular is that software reuse is more important than before. As software systems become more complicated and expensive to maintain, the importance of software reuse becomes increasingly apparent. One study[27] suggests that 60 to 85 percents of the cost of software is due to maintenance. If programmers use software components in their systems, then total maintenance cost can be greatly reduced.

Object-oriented programming has encouraged software reuse in a number of forms. Abstract and concrete class definitions provide a mechanism to encourage source code reuse. Polymorphism helps reduce the number of methods needed and thus also reduces the amount of code.

Software reuse can be a confusing term, as it is often misused to refer to the use of an API by the source code of a project. In general, *reuse* is a high level term which refers to a conceptual design idea that is used repeatedly in many software systems.

2.3 Common Structural Patterns used in Frameworks

In each framework, structural design patterns are frequently incorporated into the design of the framework classes and interfaces. These structural design patterns are explained in detail in "Design Patterns"[17].

We devote a large portion of this chapter to outlining several structural patterns commonly used in frameworks. We will discuss how these design can be used, in general, to help solve the problem of framework migration. We will revisit some of these patterns in a later chapter and explain how they were specifically used in our approach to framework migration.

2.3.1 Façade pattern

Façade is the first design pattern that came to our attention, since we would like to control access to the underlying framework's concrete classes from the application. A façade interface can provide us with a small set of entry points to entire subsystems.

The primary advantage of façade is that it can help identify possible flaws in the system design, since there are a small number of entry points. These entry points can be used as break points when debugging the code, allowing stack traces to be produced. This helps to isolate errors in modules.

One example of this kind of error occurs when a program crashes or generates run-time exceptions, usually causing the program to halt. In the case of the Java runtime system, an exception is thrown to indicate the location of the statement which caused the halt. Possible reasons for this halt include a variable not being initialized properly or side effects caused by other objects in that program. However, finding which object caused the side effects can be simplified if the façade pattern is used in this application, as it will isolate sections of code where the error might occur. Errors are therefore easier to find among groups of back-end façade objects.

Another benefit of this pattern is the elimination of import statement processing. When the execution environments changes, a single set of façade packages will be provided to access other concrete frameworks' packages.

Figure 2.3 illustrates this pattern.

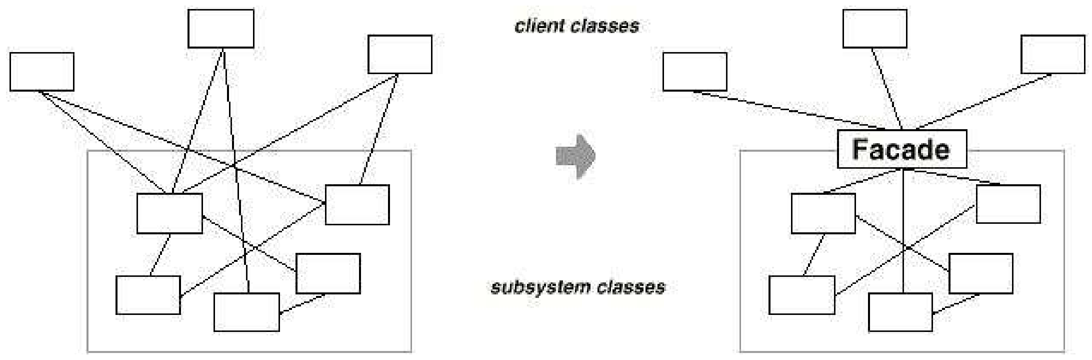


Figure 2.3: Façade Interfaces[17, p.185]

2.3.2 Bridge pattern

The Bridge pattern is used to decouple an abstraction from its implementation so that they can vary independently. When an abstraction can have several possible implementations, inheritance is the typical way to represent it. Some implementations may bind to the abstraction permanently, although this may cause problems.

A classic example is the windows hierarchy as shown in Figure 2.4. In this example, concrete implementations are provided for two window system, XWindow and PMWindow. If we refine the abstraction for the window system as IconWindow, two concrete implementations of the IconWindow are needed for each concrete window systems.

In this scenario, the application might permanently bind to one of these concrete implementations. It is not a desirable practice.

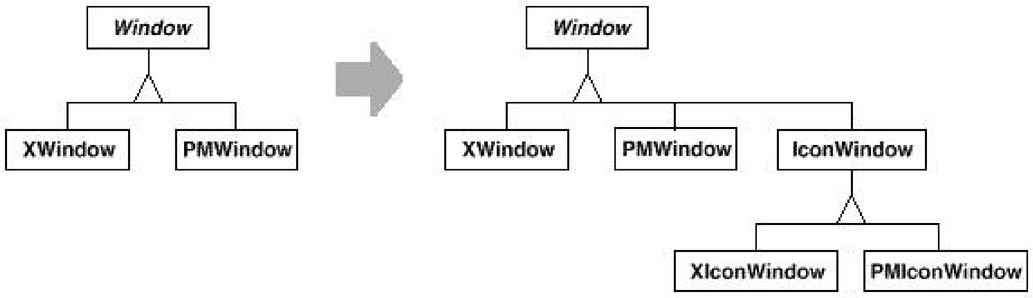


Figure 2.4: Bridge pattern[17, p.151]

The bridge pattern solves this problem by creating two separate class hierarchies. One for the abstract window interfaces and another for the platform specific window implementations. The design for the virtualized framework, our work, also follows this principle which provides flexibility. We will examine such a situation, namely integrating Swing[7] and SWT[2] in a new GUI toolkit, in Chapter 4.

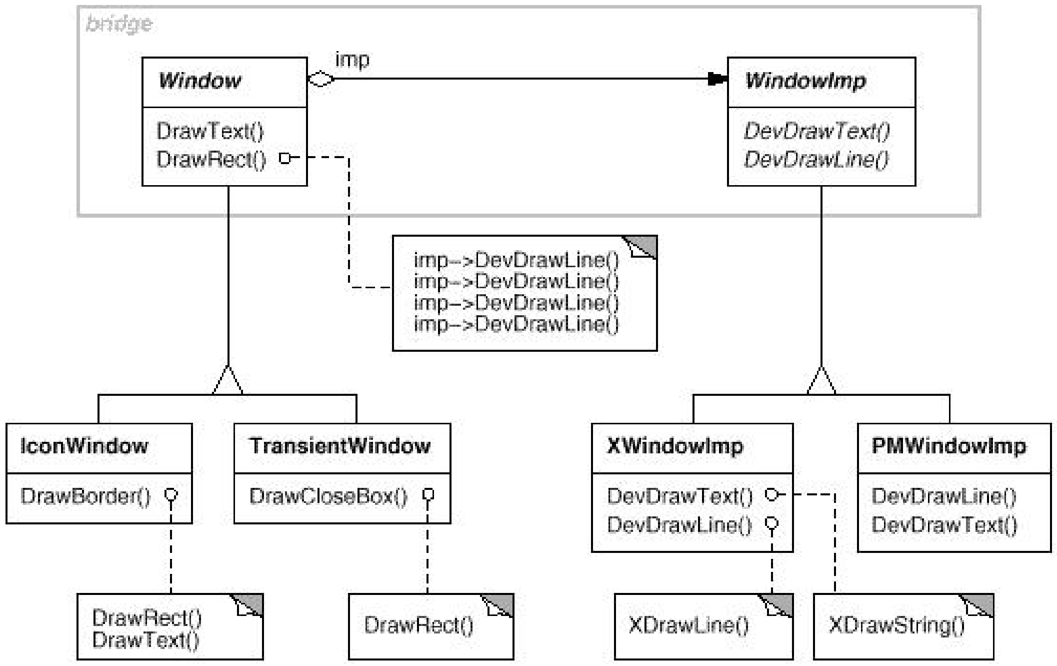


Figure 2.5: An example of the class hierarchies linked by a bridge[17, p.152]

However, although this pattern initially appeared useful, it turned out to be inappropriate for our work. Instead we used a Proxy-Adapter, which combined these two patterns in order to separate the abstraction and implementation in the virtualized version of the framework. The Bridge pattern works best when two objects share an interface. This was not the case for the frameworks we were interested in. The Proxy-Adapter approach therefore seemed to be a better solution for the source migration from one framework to multiple target frameworks. More details about this combined pattern, used in a virtualized framework, will be discussed in Section 3.3.

2.3.3 Proxy pattern

The purpose of the proxy design pattern is to provide a surrogate for another object which is then used to control access to the original object.

In Figure 2.6 we can see an example illustrating the relationship between a proxy and the object it contains. In this example, a proxy is created to represent the real image. However, the real image object will not be instantiated until the proxy receives requests from the application. This can be useful in many problem domains, and the proxy pattern is commonly found in on-demand applications[17].

In our proposed GUI toolkit, which will be introduced in our case study later in Chapter 4, we provide a proxy class for each UI component class. The actual object creation is always deferred until the contained UI proxy object is passed as an argument to the add method of the containing proxy object. The reason for this is not resource management, but due to difference in protocols between the proxy objects. We will explain the reasoning behind this in Chapter 4 where we will further examine the problem of protocol differences between proxy objects.

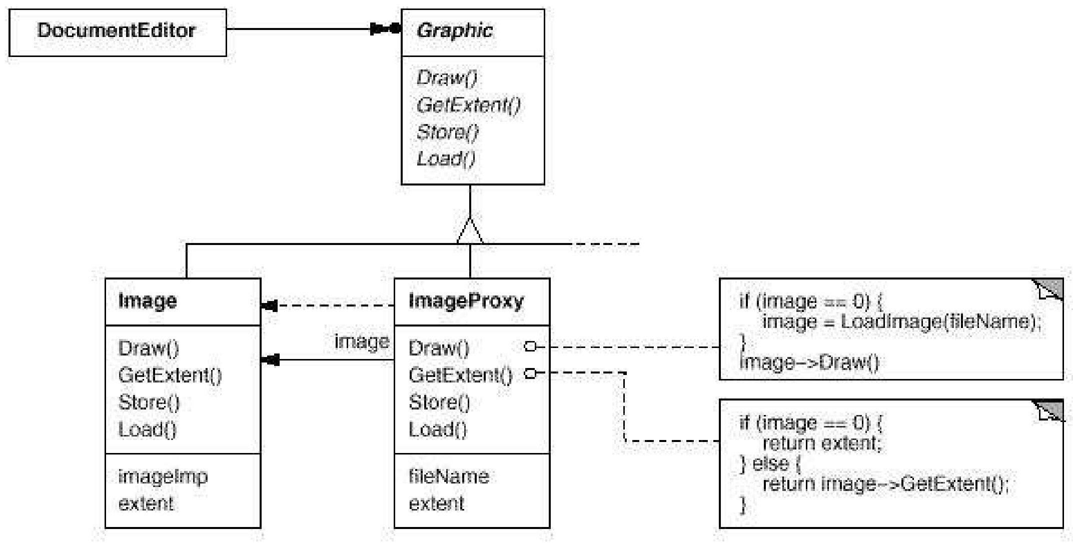


Figure 2.6: Proxy pattern[17, p.208]

Another reason for using this pattern in the proposed virtual framework is to load different types of adapter objects. The proxy class must communicate with a variety of adapter classes. It is similar to the plug-in mechanism found in many modern software designs. Thus, the application will behave differently according to which adapters are loaded, as chosen by the proxy at run-time.

2.3.4 Adapter pattern

The Adapter pattern converts an interface of a class into another interface.

There are two methods for implementing an adapter. The first is by using multiple inheritance. The second is through the use of a reference to connect the interfaces. These two methods are illustrated in Figures 2.7 and 2.8. Since multiple inheritance is not supported in Java, we will use the reference approach in our case study.

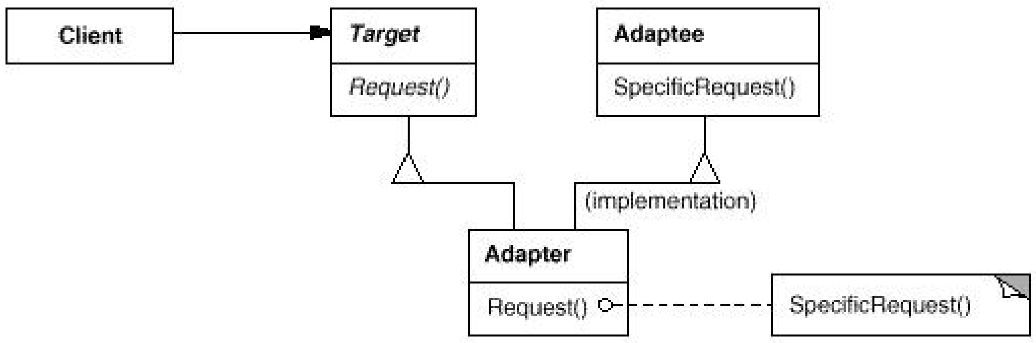


Figure 2.7: Adapter support multiple-inheritance[17, p.141]

In our proposed virtual UI framework, we need to provide an adapter for each proxy class in each implementation. This will allow proxies to load the appropriate adapter object. Each adapter implements a single interface design, predefined by its respective proxy class.

Once we have adapters for each implementation, the proxy objects can communicate

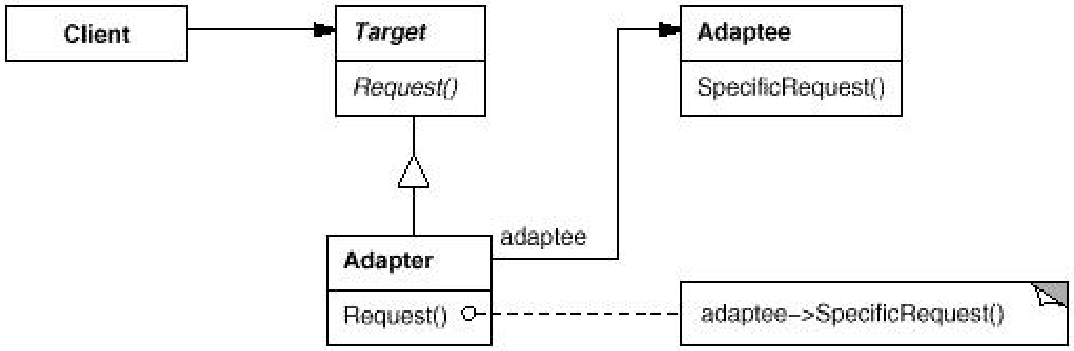


Figure 2.8: Adapter with single-inheritance[17, p.141]

with each other, even proxies representing parts of different frameworks. The application will not be aware of any differences between frameworks chosen at run time.

2.3.5 Decorator pattern

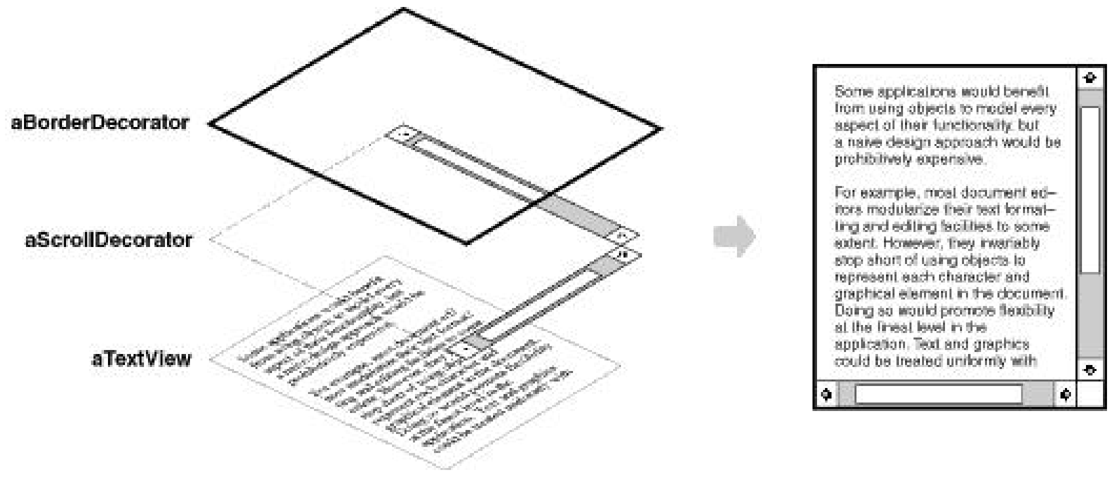
The decorator pattern is also known as *wrapper*, since it describes a pattern in which an object can be enclosed by another object, with the enclosing object controlling input and output. By using this pattern, programmers can dynamically attach functionalities to an object without modifying its internals.

Sometimes we want to give additional responsibilities to an individual object, but not to other objects of the same class. For example, a graphical user interface toolkit should let programmers set the properties and behavior of any UI component.

Using inheritance is one of these choices. It is inflexible, however, because all the properties and behaviors will be inherited statically. A more flexible approach is to enclose the component in another object. This enclosing object will be responsible for adding any desired functionalities to the enclosed object without requiring inheritance.

The enclosing object is called a decorator. The decorator conforms to the interface of the enclosed object so that the presence of the decorator is transparent to the enclosed object and any of its clients. The decorator forwards requests to the enclosed object and may perform additional tasks while routing requests. This transparency lets us nest decorators.

An example found in "Design Patterns"[17] is presented in Figure 2.9. It is a UI component which renders text on a display. Two nested decorators enclose this UI component to enable scrolling of the text and to draw a border for the UI component.



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Figure 2.9: Decorator pattern[17, p.175]

In our proposed virtual framework, we use this pattern everywhere. It is not only used in the design of proxy classes but also in concrete framework adapters.

2.3.6 Composite pattern

The Composite pattern is found in almost every graphical user interface toolkit. Java's Swing[7] and SWT[2] are no exceptions. This pattern allows users to construct a complex user interface from a set of simple components.

In this pattern, two types of entities need to be identified. The first one is a primitive. A primitive in UI terminology is a component which can be used to compose the user interface design and can be extended or used directly by an enclosing UI component. A container is also a user interface component. However, this component can enclose primitives or other containers.

Our research work places a focus on the use of this pattern and has lifted this pattern to a higher level of abstraction so that programmers are able to compose user interfaces abstractly at the proxy class level. Furthermore, the contained proxy objects defer binding to underlying implementations until they are added to the containing proxy objects.

Hence, in our proposed UI toolkit, programmers have the ability to construct a user interface in a manner similar to how they have used other graphical user interface toolkits, but working at a higher level.

In Figure 2.10 this pattern is illustrated as a complex graphic object composed from primitive objects and other graphic objects. The Picture class is also a Graphic class since it extends Graphic.

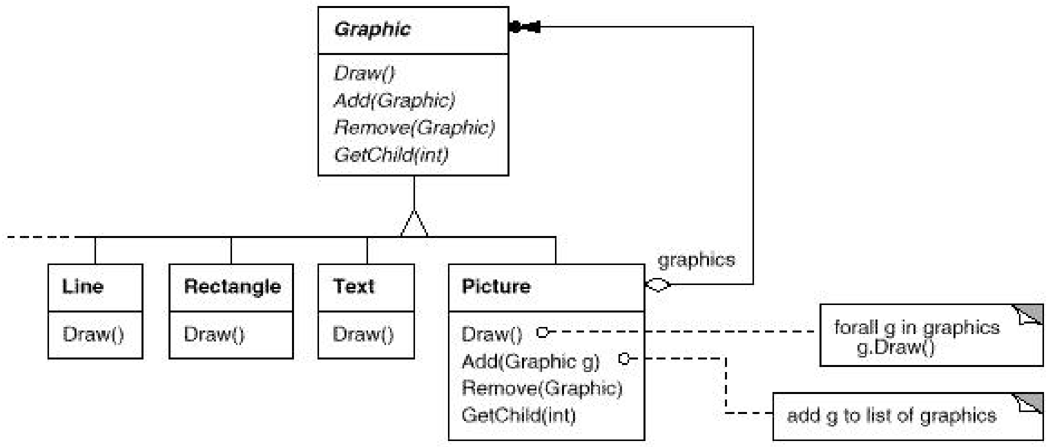


Figure 2.10: Composite pattern[17, p.163]

2.3.7 Flyweight pattern

A flyweight object is an object that can be used in different contexts. Such an object can not assume any specific context and must be indistinguishable from non-flyweight objects.

An important concept about flyweight objects is the distinction between intrinsic and extrinsic states. Intrinsic state is the information that is stored in the flyweight object, the meaning of which is independent of the context where the application is running. An extrinsic state is dependent on a flyweight object's context and is not stored by the object. The client code needs to pass an extrinsic state to a flyweight object to deal with dependency in the execution context.

Figure 2.11 is a Glyph object with some of its derived classes. A draw method is defined and a drawing Context will be passed as an argument to this method. This method is responsible for drawing entities in the correct manner in this context.

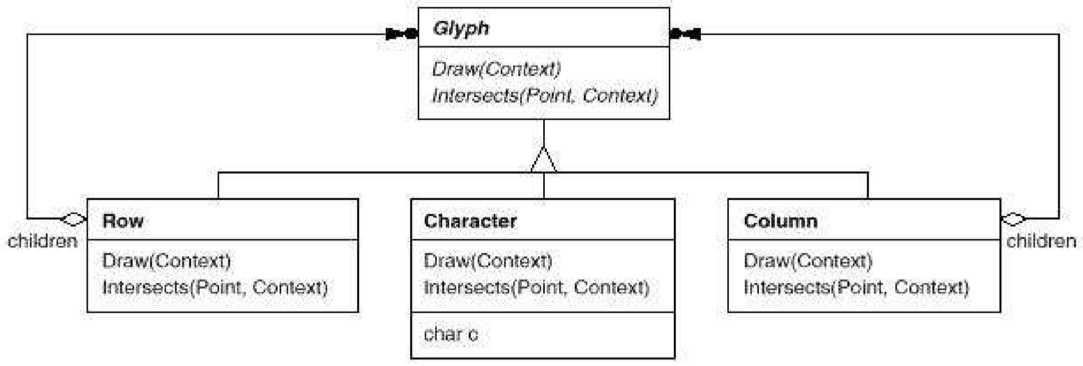

Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
children
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children

Figure 2.11: Flyweight pattern[17, p.197]

Our proposed UI toolkit also incorporates flyweight patterns. The GUI proxies used in the new version of the landscape editor are non-flyweight objects and their extrinsic states are stored within their classes. This is done as these proxy class objects will load other platform dependent adapter class objects. Non-GUI classes in the landscape editor are flyweight objects.

One of our goals is for applications using this toolkit to be able to run on different platforms without platform specific code. This makes applications easier to maintain, as there is a single version of the source. This also helps achieve the goal of maintainable source.

2.4 Examples of General Application frameworks and Toolkits

Many frameworks exist that help programmers build applications. Some of these are general purpose, while others are designed specifically for a particular application domain. We will give several short examples of such frameworks.

This first framework we will present is the Java Collection Framework, a general purpose framework. It is an example of a framework which helps reduce programmers' workload by offering classes and interfaces rather than forcing programmers to implement functionality themselves.

In our work, we focused on the problem of integrating Java applications with different GUI frameworks. We will therefore introduce some popular Java GUI frameworks and give a brief description of their internals.

One such framework is the Swing framework [7]. This is Sun's answer to the typical problems of GUI applications. We will then examine the Abstract Window Toolkit, the base of the Swing framework. SWT and JFace will also be introduced.

After providing details about Swing we will discuss an open source IDE project, NetBeans [9], which uses and extends Swing for its UI components.

Finally, the Eclipse project[4] and its framework will be examined, which has interesting features and problems that need to be overcome in order to integrate Swing GUI applications with this technology.

2.4.1 Java Collection Framework

A collection is an object that represents a group of objects. The Java Collection Framework[6] is an architecture that allows manipulations of objects independent of their actual representations. This framework provides solutions to several general problems in computing. These problems are: Set, List, and Map.

A *set* is an object that can store other objects, of arbitrary type, with no duplicates. The order of elements in a set is not guaranteed.

A *list* is different from a set in that the elements of a list are ordered and can be accessed by specifying an index from the starting position. Duplicate items are permitted.

A *map* is another type of object that can be used to store paired keys and values. The function of a Map is similar to a database or a dictionary. We can use the key to quickly locate the value associated with it.

Programmers can apply the same algorithms in the same way to each of these data structures. Consider the example of bubble sort. This sort algorithm examines the data elements of a collection of objects by comparing the values of adjacent elements and swapping their positions in the collection if needed. Such a an operation can be applied to any of the three data structures presented, in much the same way.

The Java Collection Framework provide solutions to commonly encountered problems and the framework has the same purpose as the C++ standard template library. These set, list and map objects are commonly used by programmers to store values and manipulate them in memory to achieve some runtime behaviors. However, some design decisions have been made by the framework designers. Hence, the programmers need only focus on implementation and using the framework's functionalities and interfaces.

2.4.2 The Swing framework and Abstract Window Toolkit

There are many examples of GUI frameworks, another type of framework. We will examine the most important examples for our work, starting with Swing.

The Swing framework[7] is commonly used by Java applications with a GUI to provide user interface components. Programmers can either directly use these or customize the Swing components.

It is a fully-featured UI library implemented entirely in the Java language. The Swing framework uses the windowing functionality of the Abstract Window Toolkit and the rendering capability of Java2D to implement its own extendable UI components.

In general, Swing's framework components can be categorized as top-level components. Such components include: layout-containers, menu components, complex data components, text components, and atomic controls.

Layout-containers can contain other types of components. Top-level components are a special type of layout containers. They can not contain other top-level component or be contained by other components.

In Figure 2.12[5], we show a variety of Java components provided by the JRE and JDK distributions. It can be clearly seen what roles these components play in these two distributions.

In Figure 2.13, it can can see that the Swing and AWT frameworks are closely related and that Swing extends AWT's base classes. A brief introduction to AWT is an important foundation in understanding the core of the Swing framework. Such an overview to AWT is provided next.

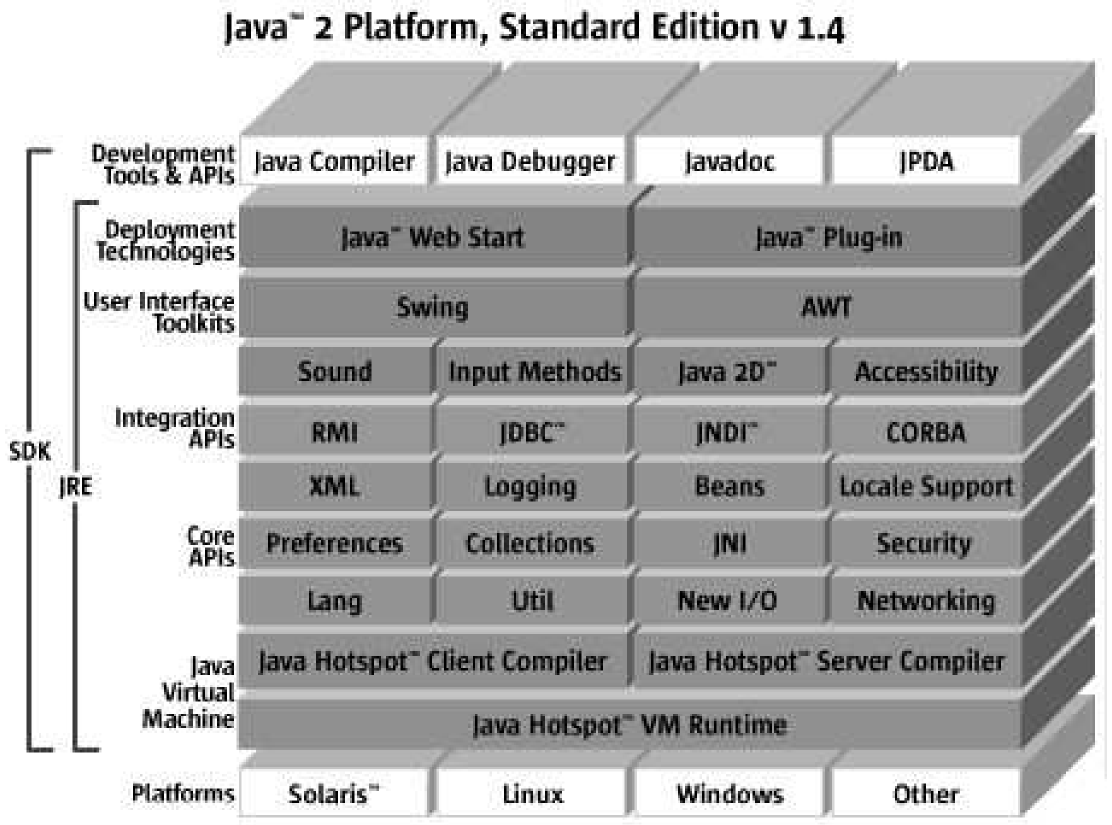


Figure 2.12: J2SE platform components[5]

The AWT classes are designed to interact with the underlying native windowing systems across many platforms. They provide a set of simple UI components that map directly onto their native counterparts.

These components therefore depend on the native widget code of each platform. They are implemented by way of native widget calls, resulting in inconsistent component behavior on different platforms. Native code blocks in AWT classes are referred to as *peer code* as they function as a gateway to the native platform libraries.

Many Swing classes extend AWT classes. Since the containment hierarchy is managed by their base classes, programmers can use AWT and Swing classes interchangeably in their applications. Figure 2.13 shows the extensible relationships between Swing components and their AWT based parents[7].

Keeping in mind Swing's foundation in Java's GUI approach, let us now examine other GUI frameworks.

2.4.3 Standard Widget Toolkit and JFace

SWT (Standard Widget Toolkit[2]), developed by IBM, is one such GUI toolkit. SWT classes do not extend Java's AWT or Swing classes. Instead, it has its own hierarchy of classes.

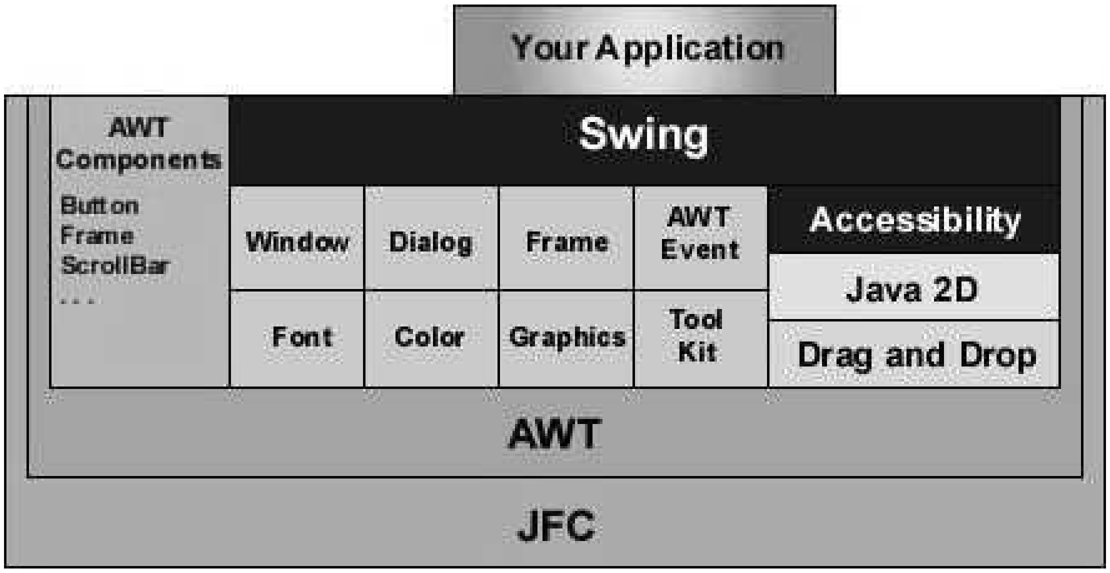


Figure 2.13: Swing and AWT class dependency relationships[7]

This makes SWT an alternative method of developing UI applications in Java. It provides a user interface which is OS independent. This provides a tighter integration with the native window widget systems as it simply uses the platform's functionality. This also provides the native window system's look and feel. For each supported platform, SWT uses native widgets wherever possible. If a widgets is not available on the platforms, SWT emulates it.

Figure 2.14 is a simplified version of the SWT hierarchy. All widgets classes in SWT extend the `Widget` class.

JFace was also developed by IBM and is another toolkit used to handle UI tasks. It is system independent, both in its API and its implementation. JFace is also designed to

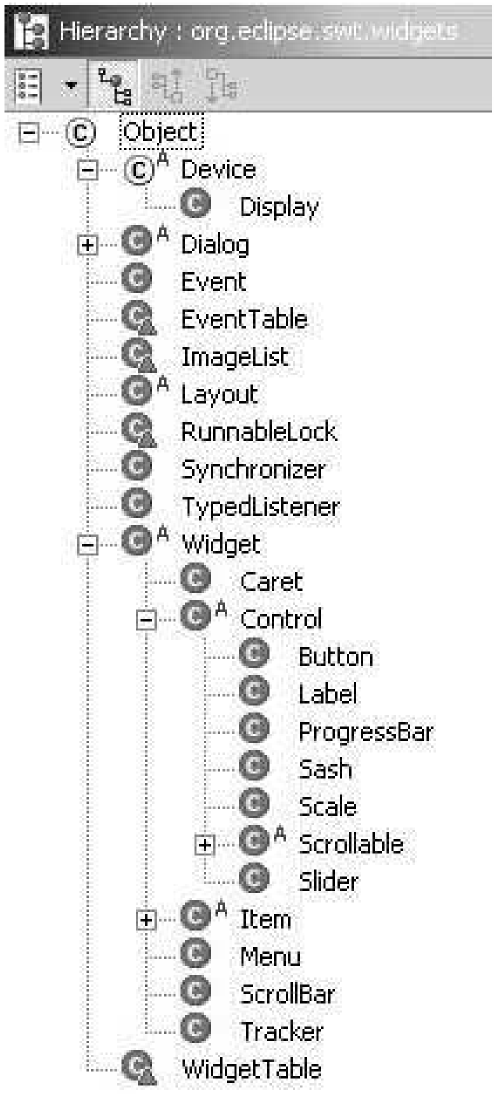


Figure 2.14: SWT class hierarchy[31]

work with SWT without hiding it. It is coded completely using Java, is compatible with SWT and is considered an extension of it.

In the following sections, we will focus on two popular IDE frameworks, highlighting parts that are related to the Swing framework and SWT.

2.4.4 Netbeans framework

NetBeans[9] is an open source Java project that provides a Delphi-like IDE. The project was first started in 1996 as a student project in the Czech Republic. A company was formed around the project, called NetBeans, and made two commercial releases.

In 1999, NetBeans was acquired by Sun Microsystems. After this acquisition, Sun released the "Forte for Java, Community Edition" IDE, the same IDE that had been in beta as the last official release of NetBeans.

NetBeans' IDE is built on top of a set of runtime libraries, called the NetBeans framework. A major design principle for this project is modularity. Any applications built using this framework consist of the framework itself plus some custom made modules. Modules can be plugged into the framework at runtime.

The framework also provides a set of well-documented interfaces called OPEN APIs.

Programmers use these APIs to build modules for their applications. Figure 2.15 shows the relationship between user created modules and the NetBeans' OPEN APIs.

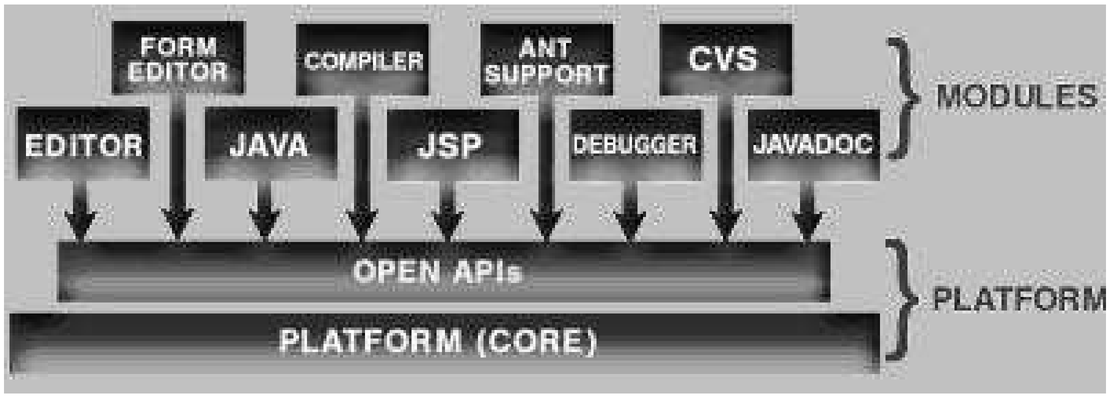


Figure 2.15: NetBeans OPEN API[9]

Programmers can use OPEN APIs to save time. APIs generally provide a default solutions for problems, which can be useful in many scenarios. Examples include user interface management, data and presentation management, code editor, setting management, the wizard framework, configuration management and storage management.

One interesting package that the OPEN APIs provide is the Window System's API. This package provides the capability for a UI component to dock or move to a new position within an IDE. To achieve this behaviors, the class `TopComponent` or its subclass must be instantiated. `TopComponent` is a Swing component, which allows programmers to use this class as a regular Swing top-level container when building their IDE UI modules.

Thus, the Swing framework and the NetBeans platform can be integrated since the Window System API is an extension of the Swing framework.

2.4.5 Eclipse framework

Eclipse[2] is another open source software project which provides a robust and open software development environment. The Eclipse platform is implemented in the Java language and consists of many sub-components.

Eclipse is intended to allow individual software vendors to integrate their development tools seamlessly to create the appearance of a single platform. Software developers can accomplish many tasks using only the Eclipse IDE; there is no need to invoking other tools. Hence, the productivity of developers can be greatly improved.

The Eclipse platform is based on a mechanism for discovering, loading and executing modules to extend the platform's functionality. This mechanism is called a *plug-in* architecture.

A tool developer can package his or her tool as an Eclipse plug-in. The packaged plug-in can interact with the *workspace*, and can have its UI components integrated with the Eclipse platform workbench. When the platform is launched, the configuration file for each plug-in will be found and indexed. This plug-in meta-information is then stored in memory. The Eclipse platform can then load plug-ins as needed.

The following figure, Figure 2.16, represents the architectural components of Eclipse. This diagram was published in the *Eclipse Platform Technical Overview*[11]. In this figure, a *Workbench* and a *Workspace* can be seen. These are the two most important components in the Eclipse platform.

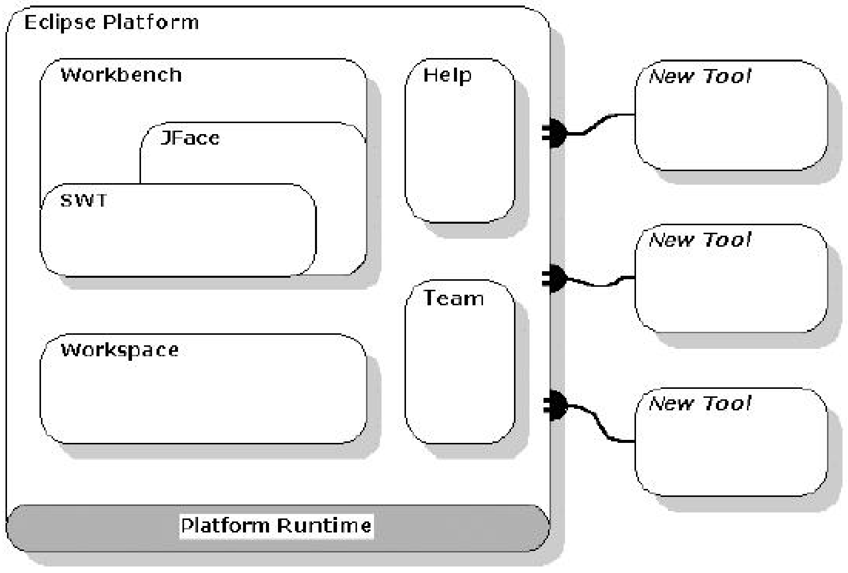


Figure 2.16: Eclipse platforms plug-in architecture[4]

Workspace in the Eclipse IDE is a component that manages the life cycles of resource entities on the local file system. It also manages the synchronization of local copies of these resources and their remote counterparts in a code repository. Eclipse is built around an entity called the *Workbench*, which provides the overall UI structure and interacts with the users.

The Eclipse UI paradigm is based on editors, views and perspectives. A workbench will render editors and views it. Perspectives allow the platform's look-and-feel to be configured, while quick-launch buttons allow users to switch between perspectives. This allows changes such as the layout of the editor and views to be customized by each user.

The Workbench API is dependent on the SWT API and to a lesser extent on the JFace API. SWT and JFace were introduced earlier in this chapter. The workbench is built using both SWT and JFace; Java's AWT and Swing were not used in the workbench implementation. This choice of APIs forms the basis of our research. We will detail this in our case study.

Chapter 3

Our Framework Migration Approach

In this chapter, we shall give an example to illustrate the ideas behind framework migration. The rationale for framework migration will be presented first. Then we will give an introduction to the *dependency inversion principles* which are at the center of our work. We propose the term “framework virtualization” for our method of performing this type of source migration.

3.1 An example of Framework Migration

One example of a GUI frameworks used in application development is Sun's Java Swing framework [7]. Sun also refers to the set of classes used for graphics and GUI in Java as "Java Foundation Classes". However, we will just use "Swing" to refer to the subset of the Java Foundation Classes we are dealing with in this work.

The Swing framework is designed to help developers construct GUI applications. Many Swing-based applications are designed to help users accomplish complicated tasks by presenting information on the display and providing visual cues to the user. A GUI can greatly reduce the time required to learn how to use a software tool and decrease the chance of making errors. They can also improve user productivity.

There are many software engineering tools implemented by using GUI components to interact with users to solve complicated problems in software engineering. *lsedit* is one such GUI application. As mentioned in the Introduction, *lsedit* is a software architecture visualization tool originally built and packaged in PBS, the Portable Bookshelf Tookit[28]. It was later integrated into a toolkit called *SWAGKIT*[10] as a single downloadable tarball by the software architecture research group at the University of Waterloo. This tool is also called the "landscape editor" as we can view and edit software components within this tool's editor pane. Users can view and edit architectural components by double-click and drag-and-drop actions using the mouse. The tool is built using the Java Swing framework.

Software evolves as requirements change. We decided to deploy our tool in a variety of integrated development environments (IDE) because we believe that the IDE allows tools to cooperate with each other tools to produce a benefit greater than the sum of its parts. The Netbeans [9] and Eclipse [4] platforms are two examples of such open source IDEs that can be easily extended.

However, integrating the landscape editor with the Eclipse platform requires massive modifications because the it is built on the Eclipse framework. This framework is not compatible with the Swing.

Migrating the landscape editor from the Swing framework to the Eclipse framework is difficult and time consuming because the Eclipse framework is not an extended version of the Swing framework. The Eclipse framework consists of many features such as the platform runtime, workbench, workspace, team management and debugging. Indeed, the Eclipse framework is more than just a GUI framework, and can be considered a framework for building applications.

There are many Eclipse plug-ins that have been released to the market and the Eclipse IDE has become the favorite tool platform for many developers we talked to. Thus, the value of working on a solution for migrating applications from Swing to the Eclipse framework can be seen as work that will benefit developers. In our research, we will try to accomplish this migration by using a simple approach we will elaborate on at the end of this chapter.

3.2 Framework Dependency Inversion

In his article *Dependency Inversion Principle*[26], Robert Martin outlines three properties that he believes result from bad design in a project. These are:

- It is hard to change because every change affects too many other parts of the system. (*Rigidity*)
- When you make a change, unexpected parts of the system break. (*Fragility*)
- It is hard to reuse in another application because it cannot be disentangled from the current application. (*Immobility*)

Many software projects that fulfill their requirements will still exhibit some of these properties. To make the framework migration process smooth, we must consider these factors carefully when planning.

The dependency relationships between modules in a software project must be dealt with carefully. Framework migration can be considered almost wholly the rearrangement of these dependency relationships.

In his book "Agile Software Development"[25], Robert Martin expands on these issues and gives advice on avoiding a bad design. Martin defines two principles for *dependency inversion*, which should be used to improve the design.

- High level modules should not depend upon low level modules. both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Traditional development will create a software structure where high-level modules use low-level modules and the abstractions depend on platform specifics. Martin use *inversion* to refer to the principles that inverts this traditional dependency method. In the next section, we shall use Martin's idea of inversion in a software maintenance strategy to perform framework migration.

3.3 Software Joint and Framework Virtualization

To make this framework migration a success, we apply the dependency inversion principles to the software to be migrated. This helps remove the rigidity and immobility of the software. However, fragility remains. The reason for this is explained later.

The landscape editor is a stand alone Java GUI application. The model-view-controller (MVC) design pattern[20] was used, although the programmers were not aware of it. In general, all GUI applications employ this design pattern, explicitly or not, to make use of components' inherent benefits, such as their extendability.

We can separate components which use such a pattern into three distinct groups: a model, a controller and a view using the MVC design pattern.

The model is the set of components that represent the state of the application. This state is changed by interactions between components or by stimulus external to the model. The model is not aware of other entities; it does not know about the controller or the view.

The view is the set of components that provides a visual representation. The view updates the screen according to the state of the model.

The controller is the set of components that deliver the user interactions or commands to the model so that its state can advance or change. Figure 3.1[1] illustrates the relationships between these three entities.

In this figure, all three entities exist within a UI component. In some cases, a model can exist external to a UI component. We analyzed the source code of the landscape editor and discovered that separating the classes in the package into three distinct groups was a difficult goal, as some classes representing the model also play a role in the view. Thus, to slice the software into three distinct pieces was almost impossible.

However, import statements in the Java source files gave us a tremendous amount of information when we analyzed the source. Dependency on other packages is a very important property of modern software components. Some dependencies, such as the landscape editor's dependence on the Java Swing packages, are not reversible.

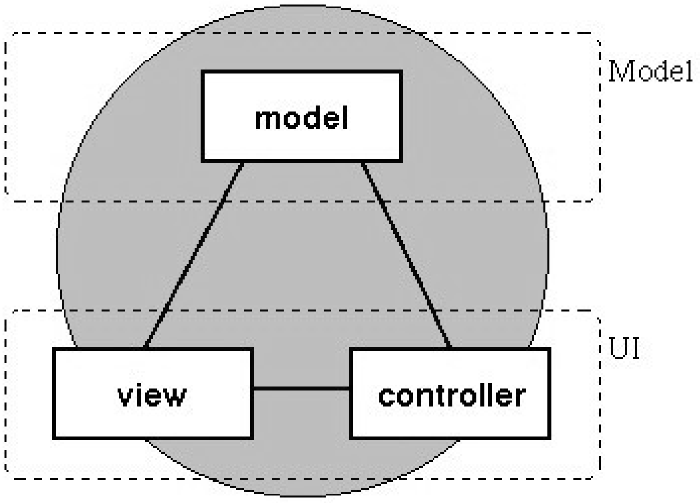


Figure 3.1: MVC illustration [1]

After translating the statements in the program into relations between objects or classes, it was easy to spot places where the landscape depends on Swing. These "hot spots" are the places where interactions happen between the landscape editor and the Java Swing components.

We were able to change the import statements at these "hot spots", in order to create a dependency from the "hot spots" to another type of GUI component. Furthermore, inserting flexible software joints into these "hot spots" enabled a greater degree of interconnections between all GUI components.

Joint is a design pattern, commonly found in the mechanical industry, which is used to transform mechanical force from one direction to another. The joint pattern can also be used in software projects to link software components. The software version of the joint pattern is detailed next. These joints also cooperate to provide an abstract shield between the application and any GUI implementations in the environments where it is executing. Hence, dependency inversion[26] is achieved through the re-engineering of these ”hot spots”.

In Figure 3.2, we use the Button class as an example of dependency inversion. In this diagram, the application was originally dependent on an object of the Button class. We provided an abstract form of the Button class so that the application could be modified to depend on this abstraction. An adapter for this Button class was also implemented to enclose the Button class and depend on Button’s abstract interface. Thus, the application depends on the abstraction and the abstraction does not depend on the details. The principle of dependency inversion is carried out in the process. This method is more flexible than traditional approach where programmers simply replace the source framework classes used in applications with the matching target framework classes. Therefore, we can deploy the application in a variety of environments. This is a very desirable feature for an application. The method of dependency inversion in practice is described next.

To make software joint components, there are several issues that need to be considered. Since joints are designed to connect families of GUI components, we must compare the different GUI class hierarchies and find out what capabilities they share. Conversely, disparities between the hierarchies’ functionalities also need to be considered.

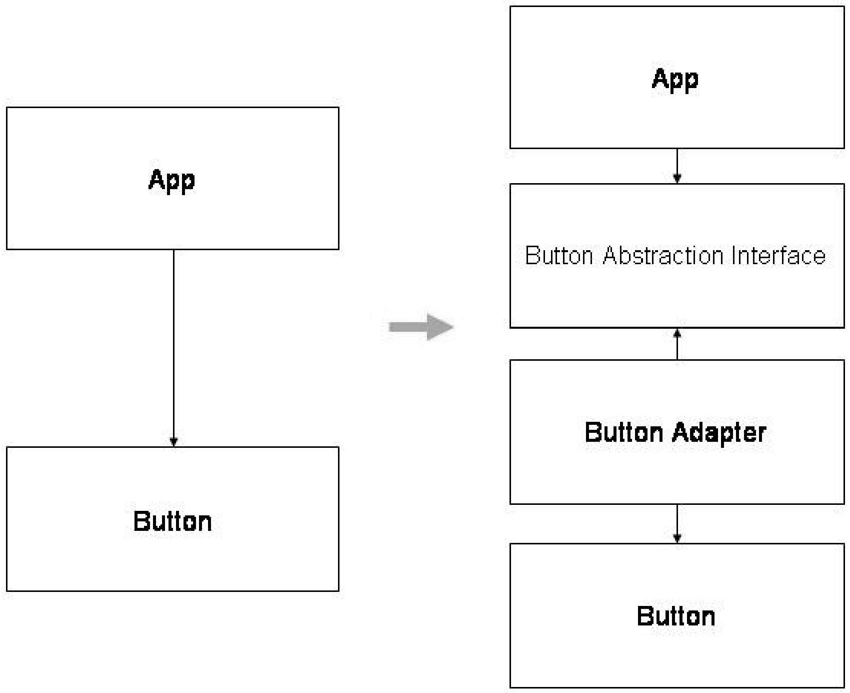


Figure 3.2: Dependency Inversion

To find an effective design for this joint mechanism, we extracted common factors from all candidate GUI frameworks. We used a combination of the structural patterns introduced in Chapter 2 to form this mechanism. Bridge and adapter patterns were the most commonly used in this joint mechanism.

The software joint also requires an interface to advertise its functionality to applications. This interface should look similar to its concrete framework's counterparts. We believe that these frameworks have captured good design patterns for GUI software development. Thus, the reuse of the design patterns in the abstract framework can greatly reduce the risks of running into design flaws in later development phases. The interface for these joints is abstract and its sole purpose is to pass method arguments to the concrete framework.

Thus, having a detailed analysis of the different GUI frameworks can contribute to a good design of the interface for the abstract software joints. This approach generally follows the rules for forward engineering. We can even use this approach when developing an application from scratch.

In our work, we needed a design for the joints. Thus, we needed a procedures to overhaul "hot spots" and a commitment to how much effort we are willing to spend on these overhauls. If we can find an effective design for the joint mechanism, the degree, breadth and depth of use of the original concrete frameworks will not play any significant role in the process of migration.

3.3.1 Migration Path

Migration by systematic removal of dependencies on the old framework and the corresponding erection of dependencies on the new one is intrusive, labor-intensive, and risky. This style migration is often expensive and simply not practical. Furthermore, the use of new target framework classes must be planned carefully, otherwise migration will overwhelm the source maintainers. Minimize the impact and providing a smooth integration process are vital. Our challenge is to make the interface identical to the previous source framework and link it to the new concrete framework using the *dependency inversion principles*. The dependency inversion is not only used when the migration is perform but it can also be used when the application is deployed in an environment with either framework. If this approach works, then the cost to maintain the "hot spots" in the application will be greatly reduced. This process is identified as the virtualization of the application frameworks. The classes and interfaces extracted in addition to other source artifacts used to link the target frameworks to the application are called the *virtual framework*.

We illustrate our idea, using the diagram shown in Figure 3.4, that software using framework X can be migrated to using virtual framework X. The virtual framework X is an abstract layer which resembles framework X's interfaces but will use adapters to communicate with another concrete framework, Y. The virtual framework X does not directly depend on concrete framework Y. Hence, the dependency inversion is achieved by using virtual frameworks to perform the framework migrations. Rigidity and immobility in the migrated software are simultaneously removed as an additional benefit.

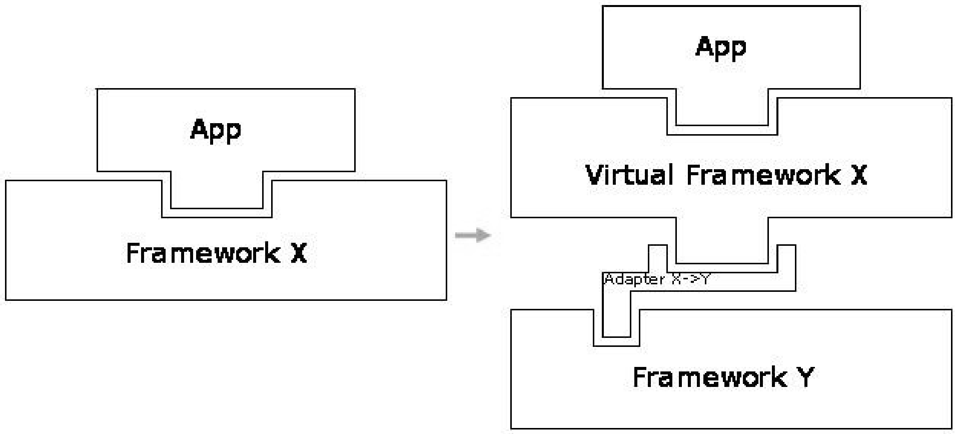


Figure 3.3: Framework Virtualization

This approach avoids the problems of “reinventing the wheel” for a given problem domain. The application logic and semantics are preserved. The application execution sequences of statements remain the same. In the ideal case, no new test cases are needed. Therefore, there is no confusion for the application maintainers.

We used this approach and made several Java packages to assist with framework migration from the Swing framework to SWT. We named this small set of Java packages the *Virtual Widget Toolkit*, as it represents the problems it aims to solve and the abstract nature of the class design. The same interface that the application originally used from Swing is now an abstract interface in the proposed toolkit.

3.4 Architecture of Virtual Framework

We generalized the idea introduced in the above section and integrated several commonly used structural design patterns into the virtual framework in order to perform additional framework migrations.

This architecture is a virtualized version of the original source framework. The virtual framework uses the proxy-adapter and other patterns, which we considered a strong approach for achieving more maintainable source. It can be also be used to migrate software from using a single framework to multiple frameworks, due to the proxy-adapter pattern which is used in this architecture. The rationale for using the proxy-adapter pattern has been discussed in the previous chapter in Section 2.3.3.

Figure 3.4 illustrates the architecture of the virtual framework used in this framework migration. As the figure shows, only proxy classes can be instantiated by the applications. The proxy classes do not interact with the objects of the concrete framework classes directly. Instead, these framework objects are wrapped by their respective adapters. These adapters collaborate with the proxy objects and forward client requests to the wrapped concrete framework objects. When a different framework is used, a new set of adapters will need to be provided. However, the proxy and adapter objects can continue to communicate in the same way they currently interact.

It is important to clarify that the proxy and adapter classes each have their own class inheritance hierarchies. For example, a proxy class may extend one of the base proxy classes or the abstract proxy classes of this virtual framework so that the migrated source is type-correct and does not generate compile time errors. The operations of casting objects in the migrated source will execute it the same way as in the original source.

On the other hand, adapter classes can also extend one of their own base adapters in the virtual framework to reuse code that has been pre-defined in the base adapter classes.

The proxy classes define the hierarchy for the abstraction, while the adapter classes form another hierarchy for the concrete implementation. Thus, the separation of the abstraction and the implementation is achieved. These two hierarchies are linked by the message forwarding mechanism. We use this new proxy-adapter pattern in the virtual

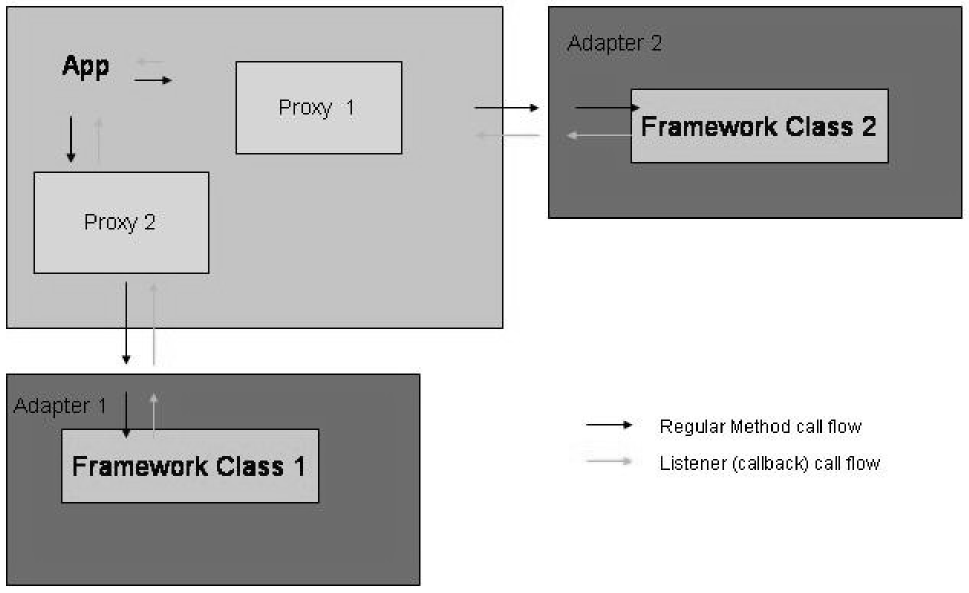


Figure 3.4: The Virtual Framework Architecture

framework. Hence, a novel form of the bridge pattern is expressed by this combination pattern.

Next we will discuss several points related to these class hierarchies and the link between them. Diagrams will be included to explain the differences between the proxy and adapter hierarchies. Two illustrations of the class hierarchy of a virtual Java Swing GUI framework are provided in Figure 3.5 and 3.6. The virtual Swing framework will be explained in detail in the next chapter.

In summary:

- the virtual framework has a class hierarchy of its own
- the proxy class hierarchy and the adapter class hierarchy are different
- a message forwarding mechanism links them

Figures 3.5 and 3.6 are class hierarchies found in two distinct Java packages in the virtual Java Swing framework. In Figure 3.5, the proxy class hierarchy has the same structure as the original Java Swing framework. If a migration is performed, the application will use the proxy classes instead of the original Java Swing classes. The casting expressions in the migrated applications will function without any modifications, since the class hierarchies of these source and target frameworks have the same structure. Thus, the migrated source is guaranteed to not produce compile-time or run-time errors due to casting expressions, unless these were present before migration.

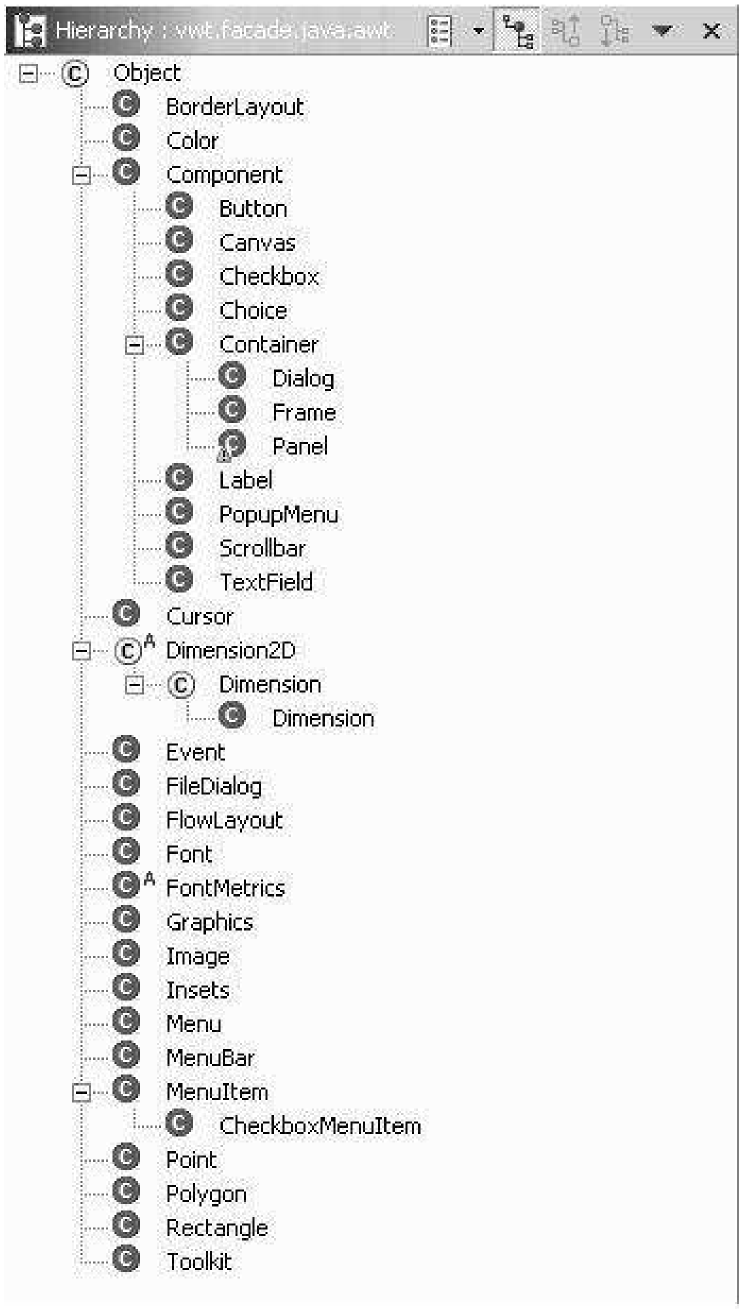
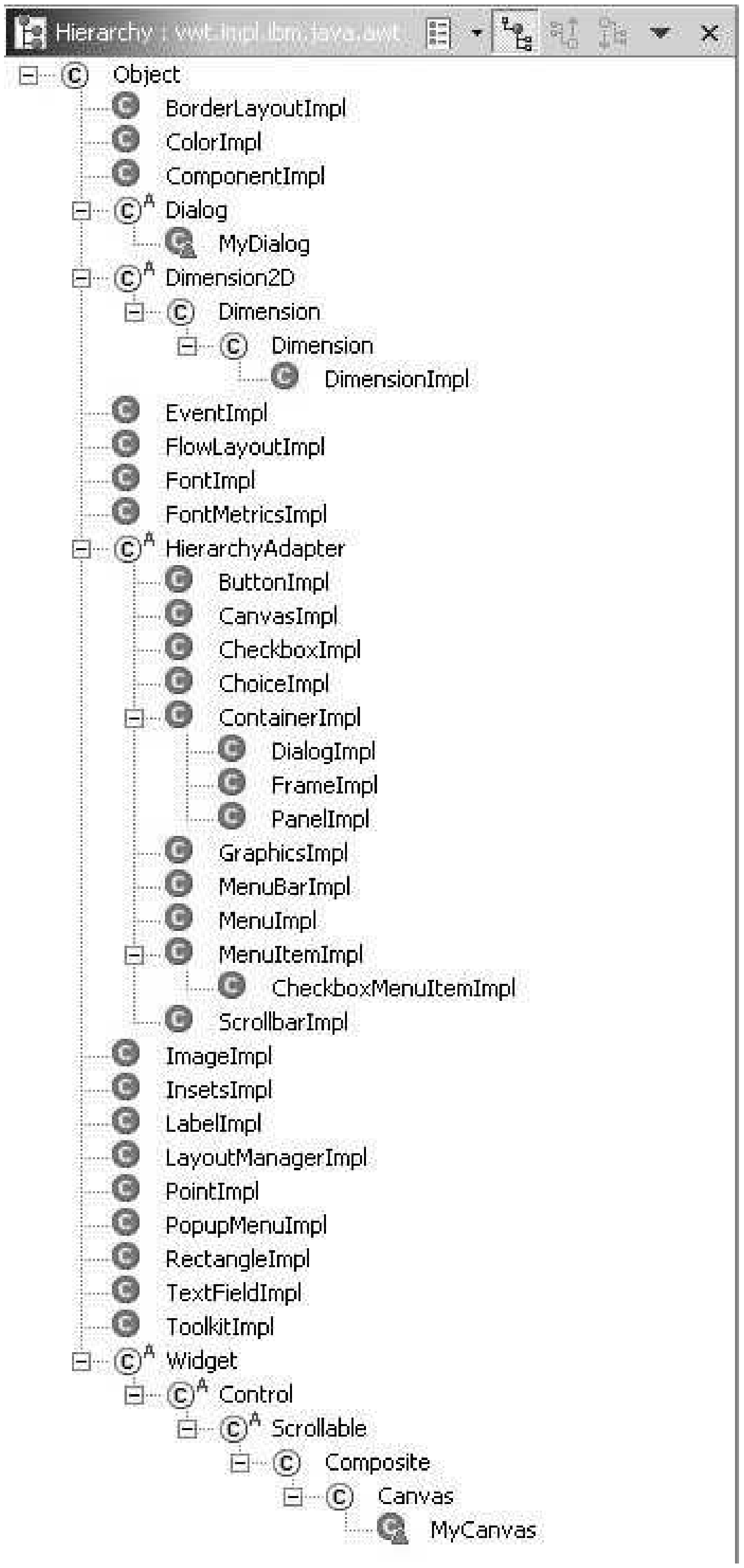


Figure 3.5: Virtual Framework Proxy Class Hierarchy

As shown in Figure 3.6, a distinct hierarchy for adapters is formed to re-use the target framework classes and the adapters themselves as much as possible. This hierarchy looks similar to the combination of the source and target framework hierarchy because the adapters need to communicate with the objects of proxy classes and the target framework classes. The mechanism for this communication is *message forwarding* and this mechanism will be explained in greater detail in the next chapter.



```
Object
    BorderLayoutImpl
    ColorImpl
    ComponentImpl
    Dialog
        MyDialog
    Dimension2D
        Dimension
            Dimension
                DimensionImpl
    EventImpl
    FlowLayoutImpl
    FontImpl
    FontMetricsImpl
    HierarchyAdapter
        ButtonImpl
        CanvasImpl
        CheckboxImpl
        ChoiceImpl
        ContainerImpl
            DialogImpl
            FrameImpl
            PanelImpl
        GraphicsImpl
        MenuBarImpl
        MenuImpl
        MenuItemImpl
            CheckboxMenuItemImpl
        ScrollbarImpl
    ImageImpl
    InsetsImpl
    LabelImpl
    LayoutManagerImpl
    PointImpl
    PopupMenuImpl
    RectangleImpl
    TextFieldImpl
    ToolkitImpl
    Widget
        Control
            Scrollable
                Composite
                    Canvas
                        MyCanvas
```

Figure 3.6: Virtual Framework Adapter Class Hierarchy

Chapter 4

Case study

We began the present study with a practical problem, namely integrating an existing reverse engineering software tool (lsedit[10]) into an Integrated Development Environment (IDE) framework with nonstandard windowing support (Eclipse [2]). Let us now look at the details of this migration example. In the present chapter we detail how the techniques outlined in Chapter 3 were applied in practice.

The tool to be migrated, formally called the Landscape Editor, is used as a case study to illustrate several issues that relate to framework migration. To provide some background, the features of this tool will be detailed. It was also desired to integrate this tool with the Eclipse platform, so this became part of the migration.

To achieve this goal, the tool could simply be configured as an external tool for the Eclipse platform. Another option would be to deploy the tool as an Eclipse plug-in. The second method was preferred because it results in better integration and is a good example of API migration[23] as we have described in Chapter 1. We will examine this method of integration and present the problems associated with this method in the later sections. Other related framework issues will also be discussed in this chapter.

4.1 Tool integrations with Eclipse platform

Many tools have been developed to help programmers speed up the development cycle and reduce the number of defects in the software. Some of these tools are quite successfully in handling a specific job. Unfortunately, many of these tools are stand-alone applications.

Compilers and linkers are two good examples of these tools. They will process input files of a format which has been carefully specified by programming language experts. Standard output or a custom GUI is used to indicate success or failure. If no error is found, object files will be produced by the compiler and will be linked to libraries to become binaries.

However, if an error is found in the processed source file, the programmer has to edit the source file using an editor. After the programmer is satisfied that the error has been removed, the compiler will be invoked again to verify that the problem has been corrected. If it has, the linker then needs to link the object code with the library.

It makes sense to streamline compilation and linking tasks. Since these tools provide a traditional text-based user interface and can return a value indicating whether execution was successful, the programmer can pipe the resulting object files from the compiler to the linker to automate the process.

However, many other tools are stand alone applications and not all of them can co-operate to solve the specific problems a programmer faces. Each has its own ways to perform tasks. Thus, integrating and using these tools together can cause confusion and problems. There are a number of specific problems that may be encountered.

The first possible problem is resource access constraints. Different tools implement their own methods for managing resource access requests. Thus, tools trying to access the same resource file need to synchronize their access sequences.

Another problem is integrating these tools' GUI components into a single window frame so that the programmer can edit the source and invoke the build process without switching window frames. Tools with GUI components are often built on different technology. Our architecture landscape editor is built with Java's Swing framework and it is difficult to integrate it with Eclipse's workbench which is based on the Standard Widget Toolkit.

Ideally, we believe that an IDE, like Eclipse, is the best place to use different tools together and can be useful to most programmers. Such an environment needs to provide infrastructure facilities for basic services to the hosted tools and can act as a coordinator if access conflicts occur. Eclipse uses a plug-in mechanism as the standard method to extend its functionality, which allows tools to be loaded and managed by a single control mechanism.

Recently, the Eclipse project[11] got many developers' and researchers' attention because of its open source approach. We believe it is useful to deploy tools within an IDE such as Eclipse. Therefore, the Eclipse platform was chosen for this purpose and our first attempt was to migrate our landscape editor onto the Eclipse platform to explore how GUI components built with different technologies can be integrate together.

4.2 Landscape Architecture Editor Tool (*lsedit*)

The landscape tool was constructed as a stand alone application initially and evolved into an applet within the bookshelf environment[28]. Later it was packaged into a software architecture analysis toolkit called *SWAGKIT*[10], and was re-named the Landscape Architecture Editor Tool. In this work, we will refer to this tool as "The Landscape Editor", or *lsedit* for short. SWAGKIT was designed to parse source, extract architectural representations and visualize software components.

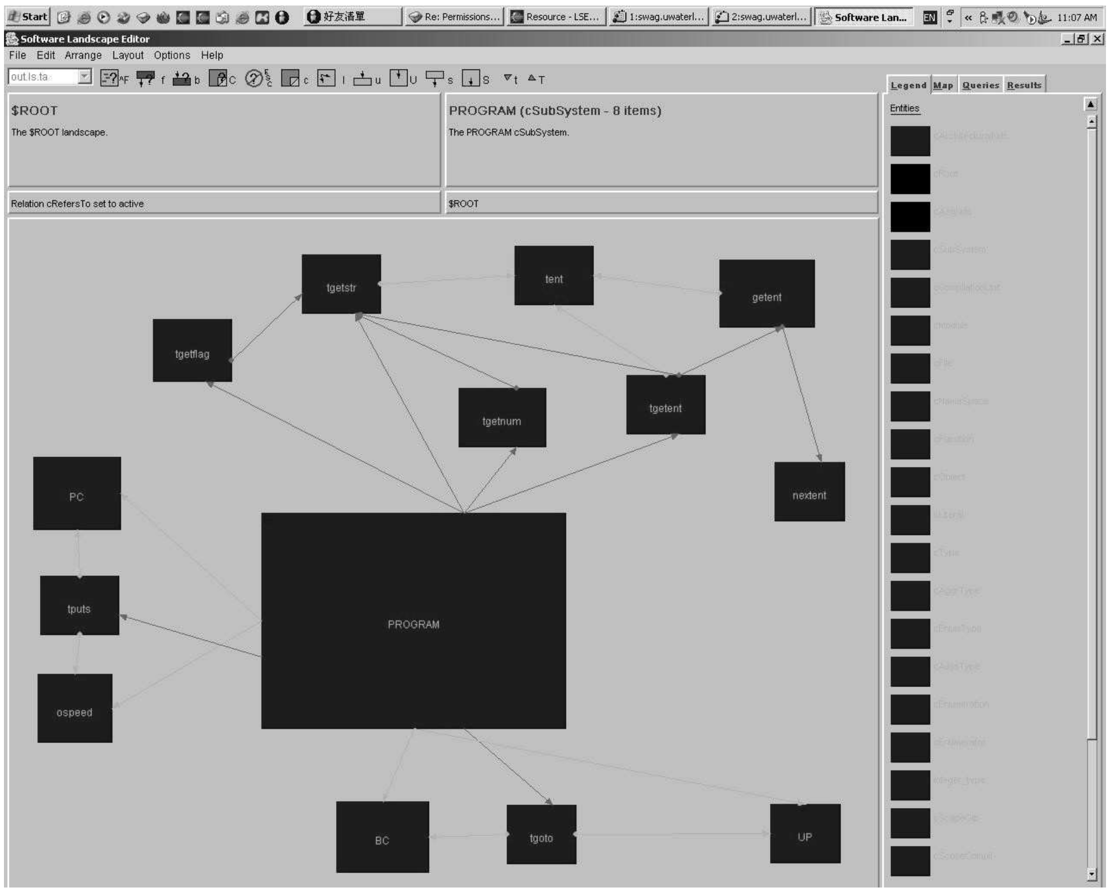
The Landscape Editor is a Java application designed to visualize the components of selected software systems. This visualization is realized by displaying architectural diagrams. Users can edit a diagram and save it to the local file system. Some simple visual queries are provided to assist the user in learning about the software architecture.

The Landscape Editor can create a diagram from a TA file or save a diagram to a TA file[3]. TA files are formatted using the Tuple-Attribute Language. This language records certain graphs as plain text. The information recorded about these graphs consists of tuples and attributes. Tuples describe the nodes and edges of the graph, while attributes describe the properties of these nodes and edges. Hence, a TA file can be seen as a graph database. We can insert or modify tuples and make queries to this database.

The relationships found in a software system can be encoded in TA format. While displaying software components by rendering diagrams, users can manipulate nodes and edges in these diagrams. Their positions can be stored as attributes in TA files.

The Landscape Editor was first designed to run on a web browser to view the architectural diagrams stored on a remote server or to run as an stand alone application on a local machine. Thus, the editor was implemented as a Java Applet and provided read-only access. The stand alone version will use the graphical rendering capability of the applet and add other TA manipulation and file access functions from the application.

Figure 4.1 shows the concept behind this tool.



$ROOT

The $ROOT landscape.

PROGRAM (cSubSystem - 8 items)

The PROGRAM cSubSystem.

Relation cRefersTo set to active

$ROOT

Figure 4.1: The Landscape Editor

4.3 Migration method chosen for the Landscape Editor

When we began this project, we performed a thorough examination of the Eclipse platform. At its core, the Eclipse platform is a plug-in discovery and loading mechanism. The rest of the functionality of Eclipse is provided by plug-ins.

The platform has provided some predefined extension points and a set of default plug-ins for these. However, platform plug-ins may define other extension points to allow the integration of third-party plug-ins.

Since the Landscape Editor is an editor, allowing the display and modification of a diagram, defining it as a UI plug-in is the natural choice. Our Landscape Editor will use this extension point and be packaged as an Eclipse plug-in so that it can be easily deployed to any Eclipse runtime environment on any operating systems.

The Eclipse platform provides an interface named `IEditPart`, which defines several methods that must be implemented. Once these methods are implemented, they are invoked by the platform so that Eclipse can manage their life cycle.

Many similarities exist between this interface and other Java component technology. Programmers only need to extend or override a set of methods for these components in order for them to be invoked. Consider the Java Applet class, the `init` method is defined to initialize the applet. Correspondingly, in `IEditPart init` is also used to initialize the editor. Other methods exist to support resource management. One of the most important methods, relating to UI component integration, is `createPartControl`. This method allocates UI component resources and creates an UI containment hierarchy for the editor plug-in. A successful return from this method is essential for the creation of a robust UI plug-in. It defines a parameter which is a `Composite` from the Standard Widget Toolkit. This `Composite` provides a UI component that the editor can place components on.

In the Standard Widget Toolkit, all concrete widgets are subclasses of `Control`. `Composite` is one of these subclasses of `Control`. A `Composite` may contain other `Composite` or `Control` objects. A `Control` can not contain anything. If we subclass `Control` or `Composite`, the instance of this class can be added to the UI component containment hierarchy. In general, as long as a widget is a subclass of `Control` and the rules above are observed, the creation of a UI containment hierarchy quite simply.

However, our Landscape Editor was built on the Java Swing framework, not SWT. The AWT and Swing classes hierarchy and Standard Widget Toolkit hierarchy form two distinct sub-hierarchy trees. Their only common parent class is `Object`. In other words, they hardly have anything in common in terms of their inheritance hierarchy. Since *lsedit* consists of only Swing components. According to the rules above, it is impossible to link the two containment hierarchies together.

Due to the previously mentioned problem, we decided to study the internals of the Landscape Editor and the two class hierarchies before we proceed further. We will use the virtual framework that we proposed earlier as the architecture for migrating the Landscape Editor to the Eclipse framework.

In this case study, we decided to leave the resource access part of the code intact since it did not interfere with performance and users were not aware of its presence or absence.

4.4 Framework classes and interfaces used in *lsedit*

To make this virtual framework a reality, we used a tool called *jclassinfo*[8] to extract the implementation information from the binary class files of *lsedit*. This tool is a parser, written in C, that is used to retrieve fields, methods and other related information from class files in binary format specified by the Java VM specification.

We wrote a script to scan all class files in the the Landscape Editor and generate a report of Swing classes that are used. This report also identifies which fields and methods of Swing classes are referenced. This results is a subset of the Java Swing classes and interfaces that must be supported for the migration to succeed.

4.5 Package names and hierarchies in Virtual Swing Framework

While migrating *lsedit* from being dependent on Sun's Swing framework to depending on our "Virtual Swing" package we looked for ways to make this transition smoother. To this end we add a prefix to each Swing package name that was being replaced. For example, a `Container` class is defined in the java.awt package. In our Virtual Swing Framework, we provide a package called vwt.java.awt and defined a `Container` class in this new package.

By putting prefixes in front of the package names, we achieve several goals. First, it resolves naming conflicts with the original Swing packages which might cause confusion in certain environments.

Secondly, the import statements relating to the original Swing packages, specified in each source file, are easily located and changed. This process can be automated by developing a script to scan the source file. Once the string patterns are identified, the script can change it accordingly. There is no table required to do the changes. The script will always insert the string *vwt* in front of each Swing package name found in the import statements of the Java source files.

Finally, the class design of the original Swing packages is preserved in the Virtual Swing Framework. The interface is extracted and not changed, so applications will not encounter compilation errors once the transition has been performed. The semantics are also preserved, so programmers can read the original Java Swing documentation when modifying the application. Thus, the need for producing documentations is also eliminated.

4.6 The Architecture of the Virtual Swing Framework

Once we have completed the virtualization of the application framework, we must deal with more difficult challenges. Since we intend to support applications running in a variety of environments, we need to provide multiple implementations of these GUI frameworks.

We began by providing two implementations packaged in distinct hierarchies, vwt.sun and vwt.ibm. These two packages provide support for the Java Swing framework and the Eclipse UI framework respectively. After a while, we found that in order to change frameworks we needed to either manually edit the import statements in the source files or run a script to change them. This approach would causes frustration for many developers. It also causes the replications of the same interfaces in two package hierarchies. Synchronization of the interfaces in these two package hierarchies is also tedious.

We looked at the Java language specification and several Java tools. Conditional import statements are not supported or defined in the language specification, nor are there any Java tools that can preprocess conditional statements in the manner that a C preprocessor can.

An alternative was chosen to unify the package hierarchies into one so that the synchronization of the interfaces and the need to change import statements when the environment is different were eliminated. We created a proxy layer to handle this jobs, as the proxy classes will load the adapter classes for each of the implementation packages as indicated in Figure 4.2. We use a `Button` and a `Container` class to illustrate this. By doing so, we were able to decouple the abstraction from the implementation in a different manner than the one suggested in "Design Patterns"[17]. As indicated in the figure, proxy, façade, bridge and adapter patterns can each be used to solve this problem. However, the bridge pattern has a different form in this architecture.

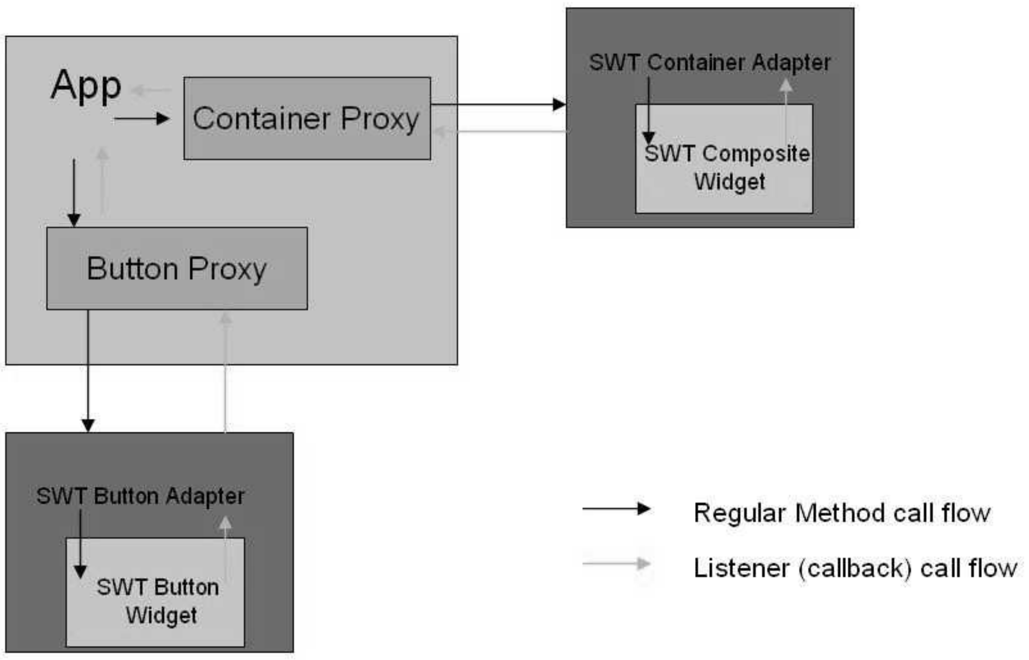


Figure 4.2: The architectural view of Virtual Swing Framework

4.7 Protocol conversion

Pluggable objects exist specifically for the purpose of adapting to various environments. The correct sequence in communication with them is essential to the success of our work. In this sections, we will explain two object interaction characteristics present in our design of the Virtual Swing Framework.

The first is multiple-stage message forwarding. The second, protocol translation, is more critical, as handshaking is the determining factor for allocating windowing system resources. We will explain both these issues in greater detail.

4.7.1 Multiple stages message forwarding

The virtual framework is designed to support applications for a variety of environments. A multi-tier architecture is used in the implementation to help allow this.

As illustrated in the previous chapter, the proxy classes are the front end used by client applications. This tier is designed to be fully compatible with the original Java Swing framework interface. Changing this dependency from the original Swing framework to our Virtual Widget Toolkit is quite straightforward.

The middle tier contains the adapters for each concrete framework class which adheres to the specifications of the reference interfaces, that is the interface the proxy classes can understand. In the case of our implementation, this is the same as the original Swing framework interface. This was done for the sake of convenience and to avoid introduction of new problems. The proxy tier can simply forward messages to this adapter tier.

The back end tier is the concrete UI component classes of the Swing framework and SWT classes. The objects in these two implementations are the actual objects that are the gatekeepers to the windowing system and are provided to satisfy clients' requests.

The architectural structure of this virtual Swing framework is layers. Communications between these layers is illustrated in Figure 4.3.

The clients' requests need to follow a predefined path. The message is eventually delivered to the concrete framework objects. These messages are either delivered intact or modified so that they can be understood at the final destinations by the Swing `Component` or the SWT `Widget` classes.

The degree of paths for these message is usually three and the arguments in the messages are usually not modified. However, the content of the message might be modified to reflect the fact that an interface mismatch has occurred.

Another term to describe this multiple-stage message forwarding is "wrapping and unwrapping the messages". As each tier receives the message, it is examined to decide if extra processing or modification is required. After making this decision, and implementing

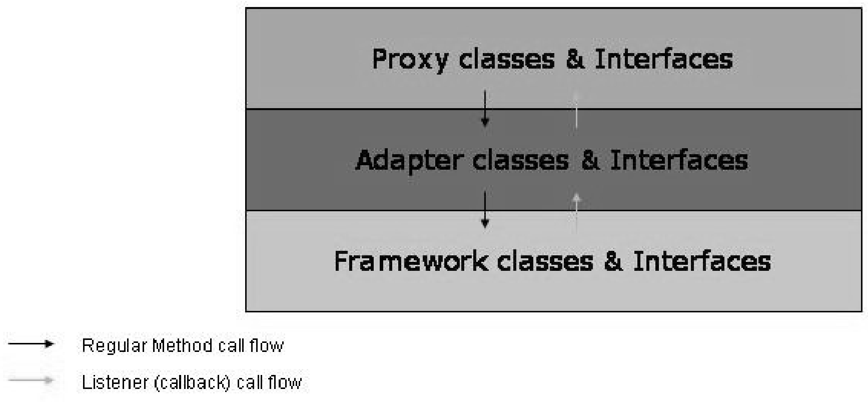

Proxy classes & Interfaces
Adapter classes & Interfaces
Framework classes & Interfaces
Regular Method call flow
Listener (callback) call flow

Figure 4.3: The message forwarding between layers in the virtual Swing framework

any such modifications the message is sent to the receiving objects at the next tier.

However, as mentioned, typically the message is passed along its paths without any modification. The most common change is converting a value from a generic data type to its class equivalent counterpart or collapsing two or more generic data type values into a single object, a simple aggregation of these primitive values.

4.7.2 Protocol translation

The second form of protocol conversion happens when UI components are instantiated, but the containment hierarchies are formed from two different implementations. Such a situation, occurring when using Swing and SWT was handled as follows.

In the Swing framework, there are usually two types of classes involved, the `Container` and `Component` classes. The constructor for the `Container` class is defined as:

```
public Container();
```

The programmer will have to create an instance of the `Container` class first. The following statement needs to be added to the program:

```
Container container = new Container();
```

Then the programmer creates an object of the `Component` class and adds it to the containment hierarchy of the `Container` object which was just created.

The constructor for the `Component` class has the general form:

```
public Component();
```

The programmer will generally use the statement below to create a new object of class `Component`:

```
Component component = new Component();
```

Once the `Container` and `Component` objects have been created, we need to invoke the `Container`'s `add` method to create the association to form the containment hierarchy as illustrated in the line below.

```
container.add(component);
```

Creating `Container` and `Component` objects is straightforward. The constructors of each class is simply called. Programmers can create the objects in any order they wish before the contained object is added to the containing object.

Between these calls, programmers might want to set the property values of the objects created. They can do so anywhere after the object's constructor is invoked. Similarly, getting the property values is possible and can also be done in any order desired.

Now consider the SWT. Programmers also invoke the constructor methods for the UI component classes in SWT, however programmers must be aware of the differences between the calls in Swing and SWT classes. These differences are subtle but require different coding practices.

In SWT, the constructor for a UI component needs parameters. The number of parameters for these constructors is usually two. The first parameter is usually the object that will contain the object being created. The second parameter is an integer value, which is a constant that specifies the style the UI component being created will have.

The statement below is the constructor method signature for SWT's `Composite` class:

public Composite(Composite parent, int style);

We need a reference to a `Composite` object and a style value when we create a `Composite` object. In this case, the reference to *parent* is a `Composite` class. There is no need to invoke the containing object's add method, and in fact no such method exists.

Thus, the program coded in SWT is much cleaner than the Swing flavor as it does not require additional method invocation. The interface design for widgets in SWT is less complicated and more efficient because a second call to create the containment hierarchy is eliminated.

Once the constructor methods are invoked for the SWT widgets, programmers can call any getter or setter methods they desire. This is similar to the approach used in the Swing framework.

Since our Virtual Widget Toolkit needs to communicate with the underlying framework layers, we need a clear understanding of how the clients' requests can be passed to the concrete framework layer correctly. Some translation or aggregation of the messages is essential.

We use the Swing protocol in our proxy layer, since an understanding of protocol translations from Swing to SWT is important for this purpose. In the following sections, we will examine and propose a solution to this problem.

4.7.3 `construct` methods

In the previous section, we introduced the approach for constructing a UI object and adding it to the containment hierarchy in both the Swing and SWT implementations. In this section, we describe the problem when making the translation.

In Swing, programmers have the choice of creating a `Component` object without any arguments passed to it. Similarly, a counterpart class in the SWT class hierarchy can be instantiate. However, a reference to the containing object is needed when this constructor is called in SWT.

The running program will not have a reference to the containing object in the proxy tier until the program advances to the statement where the add method of the containing object is located.

If we invoke the constructor for the SWT widgets with a null reference a run-time error occurs and an exception is thrown. The program then halts where the constructor was invoked. Constructors for SWT widgets can not be invoked this way since we are unable to obtain a reference to the containing object at that point in the execution context.

We must therefore defer the construction of the widget until we are able to obtain the needed references to the containing objects. This behavior is also known as late binding, a concept commonly found in OOP[12]. The usual reason for late binding is to achieve a high level of polymorphism. In our work, late binding is used for resolution of references to containing objects, since we can not have a valid reference until the add method of the containing object is invoked.

After a valid reference to the contained object is obtained through the add method of the containing object, we can create the actual SWT widget correctly. The contained object is passed as an argument to the add method of the containing object. Thus, the containing object will be able to invoke the widget's constructor accordingly.

For this reason, we provide a new set of methods named *construct*, in the adapters, to avoid the confusion that would occur if the constructors defined for the adapters are called a second time. This would not be desirable, and in fact might not be legal at all.

4.7.4 Method call queuing

In the previous section, we briefly explained the problems and solutions occurring when we translate the protocols used between the proxy, adapter and concrete framework layers. In this section, we will examine another problem with these protocols, the method call sequencing issue.

As we explained before, the client is free to access components' property values through the getter and setter methods in any order they desire after the objects have been instantiated.

Object oriented programming principles do not enforce any message sequencing constraints onto any objects. It is up to the applications to handle these properly. This self-management scheme gives us flexibility when developing software system.

In the Swing framework classes, the setter and getter methods can be called at any times after the component in question has been created. However, this free form of call sequences causes some trouble when late binding occurs in the underlying framework implementations. If programmers calls a setter or getter method before late binding has happened, a null exception occurs.

A simple solution to this problem is to queue such method calls in the original application into a vector. This technique is illustrated in Figure 4.4. This queue can be found in each contained adapter class. The Java development kit has provided a package, java.lang.reflect, which is convenient to use to resolve the problem.

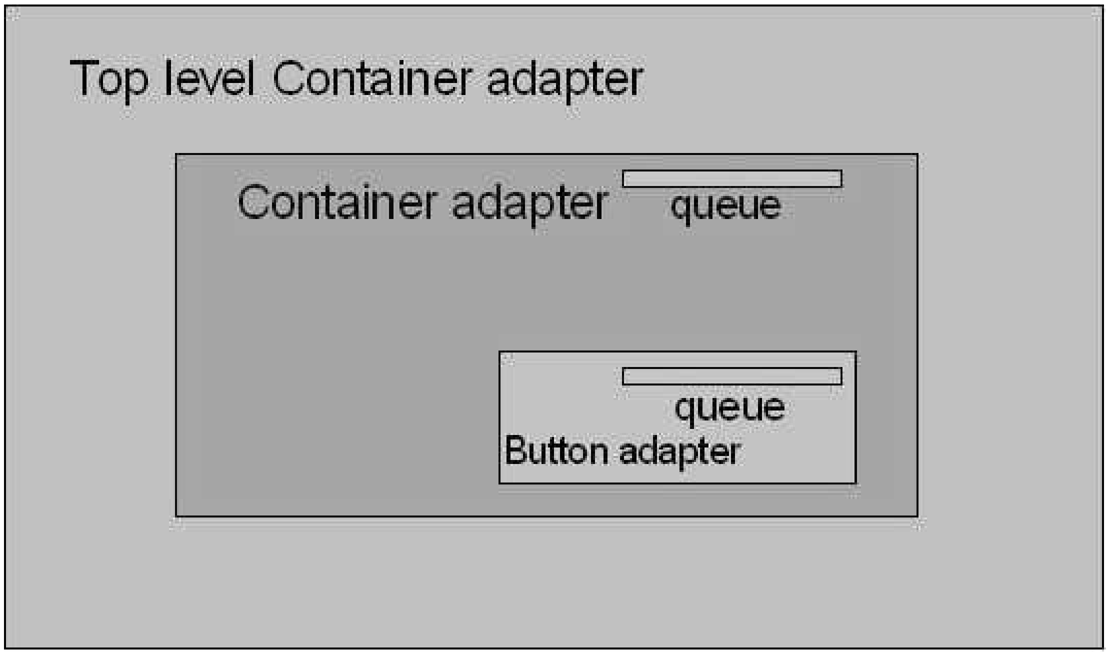


Figure 4.4: Method calls queuing

The queued method calls will be fired sequentially after the late binding is completed. If the construct's method in the adapter class is called, all the queued method calls are executed after the construct's method has returned.

In extreme cases, the construct method of a contained object is also queued because the late-binding for the containing object is also pending its own method call queue. Thus, this method can not be executed and has to defer its execution until a later time. When the topmost overdue containing object's late-binding is complete, all the queued method calls in each of the contained objects will be fired sequentially and recursively, according to the positions of the contained objects in the containment hierarchy and the method call locations in the queues, until all the overdue method calls are executed.

For example, a `construct` method is queued for a containing object. An event occurs when its containing object is instantiated or the late-binding that has just completed will cause the pool of calls to be executed. When this `construct` method is executed, it will trigger its contained objects' queued method calls to be executed. In turn, if the contained object has the construct method queued, it will cause its nested contained pool of method calls to be executed accordingly.

Setters and getters work properly if we employ this mechanism . However, some minor problems might occur if we need to immediately see some side effects from a setter which is queued in the vector. Such a desired side effects will not occur until after the late-binding for the widget has completed.

The getter might also cause problems. If the widgets have not been instantiated, the program has no way of getting the property values from the widget. If the property is a complex type but a generic data type, we can update the encapsulated value of the complex type object at a later time before the values are accessed in the client.

In the case of a generic data type, the same approach applies. We just need to pass a reference to the generic type value as an argument to the queued method calls. The generic type values are updated accordingly when the queued method calls are executed.

However, the same rule apply. The client can not compute a new value, based on values just returned, before the queued method which is responsible for updating the return value is executed.

4.8 Virtual framework mechanism - Virtual Events

A very important aspect of the Virtual Widget Toolkit is that both the latest Swing and SWT implementations support the Java event model used since JDK 1.2. To provide backward compatibility with programs developed using JDK 1.0 or 1.1, the Swing framework supports the 1.0 event model as well. This means that 1.0 and 1.2 event models are both supported in the Swing implementation.

In our Virtual Widget Toolkit, we also need to support these two event models because the classes in the proxy tier of the Virtual Widget Toolkit were written for the Swing framework. Thus, Swing and SWT Adapters can dispatch the events to the right event handlers previously defined in the derived proxy classes. By using the Virtual Widget Toolkit, programmers can specify listeners for proxy classes or just override methods to obtain the desired event handling functionality.

In our implementation, we provide a default event listener for each of the adapters. These event listeners choose the right method to invoke for derived classes in the proxy tier. The concrete framework events need some wrapping to become virtual events before they are passed as arguments to proxy tier methods.

To enable delivery of messages back to the proxy tier, each object in the adapter tier needs a reference to the proxy object which created the adapter object. Thus, communication between the proxy and adapter tiers is bi-directional.

4.9 Framework mismatches

In this section, we examine the two GUI frameworks at the implementation, that is source code, level and compare them in order to illustrate problems that exist. An exhaustive list of items examined will not be provided due to space constraints.

In this section, we first examine the hierarchies of the two frameworks to see how similar they are to each other. Reasons why one does not fit with the other will then be considered. Relationships extracted from the frameworks will be illustrated. We will then present methods for comparing classes which show syntax differences at the method level. Finally, other mismatches, which relate to our migration project for the Landscape Editor to the Eclipse platform, will be explained. Graphic context is one of these examples we will expand on.

4.9.1 Two Class hierarchies

Consider the class hierarchies of the two frameworks we focused our work on, Eclipse and Swing. In the following comparison, we will be focusing on the classes of the Java AWT since it contains the base classes that most Swing components are derived from.

In Figure 4.5, there is a class hierarchies for each of these graphical user interface toolkit. We illustrate here the component classes that are used to construct the user interface. In this figure, only the classes derived from the Object class, which are inherently incompatible with each other, are shown.

Since few readers are familiar with both AWT and Swing components, we begin with an introduction to AWT classes in the next section. We will then match these with their counterparts classes in SWT, if such a counterpart exists.

4.9.2 `Component` class versus `Widget` class

We first introduce the base classes at the top of each class hierarchy. On the left hand side in Figure 4.5, near the the top of the hierarchy of AWT packages, readers will find a class named `Component`. This is the base class that all AWT and Swing components extend directly or indirectly.

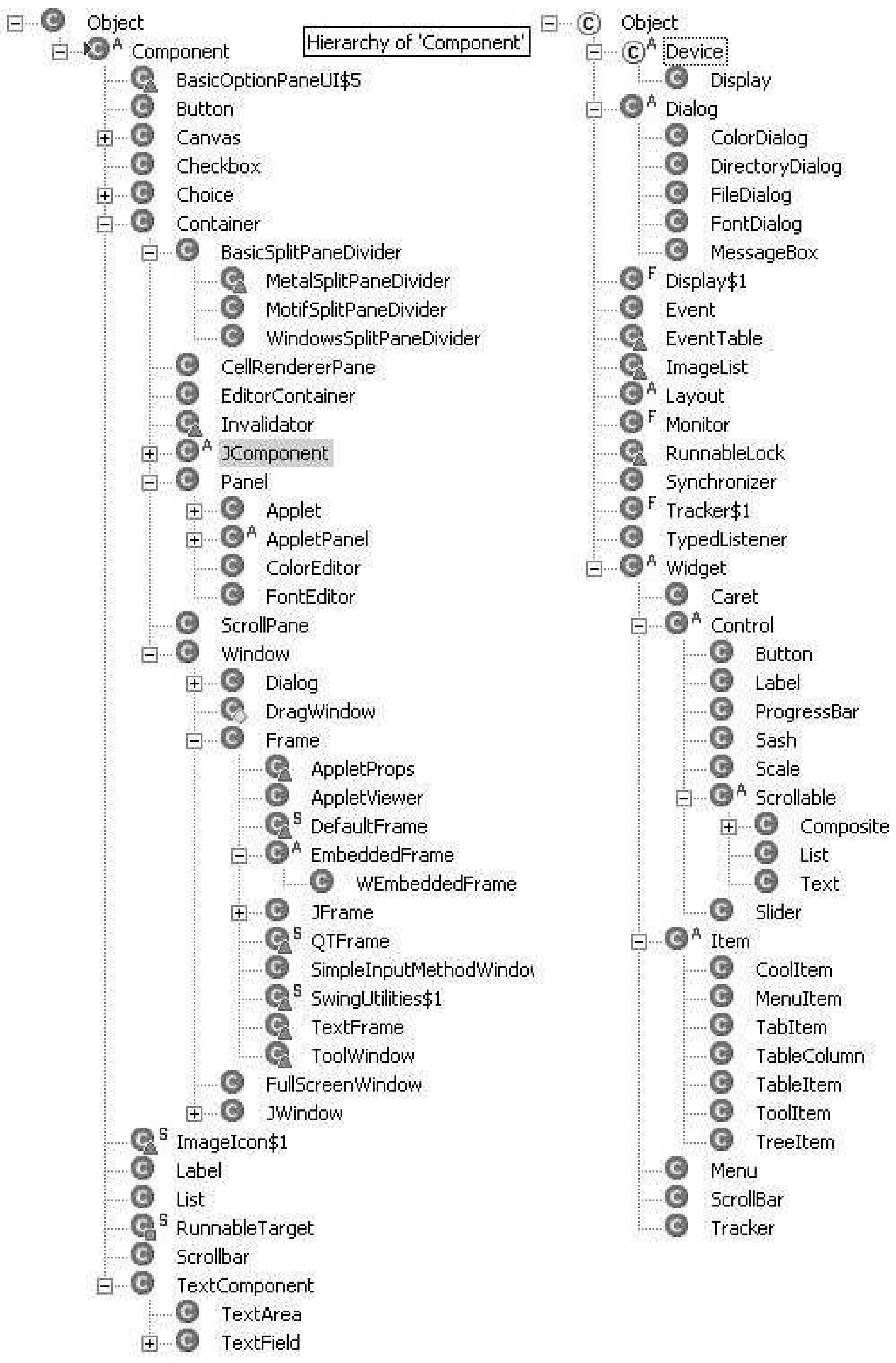


Figure 4.5: Swing/AWT and SWT Hierarchies

Many basic operations have been predefined such that derived classes can simply override these methods or omit the redefinitions of predefined methods. This class also serves as the gateway to communicating with the platform windowing systems.

The derived classes can also add new methods which refine their behaviors. Thus, multiple levels of polymorphism can be achieved through the reuse and overriding of methods defined in the base classes.

Similarly, readers will find a SWT class named `Widget` located in the middle of the hierarchy on the right hand side of the same figure. This `Widget` class is the gatekeeper that controls access to the platform windowing systems. The `Widget` class is responsible for sending requests for memory allocation and graphic resources to the windowing platforms.

However, the `Widget` class is further extended. A `Control` class extends the `Widget` class. Both `Widget` and `Control` classes are abstract, that is they can not be instantiated directly. Programmers must extend them to provide a concrete implementation in order to use them. Nevertheless, the `Control` class is considered the equivalent counterpart in SWT for the `Component` class in the AWT.

Some methods found in the AWT `Component` class follow the setter and getter method convention defined by the Java Bean Specification. The SWT `Control` class also follows this guideline. However, some mismatches occur in the `Control` class of SWT. Readers will find a new method called `setToolTipText` which can not be found in the AWT `Component` class. Similar mismatches will be found in many of the derived classes for the `Component` class and the `Control` class.

4.9.3 `Container` and `Composite` classes

The `Container` class in AWT and the `Composite` class in SWT are comparable in the sense that they both can contain themselves or other types of `Component` or `Control` objects, respectively.

For example, an AWT `Container` class is a concrete class that is used to hold other types of `Component` objects. Complicated user interface construction is possible through the creation of a containment hierarchy.

Similarly, a SWT `Composite` object is able to be constructed to hold other types of `Control` objects. The parent class of the `Composite` class is the `Control` class.

A mapping is found from the `Container` class to the `Composite` class. However, differences exist between the two hierarchies. As we described earlier in the section about structural patterns, a *decorator pattern* is used to incorporate additional functionalities without the use of inheritance commonly found in object-oriented programming development.

A concrete example is exploited in the AWT implementation. In AWT, a ScrollPane is provided to give scrolling functionality to AWT components. The AWT `Component` object is passed as an argument to the class ScrollPane's add method. The hierarchy in the AWT does not play any significant role.

However, the `Composite` class in SWT is derived directly from the class `Scrollable`. Inheritance plays a significant role in their scrolling behaviors. Some unnecessary overhead might exist which hinders performance in certain scenarios. Benefits that arise in other cases where a scrolling feature is desired is that the size of the code is minimized, because the feature is provided automatically through inheritance.

There are many other match and mismatch cases found in the two toolkits, but they are all special cases of these. No further examinations of these base classes will be provided in this work.

4.9.4 Constructor parameter mismatches

The constructors found in the two toolkits are similar. They are used to notify the windowing system, at runtime, to allocate resources for client requests. However, they have different forms.

The Swing framework follows an abstract method for specifying names for the constructors for concrete UI components. In the case of a Button class, programmers can specify the constructor without a reference to the containing object, of class `Container`, with the statement "new Button();".

However, the requirement for specifying constructor signatures in SWT is different from the requirement of the Swing framework. Programmers must specify a reference to the containing object of type `Composite` when the constructor is called. If null is used instead of the reference to the containing object, the SWT runtime library will throw an exception to force the application programmer to handle the exception explicitly. In such a case, the constructor's signature for the SWT version is "Button(Composite composite, int style);". The contained objects are instantiated and bound when the constructor is called. The relationship of the containing and contained objects is determined at the time of instantiation.

The Swing framework manages this problem by providing an *add* method for the containing object to get a reference to the contained object. A reference to the contained object is passed as an argument to this add method. The precise location of the add method invocation of this containing object can be anywhere within the source scope where references to both the containing and contained objects are still valid. This created a large challenge for the migration, but has been addressed in an earlier discussion in Section 4.7.4.

4.9.5 Graphic context mismatch

Framework mismatches also happen in the graphic context. The graphic context is Java's terminology for an object that manipulates the hardware display device on which programmers invoke drawing primitives to render shapes like lines or rectangles with different foreground and background colors.

We begin with a short introduction to the graphic context as provided by AWT. There are two methods programmers can use to get a reference to an object in a graphic context. The first is to invoke the `getGraphicContext` method on any visible UI component. This method will return a reference to the component's graphic context object.

The second method is to add a paint listener to a UI component. Programmers need to redefine the *paint* method to have a graphic context as the parameter. The repaint event will be captured by the windowing system and an update request will be delivered to the `paint` method with the associated component's graphic context as the argument to the `paint` method.

In either of these cases, we can only obtain a reference to the graphic context object associated with the UI component. We can not create graphic context objects explicitly.

In SWT, we also found two ways of getting the graphic context object. The paint listener approach is also supported through the overriding of the `paint` method. Another approach is to call the constructor for the graphic context in SWT and pass a reference to a UI component as an argument in this constructor. In this way, not only do we get a reference to a graphic context but we also instantiate a graphic context object explicitly. Hence, SWT has the ability to create the graphic context of objects on demand. Conversely, AWT will only provide a singleton graphic context object for any UI component.

There are small difference between the functionalities provide by the different frameworks. For example, in the AWT toolkit programmers can create a graphic context object which has the origin coordinates relocated to a location on the screen relative to the origin coordinates of the previously obtained graphic context object. In SWT, the ability to redefine origin coordinates does not exist.

Other such subtle distinctions exist. In AWT, programmers can set colors before they invoke any graphic context primitive methods. In SWT, there is even greater control over colors and programmers can even set foreground and background colors separately. Drawing a 3D rectangle within a primitive in SWT is not supported. However, we were able to emulate a 3D effect in our *Virtual Widget Toolkit's* graphic context adapter objects which was an intriguing feature.

4.10 Virtual Swing packages as an Eclipse plug-in

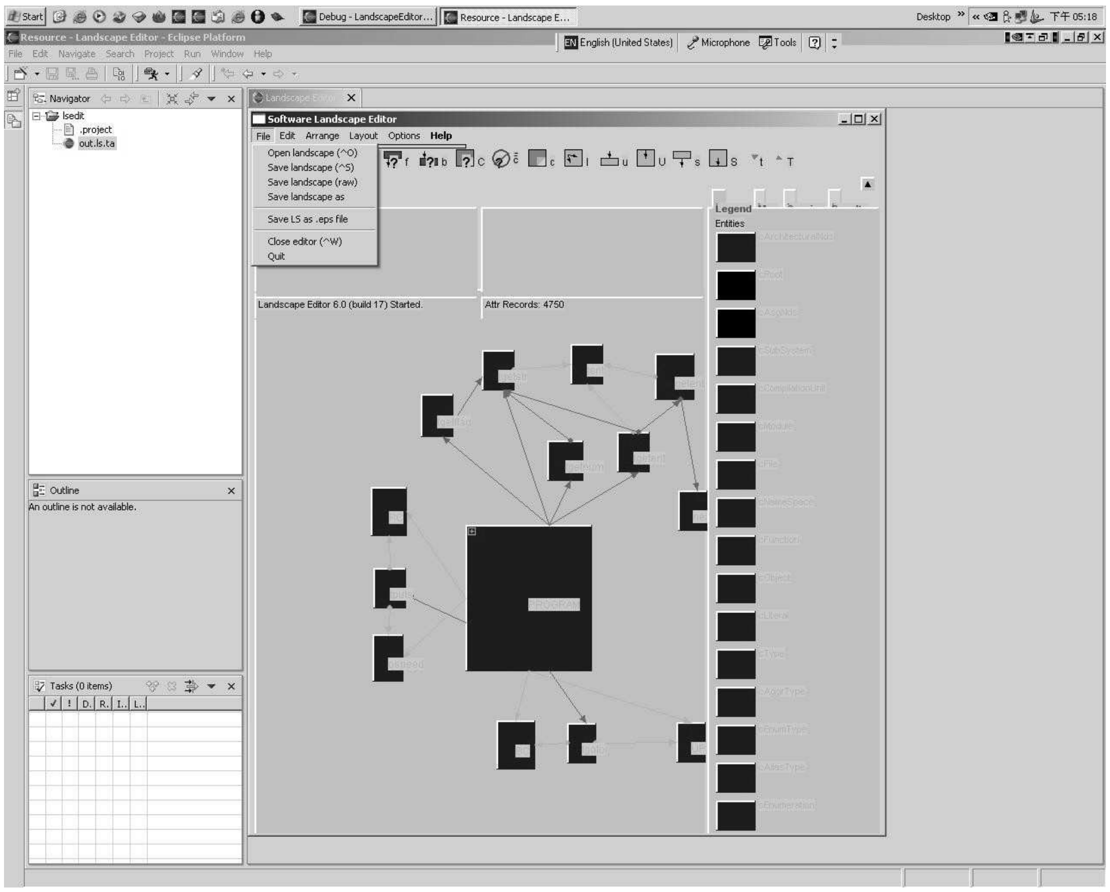
Programmers can choose to import the packages of the Virtual Widget Toolkit and integrate them with their applications within a single Eclipse plug-in project. They can make their applications and the virtual toolkit deployable as a single Eclipse UI plug-in, compressed as a jar file and with an XML included as a deployment descriptor.

However, another alternative exists to package the virtual toolkit as a single plug-in. Users of the Eclipse platform can then download this plug-in and install it manually or use the platform software update mechanism to do this. Thus, programmers can package their applications as a separate plug-ins. The build process of the Virtual Widget Toolkit and the application can be separated, and these processes will consume less time and resources.

4.11 Result

This proposed approach was a success. As can be seen in the Figure 4.6, the same landscape editor is deployed to the Eclipse platform as a plug-in.

The plug-in can recognize *TA* files, which is a format for storing architectural information using the Tuple-Attribute Language. After double-clicking on the TA file, developers can view the architectural components within the Eclipse IDE without launching a stand


Resource - Landscape Editor - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
lsedit
.project
out.ls.ta
Outline
An outline is not available.
Tasks (0 items)
Software Landscape Editor
File Edit Arrange Layout Options Help
Open landscape (^O)
Save landscape (^S)
Save landscape (raw)
Save landscape as
Save LS as .eps file
Close editor (^W)
Quit
Legend
Entities

Landscape Editor 6.0 (build 17) Started.

Attr Records: 4750

Figure 4.6: Landscape Editor Eclipse Plug-in

alone version of the Landscape Editor. The editor pane is tightly integrated with the Eclipse workbench. Our goals for this work have been accomplished.

Chapter 5

Conclusion

In the case study, we looked at the problem of integrations within the Eclipse platform. This platform is open, both in source license and architecture. It uses a plug-in architecture and predefines many extension points such that users can develop or install plug-ins easily. Software engineers and researchers can also derive increased benefit when their tools cooperate on such a unified tool platform.

However, there are problems for developers to integrate existing tools into this platform, as it uses the SWT and JFace toolkits as its foundation. These toolkits are not compatible with the Swing framework, because their widget classes reside in different class hierarchy sub-trees. This causes difficulties when migrating from Swing to another of these frameworks.

We attempted to integrate a landscape editor which was originally built using the Swing framework. The original version of this tool could not be integrated with the Eclipse platform and could not use the workbench's docking or drag-and-drop features unless the Landscape Editor was extensively modified to use the SWT and JFace toolkits. This case study was a practical example of framework migration.

Our first priority was keeping the source unmodified in order that the maintenance cost not increase. We provided a virtual layer design, which is compatible with the original Swing interface, and adapters for each proxy class to use SWT classes.

In our research, we encountered many problems. We exploited the *structural design patterns* to solve these problems and to ease framework migration. We also applied late-binding techniques to solve protocol problems when creating the UI containment hierarchy.

We also deferred method calls and put them in a queue in order to resolve a problem of invoking the getter and setter methods before the actual widget is instantiated. This occurred when we used late-binding, a good example of the constructor signature mismatch when we were performing a framework migration.

Our experiment was successful, as the Landscape Editor has been deployed as an Eclipse plug-in. Integration with the Eclipse platform now simply involves changing the import statements in the application to other packages. Due to this work, these other packages now provide compatible classes and interfaces. Hence, dependency inversion is achieved with ease through the use of this virtual framework for source migration. Furthermore, a maintainable source is attained, or at least the maintainability of the source does not decrease, which was the third goal of source migration as outlined in *The Migration BarBell*[23].

Chapter 6

Future work

In our case study, we have applied the virtual framework approach to perform framework migration from Swing to SWT. However, there are many issues which have not been examined. We mention one of them in the following section as this issue relates closely to our research work.

6.1 Threading issues in framework migration

In our case study, we have not applied the virtual framework technique to threading issues, because the Landscape Editor does not create separate threads. Thread are used to facilitate computation and the Swing framework does not enforce any threading constraints. However, managing threads is critical when we are dealing with framework migrations. An example is given below to illustrate this problem.

To start the event loop in SWT, programmers need to set up a loop and dispatch the events explicitly in the code. SWT also forces some constraint rules when applications access widget resources from other threads which do not create the UI resources themselves.

These combinations of rules cause some difficulties when performing a framework migration. In the original Swing framework, we do not explicitly invoke any methods to start the event loop. In the virtual framework, the loop is started by statements in the `show` method of class Frame and its subclasses.

However, programmers have the freedom to put statements in any order they want. The problem with this is that the statements defined later in the scope, after Frame.show(), will not be executed at all because the show method will execute the event loop infinitely.

We have tried to use SWT utility classes to create a second thread to access the UI resources with little success. Thus, attempting to start the event loop in the new thread is also a failure.

However, creating a thread in the applications using a virtual framework can be achieved if we provide a virtual `Thread` class to wrap the original `Thread` class. In the virtual `Thread` class, we can use the SWT utility classes to create a thread and access the UI resources with it. We have not performed experiments on this issue yet, although we hope to incorporate this feature in later releases of the toolkit.

Bibliography

[1] Mvc meets swing, java world online magazine. http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html, 1998.

[2] eclipse.org - what's new history. http://www.eclipse.org/whatsnewhistory.html, 2003.

[3] An introduction to ta: The tuple-attribute language. http://www.swag.uwaterloo.ca/pbs/ref/lsedit/index.html, 2004.

[4] eclipse.org. http://www.eclipse.org/, 2004.

[5] Java 2 platform, standard edition v1.4.2 overview:. http://java.sun.com/j2se/1.4.2/, 2004.

[6] Java collection framework. http://java.sun.com/j2se/1.4.2/docs/guide/collections/, 2004.

[7] Java foundation classes:cross-platform guis & graphics:. http://java.sun.com/products/jfc/index.html, 2004.

[8] jclassinfo. `http://jclassinfo.sourceforge.net/`, 2004.

[9] netbeans.org. `http://www.netbeans.org/`, 2004.

[10] The software architecture toolkit. `http://www.swag.uwaterloo.ca/swagkit/`, 2004.

[11] Eclipse platform technical overview,object technology international, inc. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`, February 2003.

[12] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

[13] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering*. Prentice Hall, 2000.

[14] John C. Champaign. An empirical study of software packaging stability, appendix b, glossary of terms. pages University of Waterloo Master Thesis 48–49, 2003.

[15] Desmond F. D'Souza and Alan C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[16] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Software, 4(4):60–72*, 1987.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[18] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming, 1(2):22–35*, Feb 1988.

[19] Paul E. Kimball. *The X Toolkit Cookbook*. Prentice-Hall, 1995.

[20] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming, 1(3):26-49*, August/September 1988.

[21] Craig Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unifed Process.* Prentice Hall PTR, 2001.

[22] B. Lientz and E. Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[23] Andrew J. Malton. The software migration barbell. *First ASERC Workshop on Software Architecture*, Aug 2001.

[24] Johannes Martin and Hausi A. Muller. C to java migration experience. In *The Sixth European conference on Software Maintaineance and Regineering (CSMR'02)*, 2002.

[25] Robert C. Martin. *Agile Software Development Principles, Patterns & Practice*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[26] Robert C Martin. The dependency inversion principle. *Engineering Notebook*, May 1996.

[27] Ware Meyers. Interview with wilma osborne. *IEEE Software, 5(3):104–105*, 1988.

[28] I. Kalas S. Kerr K. Kontogiannis H. Muller J. Mylopoulos S. Perelgut M. Stanley P. Finnigan, R. Holt and K. Wong. The software bookshelf. *IBM Systems Journal, Vol. 36, No. 4, pp. 564-593*, November 1997.

[29] Roger S. Pressman. *Software Engineering*. McGraw-Hill, third edition, 1997.

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1992. P C++ 97:2 1.Ex.

[31] Joe Winchester and Steve Northover. Swt: A native widget toolkit for java, part i of ii. *Java Developer Journal, Volume 8 Issue 4, May 2003.*

Appendix A

Glossary of Terms [14]

framework The Gang of Four describe a Framework as a set of cooperating classes that make up a reusable design. The framework dictates the architecture of the application, while the developer provides the functionality. [17]

namespace Scope. Basically defines a section of a codebase that can make calls within itself easily, but anything outside must follow a more formal interaction with it. [30]

package "A (source) package is a basic development unit which can be separately created, maintained, released, tested, and assigned to a team." [15] "The unit of release (e.g. a jar file)." [25].

release A version created for users [13].

repository A library of releases [13].

toolkit According to the Gang of Four, a toolkit is a "set of related and reusable classes designed to provide useful, general-purpose functionality." Rather then providing a design, they simply offer functionality. [17]

version The complete state of all parts of a software system at a certain point in time [29].