SwapBox: a Hot-Swapping Framework for Swappable JavaBeans

By

Lei Tan

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of

Master of Science

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
1125 Colonel Drive
Ottawa, Ontario, Canada
K1S 5B6

September 4, 2001

The undersigned hereby recommend to

the faculty of Graduate Studies and Research

acceptance of the thesis

SwapBox: a Hot Swapping Framework for Swappable JavaBeans

submitted by

Lei Tan

in partial fulfilment of the requirements for the degree of Master of Science

---

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

September 4, 2001

Abstract

Software hot swapping refers to the replacement of a part of a program with a new version at runtime. Increasing demands for on-line software upgrading in safety- and mission-critical systems drive the research. This thesis proposes a new hot swapping infrastructure for hot swapping software applications.

A set of the issues facing hot swapping systems design is derived from state-of-the-art research. A hot swapping prototype for swappable JavaBeans is proposed. The prototype is implemented as SwapBox, which is a running environment and hot swapping management tool for swappable JavaBeans. The SwapBox allows implementation change, incremental and decremental interface change, as well as data structure change between versions. It is designed as an extensible framework so that future research could add new hot swapping strategies into it.

Two sample applications are developed to completely test the SwapBox. They demonstrate that the SwapBox is able to handle complex and diversified hot swapping work. The development of swappable JavaBeans is also simplified because of the existence of the SwapBox.

This thesis provides a new method for on-line software upgrading.

Acknowledgement

I would like to express thanks to my supervisor, Professor Babak Esfandiari, for his guidance and encouragement for this thesis, and many interesting discussions we had. With his help, I greatly enjoyed the beauty of research. I would also like to give my thanks to Professor Bernard Pagurek, whose valuable suggestions and detailed comments help me clarify my ideas. My colleagues at Carleton University, especially Yang Wang, have also been very helpful at providing me with constructive feedback and encouragement throughout my academic pursuits.

The financial support from Communication and Information Technology Ontario, and Carleton University are gratefully appreciated.

My parents deserve special mentions. From the beginning, they have encouraged me to strive for excellence. Without their love, support, and understanding, I would never have a chance to finish this thesis.

Table of Contents

Table of Figures

Chapter 1 Introduction

1.1 Motivations

1.1.1 The Significance of Hot Swapping

Software systems are becoming larger and more complex. However, these systems are neither error-free nor can they satisfy every anticipated need. Software systems have to change over time. Changing business practices, the relentless advance of new technology, and the demands of end users drive this evolution. At the other end of the spectrum, software systems need patches from time to time to fix bugs. Even though the program is perfectly adequate, the environment within which the program is running might change over time. A changing environment may require that the running programs be updated. These changes, without a particular support mechanism, will need shutting down the system, modifying the code, recompiling, re-linking, reloading, and restarting the program. In other words, they may cause downtime.

For a class of safety- and mission-critical software systems, shutting down and restarting the system for upgrades incurs risks, unacceptable delays, and increased cost. Upgrading the software that controls an orbiting spacecraft, for example, cannot be done at all if it means disabling the life-support system. In addition, although not safety-threatening, disabling a bank transaction processing system may have significant economic

consequences, particularly if the companies involved have a reputation for providing a highly available service. In the telecommunications domain, switching systems have a maximum downtime requirement of less than two hours within 40 years!

Obviously, there is a need for maintenance approaches that do not interrupt system operation for long periods. In the literature, there are many approaches [10, 21, 17, 7, 9, 13, and 19] proposed to attack software on-line change problems. They are named as software runtime evolution [17], on-the-fly software replacement [10], dynamic program updating [21], etc. Beyond research projects, a growing class of commercial software applications exhibits similar properties in an effort to provide end-user customizability and extensibility, even though the software application is not safety-intensive or mission-critical. Runtime extension facilities have become readily available in popular operating systems (e.g., dynamic link libraries in UNIX and Microsoft Windows) and component object models (e.g., dynamic object binding services in CORBA [15] and COM [2]). These facilities enable system evolution without recompilation by allowing new components to be located, loaded, and executed during runtime.

Software hot swapping is one such maintenance approach. It refers to the process by which a part (the old module) of a running software application is replaced by a new version of the program (the new module). The replacement takes place at runtime. Implementations of the new and old versions are different, while interfaces and data structures may or may not be the same.

Hot swapping is a subset of runtime software evolution, which includes runtime reconfiguration, dynamically adding/deleting/replacing a component. It has the same meaning with on-the-fly software replacement, and almost the same meaning with dynamic program updating. In the context of this thesis, hot swapping, on-the-fly software replacement, and dynamic program updating are used interchangeably.

1.1.2 S-Module: A Proxy Pattern-Based Solution

Ning Feng [4] and Gang Ao [1] have proposed a software-based hot swapping approach for software applications written in Java. In their approach, a program is composed of swappable and non-swappable modules. The proxy pattern [38] was selected to design swappable modules. A swappable module consists of an S-Proxy and an S-Module. The outside world only has reference to the S-Proxy, which has reference to its S-Module. Any method invocation to the S-Module must go through the corresponding S-Proxy. The reference indirection to the S-Module ensures its ability to be replaced at runtime. A swap manager was proposed to take care of the hot swapping transaction. It is incorporated into the swappable application. The hot swapping can only occur when the old S-Module is in idle state. The new S-Module could have a different interface to the old S-Module. Java reflection is used to invoke the methods provided at the new S-Module but not at the old S-Module. In order to hot swap more than one S-Module in one transaction, a two phase commit transaction model was proposed. It ensures that either all of the participating S-Modules or none of them are swapped out.

The S-Module approach achieves the basic goal of hot swapping. However, there are a few problems that are not yet addressed or not addressed well. For example, the application programmer has to code the S-Proxy. The use of Java reflection to invoke the new methods at the new S-Module suffers a performance penalty. No concrete solution is provided to transfer the state from the old S-Module to the new S-Module at the time of hot swapping.

1.1.3 JavaBean: A Model that could Facilitate Hot Swapping

JavaBean is a software component model of the Java programming language. A Java class could easily be rewritten to be a JavaBean, thus benefiting from the JavaBean component model. These advantages, including easy composition, a well-defined public interface, event communication model, on-the-fly property modification, etc, may facilitate hot swapping not only at the time of on-line change but also in the application development stage.

Software component technology has emerged as an important element in the development of complex software systems. It allows complex applications to be built by composing them from existing applications. There are quite a lot of definitions for the software component. This thesis makes use of the one presented by Szyperski in [28], which is: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". Ideally, a complex system would be able to be built by selecting a set of predefined components

and assembling them together to have the functionality of a new application. Those predefined components are not specific to any particular application. They are functionality-oriented rather than application-oriented. However they provide a means to allow an application assembler adjusting them to fit them into a particular application. Re-use is the most significant yet not the only benefit associated with software component technology. Easy maintenance and evolution is another benefit. By creating a system that is highly componentized, system updates could be localized to a particular component without affecting the rest of the application. Moreover, some component models, such as CORBA [15] and COM [2], provide facilities to allow the dynamic addition new component at runtime. This facilitates the software evolution process.

A formal definition for JavaBean is that: " A Java Bean is a reusable software component that can be manipulated visually in a builder tool"[29]. Builder tools may include web page builders, visual application builders, GUI layout builders, or even server application builders. A JavaBean is not required to inherit from any particular base class or interface. The three most important features of a bean[1] are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires. The JavaBean model specifies standard naming and type signature conventions for methods that a bean uses to expose its properties, events, and methods. Visual builder tools use these naming conventions to analyze a bean and discover what the bean has. This ability enables JavaBeans to be wired up into large and complex applications. More important for hot

[1] In the context of this thesis, the term JavaBean and bean are used interchangeably

swapping, it potentially allows on-the-fly replacement of JavaBeans, with little or no modifications to the existing model.

There are many JavaBeans available for composing complex applications. IBM's alphaworks web site [30], for example, has many beans that could be downloaded and wired into non-trivial applications. JAIN (Java APIs for Integrated Network) is suggested for use with the JavaBean model to provide across-vendor telecommunication services for subscribers [35]. Considering the wide acceptance of the JavaBean component model and its built-in characters with regard to re-use and easy maintenance, it is worthwhile investigating a hot swapping approach for the JavaBean model, to see how JavaBean can facilitate hot swapping. This new idea stems from S-Module approach (e.g., the SwapManager is kept for co-ordinating hot swapping; the change is based on the module level, etc). Moreover, JavaBean's features can be exploited to simplify the development of swappable JavaBean applications and facilitate hot swapping work at the time of upgrade.

1.2 Objectives

The main objectives of this thesis are as follows:

1. Investigate existing runtime software evolution techniques. A set of common design issues that are faced with all hot swapping approaches is abstracted.

2. Propose a new software design/implementation framework for software hot swapping applications, based on the JavaBean component model. The framework must be extensible for future research.
3. Design and develop a hot swapping environment that could be used as a container and hot swapping management of swappable applications.
4. Apply the new technique to develop applications. Completely test and evaluate the new technique.

1.3 Organization

The rest of this thesis is organized as follows:

Chapter 2 illustrates related work in hot swapping. It begins with a hardware-based approach, and elaborates on software-based solutions. At the end of the Chapter, a set of common design issues faced by all hot swapping techniques is abstracted out.

Chapter 3 enumerates related technologies used in the thesis, and outlines the BeanBox, which is a test container for JavaBeans.

Chapter 4 proposes a hot swapping prototype for swappable JavaBeans, and elaborates the design and implementation of SwapBox, which combines the prototype and the BeanBox together to provide a running environment and swap management for swappable JavaBean applications.

Chapter 5 presents two sample applications. One is Conway's game of life, the other is a sorting application, which will completely test the SwapBox.

Chapter 6 concludes the thesis. Contributions, drawbacks and limitations, and future work are also laid out here.

Chapter 2 Related Work

2.1 Hardware-based Solution

The term *hot swapping* has its origin from the hardware domain, where important hardware units are often constructed in redundancy. If a working unit encounters a fatal fault, the backup unit gets into running state to take over the role of the broken one. The faulty unit can then be replaced without the whole system being shut down.

A slight, yet more commonly used, variation of the original hot swapping technique is hardware-based dynamic updates on software programs. In a system that uses hardware-based dynamic updating, an entire running program is dynamically updated on a second computer system on which the new version of the program is loaded, while the first computer continues to execute the older version. Programs can be updated without or with minimal downtime.

Many systems in the telecommunications domain have the capacity to support hardware-based hot swapping. Nokia's DX 200 mobile switching centre (MSC) is an example. An MSC must always be available to service both basic and advanced mobile call requests. To achieve the desired performance, fault tolerance, and availability goals, an MSC is constructed with redundant computer and communication hardware for most of its units. In a dynamic update, the user first updates the backup unit while the working units keep

servicing requests. After the old programs are replaced at the backup unit, the user activates it to working state to take care of servicing requests. The programs at the working unit that handled the requests during the initial update are then replaced.

The most significant advantage of this technique is high reliability. Compared to Internet service, the telephone network enjoys a very high reputation for its reliable services. The principal disadvantage of this approach is its substantial cost. The redundancy inevitably increases the cost. It is typically used in mission-critical systems. In addition to cost, building a redundant computer system and properly connecting it to the main computer system is both difficult and expensive. Not only must the hardware be interacted, but the software shared between the systems (such as databases) must also be kept consistent. Moreover, such an approach, which requires close synchronisation between systems, does not scale to distributed systems.

2.2 Software-Based Solution

A software-based solution for runtime replacement is the next research step, because it is not as expensive as the hardware-based solution. Normally software-based solutions do not require redundant hardware units (although some may need specific hardware support). This reduces the cost of building highly reliable systems. Software-based solutions tend to be more diversified because the application domain, the desired performance, and correctness guarantees (i.e., how correctly the program is running at the time of hot swapping) influence the techniques a system uses for hot swapping. For

example, a system for hot swapping an information server used in a time-sharing environment would generally not be appropriate for hot swapping a real-time process-control program. Another example is transferring states between versions. Some systems do not allow state loss and force the hot swapping to occur only after the old version of the program has reached the idle state; others may tolerate state loss. Such systems typically detect state loss and switch to a degraded mode of operation while recovering.

2.2.1 General Characteristics

Even though there are various hot swapping techniques, there are several characteristics all hot swapping systems should possess, regardless of their intended use. Segal and Frieder gave a list of such characteristics in [21]. They are:

- Preserve program correctness. Program correctness must be preserved during the update as well as at times when no updates are in progress.
- Minimizing human intervention. Part of preserving program correctness during an update means ensuring that the updating components are applied in the correct order and at the right time. Even a meticulous person can perform an update improperly.
- Support low-level program changes. To dynamically update a range of programs, an updating system must support a variety of low-level program changes. The simplest kind of change is to replace a module with a new one that is implemented differently. More complicated changes include changing the module's interface, having the module retain state between invocations, changing

the state's implementation, and changing the implementations of both the interface and state variables.

- Support code restructuring. Significant code restructuring can occur during maintenance – a change beyond simple module replacement.
- Update distributed programs. Many programs that benefit from dynamic updating are distributed by nature.
- Do not require special-purpose hardware.
- Do not constrain the language and environment. They must be free to choose a language and system environment. An updating system must not force programmers to write code or call operating system primitives in a radically different manner.

It is worthy to note that the relative importance of these characteristics varies with the application domain. For example, almost all the runtime updating and hot swapping systems described in the following sections place emphasis on preserving program correctness. Some of them (e.g., [1, 4, and 26]) take a conservative approach, i.e., avoid beginning replacement when the old version is still busy. Since the program is idle at the time of change, it is easier to preserve program correctness than when the program is busy. Only a few systems made explicit attempts to minimize human intervention, though this does not necessarily imply high level human intervention during hot swapping. All systems support implementation replacement. Some of them support interface or data implementation change [1, 4, 10, and 14]. Even though supporting distributed program updating is a wide research area, it is not a must for all hot swapping systems. Most

research papers did not mention hardware, therefore they do not need special-purpose hardware support (one exception is DAS [8]). Language and environment support are occasionally needed, e.g., operating system call Thread::abort in Chorus is used to signal the beginning of a runtime software replacement in Hauptman and Wasel's approach [10].

All the techniques used in hot swapping systems aim to make the system as transparent as possible to both its users and programmers and its execution environment. The more transparent a hot swapping system, the more likely it would be used. Unfortunately, due to the inherent difficulty of runtime software replacement and the variety of application program structures as well as diversified domain requirements, no sole hot swapping system is able to incorporate all the possible changes into any program structures. Segal and Frider [21] regarded this as "why those researching dynamic updating have concentrated on creating dynamic updating techniques for specific well-accepted and well-understood program structures".

2.2.2 Dynamic Linking

Dynamic linking [5 and 11] allows names to be bound when the program begins execution. It lets a program link the reference to an external procedure (usually part of the operating system or a library) to the actual procedure when the program is run or when the external procedure is first referenced during the run. This ensures that the most up-to-date external procedure is bound and executed. Once done, this binding cannot be changed without restarting the program. The common point among all traditional states

of binding is that any type or method name can only be bound once across all phases. Even if dynamic linking were possible on a per-invocation basis (i.e., each time the reference is encountered within the program, it is resolved to bind to the external procedure), it does not contain a mechanism to preserve program correctness. Dynamic linking is widely used in commercial software products such as UNIX and Microsoft's Windows operating system.

2.2.3 Dynamic Class

2.2.3.1 Language Support Dynamic Class

CLOS (Common Lisp Object System) [24] and Smalltalk [6] support dynamic typing, in which the type descriptor of an object is able to change freely at runtime. Method code may be modified. Data fields and methods may be added or removed, etc. For example, the Information Bus distributed systems architecture [16] uses a CLOS-derived language to implement dynamic classes. Fabry [3] implemented a dynamic type system using capabilities. Widening [22] provides a mechanism for constrained dynamic type changes, in which objects may be temporarily "widened" to a subtype of their defining class. Dynamic typing, in its unconstrained form, supports the greatest flexibility. It supports hot swapping with respect to modifying interface, implementation, and data structure. However, static type checking of any kind becomes unfeasible. The runtime system must therefore support complete runtime type checking, with all the associated overhead.

2.2.3.2 DVM: Virtual Machine Support Dynamic Class

Malabara et al. [14] presents a modified Java Virtual Machine – DVM (Dynamic Class-based Virtual Machine) for dynamically updating running Java classes. DVM has a class named *DynamicClassLoader* derived from *ClassLoader*. *DynamicClassLoader* has two methods (i.e., reloadClass and replaceClass) that can reload an active class and replace it with a new version. The new class does not have to be the subclass of the old class. However, the dynamic change has to be type safe. A valid (type-safe) dynamic change to the old class *C* has to fulfil two conditions. Firstly, no class defined within the application depends on the fields or methods being removed from *C*. Secondly, an element of *C*'s type set (which is the set of all classes and interfaces to which an instance of *C* can be cast) cannot be removed if other class depends on it. The update is based on class, i.e., all instances of the old class have to be updated to instances of the new class. In order to achieve a type-safe and class-based dynamic update, DVM addresses two issues. One is to update instances; the other is to update dependent classes.

Many JVM implementations, including JDK 1.2, divide the Java heap into a handle pool and an object pool. Java objects are always addressed indirectly through their handles. The DVM uses this address indirection to handle instances updates. It allocates new space for an object when updating it, without changing the handle used to reference the object. When the old object is moved, only the pointer in its corresponding handler needs to be updated; the handle never moves. The DVM uses an incremental mark-and-sweep approach to update instances. During the mark phase, objects are identified. They are actually updated in the sweep phase. The mark phase is atomic, and the sweep phase

proceeds incrementally. For updating dependent classes, the DVM first identifies such classes by scanning the constant pool of all loaded classes for *C*. Then it updates each one according to its relation to *C* (e.g., subclass, method usage, etc). The constant pool of a loaded class contains symbolic references to other class objects and their methods and fields. The JVM resolves such symbolic references when they are invoked the first time. It replaces these references with pointers to the referenced object. When class *C* is updated to a new class *C'*, any pointers to *C* become invalid, and the DVM replaces all resolved references to *C* with original symbolic references. It then resolves the class and replaces the references with pointers to *C'*. This approach uses a simplified mechanism to handle state transferring between versions. The DVM gets the values of the old states that exist at the new class, and sets them to the new instance. It initialises all added fields at the new objects to be NULL.

DVM allows class updates even though some of its methods are active. The active methods, however, cannot be modified because it is difficult to map the stopping point at the old version to the restarting point at the new version if the method implementations are different. DVM enables changes to the implementation, data structure, and interface (provided it is type safe). The principal disadvantage of DVM is the modification to the JVM. Moreover, the efforts to achieve type safety and avoid race conditions in multi-threaded applications tend to complicate the design of DVM. Native method execution is another problem. The proper class updates makes use of the assumption that all native methods can be trusted to behave properly.

2.2.4 DAS: An Example with Hardware Support

DAS (Dynamically Alterable System) [8] provides support for dynamic updating of application programs by letting a module be replaced with a new module that has the same interface. It performs dynamic updating using “replugging”, a mechanism built on DAS’s address-space management system. This is, in turn, built on the addressing hardware of DEC’s PDP 11/40E. The information used to keep track of each module is stored in a linked list of descriptors called a descriptor chain. Replugging is done by changing the links within the descriptor chain. When a procedure in a different module is called, DAS performs an address-space transition by placing the descriptor table of the new module into the address-mapping hardware, while saving the current entries on a stack. Only one code segment is kept in the process’ virtual address space. One significant problem with DAS is its use of virtual memory, which is both overly complex and inefficient, to aid updating. Another problem is that DAS fails to provide mechanisms for procedures whose interfaces change between versions. DAS is an early example of software dynamic updating systems, developed in the late 1970s (the software-based dynamic updating system emerged in the 1970s). It requires hardware support. Most of today’s dynamic updating systems have no such requirements.

2.2.5 Solutions with Special Support from the Operating System

All hot swapping systems must have support from the underlying operating system to support dynamic loading, dynamic linking, and dynamic deleting of parts of executable programs during the process of on-line upgrading [1]. Some solutions go even further.

They need special support from the operating system to facilitate runtime replacement for particular domains.

2.2.5.1 An Approach Based on Chorus

Hauptman and Wasel [10] proposed an approach to support on-the-fly software replacement. The approach is based on Chorus, a modern, distributed, multi-threading, real-time operating system. C++ is the programming language. Their approach simplifies the task of replacing a group of actors into a sequential replacement of one actor after another. An actor is similar to a heavyweight process. Actors have ports attached to them. An actor can receive messages only via its ports. In the Chorus system, ports can be migrated between actors with virtually no degradation of their functionality. Each replaceable actor is given an additional thread (exchange thread), which organizes the replacement procedure within the actor, as well as an additional management port, which is used for all replacement specific communication.

The replacement cannot occur at arbitrary points of the old actor. Instead there are exchange points at which all threads can be blocked (due to mutexes waiting or in an idle state, etc). It is demonstrated that such points always exist if the CPU load is less than 100% and only one processor is used. If the execution time between two exchange points is too long, artificial exchange points could be inserted. The new actor could be different from the old one in code and object structure, but it has to provide one-to-one restarting points corresponding to exchange points on the old actor. The application programmer has to declare such restarting points and, in case of data implementation changes, one or more state transformation functions. Additional code with *goto* clause must be inserted

into the application code to navigate the program to the restarting point. The code could be automatically included with the help of a pre-processing tool. Each object has to provide two methods to access its states (i.e., one to get and one to set). They are used to transfer the old actor's state to the new actor.

Upon getting a replacement request and once an exchange point is reached (i.e., all threads are in block state), the exchange thread calls a Chorus operating system call *Thread::abort*, which unblocks the blocked thread, and the return value signals the beginning of a replacement. Those unblocked threads collect their objects' states via the state access method. The new actor then jumps to the restarting point by executing replacement related code, and rebuilds the thread state list. The last step of replacement is port migration. Ports are known entry points of external clients. They are guaranteed to stay alive during the replacement process and should be migrated from the old component to the new one. The port migration is easy to implement because the underlying operating system, Chorus, supports it. After migrating ports, the replacement finishes and all new communication is redirected to the new actor.

2.2.5.2 Port-based Object Solution on Chimera

Steward et al. [27] presented a software framework using a port-based object (PBO) to design dynamically reconfigurable real-time software. The main goal is to reconfigure a real-time software program associated with a robotic system when such a system is dynamically reconfigured. It is supported by an implementation using domain-specific communication mechanisms and templates that have been incorporated into the Chimera

Real-time Operating System [26]. The term *object* does not imply *object-oriented design*; rather it refers to *object-based design*. A PBO is an independent concurrent process, whose functionality is defined by the methods of a standard object (which is predefined, therefore not changed at runtime). Communication with other modules is restricted to its input ports and output ports. There is no explicit synchronisation with other processes; when a PBO needs information, it obtains the most recent data available from its input ports, and there is no knowledge as to the origin of the information. When a process generates new information that might be needed by other processes, it sends this information to its output ports, and there is no knowledge as to which processes might look at this information. A variable type mechanism is used so that data transmitted over the ports can be any type. The approach implements input and output ports as state variables stored in global and local tables. A PBO can only access the local table, where only the subset of data from the global table that is needed by that PBO is kept. Whenever a PBO produces output states, the global table is updated.

Because PBOs implement the same set of methods predefined by a standard object and they communicate each other via global and local state tables without direct or indirect references, this approach supports on-line reconfiguration. An engineer can reconfigure a running real-time robotic control program, e.g., replace a PBO with a new one, add a new PBO, and so on, provided the new configuration is a valid one. In order to achieve stable execution during a reconfiguration, the robot is temporarily at rest (i.e., velocity and acceleration are both zero before dynamic reconfiguration begins). Therefore no state transfer is needed. One problem with this approach is that it relies on operating system

services (e.g., a PBO framework process, multiprocessor state variable communication mechanism) to achieve dynamic reconfiguration. This restricts the approach from being applied to other domains. Moreover, this approach does not allow interface change.

2.2.6 Java-C2: an Architecture-Based Solution

Oreizy et al. [17] proposed an architecture-based approach to runtime software evolution. Software architectures [23, 31] are used to provide a foundation to facilitate such evolution. Each application is composed of components and connectors. Components are responsible for implementing application behaviour. A component must provide a minimal amount of functional behaviour to participate in runtime change. To support runtime addition and removal, components must be packaged in a form that the underlying runtime environment can load dynamically. To support runtime reconfiguration, components must be able to alter their connector bindings. These additional behaviours are provided in the form of reusable code libraries that act as a wrapper to the actual component. The introduction of connectors is a distinctive feature of software architectures. They are explicit architectural entities that bind components together and act as mediators between them. They encapsulate component interactions and localize decisions regarding communication policy and mechanism.

This approach focuses on supporting architectures in a layered, event-based architectural style, called C2. In the C2-style, all communication among components occurs via connections, thus minimizing component interdependencies and strictly separating computation from communication. The C2-style also imposes topological constraints -

every component has a “top” and a “bottom” side, with a single communication port on each side. This restriction simplifies the task of adding, removing, or reconnecting a component. A C2 connector also has a top and bottom, but the number of communication ports is determined by the components attached to it. This enables C2 connectors to accommodate runtime rebinding.

A tool suite named Archstudio has been developed. Archstudio provides graphical and command-line tools used to modify a Java-C2 program specification at runtime. An attempt to change the specification invokes an Architecture Evolution Manager, which checks the request for validity, and modifies the program implementation accordingly. The runtime architecture infrastructure supports the addition and removal of components and connectors, and the reconfiguration and querying of the architectural model. There is no support for component replacement. This approach is an example of runtime software evolution. It concentrates more on high-level software change management than the fine-grained details. The main limitation of this prototype is that all components and connectors have to be written using the Java-C2 class framework.

2.2.7 S-Module in Detail

In our lab, Ning Feng [4] and Gang Ao [1] developed the S-Module approach for the hot swapping problem. Since this thesis attempts to improve the S-Module approach, it is worthwhile to describe the S-Module in detail.

2.2.7.1 Basic Idea

Ning Feng and Gang Ao made use of proxy pattern [38] to design hot-swappable applications, which consist of swappable and non-swappable modules. Each swappable module is further composed of an S-Proxy and an S-Module. The S-Module is swappable and the S-Proxy cannot be swapped out. Figure 2-1 shows how the S-Proxy hides the S-Module from its clients.

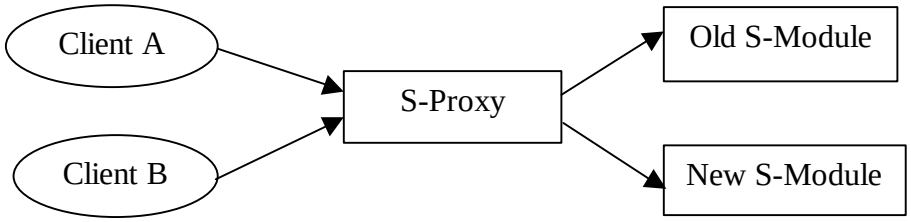

FIGURE 2-1 PROXY APPROACH FOR REFERENCE INDIRECTION

The S-Proxy hides the real handle of the S-Module from clients while these clients only get the handle of the S-Proxy. When an S-Module is swapped, only the S-Proxy switches the handle to the new S-Module while the clients retain their relationship with the S-Proxy. The new S-Module should inherit from the old S-Module. In other words, it provides all methods implemented by the old S-Module.

Besides swappable and non-swappable modules, an S-Application also contains one swap manager. The swap manager has access to all S-Modules. It co-ordinates the hot swapping transaction. Upon getting a hot swapping request, the swap manager may determine whether or not it is possible to begin hot swapping. It makes the decision based on what state the S-Module is in. The swap manager is also responsible for mapping attributes from the old S-Module to the new S-Module and blocking the new method invocations to the S-Module that is going to be swapped out.

2.2.7.2 S-Module States

The S-Module has five states, i.e., initialising state, swappable state, busy state, blocked state and swapping. The initialising state is the one during which S-Module is loaded into the program. In the busy state, one or more methods of an S-Module are called by other modules in the program. In the blocked state, the S-Module itself is calling other methods of the program, or using the system resources, and waiting for the completion of the operations. The hot swapping transaction will take place if and only if an S-Module is in its swappable state. When an S-Module is in either the busy or the blocked state, and receives a request to be swapped, the S-Module is forced to enter the swapping state. In the swapping state the S-Module should either continue its normal operations or be asked to terminate its services in a safe way and release the system resources it holds. After that, the S-Module enters the swappable state.

2.2.7.3 Handle Interface Change

The service interface of an S-Proxy is the same as the old S-Module that is originally represented. If the new S-Module changes its interface, the S-Proxy cannot change accordingly because the S-Proxy is not swappable. If the new S-Module has a newMethod interface that does not exist in the old S-Module as well as in its S-Proxy, this newMethod cannot be invoked via S-Proxy's service interface. To address this problem, Java reflection is used. An S-Proxy provides a newService interface to adapt to the case when a new S-Module has a new method.

In Figure 2-2, a new client knows the newMethod of the new S-Module can call the newService interface of the S-Proxy with the parameter of the new method's name and parameters. The S-Proxy can then invoke the newMethod of the new S-Module by Java reflection. The dash line represents method invocation for the new S-Module.

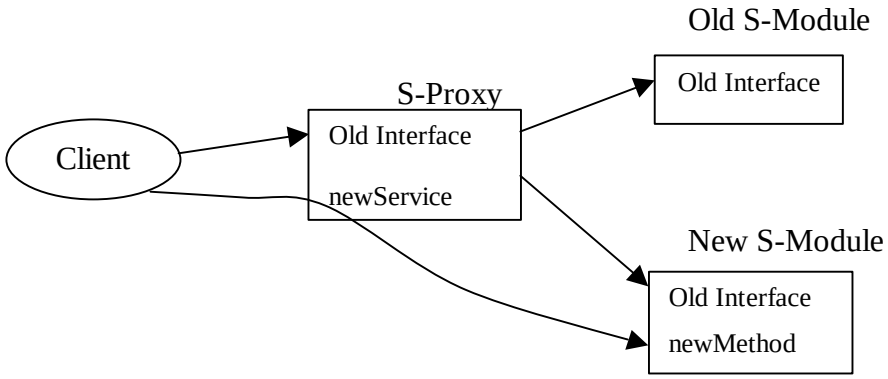

FIGURE 2-2 NEW SERVICE INTERFACE FOR S-PROXY

2.2.7.4 Two-Phase Commit Transaction

In order to support atomic hot swapping for multiple S-Modules in one transaction, a two-phase commit transaction model is proposed. Each participant (S-Module) in such a transaction has three kinds of tasks to do, i.e., prepare-task, commit-task, and abort-task. The prepare-task moves the old S-Module from the busy state to the idle state with the help of a swap manager. The S-Module then transfers its state to its corresponding new S-Module. The successful completion of this task leads the task to be marked as PREPARED. In the commit-task every participating old S-Module is removed from the application and every new S-Module is ready to provide the application services. If the task fails, it will be marked as ABORTED. In abort-task, every participating old S-

Module is ready to continue its application services, and all the new S-Modules are removed.

In a sequential transaction model, every participant in the swap transaction will line up to process its swap transaction. One participant will do its prepare-task first. If the result is PREPARED, then the next participant will do the same job. If every participant is PREPARED, then every participant will do its commit-task on by one. If any one is ABORTED, then every participant has to do its abort-task.

2.2.7.5 Problems

One problem with this approach is that it can only support incremental functional modification and extension. The technique used to handle interface change cannot support decremental functional modification. This means that a new S-Module has to keep all the interfaces its corresponding old S-Module has provided. Otherwise it will violate Java's type system. Another problem is the use of Java reflection in invoking the new method. It has to pay a performance penalty. In the worst case, if the evolution of the S-module leads to all methods being changed at the end, the whole application performance will suffer a great deal, because all method invocations have to go through reflection. Moreover, the approach does not explicitly provide a mechanism to transfer states between versions.

2.2.8 Others

All the above approaches treat hot swapping, runtime evolution and reconfiguration at component level (i.e., S-Module, Component and PBO). Gupta et al [9]. describes an approach to modelling changes at the statement- and procedure-level for a simple theoretical imperative programming language. The basic idea of the technique is to locate the program control points at which all variables affected by a change are guaranteed to be redefined before use. By redefining, a variable gets a new value in the new version program. It was demonstrated that in such points on-line change would be safe. At these points, old version program's running stack is captured and used to construct new version program's running stack. They show that in the general case locating all such control points is undecidable, and approximate techniques based on source code data-flow analysis and developer knowledge are required. A very short piece of program is used to demonstrate the idea. However, scaling up this approach to manage change in large systems written in complex programming languages is still an open research problem.

2.3 Summary

2.3.1 Common Design Issues in Designing Hot Swapping Systems

The approaches listed above exhibit diversified techniques used to address runtime software replacement problem. There is, however, a set of common design issues faced by the developer when designing hot swapping systems. The strategies adopted to

address these issues determine the nature of the hot swapping system. These common issues are:

- Granularity, which is the basic unit for hot swapping. The granularity for the DVM, for example, is based on the Java class.
- Reference indirection. A hot swappable module must not be directly referenced by the other parts of the program. S-Module [4, 1] uses proxy patterns to separate a hot swappable object from the outside world. Steward's approach [27] exploits ports, implemented as global and local variable tables, to hide the swappable objects.
- Constructing new states. Constructing states in the new version program is a challenging task. The simplest solution is that the new version program has default values that do not need state values at the old version program. In cases where new states are constructed based in the state values at the old version program, there is a need to transfer old states to new states. DVM [14] is an approach that combines both solutions. At the time of change, it copies the values of old states that existed at the new class to instances of the new class, and set newly added fields to NULL.
- Levels of change. There are three levels of changes allowed for the new version program, i.e., implementation change, interface change, and data structure change. DAS [8] only supports implementation change. The S-module [4, 1] supports both implementation and some interface changes (i.e., incremental interface change). While not explicitly mentioned, it also supports data structure change.

- Timing for upgrading. Hot swapping cannot take place at arbitrary time. The hot swapping system must determine in what circumstances the upgrading can occur. In DVM [14], it could occur at any time provided the implementation of the currently active method is not changed. In S-Module [4, 1], hot swapping can only take place when the old module is in idle state and agrees to be swapped out. In Hauptman's approach [10], the stopping points at the old actor are hard coded into application code. Only when the stopping points are reached and a replacement job is pending could the runtime replacement take place.

2.3.2 Generic Procedure for Hot Swapping

At the time of on-line upgrading, there are several steps that all hot swapping systems will go through to replace the old version of the program with the new one. Highlighting these steps facilitates the understanding of the basic functionality of hot swapping systems. These steps are:

- S1: Determine safe points in the old version of the program to begin a runtime update
- S2: Reconstruct runtime states in the new version program
- S3: Determine appropriate points in the new version of the program to restart the updated program
- S4: If the hot swapping system allows interface change and there is an interface change, handle the interface mismatch between versions

- S5: Bring the new version of the program to the application so that all forthcoming service requests destined for the old version of the program will be redirected to the new version of the program

This chapter reviewed the related work in the hot swapping research. It could be seen that because of the inherent difficulty of the problem, all of approaches have concentrated on creating techniques for specific program structures.

The software component is highly decoupled with each other because it is designed with contractually specified interfaces and explicit context dependencies. We believe this character, plus its abilities to be deployed independently and be subject to composition by third parties, can facilitate and simplify hot swapping at the component level. For example, the JavaBean's ability of dynamically adding/removing interactions with other beans can be used to handle both incremental and decremental interface changes of the new bean.

As described in section 1.2, the fundamental objective of this thesis is to design a new hot swapping infrastructure with particular focus on exploiting JavaBean's features to simplify development of hot swappable beans and facilitate hot swapping transactions as well. Instead of being incorporated into hot swappable applications, the new infrastructure should be a running environment and a swapping management tool for hot swappable beans. In other words, a test container for hot swappable JavaBeans is needed.

BeanBox [36] was developed by Sun Microsystems as a test container for JavaBeans. It can be extended to implement the new proposed hot swapping infrastructure.

The next chapter briefly describes related technologies such as JavaBean component model, Java object serialization, XML, and the BeanBox. All of them will be used to develop the SwapBox, which extends from the BeanBox and works as a test container for hot swappable JavaBeans.

Chapter 3 Related Technologies and the BeanBox

3.1 Related Technologies

3.1.1 JavaBeans Component Model

3.1.1.1 The Goal and Features

The JavaBeans specification [30] describes a component architecture for Java. It addresses the needs of two sets of programmers:

- Component developers who write code at the source level, and
- Component assemblers who create large applications by combining beans, either visually or by writing some glue code (or both)

The goal of JavaBeans APIs is to define a software component model for Java, so that third-party developers can create and ship Java components that can be composed together into applications by end users. The JavaBeans component model is based on a Java class. Almost any class written in Java can be made into a bean. The model simply adds a few rules that a programmer must follow to make classes toolable and reusable. This section is not trying to cover all these rules in detail. Instead, it focuses on some rules that are heavily used in the SwapBox to facilitate hot swapping.

Individual JavaBeans vary in the functionality they support. Some beans may be simple GUI elements such as buttons and sliders; others may be sophisticated visual software components such as database viewers. Some JavaBeans may have no GUI appearance of their own, but may still be put together visually using an application builder. Though the variety of functional behaviours, JavaBeans distinguish themselves from normal Java classes because they support the following features:

- Support for “introspection” so that a builder tool can analyze how a bean works
- Support for “customization” so that when using an application builder a user can customize the appearance and behaviour of a bean
- Support for “events” as a simple communication metaphor that can be used to connect beans
- Support for “properties”, both for customization and for programmatic use
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved and rebuilt later

3.1.1.2 Event Communication: an In-Depth Look

Among these features, the event communication model plays an important role in the SwapBox. Events provide a mechanism for allowing components to be plugged together in an application builder, by allowing some components to act as sources for event notifications that can then be caught and processed either by scripting environments or by other components. In the Java event model, an event notification is propagated from the source object to the target listener object by a direct Java method invocation on the target object. A source object may fire out several events to target objects. The state associated with an event notification is normally encapsulated in an event object that inherits from

java.util.EventObject and which is passed as the sole argument to the event method at the target listener object. Each distinct kind of event is targeted at a distinct event method. Therefore a particular event notification is defined by its event method, which is supposed to take the event and handle it. These methods are then grouped in interfaces that inherit from java.util.EventListener. Event listener classes identify themselves as being interested in a particular set of events by implementing some set of EventListener interfaces.

Event sources identify themselves as a source of particular events by defining registration methods that conform to a specific design pattern (not the design pattern in OO methodology, but rather a naming convention) and accept references to instances of particular EventListener interface. The standard design pattern for EventListener registration is:

public void add<ListenerType> (<ListenerType> listener);

public void remove<ListenerType> (<ListenerType> listener);

The presence of this pattern identifies the implementation as a standard event source for the listener interface specified by *ListenerType.*

In circumstances where listeners cannot directly implement a particular interface, or where some additional behaviour is required, an instance of an event adapter class could be interposed between a source and one or more target listeners in order to establish the relationship or to augment behaviour. The primary role of event adapters is to conform to

particular EventListener interfaces expected by the event source, and/or to decouple the incoming event notifications on the interface from the actual listeners.

3.1.1.3 JavaBean Introspection

Visual builder tools use introspection to discover a bean's behaviour, i.e., methods provided, events fired out, and properties exposed. JavaBean introspection uses the *Class* class and its helper classes to the *java.lang.reflect* package to discover and use bean information. In other words, the bean introspection is above Java reflection. The *Class* class is the mother of all reflection classes. Every instance of a Java class has a *Class* object associated with it. This object contains all the information about the Java class, e.g., its name, superclass, loader, the interface it supports, etc. The bean introspection abstracts it into bean properties, methods, and events.

3.1.2 Java Serialization

JavaBeans take advantage of the Java object serialization service to automatically save and restore their states. The serialization service also supports a simple form of versioning, which is an important requirement in component environments. It enforces simple rules that let newer beans load state written by their older versions. It also lets newer beans store states in a format that is consistent with older versions. However, Java serialization has stringent rules on what changes it supports between versions.

Java object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for Java developers to handle object serialization. The essential foundation of Java Serialization is that with methods *ObjectOutputStream.writeObject()* and *ObjectInputStream.readObject()*, a serializable object could be stored into a file or be sent across the network.

An object must implement *java.io.Serializable* or *java.io.Externalizable* interface in order to be serialized. *Serializable* interface is no more than a marker indicating whether or not an object could be serialized and de-serialized. There is no method defined within *java.io.Serializable* interface. When serialization occurs, only the object's state is saved. The object's class file and methods are not saved. Thus the class file must be accessible from the system in which the restoration occurs. Most Java classes and their subclasses can implement *java.io.Serializable* interface. There are, however, certain system-level classes such as *Thread*, *OutputStream* and its subclasses, and *Socket* that are not serializable. If a serializable class contains an instance of these classes, they must be marked as *transient* so that they will not be serialized. Indeed, *transient* is used to mark any field that either cannot be serialized or is not intended to be serialized.

When Java objects use serialization to save states in files, or as blocs in databases, there is a possibility that the version of a class reading the data may be different from the version that wrote the data. Versioning raises some fundamental questions about the identity of a class, including what constitutes a compatible change. A compatible change

is a change that does not affect the contract between the class and its callers. Java Language Specification defines compatible changes as follows:

- Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type.
- Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes.
- Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted.
- Adding writeObject/readObject methods
- Removing writeObject/readObject methods
- Adding java.io.Serializable
- Removing java.io.Serializable so that it is no longer Serializable
- Changing the access to a field - The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to non-static or transient to non-transient

When writing a serialized object, the Serialization API simultaneously writes a 64-bit safe hash code (called *SerialVersionUID*) of the following information about the class:

- The class name
- The class modifiers

- A sorted list of interface names implemented by the class
- The name, modifiers, and descriptor of each field, sorted by field name (except for private static and private transient fields)
- The name, modifiers, and signature of each method, sorted by method name (except for private methods and constructors)

When restoration occurs, the class loader of the system loads the class file of the class (if necessary), and then calculates the class's *SerialVersionUID*. If the calculated value (for the local class) doesn't exactly match the value stored previously by the output stream, the Java serialization process throws *java.io.InvalidClassException*, thereby refusing to load the stream. However, before calculating the *SerialVersionUID*, the Java serialization process checks the class it is serializing for a variable:

static final long serialVersionUID

If it finds the variable, that number, instead of the calculated value, is used to make the comparison. Therefore, if in the new version the variable is set to the *SerialVersionUID* of a previous class, the Java serialization process will get the number it expects. In this way, no *java.io.InvalidClassException* would be thrown out. However, some changes made to a Java class may cause other exceptions that make restoration fail, even though *InvalidClassException* is not thrown out. These changes are called incompatible changes.

3.1.3 XML

At the time of upgrading, it is necessary to have a configuration file that is used to regulate how to do a particular hot swapping. Such a configuration file must be well

structured and easy to extend. We decided the configuration file should follow the XML format because it has a good structure and is easy to extend. XML stands for Extensible Markup Language. It is a text-based markup language for data interchanging. Similar to HTML, XML identifies data using tags. But unlike HTML, XML tags signify what the data means, rather than how to display it. HTML is a markup language; XML is more than that. It is a metalanguage – a language used to define new markup languages. An XML document must be well formed so that the XML parser can correctly read all tags. A well formed XML document is simply one that follows all of the notational and structural rules of XML, e.g., no unclosed tags (every start tag must have a corresponding end tag) and no overlapping tags (a tag that opens inside another tag must close before the containing tag closes).

DTD (Data Type Definition) is used to verify the validity of an XML document. It is like a grammar for a markup language. The DTD specifies what elements may exist, what attributes the elements may have, what elements may or must be found inside other elements, and in what order. XML parsers can be divided into two types with respect to document validity checking. A non-validating parser reads the XML document and, if it is well formed, presents the document structure as a tree of objects. A validating parser does not simply read an XML document and verify that it is well formed, but goes a step further to determine whether the document element tags are legal, whether the attribute names make sense, whether every element nested inside another element belongs there, and so on.

3.2 BeanBox: the Starting Point for SwapBox

3.2.1 BeanBox Overview

The main goal of the BeanBox is to present users with an environment that could test whether or not beans can work properly. The BeanBox allows users to:

- Drop beans into a composition window
- Resize and move beans around
- Edit the exported properties of a bean
- Run a customizer to configure a bean
- Connect a bean event source to an event handler method
- Connect together bound properties of different beans
- Save and restore sets of beans
- Make applets from beans
- Get an introspection report on beans
- Add new beans from JAR files

The BeanBox is a standalone application. Figure 3-1 is a snapshot of the BeanBox. The BeanBox is mainly composed of three parts. From left to right in Figure 3-1, the ToolBox, BeanBox, and Properties sheet are shown. The ToolBox and Properties sheet provide supporting facilities to BeanBox, which plays a significant role in the whole framework. ToolBox instantiates available JavaBeans from JAR files located in a predefined directory. It lists the names of all beans found in a panel so that they can be

dragged and dropped into the BeanBox. The Properties sheet handles the modification of properties.

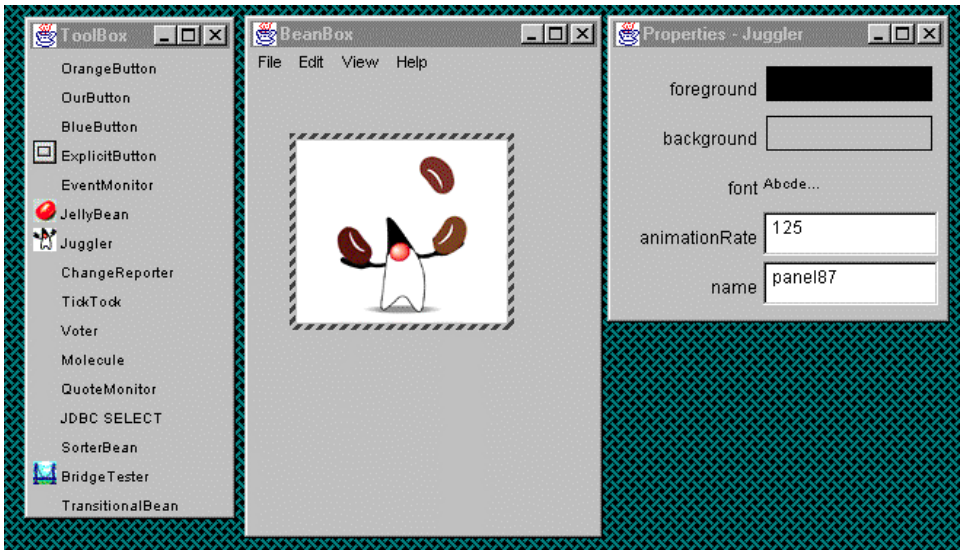

FIGURE 3-1 SNAPSHOT OF THE BEANBOX

3.2.2 Connecting Beans: Behind the Scene

After being dragged from the ToolBox and dropped into the BeanBox, a bean can be wired up by connecting events and properties to compose applications. Applications created in the BeanBox are event driven. Interactions between beans are carried out via event delivery. A bean has no direct reference to beans with which it interacts. An adapter is interposed into a source bean and the target bean. The source bean only knows the reference to the event adapter, while the event adapter knows the reference to the target bean. When the source bean attempts to fire out an event to the target bean, it actually sends the event to an event adapter, which then invokes a corresponding method at the target bean. The BeanBox automatically generates, compiles and loads the adapter class on the fly. Because the "glue code" (event adapter) is created dynamically, this

approach is rather flexible. This flexibility defers the establishment of interactions between beans until the last minute, and offers the potential to replace a bean at runtime.

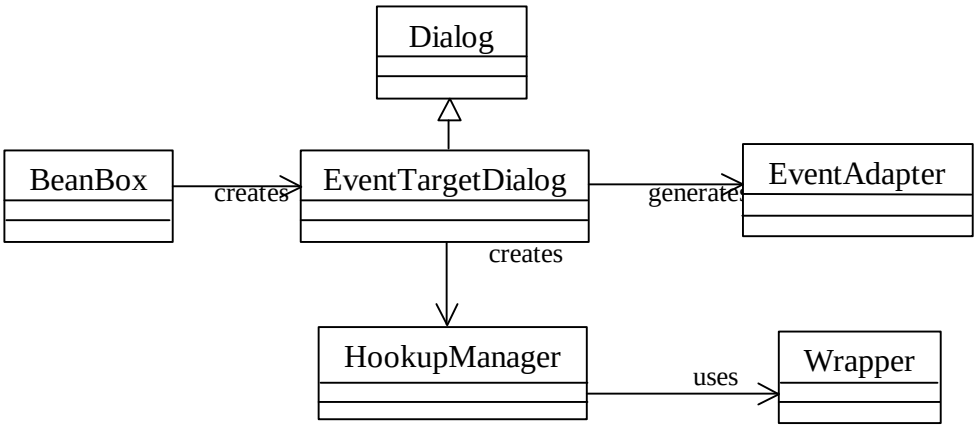

FIGURE 3-2 SIMPLIFIED CLASS DIAGRAM FOR GENERATING EVENT ADAPTERS

Figure 3-2 shows a simplified class diagram for generating event adapters in the BeanBox. A user has to identify the source bean, the event, the target bean, and the target method, in order to bind two beans with one event communication. The BeanBox composition window is used to select the source bean and the event. After that a red rubber hand appears, enabling the selection of the target bean. The EventTargetDialog pops up after the user selects the target bean. It lists all feasible methods at the target bean that are able to accept and handle the incoming event. Recall section 3.1.1, where it was stated that each distinct kind of event is targeted at a distinct target method. Such a relationship is established by the identity between event type and the type of the sole argument of the target method. In other words, a feasible method is one with only one argument, whose type is the same as the event type. The EventTargetDialog lists not only these methods, but also methods with no arguments. Void methods are selected because event adapters are interposed between the source and target bean. The target bean does

not have to implement a particular EventListener interface in order to accept an event. Figure 3-3 shows a piece of event adapter code. The adapter implements the ActionListener interface, and therefore accepts an ActionEvent with an actionPerformed method, which in turn invokes a void method at the target bean. The argument passed to actionPerformed is simply abandoned, and the target bean does not implement ActionListener interface. After the target method was selected, the EventTarget Dialog calls a method at HookupManager to generate, compile, and deploy the adapter. Along with generating event adapters, the HookupManager updates the source bean's wrapper to record the addition of a new interaction (an instance of the Wrapper class is created for each bean when the bean is dropped into the BeanBox. It provides facilities to support event binding, property binding, and other common functionality).

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import GameBoardBean;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_17312c5c50 implements java.awt.event.ActionListener,
                                             java.io.Serializable {
   public void setTarget(GameBoardBean t) {
      target = t;
   }
   public void actionPerformed(java.awt.event.ActionEvent arg0) {
      target.stop();
   }
   private GameBoardBean target;
}
```

FIGURE 3-3 A PIECE OF EVENT ADAPTER CODE

3.2.3 From BeanBox to SwapBox

The most significant reason that the BeanBox is selected as the starting point to build a new hot swapping infrastructure is that BeanBox itself is a test container for JavaBeans. By using the BeanBox, the only thing left is to incorporate it with hot swapping management such that users could visually manage hot swapping transactions. In addition, the BeanBox has been selected for the following reasons:

- Source code available. The BeanBox source code is available at Sun Microsystems' web site [36], along with the JavaBean model.
- Full support of JavaBean features. Even though the BeanBox is only a test container for JavaBeans, it supports all the features of the JavaBean model.
- Simplicity. The architecture of the BeanBox is simple but well organized. This reduces the work on coding extension so that the research can be concentrated on the strategies that should be considered for swappable beans.

In the literature, there are several groups that modified the BeanBox to do their research. To name two of them, Effaris [40] project modified the BeanBox as a visual platform for multi-agents, and EVOLVE [41] project modified the BeanBox as a runtime tailoring platform for group-aware applications.

This chapter reviewed the JavaBean component model, Java object serialization, XML, and the BeanBox. These technologies are used to develop the SwapBox. The next chapter describes in detail the design and implementation of the SwapBox.

Chapter 4 Design and Implementation of the SwapBox

4.1 General Principles

4.1.1 Design Issues

Recall section 2.3.1, which dealt with a set of common issues faced in the design of hot swapping systems, i.e., granularity, reference indirection, constructing new states, levels of change, and timing for upgrading. The design for the hot swapping system proposed in this chapter tries to address all of these issues to some extent. It aims to provide hot swapping capability for event-driven, adapter-connected JavaBean applications. Following is a complete list on how the new hot swapping system does with respect to these issues.

- Granularity. The basic unit for JavaBean hot swapping is the JavaBean component.
- Reference indirection. When the event communication model is applied to beans, adapters can be interposed between beans. They are able to provide a reference indirection to swappable JavaBeans. Therefore, the hot swapping system requires the beans to be connected with event adapters. Compared with the proxy pattern

in Ning Feng's work [4], each swappable bean has an unspecified number of event adapters while S-Module only has one proxy. The event adapters, unlike the S-Proxies, are generated automatically.

- Constructing new states. The hot swapping system captures the old bean's object states and assigns them to the corresponding new bean's object states. Mapping rules are used to establish a one-to-one relationship between the old state and the new state. The hot swapping system does not capture the running stack of the old bean in order to construct new bean's running stack. Such an attempt requires semantic knowledge of the new bean and the old bean's implementation to analyse the points at which the hot swapping could occur. Gupta et al. demonstrated in their work [9] that the data-flow analysis for such points is tedious, and how to scale it up to large programs is still an open research area.
- Levels of change. The hot swapping system supports implementation, interface, and data structure change. Reference indirection itself is enough to support implementation change. In order to support interface change, the hot swapping system has to deal with interface mismatch in which the new bean and the old bean have different interfaces. It exploits JavaBean's ability to dynamically add/remove interactions to make the interface change possible. The new bean does not have to be the subclass of the old bean, which was required in S-Module approach. The data structure change is normally an internal change with ability to affect public interface. Since the hot swapping system supports both implementation and interface changes, it is able to support data structure change as well.

- Timing for upgrading. The timing problem is indeed the problem of preserving the program correctness at the time of upgrading. Any time that the guarantee exists, the hot swapping can take place. Unfortunately, the method of keeping the program stable and correct at the time of upgrading is application-specific because different applications have different program structures and different requirements on what is "correct". Some applications, for example, may allow the upgrading to occur only when the old bean is idle. Others, on the contrary, may tolerate state loss and allow the upgrading to occur when the old bean is busy. In recognising the problem with differences in timing, the new hot swapping system is designed to provide several solutions for timing problem that can be selected at runtime. This increases its flexibility.

4.1.2 Terms

In event-driven JavaBean applications, method invocations are realised through event delivery. These event deliveries, along with beans, are composed of a graph within which beans are nodes and event deliveries are edges. The edge has direction, i.e., starting from the source bean and ending at the target bean. A node (JavaBean) may have many edges connecting with it. In the context of this thesis, an **interaction** is defined as an edge connecting two nodes. It is identified by the source bean, the target bean, the event, and the target method. A bean's **partners** are defined as neighbour nodes that are directly connected to the bean. Partners are either sending events to the bean or receiving events from the bean.

The hot swapping, like network management, belongs to the management work. In the traditional JavaBean component model, the bean developer develops beans, the application assembler assembles beans into applications, and end-users use the application. Figure 4-1(a) shows the use case diagram for the three roles in a normal component system. There is no role designated to take care of hot swapping work. This thesis proposes an extra role named **swap administrator** to deal with the hot swapping work. Figure 4-1(b) shows a use case diagram for the four roles. In practice, the swap administrator could be merged with the application assembler to the same personnel.

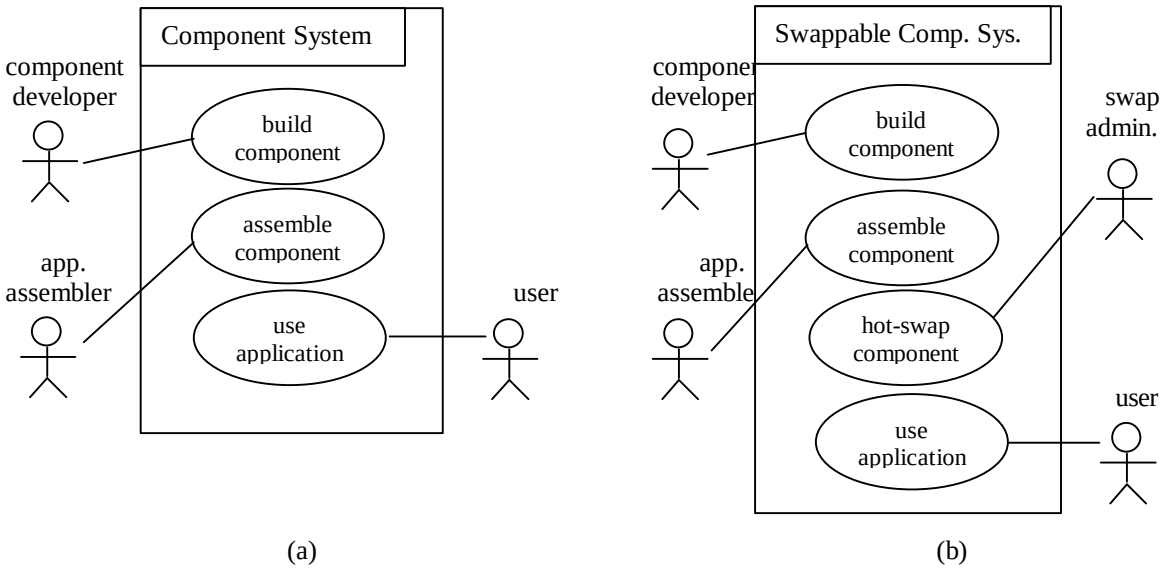

FIGURE 4-1 USE CASE DIAGRAM FOR ROLES IN TWO COMPONENT SYSTEMS

4.1.3 SwapBox Overview

The Swapbox incorporates the general principles proposed above into the BeanBox. Similar to the BeanBox, the SwapBox is the execution container for swappable JavaBeans. Moreover, it provides facilities for hot swapping old beans with new beans. Applications created within the SwapBox are event-driven and adapter-connected. The

SwapBox does not directly modify the BeanBox's code. Instead, it inherits the BeanBox's classes and overrides methods that have different behaviours. All the classes belonging to the SwapBox are at the carleton.swapbox package, which works with the sun.beanbox package to play the role of the SwapBox. It is worth noting that some methods' modifiers have been changed from *private* to *protected*. This is because the extended code in the SwapBox must make use of these methods. Without alteration, these methods are not visible to the classes in the carleton.swapbox package. The concept of the SwapManager, proposed by Feng and Ao in [4, 1] is kept. It is the co-ordinator and executor of the swap transaction. This will be discussed in section 4.6. A set of GUI is provided to configure hot swapping policy, which is then saved as an XML file. The significance of the hot swapping policy and how to configure it is discussed in section 4.3. State transferring and interaction handling are discussed in section 4.4 and 4.5, respectively.

4.2 Reference Indirection: Event Adapters

4.2.1 Functionality and Design

Event adapters provide reference indirection to swappable JavaBeans. They are like the proxies in Feng and Ao's work [4,1]. The functionality of event adapters is as follows:

1. Relay event delivery from the source bean to the target bean. This is the basic functionality of event adapters. Without it, the hot swappable application cannot work.

2. Block and unblock event deliveries as they are requested. Hot swapping involves special operations of the application. The execution of these special operations should not be overlapped with normal operations. Therefore, after on-line upgrading takes place, the old bean will not immediately react to service requests (event deliveries). Such requests are blocked. In other words, when the hot swapping framework blocks requests to a given bean, event deliveries to the bean are cut off. At this time, the event adapter does not forward any incoming events to the bean. As soon as hot swapping is finished, event adapters resume their normal state. Service requests are unblocked.
3. Queuing events at blocking time. The hot swapping is expected not to disturb the normal operations of the application. When event adapters for a particular bean are blocked, event deliveries should be queued instead of discarded in order that they will be handled after the hot swapping is finished. Event adapters must provide a mechanism to queue events and deliver them to the right bean (i.e., the new bean when hot swapping succeeds, or the old bean when hot swapping fails) after hot swapping.

Event adapters in BeanBox are relatively simple. A piece of code is listed in Figure 3-3. An adapter takes an event fired out from the source bean, and invokes a method at the target bean. Each adapter has a method in common called *setTarget*, which is used to set the target bean. Therefore the target bean could be changed after the event adapter is created. Because of this method, the BeanBox event adapter model is able to handle

simple hot swapping jobs. A new bean hot swaps the old bean by invoking the *setTarget* method so that forthcoming events are forwarded to the new bean.

However, for the reasons listed below, the simple event adapter model and *setTarget* approach does not meet the functionality previously discussed.

- There is no mechanism to temporarily block and queue incoming events while the target bean is in the swapping process
- Event delivery is synchronous which prohibits any possibility of doing additional work during event dispatch
- With the *setTarget* method, the new bean has to be a subclass of the old bean. Otherwise the method fails because of argument type mismatch.

Due to these limitations, the BeanBox event adapter model cannot handle complicated hot swapping jobs. For example, in a case where the old bean services a couple of different components, and the arrival time of requests is a function of random time. When hot swapping is initiated to this bean, the hot swapping system must somehow block the service requests such that the bean can get into idle state in order to be swapped out. The BeanBox event adapter cannot do this. A new kind of event adapter is needed to achieve this complex functionality.

The first step in designing an event adapter is to identify its states, which fall into two categories. One is the servicing state; the other is the non-servicing state. If the event adapter is loaded into memory, and is ready to accept events as well as invoke methods, it

is in servicing state. If the event adapter is loaded into memory but not ready to handle events, it is in non-servicing state, which discards incoming events. The servicing state is further divided into blocked and working states. An event adapter queues incoming events when it is in blocked state. Queued events are released to invoke a corresponding target method when the adapter moves into working state. Figure 4-2 shows finite state machine for event adapter states.

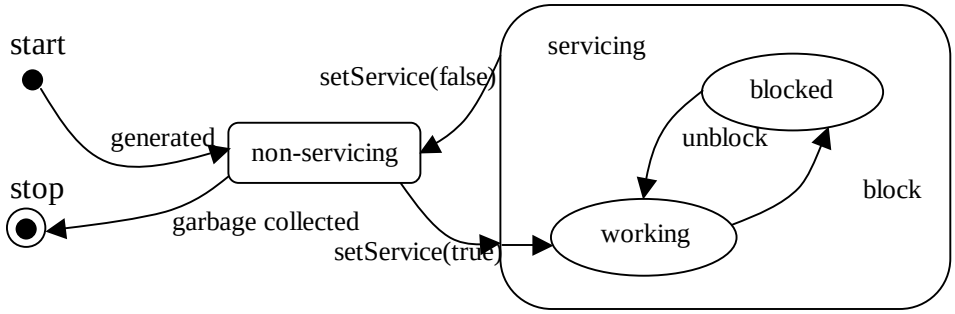

FIGURE 4-2 EVENT ADAPTER FSM

Each event adapter has three methods in common, i.e., setTarget, setService, and setBlock. The latter two are used to manipulate event adapter states. Event adapters use an asynchronous method to forward event delivery. Incoming events are deposited into a vector, while a separate thread takes events out from the vector and invokes a corresponding method at the target bean. Adapters implement java.lang.Runnable interface.

The code of the new event adapter is a little bit longer than that of the BeanBox adapter. This inevitably introduces overhead in terms of memory usage. If an application has many event adapters connected to beans that may never need to be swapped out, the overhead will be huge and unnecessary. To eliminate this, SwapBox has a sub-menu,

which can "turn off" the hot swapping capability so that the SwapBox shrinks to a BeanBox, and generates simple event adapters. In this way, application assemblers are able to decide what kind of adapters should be used in an application. An application could have both kinds of event adapters. Any bean connected by the simple event adapters is not swappable.

The SwapBox does not use *setTarget* method to do a hot swapping job, because it not only requires the new bean to be subclass of the old bean but also complicates the functionality of the event adapter, which is complex enough after adding block/unblock/queuing methods. Instead, the SwapBox generates a new set of event adapters for the new bean at the beginning of hot swapping. The hot swapping benefits in two aspects from this approach:

1. Simplicity. The two candidates of the program, the new bean with new adapters and the old bean with old adapters, co-exist in the application. If hot swapping succeeds, the old bean and its adapters are removed from the application. If hot swapping fails, the new bean and its adapters are removed.
2. Normal operation is not affected. Both the new bean and the old bean's adapters are in the blocked state when the hot swapping takes place. Therefore events will be queued in both the new and old bean's adapters. This ensures that the application's execution is not affected by the hot swapping because no matter which bean is used after hot swapping, its adapters have a complete list of undelivered events and will forward them to the bean.

4.2.2 Automatically Generating Event Adapter

Similar to event adapter generation in the BeanBox, the SwapBox automatically generates enhanced event adapters on the fly. This simplifies the bean developer's work when developing hot swappable applications. Figure 4-3 shows a class diagram for classes generating event adapters. Among the classes presented, the class SwapBox extends from the BeanBox class. It has the same sub-menu to bind events that BeanBox does. However, instead of creating an EventTargetDialog instance, it creates a SwapEventTargetDialog instance, which in turn creates SwapHookupManager to carry out event adapter code generation, as well as compile the code and load the adapter into memory. SwapHookupManager extends from HookupManager. It has a very big static method called *generate*, which is used to generate the code for an event adapter. This method overrides the same signature method in the HookupManager class. Moreover, SwapHookupManager puts an instance of SwapEventInfo into AdapterCenter after the newly created event adapter is loaded into memory. Section 4.4.3 discusses AdapterCenter and SwapEventInfo in detail.

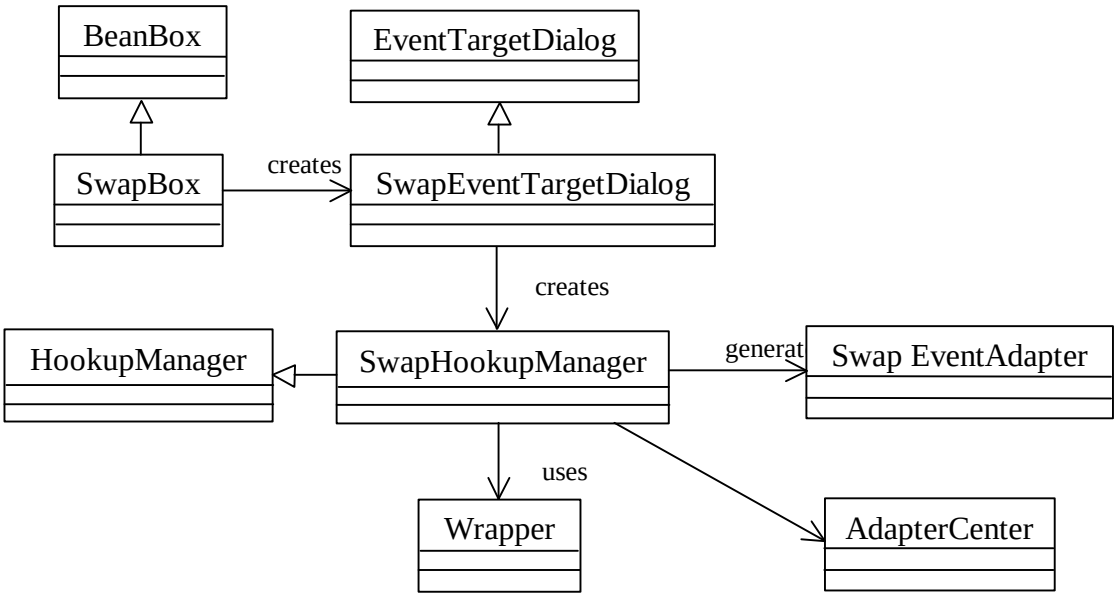

FIGURE 4-3 SIMPLIFIED CLASS DIAGRAM FROM GENERATING EVENT ADAPTER

The SwapBox provides two possible approaches to initiating the process of creating event adapters, while BeanBox only has one. The application assembler can initiate this process by clicking a sub-menu, and identifying the source event and target method. This is what BeaBox did. Newly created adapters get into the working state immediately after being loaded into memory. At the other end of the spectrum, SwapManager is also able to initiate an event adapter generation process. This ability is necessary in hot swapping. In order to bring the new bean into the running application, SwapManager has to generate event adapters for the new bean. It is like a normal adapter generation with a slight difference to the adapter's initial state. If created upon request from the SwapManager, the adapter should stay in the non-servicing state, waiting for the remaining preparation work to be finished, then switching into the working state. The SwapHookupManager is expected to distinguish two kinds of adapter generation. A formal argument of the *generate* method specifies the kind of event adapter generation which is expected.

4.3 Configuring the Hot Swapping Policy

4.3.1 Hot Swapping Policy

A hot swapping policy regulates how to swap out an old bean and install a new bean. Due to the diversified functionality of applications, there are many different ways to carry out hot swapping. Some applications, for example, may have strict requirements on transferring states, others may tolerate transient state loss and degrade into a “lower quality” service at the time of upgrading. Some applications tend to swap the old module immediately, while others might be patient enough to allow the current work to end at the old module and direct forthcoming requests to the new module. As a hot swapping management tool, the SwapBox is expected to provide diversified and well-organised approaches carrying out a hot swapping job. A swap administrator could use a hot swapping policy to select one of several different approaches and trim them to fit the requirements of a particular hot swapping task. Therefore each hot swapping transaction has its own swapping policy.

A hot swapping policy not only benefits diversified swapping requirements, it also makes swapping safer. Recall section 2.2.1, which stated that a requirement for hot swapping is to reduce human involvement when change is in progress. Excessive human involvement at such a critical moment introduces a higher risk of mistakes.

The SwapBox addresses this issue with the provision of a swapping policy file in XML format. The hot swapping policy specific to a transaction is stored in the file, which is parsed by the SwapBox at swapping time. A swap administrator is able to configure a hot swapping transaction on the fly without interfering with the program's normal operation. The configuration is recorded in the policy file, including how to handle interactions in case of interface mismatch, how to transfer states, etc. The swap administrator initiates a swapping job by visually identifying the old and new beans, and selecting the policy file. SwapBox does the swapping work according to the contents of the file. The result is either success or failure. Human interaction in the process of hot swapping is not necessary and is prohibited.

XML format is selected because it is universal. Many programming languages are able to parse an XML document. Another consideration is extensibility. A swapping management tool is supposed to provide diversified services for hot swapping. Some of them are identified in this thesis, e.g., interaction handling, set bound time for hot swapping, and transferring object states. They are implemented in the SwapBox. Future research may explore other services and incorporate them into the SwapBox. As a framework, the SwapBox is designed to accommodate swapping policies which are not presently implemented but which may come up in the future. XML is valuable for extension, therefore it has been selected to express the swapping policies. Figure 4-4 gives an example of hot swapping policy.

```
<?xml version='1.0' encoding='us-ascii'?>
<swap>
   <swap_type>Default</swap_type>
   <pre_process>
      <time>123456</time>
   </pre_process>
   <post_process>
      <swap_method>newStart</swap_method>
   </post_process>
   <state_policy>
      <Serialization>false</Serialization>
      <state newName="NewDim" oldName="Dimension">
      </state>
      <state newName="NewWidth" oldName="Width">
      </state>
      <state newName="NewRate" oldName="Rate">
      </state>
      <state newName="Status" oldName="Status">
      </state>
      <state newName="Running" oldName="Running">
      </state>
   </state_policy>
   <interaction_policy>
      <change_TargetMethod>
            <event_source>Start Button</event_source>
            <event_name>button push</event_name>
            <old_method>
                <method_name>start</method_name>
            </old_method>
            <new_method>
                <method_name>newStart</method_name>
            </new_method>
      </change_TargetMethod>
   </interaction_policy>
</swap>
```

FIGURE 4-4 AN EXAMPLE OF HOT SWAPPING POLICY IN XML FORMAT

The swapping policy can be said to consist of five parts, i.e., swap_type, pre_process, post_process, state_policy, and interaction_policy. The explanation for each part is listed as follows:

- **swap_type** specifies which swap manager will be used in the hot swapping job. The swap manager is responsible for co-ordinating and executing the hot swapping work. Three different swap managers are implemented within the SwapBox (section 4.6 gives a detailed description on the design and implementation of the swap manager). A swap administrator selects a swap manager that fits into the particular hot swapping requirement. This is the manager that will execute current hot swapping work.
- **pre_process** gives the swap manager a chance to do some extra work prior to the hot swapping beginning. At present only a time constraint is provided. Hot swapping work is like real-time work. The correct execution depends not only on the right answer but also on whether or not the work is finished in time. The time constraint specifies the time limit allowed for a valid hot swapping. The swap manager sets up a timer before beginning hot swapping. If the hot swapping execution exceeds the time constraint, the swapping is not acceptable and should be rolled back to the state immediately before the swapping.
- **post_process** enables the swap manager to carry out extra work after a hot swapping task succeeds. This may include re-opening a file or network resource, invoking a specific method to execute some code related to hot swapping, and so on. Currently only the swap_method is provided. It is used to specify a void method at the new bean. The swap manager invokes this method by reflection after hot swapping is

finished. Invocation of such a method is needed if the old bean is swapped out while it is not in the idle state. More discussions on this are included in section 4.6.3.

- **state_policy** is used to specify which old states need to be transferred, and their correspondent states in the new bean. Section 4.5 discusses state transferring in detail.
- **interaction_policy** is used to specify interaction handling if the new bean and the old bean have different interfaces (interface mismatch). Section 4.4 discusses interaction handling in detail.

4.3.2 How to set up a Swapping Policy

Rather than writing a policy file directly, a swap administrator composes it with the help of a series of GUI in the SwapBox. State pattern [38] is used for this purpose. Figure 4-5 shows the class diagram for classes used to graphically create a swapping policy file. The SwapBox uses four states (panels) to create the file. Each concrete state (except for the TerminalState) corresponds to a panel. The SwapConfigEditor inherits from the Frame class. It is the container for different panels (states). Figure 4-6 shows a sample snapshot of the InteractionState.

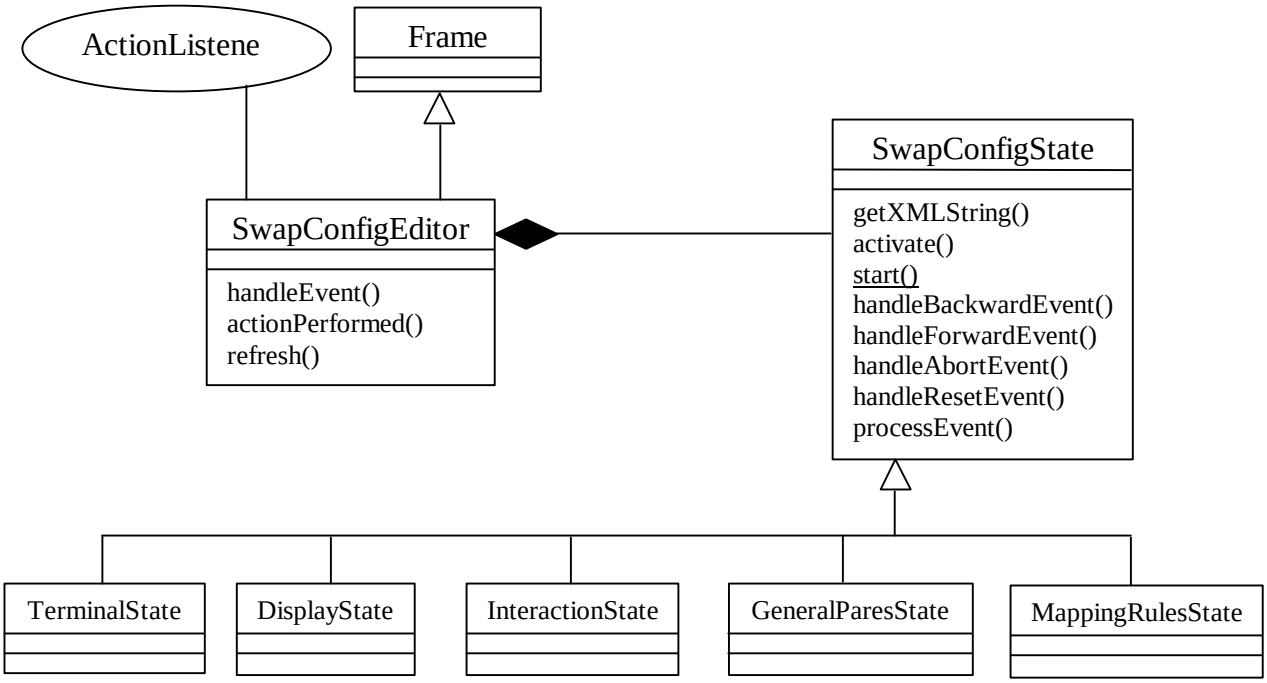

FIGURE 4-5 SIMPLIFIED CLASS DIAGRAM FOR CREATING CONFIGURATION FILE

The TerminalState is a pseudo-state to mark the end of the creation of a configuration file. The other four states, i.e., DisplayState, InteractionState, GeneralParesState, and MappingRulesState, are used to make up an XML-based policy file. Within the constructor of these states, all possible selections are laid out. A swap administrator only needs to click on the selections appropriate to the current hot swapping task. The settings are converted to XML text within the *getXMLString* method, which is invoked at the DisplayState. Four methods, i.e., handleBackwardEvent, handleForwardEvent, handleAbortEvent, and handleResetEvent, link states (panels) together so that the swap administrator can go through them at will to modify his selections.

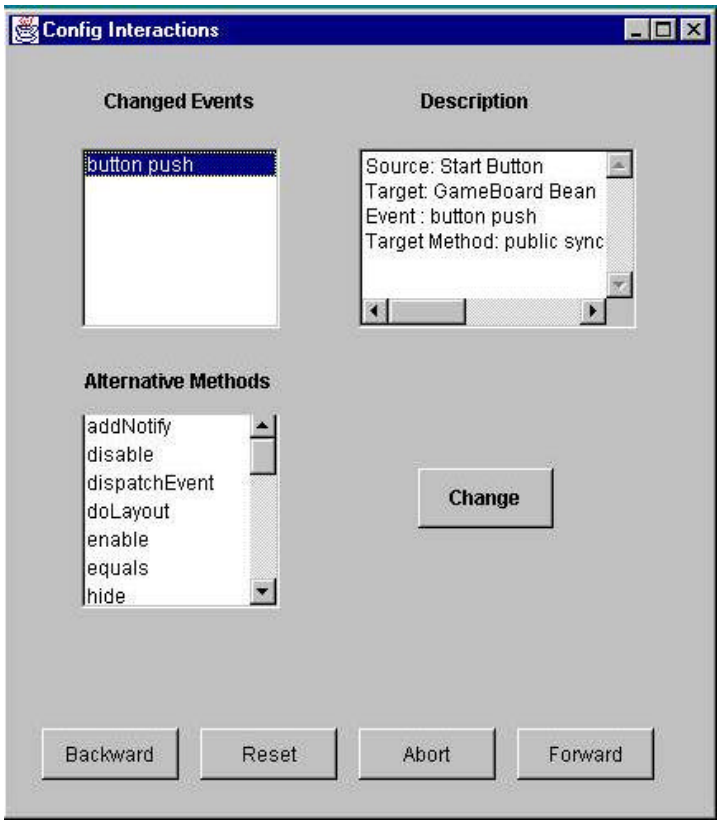

FIGURE 4-6 SNAPSHOT OF AN INTERACTION CHANGE HANDLING

GeneralParesState specifies policies on time constraint, swap manager type, and the method to be invoked after hot swapping. InteractionState deals with interface change. MappingRulesState specifies state transferring policy. DisplayState presents policies in XML format, and enables the swap administrator to save it as a file. Any time the Abort button is clicked, or the policy is saved as a file, TerminalState is reached, and the whole process stops. The former three states (panels) could be replaced by other states (panels) with different layouts for different policies. In this way, SwapBox is easily able to accommodate new settings for hot swapping configurations. It is worth noting that modifications to the current XML format inevitably cause modifications to the

SwapManager because SwapManager's behaviour relies on the parsing of the configuration file. Section 4.6 contains a detailed discussion on SwapManager.

4.4 Interaction Handling

4.4.1 Transparent and Non-Transparent Hot Swapping

As a component, a bean's functionality can generally be divided into two aspects. One is the computational aspect, which carries out a computational tasks. The other is the interaction aspect, which interacts with other beans to deliver the output of the computational task and to receive inputs. As described in section 3.1, the three most important features of a bean are the properties, methods, and events it exposes to the outside world. They are a bean's **interface**, and responsible for interaction tasks. Note that the concept of "interface" should be distinguished from the concept of "active interface". In this thesis, the term **active interface** is used to define the set of a bean's properties, events, and methods that are actually referenced by the bean's partners. A bean's active interface is a subset of the bean's interface. The active interface can only be determined on-line, after the beans have been composed as an application and the interactions established.

One important aspect of the hot swapping issue is how to handle inter-module interactions at the time of upgrading. Based on the interactions, hot swapping can be classified into two types, i.e., **transparent hot swapping** and **non-transparent hot**

swapping. If the active interface of the old bean is a subset of the new bean's interface, then the hot swapping falls into the transparent category. If the active interface of the old bean is not a subset of the new bean's interface, then the hot swapping is non-transparent. In other words, if the interactions of the old bean can be ported to the new bean without any change, then the hot swapping is transparent; otherwise it is non-transparent.

The handling work related to a transparent hot swapping is straightforward. Since the new bean's interface is the superset of the old bean's active interface, there is no need to create new interactions for the new bean at the time of hot swapping. The old bean's interactions are "cloned" to the new bean without any change. A SwapManager can either simply switch event adapters associated with the old bean to the new bean, or create a new set of event adapters to take care of the new bean's interactions.

A non-transparent hot swapping is more complex. A non-transparent hot swapping means that there are either events fired out by the old bean or methods invoked by the bean's partners which no longer exist at the new bean. In other words, the new bean breaks the promise made by its predecessor. There are two ways to handle this. The first is to carry out a non-transparent swap only if all partner beans affected by the change give their go-ahead to the change. For example, old bean A has a method M responsible for taking event E fired by bean B and C. The new bean A' does not have method M. The swapping framework should get the go-ahead from both B and C before continuing the swap job. If either B or C disagrees with such a change, the swap transaction fails. In contrast, the second approach does not consult B and C for the swapability of A. It assumes the swap

administrator has full knowledge of interactions within the application. Therefore the lack of method M at A' will not ruin the whole application, and B and C will no longer have events delivered to A' to invoke method M. JavaBean's event model, combined with *addListene*r and *removeListener* methods at B and C, make this alteration feasible. The swap manager dynamically invokes the *removeListener* method at B and C to remove A from the interested listener list so that the next time the E is fired out, it will no longer be delivered to A.

4.4.2 Implementation Change and Interface Change

As discussed in section 2.3, changes a new bean makes to the old bean occur at three levels, i.e., the implementation level, interface level, and data structure level. If the new bean doesn't change the interface, but only modifies the method implementation, the hot swapping is at the implementation level. If the new bean has a different interface (i.e., different set of properties, events, and methods) to the old bean, the change is at the interface level. If the new bean changes the internal data structure (e.g., adds new variable, changes an array to a vector, etc), the change is at the data structure level. Data structure change is normally an internal behaviour. There may be cases where data structure changes lead to interface changes. However, with respect to inter-module interactions, they could be classified as implementation or interface changes rather than data structure changes. Therefore, in terms of interaction handling, there are only two types of changes, i.e., implementation change and interface change.

If the new bean only has implementation changes, the hot swapping is transparent. If the new bean has interface changes, the hot swapping might be transparent or non-transparent. Figure 4-7 illustrates the distinctions.

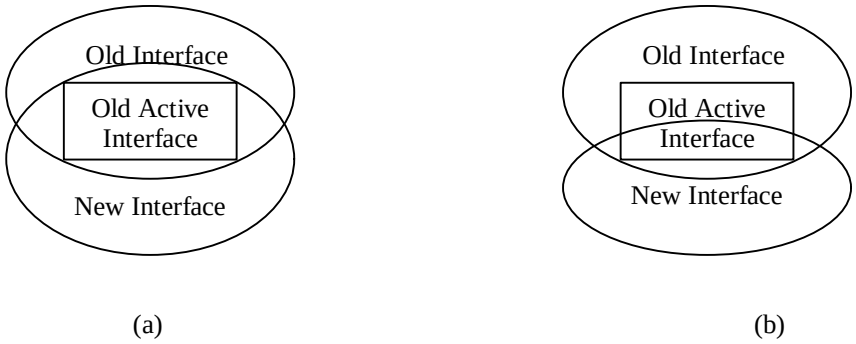

FIGURE 4-7 TWO DIFFERENT INTERFACE CHANGES

(a) The new interface is the superset of the old bean's active interface; (b) The new interface is not the superset of the old bean's active interface

It can be seen from Figure 4-7(a) that even though the new bean has an interface change, those changes do not affect the old bean's active interface. Therefore all interactions attached to the old bean can be ported seamlessly to the new bean. This is a transparent hot swapping. By contrast, Figure 4-7(b) shows that the new interface does not include all of the old active interface. In other words, the new bean does not provide certain properties, methods, or events that are necessary to rebuild the old bean's interactions at the new bean. This is a non-transparent hot swapping. It is worth noting that both (a) and (b) show that the new bean provides some additional interfaces (i.e., the part outside the old interface circle). A brief discussion on how to invoke these additional methods is contained in Section 4.4.3.4.

4.4.3 Interaction Handling in the SwapBox

4.4.3.1 AdapterCenter: the Repository for Interactions

No matter what kind of hot swapping it is (i.e., transparent or non-transparent), the SwapBox has to ascertain the active interface offered by the old bean. Since the interactions are created on the fly, it is not possible for the SwapBox to analyse a bean's active interface statically. This is where the AdapterCenter comes in.

AdapterCenter is a repository for all interactions which exist in the SwapBox when it is running. In addition, it provides further functionality to compare two beans, as well as identify whether or not a bean is in the progress of hot swapping. Each time an event binding is established, a SwapEventInfo instance is added to the AdapterCenter. This SwapEventInfo instance represents the interaction being added. In order to record an interaction between a source bean and a target bean, the SwapEventInfo must have information like the source and target bean's reference, reference to the event adapter, event name, and target method name. Because the interaction is established at runtime, records in the AdapterCenter are changed from time to time, as is the old bean's active interface. When an interaction is created, a record of the SwapEventInfo object is added to the AdapterCenter. The record is deleted when the interaction no longer exists.

Based on SwapEventInfo records, the AdapterCenter is able to provide support for hot swapping. The AdapterCenter uses two important methods to do this. One is the *getBeanReport*, the other is the *getSwapReport*. The former allows the SwapBox to

ascertain all the SwapEventInfo instances associated with a given bean. This bean is either the source or the target in the SwapEventInfo. The latter compares the two beans, the old and new bean, to return an instance of SwapReport, which contains all of the changed and unchanged interactions. They are stored as SwapEventInfo instances in two vectors, separately. Suppose the old bean A has two interactions with other beans. The first is A firing out an event E to bean B. The other is method M at A invoked upon receiving an event from another bean. After comparing A and its substitute A', AdapterCenter is able to tell whether or not A' has the same event as E and the same method as M. Using the comparison result, the SwapManager is able to determine the hot swapping type (i.e., transparent or non-transparent), and behaves accordingly. The AdapterCenter uses JavaBean's introspection to analyze and compare beans.

4.4.3.2 When only Implementation Change Occurs

If the new bean only has implementation changes, the hot swapping is certainly a transparent one. For transparent hot swapping, a swap manager first ascertains all the interactions associated with the old bean by calling the *getBeanReport* method at the AdapterCenter. Based on the knowledge of the old bean's interactions, the swap manager could generate adapters for the new bean by calling the *generate* method at SwapHookupManager. Then the swap manager establishes the interactions between the new bean and the old bean's partners by invoking the *addListener* method to properly hook up the new bean.

4.4.3.3 When the New Bean has Fewer Methods

Figure 4-7(b) shows an example of where the new bean has fewer methods, meaning that the hot swapping is non-transparent. If the swapping is non-transparent, the old bean's interactions fall into two categories, i.e., changed and unchanged. Unchanged interactions are those that can be ported to the new bean. Changed interactions are those that cannot be ported to the new bean. Unchanged interactions are treated like those in a transparent swap. For changed interactions, the SwapBox gives the swap administrator a chance to reconfigure them. The swap administrator can decide not to reconfigure the changed interactions, thus deleting such interactions in the new bean. The SwapBox provides GUI to enable the swap administrator to reconfigure changed interactions (e.g., pick up a method at the new bean to take an incoming event). Such change is recorded as a part of a hot swapping policy and stored in the XML file.

When configuring hot swapping policies for interaction changes, all changed interactions are presented to the swap administrator. The SwapBox receives information about the changed interactions by consulting the AdapterCenter. It calls *compare* method at the AdapterCenter, putting the new and old beans as arguments. Upon getting the result, the SwapBox visually lists all of the changed interactions, if there are any. In addition, it analyses the new bean with Java reflection and lists all possible alternative methods. A method is alternative if it has no argument or the same arguments as the old target method, which is specified at the SwapEventInfo instance. Figure 4-6 shows a snapshot of a GUI handling interaction change.

A swap administrator can either select an alternative method to make the alteration or ignore the change. By selecting an alternative method, the administrator establishes a brand new interaction at the new bean. If no alternative method is selected, the changed interaction no longer exists at the new bean. The SwapManager parses this file at swap time, and behaves accordingly. Figure 4-8 shows an example of such an XML document.

```
<interaction_policy>
   <change_TargetMethod>
         <event_source>Start Button</event_source>
         <event_name>button push</event_name>
         <old_method>
               <method_name>start</method_name>
         </old_method>
         <new_method>
               <method_name>newStart</method_name>
         </new_method>
   </change_TargetMethod>
</interaction_policy>
```

FIGURE 4-8 HOT SWAPPING POLICIES ON INTERACTION HANDLING

Element change_TargetMethod is repeatable. For each change_TargetMethod element, event_source, event_name, and old_method identifies an interaction at the old bean. By identifying the interaction, its corresponding SwapEventInfo instance is located at the AdapterCenter. Element new_method records a method at the new bean. This method is expected to take over old_method to handle the event, whose name and source are recorded in event_name and event_source, respectively. At swapping time, the SwapManager ascertains the changed SwapEventInfo and the new method, then

generates a new event adapter for this changed interaction. In this way, SwapBox solves the problem of interface mismatch between the old and the new bean.

4.4.3.4 When the New Bean has Additional Methods

If the new bean has additional methods other than that of the old bean, just as Figure 4-7 (a) and (b) shows (the new interface area outside the old interface circle represents additional methods), the swapping framework must provide a mechanism to invoke these additional methods if necessary. Fortunately, the BeanBox, which is the ancestor of the SwapBox, already allows users to wire up beans dynamically. All of the events are bound up on the fly in the BeanBox. The SwapBox just makes use of this facility to provide a solution for invoking the new bean's additional methods. However, the SwapBox must take synchronisation into consideration. A swap administrator can wire up new events before the swap takes place, setting a new event adapter into non-servicing state. After the swap job is finished, the SwapManager brings these events to working state. Another possible approach is to wire up a new event after the swap transaction is finished. With the latter method, the event binding is the same as the BeanBox.

4.5 Transferring States

4.5.1 The Significance of Transferring States

The very basic idea of hot swapping is to replace an old software program with a new one while not disturbing the normal operation of the whole application. In other words, a hot swapping technique has to ensure that the application is stable at the time of upgrading. State transferring plays a significant role in achieving this goal. It includes capturing the old states from the old module, and re-constructing the new states at the new module. A process P containing the old version program X runs from the very beginning, possibly with input from the user. It starts from the initial state, and the states afterwards are the result of the program execution. When the new version program X' has been swapped in P, the hot swapping management tool must specify from which state X' should begin to execute. Intuitively a consistent hot swap is one where, after the swap transaction, the process P behaves as if X' has been running from its initial state. In order to achieve a persistent hot swapping, the state s in X where the hot swapping takes place must be captured, and translated (mapped) to an intermediate state s' of X'. The term intermediate signifies that state s' is one that can be produced by running X' from its initial state.

It is well known that keeping states persistent at the time of changing is one of the most important yet challenging tasks. The difficulty stems from two sources. First, it has to rely on the underlying operating system or virtual machine to capture the running environment (e.g., stack, program counter, register, etc) for the old module. Sometimes this support does not exist; e.g., Java Virtual Machine just disallows such an attempt. Second, even if the running environment of the old module could be retrieved easily, the

fact that the new module and the old module have different implementations hinders the construction of new states from the retrieved environment. Also, the term “persistent” has different requirements for different applications. Hot swapping is relatively simple when the old module is stateless or belongs to systems specifically designed to tolerate state loss. In such cases, hot swapping has nothing to do with state transfer, or only transfers some states that are not frequently changing. Unfortunately many systems are either “stateful” or do not tolerate dramatic state loss at the time of upgrading.

Considering the diversified types of underlying systems and application domain, it is usually domain (or even application) specific to develop state policies that ensure persistent execution during hot swapping. Several approaches for preserving component states and preventing communication loss during runtime change have been proposed at [32, 33, 34]. Hofmeister’s approach [34] requires each component to provide two interface methods: one for divulging state information, and the other for performing initialization when replacing another component. Feng [4] discussed the possibility of using mapping rules to transfer the states between the old S-Module and the new S-Module.

4.5.2 A Possible Solution: Java Serialization

Java object serialization provides a way to store away a Java object state and rebuild it later, possibly in another name space. It is easy to use. The application programmer does not have to write too much code to serialize and deserialize. However, it has versioning

restrictions on classes which serialize and deserialize states. Section 3.1.2 lists compatible changes for Java serialization. If changes between versions are not compatible, the serialization will fail. Since it cannot predict what changes the new bean may have, serialization cannot generally be used for state transferring.

Another problem of Java serialization for transferring states in a generic framework like the SwapBox is that the class type has to be known as a priori at the compile time. The *readObject* method at ObjectInputStream returns an instance of Object class. It has to be explicitly downcast to the type of the class that is deserializing the object. The application programmer must specify the type of class. It cannot be dynamically ascertained using Java reflection. Because the SwapBox is a generic environment for hot swapping all kinds of JavaBeans, it is impossible to hard code those class types into the SwapBox. Therefore the SwapBox cannot make use of Java serialization even if the changes between versions are compatible. However, it is worth noting that the Java serialization still has merits for transferring states in situations where there is no generic environment but a specific hot swappable application is developed. In such situations, the class type is known at the compile time, and the application programmer can explicitly downcast the type of return instance from *readObject* method to the one that is actually reading the object.

4.5.3 Approach Used in the SwapBox: Mapping Rules + Accessor Methods

The SwapBox uses accessor methods plus mapping rules to transfer object states between versions. Accessor methods are used to get and set state values. Mapping rules, defined by the swap administrator and used by the SwapManager, are used to dynamically link old states and new states.

Like Hofmeiter's approach, each bean which expects to transfer states in time of hot swapping has to provide interface methods to divulge state information and/or methods to initialise states. An old bean has to at least provide methods to divulge state information, whilst a new bean provides methods to initialise states. By providing both kinds of methods, a bean is capable of replacing the other bean, as well as being replaced by the other bean. As described in section 3.1.1, JavaBean already has a mechanism called "property" to export/import states. These properties can be changed at runtime. Each property has a pair of methods to set and get its value. A naming convention is imposed for the name of these two methods. The SwapBox makes use of this mechanism to divulge state information at the old bean and initialise states at the new bean. For each state that is needed at the time of hot swapping, it should be defined as a JavaBean property, i.e., a pair of methods which comply with the naming convention have to be provided for the state. There are, however, situations in which a state is needed in time of hot swapping but need not be displayed in the property sheet of a JavaBean visual assembler tool like the SwapBox. For example, some computational states, such as an array used for sorting, may never need to be displayed in the property sheet because it does not make sense to modify a sorting array when the sorting is in progress. It is also impossible to display all records (especially if the volume is extremely big) of an array in

a small property sheet. Fortunately, JavaBean provides a process called customization that gives programmers control over what states they are willing to display in the property sheet. With the help of customization, a property does not have to be displayed in the property sheet. In order to exploit the merits of customization, a programmer has to carry out extra coding to provide a BeanInfo class to specify what states are displayed. The SwapBox, in the meantime, provides a simple yet efficient alternative to customization to identify states that do not have to be displayed in the property sheet. It is similar to the naming convention used for a JavaBean property. A bean willing to export/import states beyond properties has to provide a pair of methods to get and set a state's value. The getter method's name starts with *swapGet*, and setter's starts with *swapSet*. Both method names end with the state name. For a state named *example* and of type *Test*, the declaration of such a pair of setter and getter methods look as follows:

Test swapGetExample();

Void swapSetExample(Test aTest);

In this way, the state *example* will not be visually displayed in the property sheet, while it can still be transferred and reconstructed at the time of hot swapping.

Getter and setter methods are not sufficient to handle state transferring problem. They only provide ways to get and set states values. There must be rules to map the old states to the appropriate corresponding new states. Feng discussed in [4] that these mapping rules cannot be hard-coded as interface methods in the old and new module; neither could they be hard-coded in a swapping management environment.

In the SwapBox, the mapping rules are actually one-to-one relationships that are expressed in XML format and can be parsed out at the time of hot swapping. The SwapManager invokes the getter and setter method for state transferring according to what it parses out from the mapping rules. Mapping rules are established at runtime, after the swap administrator identifies an old bean and a new bean. Because the old and new beans are identified, the SwapBox is able to extract state information on both beans using Java reflection. Firstly, it searches methods at the old bean, fetches all methods used to divulge state information, and visually lists the old state names. Secondly, it searches methods at the new bean, fetches all the methods used to perform state initialisation, and visually lists the new state names. A swap administrator can then select an old state name and a new state name to establish a one-to-one mapping relation. Figure 4-9 shows a snapshot of GUI for this purpose.

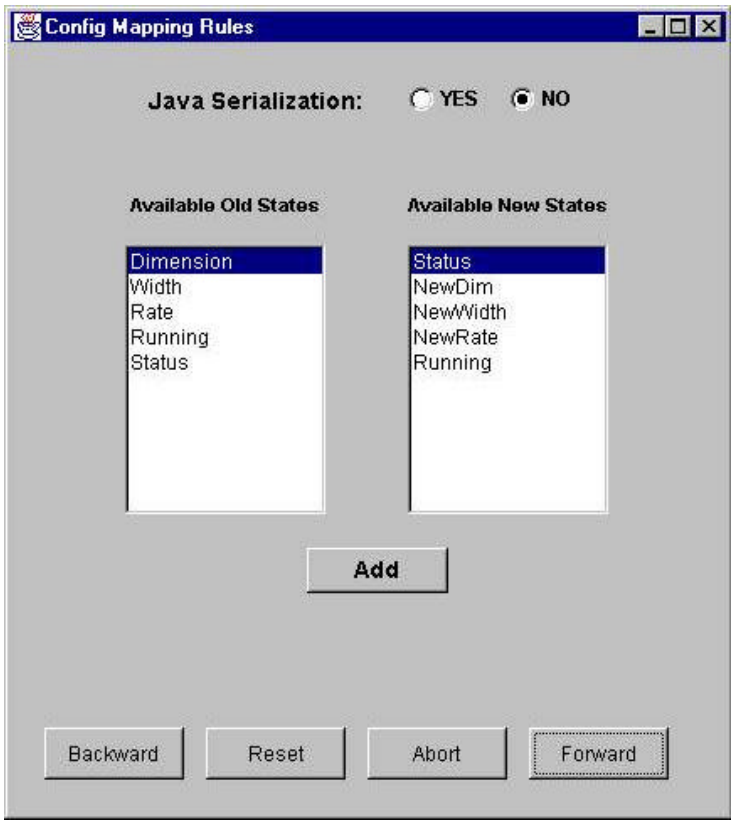

FIGURE 4-9 SNAPSHOT OF GUI TO ESTABLISH MAPPING RULES

The SwapBox does not provide a facility to support many-to-one mapping relation. Such support can easily be added to the SwapBox. It needs, however, co-operation from the new bean, i.e., an initialisation method has to take more than one argument. Normally one-to-one mapping is sufficient to handle most state transferring. If the changes between versions are only at implementation or interface level, the mapping rules are simple. Because no data structure is changed, each state at the old bean must map to the same state at the new bean. The states transferring approach proposed here can easily handle these two kinds of changes. If the data structure is changed (e.g., the name of the state is changed for some reason, or an object state is deleted, or a new state is added, etc), the approach can handle some of them without modification. For example, if the name of the state is changed, the mapping rules can easily bridge two different-name states with a

one-to-one relation. However, it must be said that for complex data structure changes, - e.g., adding a new object state whose value is determined by more than one state at the old bean, - the approach cannot cope, even though it has potential for extension.

Mapping rules are stored finally in XML format as part of the hot swapping policy file. Figure 4-10 shows an example of mapping rules. The element of *state* is repeatable. A *state* element has no value but two attributes, i.e., *newName* and *oldName*. The *newName* attribute gives the state name at the new bean, while the *oldName* attribute gives the corresponding state name at the old bean. At the time of state transfer, the SwapManager parses out the mapping rules from the swapping policy file. It looks up the getter method at the old bean by first looking at the method with the name g*etStateName*, and if the method is not found then it looks at the method with the name s*wap*G*etStateName*. The setter method at the new bean is found in the same way. State transferring takes place by first invoking the getter method at the old bean with no argument, and then invoking the setter method at the new bean with the argument taken from the return value of the first invocation.

```
<state_policy>
    <Serialization>false</Serialization>
    <state newName="NewDim" oldName="Dimension">
    </state>
    <state newName="NewWidth" oldName="Width">
    </state>
    <state newName="NewRate" oldName="Rate">
    </state>
    <state newName="Status" oldName="Status">
    </state>
    <state newName="Running" oldName="Running">
```

```
        </state>
    </state_policy>
```

FIGURE 4-10 EXAMPLE FOR MAPPING RULES

4.6 Putting it together: SwapManager

4.6.1 Design of the SwapManager

When a swap request is identified (i.e., the old bean, new bean, and XML policy file are selected), the SwapManager comes into play to co-ordinate the swap transaction. Its main responsibilities include:

- Parsing the XML-based hot swapping configuration file to retrieve parameters specific to the transaction
- Setting the timer such that the swap transaction (no matter whether it succeeds or not) takes place within a specific time
- Creating event adapters for the new bean
- Based on states at the old bean and the mapping rules, creating new states
- Cleaning up the old bean when transaction is successful
- Rolling back to the old bean when exceptions arise

The strategy pattern [38] is used to design the SwapManager. The pattern is selected because it will be easy to incorporate different swap managers in the future. The SwapBox currently has three SwapManagers, i.e., DefaultSwapManager,

Option1SwapManager, and Option2SwapManager. By implementing the strategy pattern, the SwapBox separates the concrete swap manager, which is easy to change, from the other parts. A swap manager with a different implementation could easily be added into the SwapBox. In this way, the SwapBox is an extensible framework. The Class diagram for the SwapManager is shown in Figure 4-11.

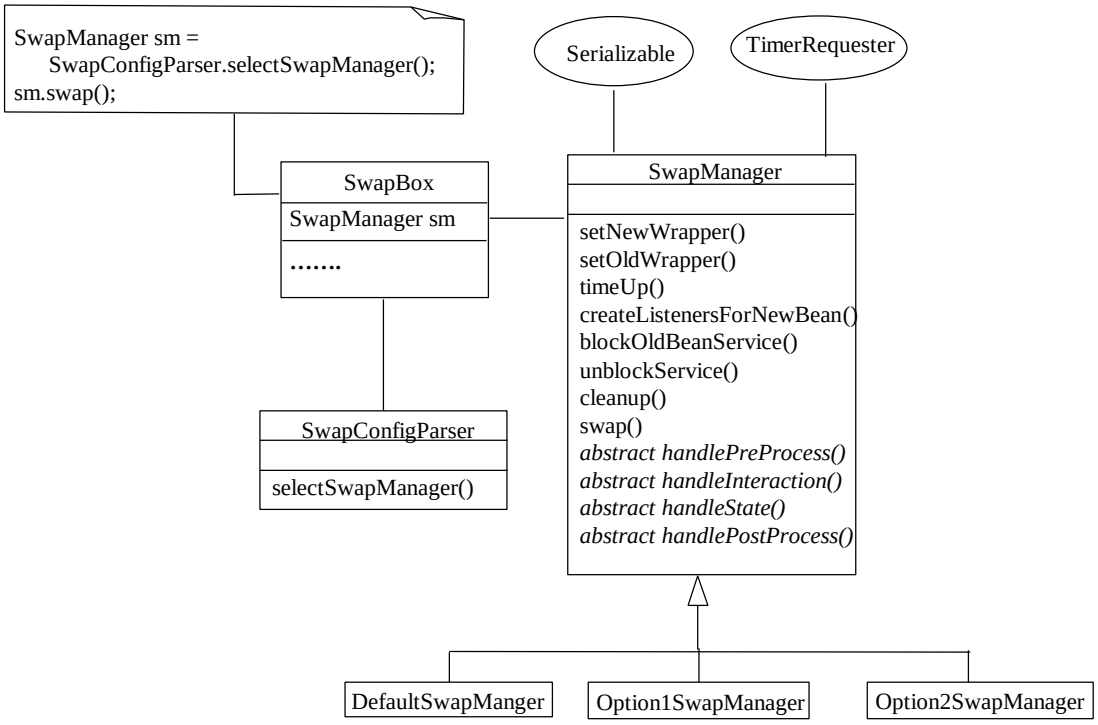

FIGURE 4-11 SIMPLIFIED CLASS DIAGRAM FOR SWAP MANAGER

The SwapManager is an abstract class. It contains methods common to all concrete swap managers, such as setting the timer and cleaning up the beans after the swap transaction is finished. In addition, it declares four abstract methods, i.e**.,** *handlePreProcess*, *handleState, handleInteraction,* and *handlePostProcess*, so that they are implemented at

the concrete swap manager class, i.e., DefaultSwapManager, Option1SwapManager, and Option2SwapManager. The most significant method at the SwapManager is *swap*, which delegates a hot swapping job to the four abstract methods. Figure 4-12 gives code for the *swap* method. Concrete swap managers differ with respect to implementing these four methods in different ways. Different implementation of these four methods results in different strategies to carry out hot swapping. Concrete swap managers are hidden from other parts of the SwapBox. The SwapParserConfig class parses the XML policy file, finds the *swap_type* element, and instantiates a concrete swap manager according to the value of the element. It finally returns this concrete swap manager of type SwapManager, whose *swap* method is then invoked to finish the hot swapping work.

```
public void swap() throws SwapException {
    NodeList nodeList = document.getElementsByTagName(preProcessTagName);

    if (nodeList.getLength() == 1 && nodeList.item(0).getNodeType() == Node.ELEMENT_NODE) {
       handlePreProcess(nodeList.item(0));
     } else {
       throw new SwapException("SwapManager: Parse pre_process node failed");
    }

    nodeList = document.getElementsByTagName(interactionPolicyTagName);
    if (nodeList.getLength() == 1 && nodeList.item(0).getNodeType() == Node.ELEMENT_NODE) {
       handleInteraction(nodeList.item(0));
     } else {
       throw new SwapException("SwapManager: Parse interaction node failed");
    }

    nodeList = document.getElementsByTagName(statesPolicyTagName);
    if (nodeList.getLength() == 1 && nodeList.item(0).getNodeType() == Node.ELEMENT_NODE) {
       handleState(nodeList.item(0));
     } else {
       throw new SwapException("SwapManager: Parse state_policy node failed");
    }
```

```
        nodeList = document.getElementsByTagName(postProcessTagName);
        if (nodeList.getLength() == 1 && nodeList.item(0).getNodeType() == Node.ELEMENT_NODE) {
           handlePostProcess(nodeList.item(0));
         } else {
           throw new SwapException("SwapManager: Parse post_process node failed");
        }
    }
```

FIGURE 4-12 CODE FOR *SWAP* METHOD AT SWAPMANAGER

The difference between DefaultSwapManager, Option1SwapManager, and Option2SwapManager is in how they handle the hot swapping timing problem. The DefaultSwapManager swaps the old bean immediately after it gets the hot swapping request, no matter whether the old bean is busy or not. Option1SwapManager is more patient. It allows the old bean to finish the current job, if there is one, before being swapped out. However, it is worth noting that the old bean must implement *isIdle* interface when the Option1SwapManager is used. The definition of *isIdle* is listed as follows:

```
public interface IsIdle {
   public boolean isIdle();
   public void addIsIdleListener(IsIdleListener l);
   public void removeIsIdleListener(IsIdleListener l);
}
```

These three methods enable the Option1SwapManager to detect the old bean's state as well as to be notified when the old bean moves from the busy state to the idle state. It is the old bean developer to decide when the old bean moves from the busy state to the idle state. During the time spent waiting for the old bean to finish its job, the Option1SwapManager blocks events from being sent to either the old or the new bean. It queues the events at both beans' adapters. However, if the time constraint allowed for the hot swapping is reached before the old bean gets into idle state, the hot swapping aborts.

All queued events will be directed to the old bean. The Option2SwapManager, on the other hand, allows the old bean to continue its work but will forward all forthcoming events to the new bean. In other words, the new bean and the old bean may be running simultaneously in memory. All concrete swap managers have the same mechanisms on interaction handling (discussed in section 4.4) and state transferring (discussed in section 4.5). Figure 4-13 is a flowchart which shows how the swap manager carries out hot swapping work.

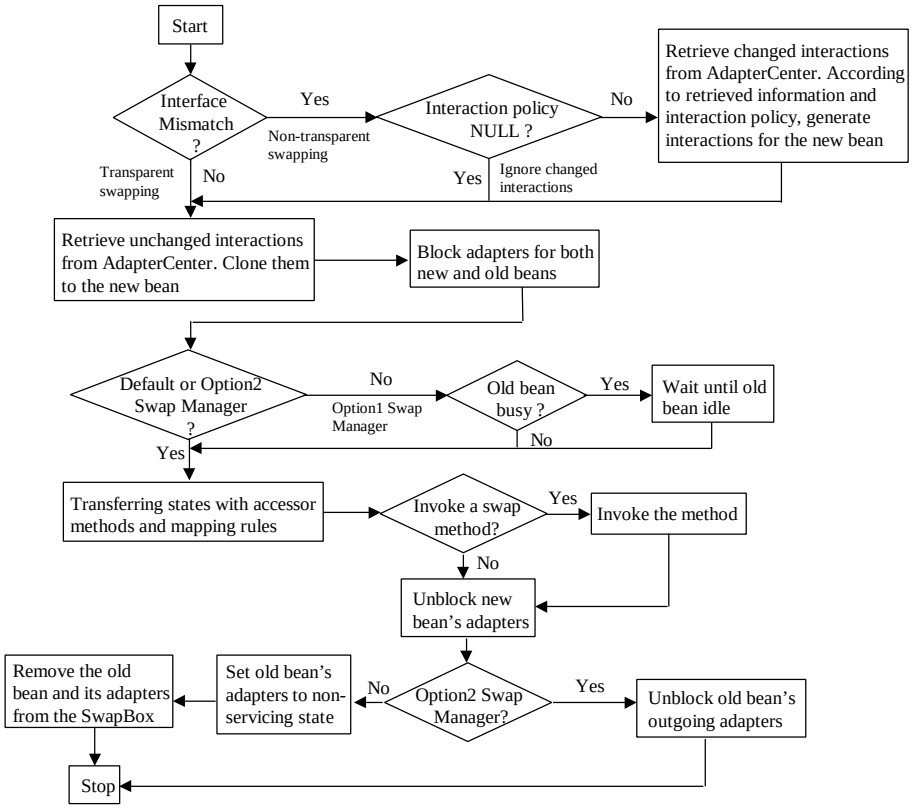

FIGURE 4-13 FLOWCHART ON SWAP MANAGER EXECUTION

4.6.2 Scenarios for Hot Swapping

The DefaultSwapManager is the reference implementation for the SwapManager in the SwapBox. Figure 4-14 is the interaction diagram which shows the DefaultSwapManager doing non-transparent swapping. Scenarios of the DefaultSwapManager carrying out transparent swapping and the other two SwapManagers carrying out hot swapping tasks can be derived easily.

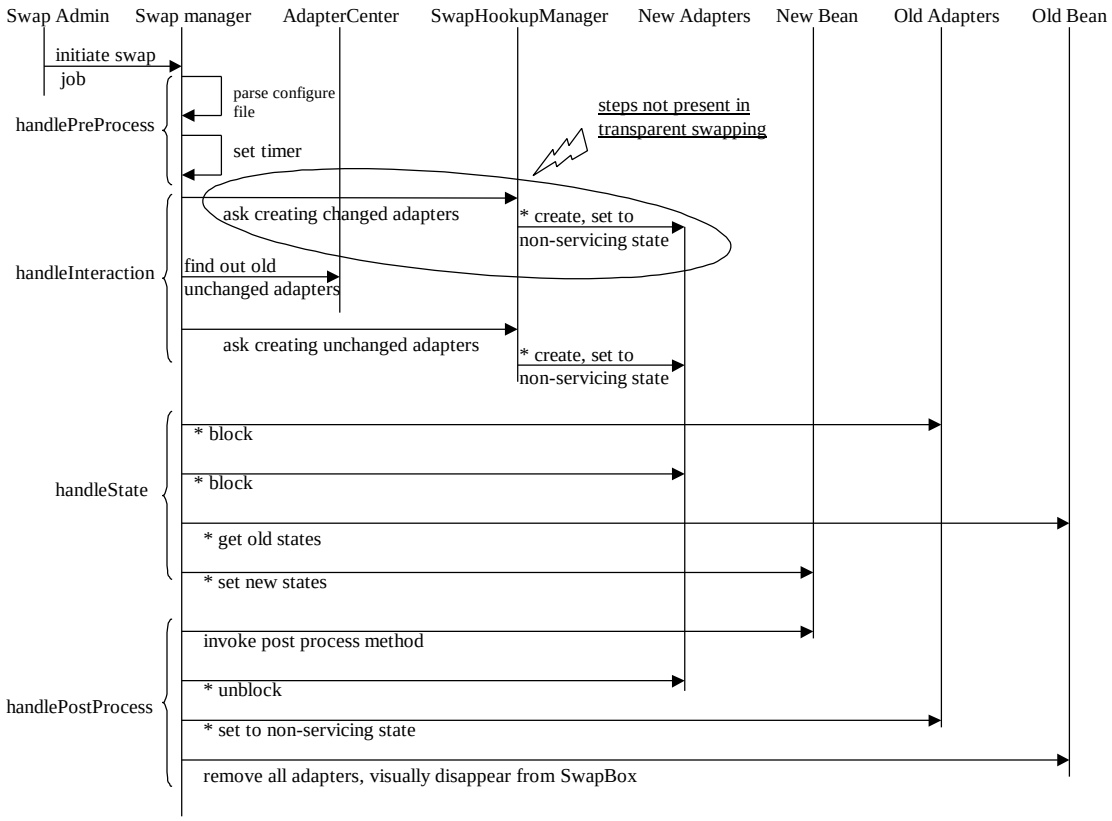

FIGURE 4-14 INTERACTION DIAGRAM FOR NON-TRANSPARENT SWAPPING

In order to "swap in" a new bean, swap manager has to create appropriate adapters for the new bean. When the case is simple, i.e., the swapping job is transparent, adapters for the

old bean are “cloned” to adapters for the new bean. SwapHookupManager is invoked to generate new adapters’ code, compile them, and load them into memory. Newly created adapters cannot work immediately after loaded into the memory. They are set to non-servicing state to discard incoming events, such that the new bean stays in idle state. After the interaction handling is finished, it is time to transfer states. From now on, the DefaultSwapManager blocks both new and old adapters so that new service requests are queued at both adapters. The DefaultSwapManager transfers states with the mapping rules discussed in section 4.5. The new adapters are unblocked after the states have been transferred and the post-process method is invoked. In the meantime, the old adapters are set to non-servicing state. The last step in event adapter manipulation is remove all the adapters attached to the old bean so that the JVM is able to garbage collect them.

4.6.3 Restarting the New Bean

When and how to restart the new bean is part of the timing problem. Many research projects adopt a rather conservative approach to addrress this problem. Stewart et al. [27], for example, sets the robot temporarily to rest (i.e., velocity and acceleration are both zero) before dynamic reconfiguration begins. Thus the new module is in idle state after replacement, waiting for the next input to behave accordingly. Feng and Ao [4, 1] require a hot swapping to begin only when the S-Module is in the idle state. The new S-Module starts execution only after receiving a new service request. In contrast, Hauptmann [10] inserts *goto* clauses into the application code to guide the execution to the restarting

point. The new actor begins execution at a fixed point, but it will jump to the appropriate restarting point with the help of the *goto* clauses.

The timing and methods of restarting the new module are related to when and how the old module is stopped. If the old module is stopped in idle state (i.e., the old bean is not handling computational tasks during the period of hot swapping), the new bean, after hot swapping, just sits there waiting for new service requests, because there is no remaining work left by the old bean. If, in a more complicated situation, the old module is stopped during execution of a method, the new module may have to go through the corresponding method to get to an appropriate point to restart new module execution.

One of the SwapManager's tasks is to restart the new bean properly. The hot swapping configure process enables the swap administrator select a void method at the new bean. This method is recorded in the XML-based hot swapping file as element *swap_method*. The swap manager invokes this method, if there is one, after the hot swapping finishes. This is similar to Hauptmann's approach. There is, however, no *goto* clause to guide the execution to a particular restarting point. The execution must be started from the beginning of the method that is selected by the swap administrator. A potential problem of this approach is that the same method may get executed twice, first in the old bean and then in the new bean. The swap administrator must use semantic knowledge of the old bean and the new bean to decide whether selecting such a method or not. If no method selected, the swap manager will just ignore this step. The new bean sits there waiting for the incoming events. This is similar to Feng and Ao's approach.

4.6.4 Swappable JavaBeans

A hot swapping system must not only be able to replace the old module with the new module efficiently, it is also expected to be as transparent as possible to both its users and programmers. The more transparent a hot swapping system is, the more likely programmers and managers are to use it. If a normal JavaBean could be converted easily to a swappable JavaBean, it will simplify the application developer's work, and more likely be adopted.

In the SwapBox, very little work is needed to make a JavaBean swappable. States transferring uses accessor methods, which is part of the JavaBean naming pattern for properties (the use of swapGet and swapSet accessor methods is only a supplement for the default approaches. They are easy to implement). Interaction handling is realised with dynamically-created event adapters. There is no need to write extra code to wrap up JavaBean for reference indirection. Different swap managers, however, may have particular requirements on what a swappable JavaBean is. The DefaultSwapManager requires a void method that could be invoked after the hot swapping transaction, if the old module is stopped in busy state. The Option1SwapManager needs JavaBean to implement *isIdle* interface so that it can detect when the old bean is idle. These requirements are associated with particular swap managers. The Option2SwapManager has no such requirements. Generally speaking, the modifications required to make a JavaBean swappable are negligible.

It is worth noting that just because a bean is swappable does not mean the bean can be swapped out at arbitrary time. The problem of determining appropriate points at the running applications to begin a hot swapping is beyond the scope of this thesis.

This chapter elaborated the design and implementation of the SwapBox. A hand of problems, including reference indirection, XML-based hot swapping policy, interaction handling, state transferring, and the SwapManager, was addressed. The next chapter describes two sample applications and the tests made to evaluate the SwapBox.

Chapter 5 Experiments

In order to evaluate the SwapBox, two test applications were developed. One is Conway's game of life [37]; the other is a sorting application. The former is used to test the basic functionality of the SwapBox, while the latter is used to show different behaviours when different swapping strategies (i.e., the DefaultSwapManager, the Option1SwapManager, and the Option2SwapManager) are applied to hot swapping.

5.1 Conway's Game of Life

Conway's game of life is played on a grid of square cells which continue infinitely in every direction. A cell can be live or dead. A live cell is shown by a marker on its square. A dead cell is shown by leaving the square empty. Each cell in the grid has a neighbourhood consisting of eight cells in each direction, including diagonals. Once the "pieces" have been placed in the starting position, the rules determine everything that happens subsequently.

GameBoard is a JavaBean which implements the game of life. It consists of a visual frame bean called the GameBoard and two button beans. The GameBoard is divided into cells, and is able to display the changing graphics periodically according to the state of each cell. The GameBoard bean determines the state (i.e., white or black) of each cell based on the current state of its neighbouring cells. A set of rules is applied to make the determination. The GameBoard bean implements *Runnable* interface. A separate thread is

responsible for calculating the state and updating the board. Two void methods, *start* and *stop*, are provided to activate and deactivate the thread. The updating rate, number of cells, running state, and the board width are exposed as JavaBean properties. The array holding state for cells is exported with the *swapGetStatus* and *swapSetStatus* methods. No events are fired out from the GameBoard bean. The start button fires out an ActionEvent (which is a core Java class) to the *start* method at the GameBoard bean. The stop button fires out another ActionEvent to the *stop* method at the GameBoard bean. Pressing the start button causes the GameBoard bean to begin changing periodically; pressing the stop button causes the alternation to stop.

For the experiment, two new beans are developed. One is NewGameBoard1; the other is NewGameBoard2. The NewGameBoard1 bean has no changes with regard to the interface; only a method implementation is changed. It displays the cell in colour rather than in black and white. The colour is randomly selected. The hot swapping of the GameBoard with the NewGameBoard1 is transparent. The NewGameBoard2 bean changes its interface. It replaces the method *start* with *newStart*. Both methods are implemented the same way. Since the method *start* belongs to the active interface of the GameBoard bean, replacing the GameBoard with the NewGameBoard2 is a non-transparent hot swapping. Figure 5-1 is a snapshot of the SwapBox when the old GameBoard bean (at the left side of the box) and the new NewGameBoard1 bean is selected. There is a rubber hand extended from the old bean to the new bean so that the swap administrator can identify them.

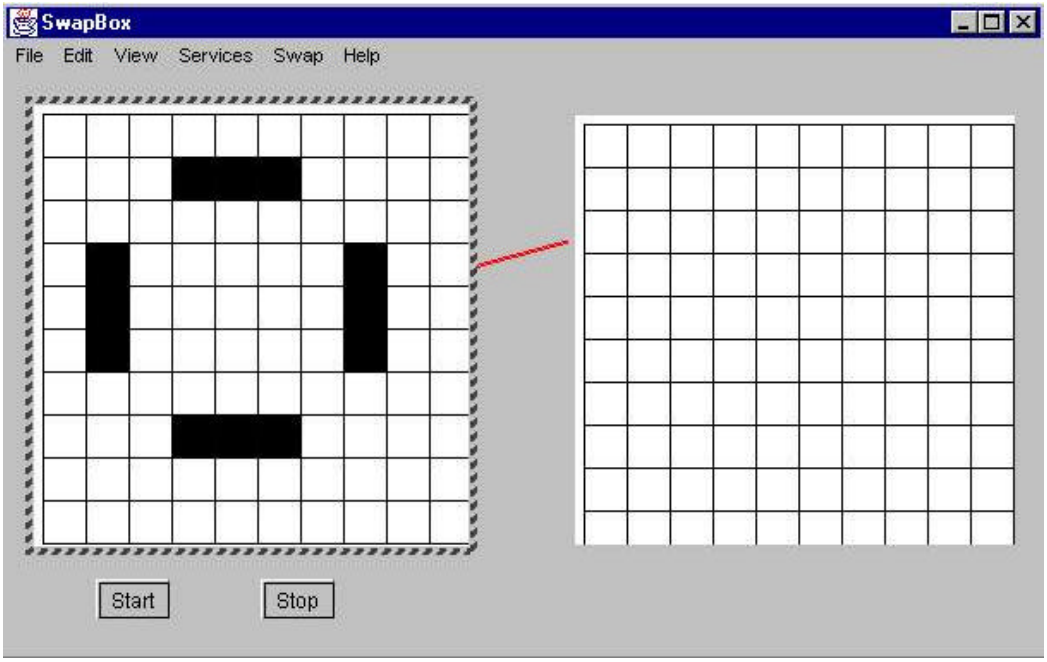

FIGURE 5-1 SNAPSHOT OF THE SWAPBOX WHEN A HOT SWAPPING TAKES PLACE

Three hot swapping tests are carried out on the GameBoard bean. In all of the tests, the GameBoard bean is playing when the hot swapping takes place. Updating rate, number of cells, running state, board width, and status array are states transferred between versions. The tests are:

1. Test with the DefaultSwapManager where NewGameBoard1 is the new bean. This is to test the basic functionality of the SwapBox, to see if it is able to hot swap a bean.
2. Test with the DefaultSwapManager where NewGameBoard2 is the new bean. The altered interaction is ignored, i.e., the hot swapping policy file does not specify to which method the start button is sending its event. This is to test the SwapBox's ability to handle decremental interface change.
3. The third test is the same as test 2 with one small change. In the hot swapping policy file, the altered interaction is re-configured so that the start button sends the event to

the *newStart* method. This is to test the SwapBox's capability to reconfigure applications at the time of upgrading.

The first step in hot swapping is to configure the hot swapping policy file. With the help of a set of GUIs, the swap administrator is guided through the process. For test 1, the DefaultSwapManager is selected. There is no need to have a method running after the hot swapping has taken place. Therefore the post_process method is selected as NULL. The time constraint is defined to be large enough to carry out a swapping. The state transfer is selected by connecting the states with the same name. There is no altered interaction. The policy is saved into a file after the configuration is complete. During hot swapping, the GameBoard and the NewGameBoard1 are selected as the old and new beans. The hot swapping policy file which has just been saved is also selected. Hot swapping begins now.

The test shows that the DefaultSwapManager successfully replaces the GameBoard with the NewGameBoard1. The GameBoard is playing before it is removed from the Swapbox. As soon as the GameBoard disappears, the NewGameBoard1 begins displaying. When the start/stop button is pressed, it starts/stops the NewGameBoard1. It is worth noting that the swappability in the busy state is application-specific. Not all JavaBean applications are able to support this capability. In the GameBoard bean example, it is the interval between the updating of the GameBoard display that enables the GameBoard bean to be swapped out even though it is playing (the GameBoard bean is actually idle in the interval).

Test 2 has the same configuration process as test 1. The changed interaction (i.e., ActionEvent to *start* method) is simply ignored. After hot swapping, the GameBoard disappears from the SwapBox, and the NewGameBoard2 begins displaying. When the stop button is pressed, the NewGameBoard2 stops playing. However, when the start button is pressed, the NewGameBoard2 does not start playing. This is because no re-configuration was made for the changed interaction in the hot swapping policy file. The interaction is just lost. This test demonstrates that the SwapBox can handle decremental interface changes.

For test 3, the changed interaction is re-configured to send to the *newStart* method at the NewGameBoard2. The test result is like that of test 1. When the start/stop button is pressed, the NewGameBoard2 starts/stops displaying the board. This demonstrates that the SwapBox can, based on information stored in the AdapterCenter and the hot swapping policy file, re-configure appropriate interactions between the new bean and old bean's partners, even though the interface is changed. However, such capability depends on the semantic knowledge of both versions. The swap administrator must know which method at the new bean is able to substitute the missing method of the old bean.

5.2 A Sorting Application

The GameBoard application only tests the basic functionality of the SwapBox. Due to the nature of the GameBoard program (i.e., it sleeps every hundred microseconds before updating the display), it cannot be used to test how different concrete swap managers deal

with the timing problem. Recall section 4.6, which stated that the DefaultSwapManager begins hot swapping immediately upon getting the request, no matter whether the old bean is busy or not. The Option1SwapManager waits until the old bean has finished its current job before beginning. The Option2SwapManager blocks incoming events to the old bean and forwards them to the new bean. In the GameBoard example, hot swapping with different swap manager exhibits the same behaviour, i.e., GameBoard is replaced immediately with either the NewGameBoard1 or NewGameBoard2 bean.

Another example application was developed to test this and to further demonstrate the busy state problem. This is a sorting application composed of a sorting bean and a GUI bean. The GUI bean generates random data set (the record size is input by the users) for sorting, and sends the request to a sorting bean. The sorting bean sorts the data and sends back the sorted data to the GUI bean for display. The old version of the sorting bean implements bubble sort algorithm, whilst the new one implements quick sort algorithm. Both sorting beans implement the BeginSortingListener, which declares only one method:

public void sort(SortingEvent e);

The GUI bean fires out a SortingEvent containing unsorted data. The sorting beans fire out a SortingDone event containing sorted data. The GUI bean implements SortingDoneListener interface, which declares only one method:

public void update(SortingDone sd);

In order to compose a sorting application, the GUI bean's SortingEvent is connected to the sorting bean's *sort* method, and the sorting bean's SortingDone event is connected to

the GUI bean's *update* method. Both the bubble sort and quick sort beans have an array to receiving incoming unsorted data. The sorting is done on this array. After sorting is finished, the array is used to compose the SortingDone event to update GUI bean. The quick sort bean makes no change to the bubble sort bean's active interface. Hence, the hot swapping is transparent.

There are five tests made to hot swapping bubble sort bean with quick sort bean. In all tests, the bubble sort bean is busy with sorting when the hot swapping occurs. The tests are:

1. Test with the DefaultSwapManager. This is a basic test.
2. Test with the Option1SwapManager where no new event arrives when the old bean is working on its current task. This is to the test basic functionality of the Option1SwapManager.
3. Test with the Option1SwapManager with one new event arriving when the old bean is working on its current task. This is to test the queuing capability of adapters.
4. The same as test 2 except the time constraint is very short, so that before the old bean can finish its current task, the time is up. This is to test that the time constraint works and that the SwapBox can roll back when an exception occurs.
5. Test with the Option2SwapManager with one new event arriving when the old bean is working on a task. This is to test the forwarding capability of the Option2SwapManager.

For test 1, the hot swapping configuration process is the same as that in the GameBoard example. However, this time a method is needed to restart the sorting after hot swapping. Therefore, a void method named *swapMethod* is selected as the post_process method. This method simply initiates sorting at the quick sort bean. During upgrading, the DefaultSwapManager copies the array, which is partly sorted, to the quick sort bean and restarts the sorting job at the quick sort bean. It is the quick sort bean rather than the bubble sort bean which delivers the SortingDone event to the GUI bean. Figure 5-2 shows a comparison of how much time is used to sort an integer data set with 30,000 records. JDK 1.3 is used. The hardware is an Intel workstation with 650 MHZ CPU and 128M RAM. It shows that a hybrid sort (i.e., a sort within which the hot swapping takes place) spends some time between the time used for the bubble sort and the quick sort. This is because this type of sort is partly done by the bubble sort bean and partly by the quick sort bean. After the bubble sort bean has finished part of the sorting job, the partly sorted array is transferred to the quick sort bean, which is very efficient compared to the bubble sort bean, to finish the remaining work. The reading of this item will be changed in a large range. It is partly determined by how fast a swap administrator does the hot swapping after the sorting begins. The faster it is, the lower the value.

	Record Size	Time (mscs)
Bubble Sort	30000	18,345
Quick Sort	30000	20
Hybrid Sort	30000	12,629

FIGURE 5-2 COMPARISON OF TIME SPENT IN SORTING

Test 2 has the same configuration as test 1. However, this time the Option1SwapManager is selected and no post_process is needed because the old bean is allowed to finish its current task and the new bean just waits for the next sorting event. Unlike test 1, test 2 shows there is no hybrid sort. The sorting task in which hot swapping occurs is completely sorted by the bubble sort bean, while the next sorting task is done by the quick sort bean. This can be recognised by reading the time taken to sort (the time the bubble sort bean takes to sort the same amount of records is much larger than the quick sort bean).

In test 3, the Option1SwapManager is selected and a new event arrives when the old bean is working on its current task. The new event is expected to be queued in both beans' adapters so that it will be directed to the new bean (if hot swapping succeeds) or the old bean (if hot swapping fails). In the test, the end user generates another sorting request after hot swapping begins and the bubble sort bean is working on the task. There is no instant reaction to this request. After the sorted data from the bubble sort bean is displayed, another piece of sorted data from the quick sort bean is displayed.

Test 4 is a non-functional test. The test is used to demonstrate that the SwapBox is capable of rolling back the old bean if hot swapping fails. The test shows that when the time constraint is up and the hot swapping is not yet finished, an error notice dialog box pops up, saying that the hot swapping fails because of the time limit. The GUI bean is still connected to the bubble sort bean. All the hot swapping work carried out previously

(such as the generation and registration of event adapters at the new bean) is abandoned. The normal operation of the old application is not affected.

Test 5 is to test the Option2SwapManager. The configuration process is the same as that in the Option1SwapManager. After hot swapping the bubble sort bean, a notice dialog box pops up, saying that the hot swapping has succeeded. From now on, when the end-user generates a sorting request, it is delivered to the quick sort bean instead of the bubble sort bean. The bubble sort bean is working on its current task when the quick sort bean accepts new request. In the test, the bubble sort bean is working on a 60,000-record sorting task when hot swapping occurs. Another 30,000-record sorting request is sent out after hot swapping, which is delivered to the quick sort bean. Since the quick sort is very efficient, the GUI bean first displays the sorted data for the 30,000-record request, then displays the sorted 60,000-record array when the bubble sort bean finishes it. Test 5 also tries to connect a SortingEvent fired out from another GUI bean to the bubble sort bean after hot swapping occurs. Because the bubble sort bean cannot accept incoming events after being swapped out, such an attempt is expected to fail. In the test, an error dialog box appears, saying that the bubble sort bean is swapped out and the new bean is the quick sort bean.

In the sorting example, the choice of swap manager affects how the sorting application runs during upgrading. The swap administrator must select a swap manager which best serves the application's need at the time of hot swapping. This is why the SwapBox

provides different swapping strategies and allows for future research to add other strategies.

In the tests above, the SwapBox times the hot swapping. A hot swapping time is the time used to execute the *swap* method at the SwapManager. It includes the time used to set the timer, generate event adapters for the new bean, carrying out the state transfer, and invoke the post-process method. The readings for hot swapping times in above tests ranged from 1 second to about 30 seconds. Figure 5-3 gives how much time used for each test case described in the previous section. In the figure, Test 1.1 refers to the first test case for game of life application, while test 2.1 refers to the first test case for sorting application, and so on. The third column is the type of SwapManager, i.e., DefaultSwapManager, Option1SwapManager, and Option2SwapManager. The fourth column is the number of adapters generated at the time of hot swapping. In all of these test cases, only one bean is swapped out.

	Time (mscs)	SwapManager Type	Num. Of Adapters
Test 1.1	2534	DefaultSwapManager	2
Test 1.2	1252	DefaultSwapManager	1
Test 1.3	2423	DefaultSwapManager	2
Test 2.1	2524	DefaultSwapManager	2
Test 2.2	17815	Option1SwapManager	2
Test 2.3	28831	Option1SwapManager	2
Test 2.4	N/A *	Option1SwapManager	2
Test 2.5	2483	Option2SwapManager	2

* In test 2.4, the hot swapping fails due to short time constraint, there is no time reading.

FIGURE 5-3 HOT SWAPPING TIME IN EACH TEST CASE

It could be seen that the hot swapping time varies a lot. The reasons for such a big range are as follows:

1. For DefaultSwapManager and Option2SwapManager, when conditions are the same, the time used to generate event adapters is determined by the number of adapters to be generated, and the speed at which the disk is accessed (a network disk is slower than a local disk). The more adapters which must be generated, the more time is needed.
2. For the Option1SwapManager, if the old bean is not idle, the executing thread for the *swap* method will wait until either the old bean gets into the idle state or the time constraint is reached. The waiting time contributes a great deal to the hot swapping time. Indeed, high hot swapping time readings (i.e., readings of Test 2.2 and Test 2.3) are all due to the use of the Option1SwapManager.

It is worth noting that the old bean is blocked for a very tiny part of the hot swapping time. Recall Figure 4-14, which showed that the old bean is blocked only after adapters for the new bean have been generated. When the old bean is blocked, the swap manager is carrying out state transfer and the post-process method invocation. Although these actions use Java reflection, it is still a faster process than generating the event adapters, which requires accessing the disk. In fact, the number of event adapters to be generated determines the hot swapping time for the DefaultSwapManager and the Option2SwapManager. For the Option1SwapManager, because of the reason 2 listed above, the hot swapping time is unbounded as long as it does not exceed the maximum time constraint given in the swapping policy file. In order to clarify how much time the old bean is unavailable, an additional test is made. This test extends the first test case made to the game of life application. The DefaultSwapManager is used, and NewGameBoardBean1 is the new bean while the GameBoardBean is the old bean. For test 1, no event adapter is connected to the GameBoardBean; for test 2, one event adapter is connected to the GameBoardBean, and so on. Figure 5-4 gives the result.

Num. Of Adapters	0	1	2	3	4
Time (mscs)	50	1251	2473	3605	4807

FIGURE 5-4 TIME FOR ONE TEST CASE WHEN NUMBER OF ADAPTERS ARE DIFFERENT

It is obvious that when no adapter is connected to the old bean, there is only a very short time needed to do the hot swapping (i.e., 50 mscs). This is the time in which the old bean is unavailable. Along with the increment of the adapters, the hot swapping time is

increased as well, in a linear way. The time used to generate the event adapters does not affect the old bean's availability.

Chapter 6 Conclusions

6.1 Conclusions

Software hot swapping reduces the cost and risk of updating software programs on the fly. This thesis proposed a new hot swapping infrastructure for event-driven, adapter-connected JavaBean applications. The new infrastructure allows implementation, interface, and data structure change between versions. It highlights the role of event adapters, which provide an address reference indirection between JavaBeans. This reference indirection enables hot swapping. The granularity of the replacement is based on JavaBean components. Chapter 2 presents state-of-the-art work in hot swapping research. A set of common issues faced when designing hot swapping systems and the general procedures for hot swapping are laid out in the same chapter. Sun Microsystems' BeanBox is introduced in Chapter 3, with a particular focus on its event communication feature. Chapter 4 describes the design and implementation of the SwapBox, which is an extension to the BeanBox. Two applications are developed in Chapter 5 to test the functionality of the SwapBox.

The SwapBox incorporates the BeanBox with the new hot swapping infrastructure. It is a running environment and swapping management tool for swappable JavaBeans. The swappability of a JavaBean is determined by the type of hot swapping strategy that is applied to hot swapping transactions. A JavaBean could easily be converted to a

swappable JavaBean with little or no extra work. This simplifies development for swappable JavaBean applications. An XML-based hot swapping policy file is proposed. The policy file contains information configuring a particular hot swapping job. The benefits of introducing a hot swapping policy include flexibility, more structured management, and reduction of human intervention, which may cause mistakes during on-line upgrading. The thesis also proposed a state transferring mechanism using accessor methods plus mapping rules. The design of the SwapBox follows Object-Oriented approach. Two design patterns (i.e., strategy pattern and state pattern) are used. This enables an easy extension of the SwapBox to incorporate new hot swapping strategies in the future research. Figure 6-1 gives a comparison on S-Module approach and the approach proposed in this thesis.

	S-Module	SwapBox
Granularity	Java Class	JavaBean
Reference Indirection	S-Proxy: provided by developer	Event Adapters, generated automatically
Transferring States	Setter approach proposed but not implemented	Accessors + Mapping Rules
Levels of Changes	New S-module must be a subclass of the old one	New Bean can be of any type
Timing Problem	When the old S-Module is idle	Provides three strategies
Performance	Every method invocation to the new method after hot swapping has to go through Java reflection.	Java reflection only used at the time of hot swapping. New method invocation does not need Java reflection
Extra Memory Usage	Need extra memory for one S-Proxy	Need extra memory for multiple event adapters

FIGURE 6-1 COMPARISON ON S-MODULE AND THE SWAPBOX

From the figure above, it could be seen that the SwapBox improves over the S-Module in the following areas.

- Has an implemented solution to transfer states between versions.
- New bean does not inherit from the old bean, while the new S-Module has to inherit from the old S-Module.
- Better performance when invoking new methods provided at the new bean but not provided at the old bean. Java reflection is not needed to invoke such methods, while in S-Module approach such invocation has to go through Java reflection.

However, the S-Module is better than the SwapBox in that there is only one S-Proxy attached with the S-Module. Therefore it does not consume too much extra memory usage. In addition, the S-Module approach can be easily applied to distributed environment, while the SwapBox has to make some modifications in order to apply to the such environment. Both the S-Module and the SwapBox do not solve the "busy state" problem (i.e., how to elegantly handle hot swapping when the old module or old bean is busy) very well.

6.2 Contributions

The contributions of this thesis are as follows:

1. A hot swapping infrastructure for event-driven, adapter-connected JavaBean applications has been proposed.
2. SwapBox [39], a running environment and swapping management tool for hot swappable JavaBeans, has been developed.

3. A state transfer mechanism has been proposed and implemented within the SwapBox. The use of this mechanism is not restricted to the SwapBox. It could be used in programming languages with reflective capability.
4. An XML-based hot swapping policy file has been proposed. It provides more structured hot swapping management, and reduces human involvement during system updates.
5. The SwapBox is designed as a framework, which provides diversified hot swapping strategies to swappable applications and could be extended in future research to accommodate new hot swapping strategies.

6.3 Drawbacks and Limitations

Due to the inherent difficulty of the hot swapping problem, the SwapBox cannot expect to solve the problem once and forever. It has some drawbacks and limitations. Following are two of them

6.2.1 Extra Memory Usage

Unlike proxies in Feng and Ao's work, a swappable bean may have many event adapters. These event adapters need extra memory usage. If there is a large number of adapters, the extra memory usage will be substantial. This drawback comes from the selection of event adapters for reference indirection. In normal JavaBean applications, extra memory used by event adapters is not expected to pose a serious problem to the virtual machine because their code is very short and the number of adapters is not large.

6.2.2 Scale to Distributed Environment

The SwapBox architecture forces beans to communicate each other via a set of event adapters. These event adapters, or interactions, have to be established before the application is run. At runtime the interactions are fixed. This structure precludes a bean from selecting where to send an event at runtime. Therefore the SwapBox cannot scale to distributed client/server model without modifications. The AdapterCenter is another aspect which prevents such scaling. Recall section 4.4.3.1, which stated that the AdapterCenter has entries for all interactions, consisting of references to source bean, target bean, and event adapter. All the references are valid only in the same JVM holding the new and the old bean. Any attempt to separate the source and target bean into different hosts inevitably brings invalid references into the AdapterCenter.

A possible remedy to this limitation is to re-organise the SwapBox architecture. If a server application is developed using JavaBeans, it can be divided into swappable and non-swappable beans. Swappable beans do not directly expose to clients. They are performing computational or database access tasks behind the scenes. Swappable beans communicate with each other through event delivery. Non-swappable beans are directly exposed to clients. They provide an accessing interface to clients through sockets or Java RMI. Meanwhile, they interact with swappable beans at the server side through event delivery. In this way, the server application has the capability to hot swap swappable beans, while keeping non-swappable beans intact. Since the main functionality of non-

swappable beans is to communicate with clients, it is possible to keep their implementation simple so that there is little chance to update them at runtime.

6.4 Suggestions for Future Work

Considering the inherent challenges of hot swapping research, this thesis is far from complete. There are many areas that deserve further investigation. Here is a short list specifying these areas:

1. XML for persistence. The state transferring process should ideally be automated, so that the old bean saves its states and the new bean is able to pick up what it is interested in. This requires that the old bean and the new bean save states in a format that is mutually understandable. In other words, a data format independent of the program is needed. XML is potential choice. In such a circumstance, a bean has an XML format to describe its data structure. When requested to save its state, a bean could generate an XML-based states file. Meanwhile, beans could load states from an XML-based states file. For each pair of new and old beans, an XSLT (Extensible Style Sheet Language: Transformations) [42] file is needed to bridge them, i.e., translate data structure format between them. How to define the structure of the XML file, how to save bean states into XML file, and how to create the XSLT file for translation is open to research.
2. Real applications. This thesis presents two relatively simple applications. They are enough to test ideas and the prototype but not sufficient for the real world. It is suggested that JAIN [35] may be a rich pool of real, complex hot swappable applications. JAIN uses the JavaBean model to provide across-vendor services to

telecommunication subscribers. A call control component is probably a target for hot swapping because it is mission critical and dynamic. Hands-on experience of these real applications would solidify hot swapping research and shed light on problems that have not been found yet.

3. General appeal of the SwapBox. Although the techniques used to address the hot swapping problem are closely associated with domain requirements and program structures, it is still desirable to take the new infrastructure proposed in this thesis beyond JavaBean. There are many component models out there, such as COM, DCOM, CORBA, .NET, and so on. It would be interesting to investigate how solutions proposed here could be applied to other component models.

References:

[1] G. Ao, Software *Hot-Swapping Techniques for upgrading Mission Critical Applications On the Fly*, Master Thesis, System and Computer Engineering Department, Carleton University, February 2000

[2] K.Brockschmidt, *Inside OLE 2*. Microsoft Press, 1994

[3] R.S. Fabry, *How to Design a System in Which Modules Can be Changed On the Fly*, in Proceedings of the 2nd International Conference on Software Engineering, 1976

[4] N. Feng, *S-Module Design for Software Hot Swapping*, Master Thesis, System and Computer Engineering Department, Carleton University, September 1999

[5] M. Franz, *Dynamic Linking of Software Components*, IEEE Computer, March 1997, pp. 74-81

[6] A. Goldberg, and D. Robson, *Smalltalk 80: The Language and its Implementation*, Addison Wesley, 1983

[7] M.M. Gorlick, R.R. Razouk, *Using Weaves for Software Construction and Analysis*, Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991

[8] H. Goullon, R.Isle, and K. Lohr, *Dynamic Restructuring in an Experimental Operating System*, IEEE Transactions on Software Engineering, vol. SE-4, no. 4, pp. 298-307, July 1978

[9] D.Gupta, P.Jalote, G. Barua. *A formal framework for on-line software version change*. IEEE Transactions on Software Engineering, vol. 22, no. 2, pp. 120-131, February 1996

[10] S. Hauptmann, and J. Wasel, *On-line Maintenance with On-the-fly Software Replacement*, in Proceedings of the 3rd International Conference on Configurable Distributed Systems, pp. 70-80, 1996

[11] W.W. Ho and R.A. Olsson, *An Approach to Genuine Dynamic Linking*, Software Practice and Experience, April 1991, pp. 375-390

[12] J. Hopkins, *Component Primer*, Communications of ACM, vol. 43, no. 10, pp. 27-30 October 2000

[13] J.Kramer, J. Magee, *The Evolving Philosophers Problem: Dynamic Change Management*, IEEE Transactions on Software Engineering, Vol. 16, No 11, November 1990

[14] S. Malabarba, R. Pandey, J. Gragg, E, Barr, and J. Fritz Barnes, *Runtime Support for*

Type-Safe Dynamic Java Classes, in Europe Conference on Object-Oriented Programming 2000 (ECOOP 2000), LNCS 1850, pp. 37-361

[15] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1996. http://www.omg.org/corba/corbiip.htm

[16] B. Oki, M. Pfluegal, A. Siegel, and D. Skeen, *The Information Bus – an Architecture for Extensible Distributed Systems*, ACM Operating Systems Review, 27(5), pp.58-68, December 1993

[17] P. Oreizt, N. Medvidovic, R. N. Taylor, *Architecture-Based Runtime Software Evolution*, in Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), pp. 177-186, 1998.

[18] D.L. Parnas, P.C. Clements, and D.M. Weiss, *The Modular Structure of Complex Systems*, IEEE Transactions on Software Engineering, vol. 11, no.3, pp. 259-266, Mar. 1985

[19] J. Peterson, P.Hudak, G.S. Ling, *Principled Dynamic Code Improvement*, Yale University Research Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997

[20] S.R. Schach, *Software Engineering*, second edition. Asken Associates, 1993

[21] M.E. Segal, and O. Frieder, *On-the-fly Program Modification: Systems for Dynamic Updating*, IEEE Software, pp. 53-65, March 1993

[22] M. Serrano, *Wide Classes*, in European Conference on Object-Oriented Programming, Springer, 1999

[23] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996

[24] S. Slade, *Object-Oriented Common Lisp*, Prentice Hall, 1998

[25] W.P. Stevens, G.J. Myers, and L.L. Constantine, *Structured Design*, IBM Systems Journal, vol. 13, no.2, pp. 115-139, 1974

[26] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, *The Chimera II Real-time Operating System for Advanced Sensor-based Control Applications*, IEEE Trans. Systems, Man, and Cybernetics, vol. 22, no. 6, pp. 1,282-1,295, Nov./Dec. 1992

[27] D.B. Stewart, R.A. Volpe, P.K. Khosla, *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Object*, IEEE Transactions on Software Engineering, vol. 23, no. 12, pp. 759-771, December 1997

[28] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addition Wesley Longman Ltd., 1998

[29] *JavaBeans specification 1.01* http://java.sun.com/products/javabeans/docs/beans.101.pdf

[30] IBM alpha works web site, http://alphaworks.ibm.com/alphabeans

[31] D.E. Perry, A.L. Wolf, *Foundations for the Study of Software Architecture*, Software Engineering Notes, vol 17, no. 4, October 1992

[32] T. Bloom, M. Day, *Reconfiguration and Module Replacement in Argus: Theory and Practice*, IEE Software Engineering Journal, vol 8, no. 2, March 1993

[33] O. Ffieder, M. Segal, *On Dynamically Updating a Computer Program: From Concept to Prototype*, Journal of Systems and Software, vol. 14, pp 111-128, 1991

[34] C.R. Hofmeister, *Dynamic Reconfiguration of Distributed Applications*, Ph.D. thesis, University of Maryland, Computer Science Department, 1993

[35] R.R Bhat, and R. Gupta, *JAIN Protocol APIs*, IEEE Communications, vol. 38, no. 1, pp. 100-107, January, 2000

[36] JavaBean Development Kit wet site,

http://java.sun.com/products/javabeans/software/bdk_download.html

[37] http://www.math.com/students/wonders/life/life.html

Berlekamp, Conway, and Guy, *Winning Ways (for your Mathematical Plays)*, vol. 2, ISBN 0-12-091152-3, 1982

[38] E. Gamma, R.Helm, R.Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, 1995

[39] L. Tan, B. Esfandiari, and B. Pagurek, *The SwapBox: A Test Container and a Framework for Hot-swappable JavaBeans*, in Proceedings of the WCOP 2001 work-shop at ECOOP 2001 (Budapest, Hungary, June 2001). On-line at: http://www.research.microsoft.com/~cszypers/events/WCOP2001/Esfandiari.doc

[40] http://www.madkit.org/

[41] Stiemerling, Oliver; Hinken, Ralph; Cremers, Armin B.: *The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware*, in Proceedings of EDOC'99, IEEE Press, pp. 106-115, 1999

[42] W3C Recommendation for the XSLT, on-line at: http://www.w3.org/TR/xslt