INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Atomic Dynamic Software Upgrades Using Software Transactional Memories

Luís Gabriel Ganchinho de Pina

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Doutor Nuno João Neves Mamede
Orientador:	Doutor João Manuel Pinheiro Cachopo
Vogais:	Doutor Luis Manuel Antunes Veiga

Outubro 2009

Acknowledgements

I would like to thank my thesis advisor, João Cachopo, for his availability, enthusiasm and encouragement. I have learned much from his experienced guidance throughout my work. His critical and careful reading significantly improved the presentation of this thesis and also improved my ability to clearly express my ideas on paper.

I am grateful to INESC-ID, specially the Software Engineering Group (ESW), for providing me with an healthy and pleasant workplace. I would like to thanks my colleagues; Hugo Rito, Ivo Anjo, Stoyan Garbatov, João Paiva and Diogo Simões; who helped me clarify many design and implementation issues throughout this work. I would also like to thank Sérgio Fernandes for his constant availability to help me using the tools that I needed to develop this work and that were new to me.

I would like to thanks the Portuguese FCT (Fundação Ciência e Tecnologia) for the scholarship granted in the scope of the Pastramy project (PTDC/EIA/72405/2006) that helped me convering my living expenses during the course of my MsC thesis.

To my mother, Conceição, and my grandmother, Idalina, I thank all the encouragement and support. A special thanks to my grandfather Manuel. Your loving memory helped to endure the most difficult parts of my work.

Finally, I would like to thank Sandra Ribeiro for her unique ability to draw a smile on my face. Thank you for all your patience and kindness, and for all the support you have given me. You have taught me how to relax during my free time, returning with fresh ideas to my thesis work.

Lisboa, October 22, 2009
Luís Gabriel Ganchinho de Pina

And now, for something completely different…
-Monty Python—The Flying Circus

Resumo

A actualização de um programa em execução é frequentemente uma operação perturbadora que implica que o programa seja parado e reiniciado. No entanto, as actualizações são operações inevitáveis. Infelizmente, as actuais soluções para actualizações dinâmicas são incompletas ou não são práticas. Sistemas com altos requisitos de disponibilidade implementam actualizações dinâmicas recorrendo a uma combinação de software específico e hardware redundante. No entanto, estas técnicas não suportam a migração do estado do programa em execução para um estado compatível com a nova versão do programa. Tipicamente, apenas migram os dados persistidos. Além disso, o processo de migração de dados pode impor uma pausa longa na execução da aplicação.

Para garantir a atomicidade da actualização e das operações de migração de dados, mas ainda permitir que se executem em paralelo com o resto do sistema, evitando portanto reiniciar o sistema ou a introdução de pausas longas, introduzo um modelo conceptual que define a semântica do sistema de actualizações, e que pode ser implementado usando qualquer STM e linguagem de programação. Além disso, o sistema de actualizações que proponho e o processo de migração de dados podem ser integrados facilmente com técnicas actuais de engenharia de software.

Descrevo também a implementação do sistema de actualizações para a plataforma Java. Implementei um protótipo do sistema de actualizações capaz de aplicar um número significativo de actualizações diferentes a uma aplicação Java sem a parar. Este protótipo representa uma prova de conceito acerca da viabilidade do sistema de actualizações que proponho.

Abstract

The upgrade of a running program is often a disruptive operation that involves stopping and restarting the program's execution. Yet, software upgrades are unavoidable. Unfortunately, current solutions for dynamic upgrades are either incomplete or not practical. Systems that have high availability requirements implement upgrades through a combination of application-specific software and redundant hardware. However, such systems lack support to migrate the state of the running program to a state that is compatible with the program that the upgrade introduces. Typically, they migrate the persisted data only and the data migration process may impose a long pause on the application.

To ensure the atomicity of the upgrade and data migration operations, but still allow them to execute in parallel with the rest of the system, thereby avoiding the restart of the system or long pauses, I introduce a conceptual model that defines the semantics of the upgrade system, and that may be implemented with any STM and programming language. Furthermore, the upgrade system that I propose and the data migration process can be integrated seamlessly with modern software engineering methodologies.

I describe an implementation of the proposed upgrade system for the Java platform that uses JVSTM. I implemented a prototype of the upgrade system that is able to perform a significant number of different types of upgrades to a running Java application without stopping it, providing thus a proof of concept that the implementation of the upgrade system is feasible.

Palavras Chave
Keywords

Palavras Chave

Memória Transaccional em Software

Actualização Dinâmica de Software

Actualização Atómica de Software

Alta Disponibilidade

Keywords

Software Transactional Memories

Dynamic Software Upgrades

Atomic Software Upgrades

High Availability

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
CLOS	Common LISP Object System
DSL	Domain Specific Languages
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine
JVSTM	Java Versioned Software Transactional Memory
OODBMS	Object-Oriented Database Management System
OPJ	Orthogonal Persistence for the Java platform
OSGi	Open Services Gateway initiative
SDK	Software Development Kit
STM	Software Transactional Memory

Chapter 1

Introduction

With the ubiquity of internet, software upgrading met a new development. The logistics of distributing applications to their customers was simplified. Instead of shipping them in a physical medium, such as a floppy disk or a CD, new upgrades can be published on a web site and the user, or the application itself, can download and install those upgrades.

Yet, upgrading software is not a trivial task. Typically, it requires restarting the application that needs to be upgraded. This method may be too restrictive for several services that run continuously and do not allow any downtime at all. On the other hand, a system that supports dynamic software upgrades can upgrade its software without stopping the provided service.

For instance, techniques such as *rolling upgrades* and *big flip* take advantage of existing redundant hardware, already used for fault tolerance and load balancing, to enable the dynamic upgrade of a system. But, typical systems also keep a state that is tightly coupled with the behaviour and that must be converted to be compatible with the new behaviour. Unfortunately, these techniques fail to support dynamic upgrades in an atomic way. With an atomic upgrade semantics, all operations submitted to the system are performed either in the previous version of the system, or in the new one, but never in an inconsistent intermediate version, a by-product of a still-in-progress upgrade process. Without an atomic upgrade semantics, programmers have the hard task of migrating the program state of a running program and assuring that there are no data corruption in the process.

The upgrade system that I propose uses a Software Transactional Memory (STM) to guarantee atomic semantics when performing software upgrades. Upgrades are performed lazily, i.e. the program state is migrated incrementally as the natural control flow touches it. Thus, it avoids any long pausing conversion operation that converts all the program state at once. Yet, it is able to provide the same semantics to the programmer writing the conversion code on most cases.

1.1 Thesis Statement

This dissertation thesis is that it is possible to implement an upgrade system that can upgrade an application without restarting it, based on programmer defined transform functions that migrate the state between two consecutive versions of the program, that uses a Software Transactional Memory to provide strong atomic semantics to the programmer writing the transform functions, and that can be integrated seamlessly with modern software engineering methodologies.

1.2 Goals

The main goal of my work is to develop an upgrade system that can upgrade the application without restarting it nor losing any part of its state due to the upgrade process. Moreover, this upgrade system must provide means for the programmer to specify how to migrate the program state from the version currently in execution to the version that the upgrade introduces.

Hicks et al [17] define several goals for dynamic upgrade systems. In an adaption of their work, I define the goals of my work.

1.2.1 Flexibility

The upgrade system must be able to upgrade any part of a running application without disrupting it.

There are several types of upgrades that an object-oriented application may suffer. For instance, a programmer may change the implementation of a method, the exported interface of a class, or even the position of a class in its hierarchy. Each type of modification raises its own issues. Therefore, the upgrade system must strive to support any possible type of modification, either isolated or used together with other types of modifications.

Furthermore, the upgrade system must not introduce any noticeable disruption to the application. Restarting the application is clearly a disruptive process that the upgrade system must avoid. But there are other sources of disruption besides the obvious restart. For instance, consider that the upgrade system migrates all the program state at once after performing an upgrade. The program state may get large, thus the migration process may impose a long pause in the application execution. This is also a disruptive process that the upgrade system must avoid.

1.2.2 Timing

The upgrade system must be able to upgrade any part of a running application at any moment, even if that part is in execution when the upgrade is performed.

The main goal states that the upgrade system must be able to convert the program state between upgrades. This raises an important question: When will the upgrade system convert the program state? To answer this question, the upgrade system must specify clear and strong semantics about when it will execute the conversion code written by the programmer.

1.2.3 Ease of Use

The upgrade system must be easy to use

The conversion code that the upgrade system requires the programmer to write may deal with several versions of the same type. This may be very confusing to the programmer. Thus, such code must be simple to write.

Moreover, the upgrade system as a whole must be pragmatic: It must integrate seamlessly in the typical software development process. When necessary, it must provide automated tools to perform repetitive and tedious tasks, that are known to be error-prone.

1.3 Structure of this document

This document is structured as follows. Chapter 2 describes two scenarios of typical applications that would benefit from dynamic upgrades and presents a series of upgrade examples around a simple geometric application theme. Chapter 3 describes the architecture of an upgradable application and proposes a framework for comparing different upgrade systems. Chapter 4 describes the state of the art on software upgrade systems, describing each approach according to the framework introduced in Chapter 3. Chapter 5 describes the solution that I propose to the dynamic software upgrade problem. Chapter 6 describes the implementation of the upgrade system that I proposed on Chapter 5. Finally, Chapter 7 presents the validation of this work, the main contributions, the greater limitations, the major challenges, future work and open issues.

Chapter 2

Motivation

The dynamic upgrade of applications is a complex problem. But before describing that problem, I must explain the reasons that make this problem relevant and interessting. To do so, I shall use two scenarios of typical applications that would take great advantage of dynamic upgrades. After describing those two scenarios that illustrate possible applications of dynamic upgrade systems, I present a simple geometric application and propose several upgrades that that application may suffer. Although simple, these examples enumerate the set of possible upgrades that an object-oriented application may suffer.

2.1 Scenarios

Upgrading software is typically a disruptive process for a running application: the application cannot be used while it is upgrading and it must be restarted after completing the upgrade. Besides, typical applications use some data that is not persisted—**transient data**—that is lost due to the upgrade process. This section describes two scenarios that help us understanding the implications of such problems related with software upgrade.

2.1.1 Server Application

As a first example, consider a server application that is used by several of users at any moment. If such an application must be restarted when performing an upgrade, the service it provides is interrupted, reducing its availability.

As pointed out by Hicks et al. [17], these server applications often overcome this availability problem with a combination of application-specific software and redundant hardware that replicates the application's behaviour (replicas). Ofter, the replicas are already present to provide fault tolerance and/or load balancing.

Brewer [7] describes some techniques that use the replicas to upgrade the application gradually without disrupting it. However, maintaining the state across all replicas when performing an upgrade is a complex and mostly unsolved problem. These applications also manipulate transient data that is lost due to the upgrade process. User sessions are a good example of transient data: often they are not persisted and the upgrade system must preserve them to be transparent to the user.

2.1.2 Desktop Application

A second example is a typical desktop application that persists most of its state. For instance, a text editor that persists the files that it edits. In these applications, usually there is some transient data that is not persisted. For instance, the text editor may be able to edit several files on separate splits. Although each file may be persisted individually, the exact state of the editor—file A is displayed in split 1 that is located on top of split 2 that displays file B—is not restored by opening all files again—only the contents of file A and B can be restored.

Typically, desktop applications must be restarted also when performing an upgrade. The user cannot use the application while it is upgrading. This is not very troublesome because the application can be upgraded when the user is not using it. But the transient data that the application manipulates is lost. To prevent that, the user may delay the upgrade. The delay that the user introduces may be very long, spanning from a day—if the user shuts down his computer daily—or more—if he hibernates his computer from day to day.

2.2 Upgrade Examples

Throughout this dissertation I shall use a series of examples to illustrate the problems that dynamic upgrade systems raise, and the respective solutions. In this section, I introduce those examples.

These examples assume that we have an application that manipulates geometric data that is able to represent points, rectangles, and squares. The examples are shown in Java.

In Section 2.2.1 I present the starting application. In the following sections—2.2.2, 2.2.3, and 2.2.4—present modifications of the basic geometric application. Please note that they are not applied in sequence. Each modification is independent and starts from the basic scenario. I shall use ellipsis to denote portions that remain equal in each example. The goal of these examples is to show different types of upgrades that an application may suffer.

2.2.1 Starting Application

In the application's first version, shown in Figure 2.1, points are represented in rectangular coordinates. The class `Point` has a method that, given two points, computes their absolute distance (modulus). Rectangles are represented by their upper left vertex, and lower right vertex. Squares extend rectangles, having a different constructor.

2.2.2 Behaviour Modification

Class `Square` delegates the area computation to its superclass, `Rectangle`. But there is a simpler area computation: the square of the length of any edge. Thus, the programmer overrides the `area()` method in class `Square`. Figure 2.2 shows this modification.

This upgrade does not change the interface exported by the modified class. Neither does it change the internal structure of the modified class.

```
class Point {
  private double x, y;

  public Point(double x,double y) {
    this.x = x; this.y = y;
  }

  double distance (Point p) {
    return sqrt( square(p.x - x) + square(p.y - y) );
  }

  String toString() { return  "("+x+";"+y+")"; }
}

class Rectangle {
  private Point topLeft, botRight;

  public Rectangle(Point topLeft,Point botRight) {
    this.topleft = topLeft; this.botRight = botRight;
  }

  double area() {
    Point topRight = new Point(botRight.x,topLeft.y);
    return  topLeft.distance(topRight)
            *
            topRight.distance(botRight);
  }
}

class Square extends Rectangle {
  public Square(Point topLeft,double side) {
    double x = topLeft.x, y = topLeft.y;
    super(topleft,new Point(x+side,y-side));
  }
}
```

Figure 2.1: First version of the geometric application example. This example is composed of three classes: (1) `Point`, which represents a point in rectangular coordinates; (2) `Rectangle`, which represents a rectangle represented by the upper left vertex and lower right vertex; and (3) `Square`, which represents a square implemented as a rectangle whose sides all have the same length.

2.2.3 Internal Structure Modification

The internal representation of the class `Point` changes: points are now represented in polar coordinates, rather than in rectangular coordinates.

Although this upgrade does not break the interface that class `Point` exports, it needs to convert old instances. We shall see the conversion method on Section 6.2.1.

2.2.4 Exported Interface Modification

The signature of method `distance` changes: instead of a method that is applied to a point, the receiver, and computes the distance to another point, the argument, it is a static method that receives the two points as arguments.

```
class Square {
  ...
  double area() {
    Point topRight = new Point(botRight.x,topLeft.y);
    return square(topLeft.distance(topRight));
  }
}
```

Figure 2.2: Behavioural modification of class `Square`. Instead of delegating the computation of the area to the parent class `Rectangle`, the method `Square:area` computes the square of the length of an edge. The ellipsis denote unchanged program code.

```
class Point {
  private double rho,theta;

  double distance (Point p) {
    double r1 = square(rho), r2 = square(p.rho);
    return sqrt( r1 + r2 + 2*rho*p.rho*cos(theta - p.theta ) );
  }

  String toString() { return  "("+rho+","+(theta*(180.00/Math.PI))+")"; }
}
```

Figure 2.3: Internal structure modification of class `Point`. Points are now internally represented in polar coordinates.

This upgrade breaks the old interface of class `Point`. As a result, all client classes of class `Point` (`Rectangle` and `Square`, in this case) must be upgraded as well.

```java
class Point {
  ...
  static double distance (Point p1,Point p2) {
    return sqrt( square(p1.x - p2.x) + square(p1.y - p2.y) );
  }
}

class Rectangle {
  ...
  double area() {
    Point topRight = new Point(botRight.x,topLeft.y);
    return  Point.distance(topLeft,topRight)
            *
            Point.distance(topRight,botRight);
  }
}

class Square extends Rectangle {
  ...
}
```

Figure 2.4: Exported interface modification of class `Point`. Method `Point.distance` becomes static and receives two arguments. Class `Rectangle` must be updated because it is a client class of the upgraded class `Point`. The ellipsis denote unchanged program code.

Chapter 3

Conceptual Model

Before describing both the existing upgrade systems and the upgrade system that I propose, it is useful to introduce some terminology that will help us to reason about upgrade systems. In this chapter, I describe the core architecture of an upgradable application. Then, I propose a framework for comparing and classifying upgrade systems according to how they solve general problems that are common to all upgrade systems. The concepts presented in this chapter establish a terminology that is used throughout this dissertation.

3.1 Software Architecture of an Upgradable Application

In this dissertation, I propose an upgrade system to a object-oriented programming language (Java). An object-oriented application may be described as a set of classes (types) whose instances interact with each other as the program runs. There is a subset of these types that the programmer writes. I assume that the programmer wants to upgrade those types only and the upgrade system that I propose supports upgrading those types only. Thus, these types are the **upgradable types** of the application. All the other types are the **non-upgradable types** of the application. As an example, all the types described in Section 2.2.1 (`Point`, `Rectangle` and `Square`) are the upgradable types of the geometric application. The non-upgradable types are, for instance, the standard Java libraries, such as `java.lang.String`.

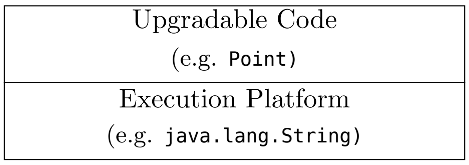


Figure 3.1: Layered view of an upgradable application. It may be seen as composed of two layers: the execution platform and the upgradable code.

For the purposes of the upgrade system, an application may be described as composed of two layers, as we can see in Figure 3.1: the **execution platform** that supports the execution of the application's code and the **upgradable code**, that contains the application itself. I assume that the upgrade system is non-upgradable because it is part of the execution platform. None of the code in the execution platform uses code from the upgradable code layer.

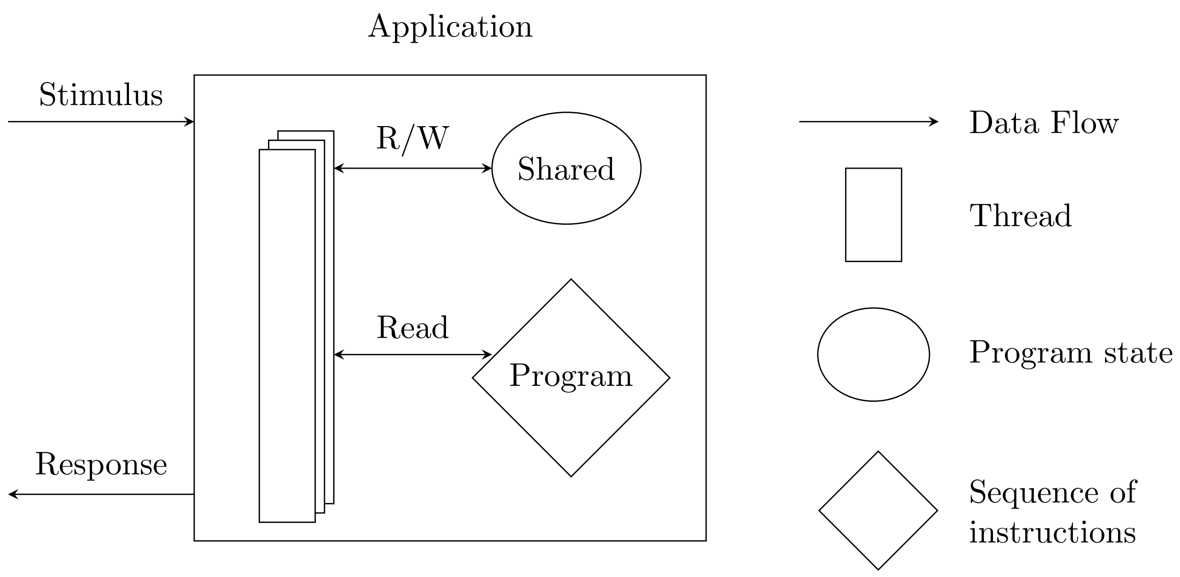


Figure 3.2: Main components of a running program.

Figure 3.2 shows the main components of an upgradable application. The application executes the **program**—a sequence of instructions that defines the behavior of the application. The application receives a **stimulus** from the exterior and reacts accordingly, producing the **response**—the result of the program execution.

The execution environment keeps the **program state**—the current state of the application. To process each stimulus, a thread accesses the program state to compute the response. All threads may access the **shared program state**.

3.2 Software Upgrades

An **upgrade** is a special kind of stimulus for a running application. It contains a new program to replace the program that is currently in execution.

Given the tight coupling of the program with the state that it manipulates, an upgrade may need to transform the program state to allow the correct operation of the new program. This transformation is done by a **transform function**—a function that initializes the new program state based on the current program state.

3.3 Upgrade System Concepts

In this section I propose a framework for comparing upgrade systems that introduces some concepts that help in understanding, classifying, and comparing the features of different upgrade systems. Using this comparison framework, I describe in Chapter 4 the existing solutions using the concepts introduced here.

The comparison framework that I propose defines a space where the upgrade systems are located. This space has three dimensions. Each dimension represents a question (problem) and each value in an dimension represents an answer (solution). The three dimensions are:

Partial Upgrades How does the upgrade system detect which portion of the program has changed with an upgrade?

Converting the Program State How does the upgrade system convert the program state between upgrades?

Executing Upgrades When does the upgrade system convert the program state?

3.3.1 Partial Upgrades

Section 3.2 defined an upgrade as a stimulus that modifies the program, replacing it by a new one. The upgrade system can take advantage of the modular structure of applications and programming languages and detect the bounds of the upgrade.

Typically, upgrades change only a portion of the program. I refer to these upgrades as **partial upgrades**. This section addresses the problem of defining partial upgrades, that is:

- Identifying the portions of the program that changed
- Defining what operation (delete, rename, user defined, ...) should be applied to each portion

For instance, an upgrade system for an object oriented programming language may detect the set of classes that an upgrade modified and replace only those classes.

There are three main approaches for detecting the portions of the program that changed: program comparison, domain specific languages, and programming language support.

Program Comparison This approach to define the program upgrade is very similar to what source code revision control systems do. The programmer works on top of the current program. When he finishes, he submits the new program to the upgrade system. Like revision control systems, the upgrade system compares the two versions of the program (the current and the new) and incorporates the changes on the current program.

With this approach, the programmer can reason about the new program without bothering about the current one, manipulating it as a whole new consistent program instead of portions of new code patched into the old program.

Domain Specific Language (DSL) Program comparison has some limitations: the upgrade system may not detect some type of upgrades. For instance, consider an upgrade where the name of a class is changed. To the upgrade system, the previous class was deleted and a new one was created. What will happen to the instances of this class? Will they be deleted?

To solve this problem, the programmer must indicate what happened. One way of doing it is with a domain specific language in which the programmer can define explicitly what portions of the program are to be upgraded and what operations should be performed on those portions.

With this language, the programmer could define completely an upgrade. Making this language small and simple facilitates its use by the programmer. The programmer is required to provide a file that describes the upgrade, written in the DSL, along with the new version of the program.

Programming Language Support The previous approach may use an external DSL [15] written in a language separate to the base language of an application. This kind of DSL has several disadvantages:

- The programmer must learn the DSL. One could argue that this language is sufficiently small and simple that it places no significant burden on the programmer, but it does not eliminate the need for a programmer to learn and use this language.
- Tools for supporting the change specification language must be developed. Even the smallest, simplest language that one can think of requires the effort of creating tools to parse it.
- Poor integration with the application's programming language. This is the worst problem of external DSLs, rendering them error-prone. A completely new language is defined, detached from the programming language used to develop the application.

All these problems could be avoided by specifying the upgrades in the base language itself. There are three approaches possible:

Define the DSL inside the application's language The programmer still needs to learn the DSL, but, with an internal DSL, the poor integration previously described does not exist anymore. The DSL is fully merged inside the application programming language.

Use the programming language to define the upgrades Some languages allow the programmer to redefine portions of a program directly, using the language itself.

Use an API defined by the upgrade system The upgrade system must define the granularity of the program that it is able to upgrade as a partial upgrade. Using an API, the program can interact with the upgrade system as it interacts with other libraries. To the programmer, the upgrade system behaves as if it was defined on the programming language itself.

3.3.2 Converting the Program State

Applications structure their data in higher abstraction level entities (abstract data types, objects, etc). Thus, as I have described in Section 3.2, when we evolve a program, we also need to convert the program state to make it compatible with the new program.

This section addresses what support the upgrade system may offer to convert the program state. It extends the model of instance conversion proposed by Dmitriev [13].

Automatic Conversion There are several situations where the upgrade system can gather enough information to make the right decision about the correct conversion of the program state.

The upgrade system can use some rules for performing default conversion. For instance, consider that a program upgrade increases the precision of an integer, changing it from 32 to 64 bits. This simple conversion is easy to perform automatically, freeing the programmer from this task.

An upgrade may also change the structure of the high abstraction level entities, inserting new data in them. In this case, the current program state does not have that new data in it. An upgrade system can initialize this data with some default value defined by the upgrade system. For instance, an upgrade system written in Java could initialize new primitive type fields to a zero like value (0 for `int`, `false` for `boolean`, and so on) and new reference type fields to `null`.

Custom Conversion The correct decision on how to convert the program state to be compatible with the new program may be arbitrarily complex. In this case, the upgrade system alone is not able to convert the program state correctly. It must provide means for the programmer to describe how to convert the program state.

Please note that, even in this case, the simple automatic conversion that the upgrade system provides frees the programmer from tedious, repetitive, and error-prone program state conversion.

3.3.3 Executing Upgrades

Consider an upgrade that is ready to be submitted to the upgrade system. If the upgrade system supports program state conversion, it must define when the program state will be converted. We shall see 3 models that define when will the conversion happen: offline, immediately or lazyly.

Offline Upgrades This upgrade model requires that the application is completely stopped (shutted down and restarted) before the new program can be put into execution. It has the following disadvantages:

- Shutting down the application means that some data cannot be saved, as it was pointed out in Section 2.1.
- The service that the application provides is disrupted during the upgrade

Despite these disadvantages, this upgrade model has a good property: the semantics of program state conversion is clear and simple. There are two distinct versions of the program and its state. There is an instant in time when the application switchs from one to another and converts all the program state. Thus, the programmer deals with two versions of the program only on the transform functions.

Immediate Upgrades Also known as *stop the world upgrades* [10, 6], this upgrade model prevents the application from executing other code while it is upgrading. It brings several benefits when compared to offline upgrades:

- The application does not shut down completely. No data is lost during the upgrade.
- The semantics of upgrading the program is also clear and simple.

On the other hand, we still have some disadvantages:

- Navigating through the program state to identify the exact portion that must be converted may take a long time.
- The service provided by the application is unavailable while the upgrade is processing.

Lazy Upgrades In this approach, once the new program is copied to the execution environment, the application can resume its execution. The program state is converted lazily: the upgrade system converts a portion of the program state only when the program tries to manipulate it.

This approach brings the following advantages:

- The disruption caused by the upgrade is minimal because no time-consuming conversion of data is needed.
- No data is lost due to the upgrading process.

This approach also introduces some disadvantages:

- The semantics of the upgrade process is not clear anymore. For instance, consider the example shown in Figure 3.3. The upgrade transaction installs an upgrade that modifies the application from the base application described in Section 2.2.1, performing an exported interface modification, as described in Section 2.2.4. Both classes `Point` and `Rectangle` are upgraded. Consider that the upgrade system uses lazy upgrade semantics, and that the application uses an instance `P` of class `Point` before using an instance `R` of class `Rectangle`. Consider also that the transform function of instance `R` uses instance `P`. When that transform function runs, the interface of instance `P` has changed. This results in runtime errors. I shall discuss the semantical implications of lazy upgrades in further detail in Section 5.2.

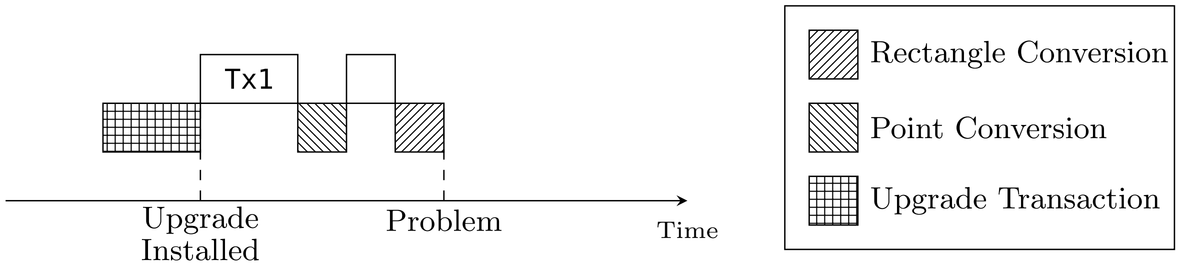


Figure 3.3: Error caused by the lazy semantics of the upgrade execution. Transaction `Tx1` uses a point `P`, triggering its conversion. Then, it uses a rectangle `R`, also triggering its conversion. The conversion code of `R` uses `P`, which has already been converted and, thus, changed its interface. This results in an error because the conversion code of rectangle `R` does not expect that new interface.

- At any instant, the application can have more than one program version. In both previous approaches, the application only dealt with two versions of the program while upgrading. After the upgrade, only one version of the program was being executed. Now, the upgrade system must cope with several versions of the program.

3.4 Discussion

In this chapter, I described the conceptual model of an upgrade system. I started by describing the software architecture of an upgradable application, then I described what a software upgrade is, and, finally, I introduced a framework for comparing upgrade systems. This framework classifies upgrade systems according to how they detect what part of the application was modified, how they enable the programmer to convert the program state, and when they perform the program state conversion.

An upgrade system that converts the program state between versions of the program must choose when will that conversion take place. It has three possibilities: (1) off-line upgrades, (2) immediate upgrades, and (3) lazy upgrades. If the upgrade system follows a lazy upgrade model, it cannot control the order in which the objects are converted. This restriction may generate some errors, as Figure 3.3 shows. This problem is harder to detect if we consider **multithreaded environments**.

Multithreaded environments are very common in modern applications. Such environments do not pose any problem for upgrade systems that follow offline and immediate upgrades because they naturally stop or suspend all threads before executing the upgrade, resuming them after all the program state is converted. However, if an upgrade systems follow lazy upgrades, it must take special care when interleaving application threads and program state conversion threads.

Consider the example shown by Figure 3.4. Transaction `Tx1` uses instance `P` of class `Point` and transaction `Tx2` uses instance `R` of class `Rectangle`. The transform function of instance `R` also uses instance `P`.

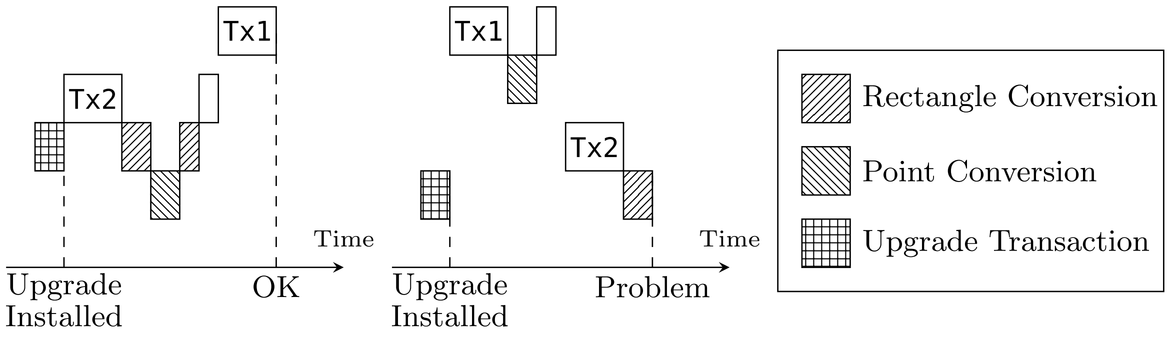


Figure 3.4: Intermittent error caused by the lazy semantics of the upgrade execution on multithreaded environments. Transaction `Tx1` uses a point `P`, triggering its conversion. Transaction `Tx2` uses a rectangle `R`, also triggering its conversion. The conversion code of `R` uses `P`. On the left side we can see a valid interleaving that converts correctly both instances `R` and `P`. On the right side we can see an invalid interleaving that fails to convert instance `R`. When the conversion code of instance `R` runs, instance `P` has already been converted. This results in an error because the conversion code of instance `R` does not expect that new interface.

If the transform function of instance `R`—triggered by transaction `Tx2`—runs before the transform function of instance `P`—triggered by transaction `Tx1`—there is no problem because the interface of instance `P` is the same that the transform function of instance `R` expects. The left side of Figure 3.4 shows this interleaving.

However, the opposite interleaving, shown by the right side of Figure 3.4, results in an runtime error because the interface of instance `P` is not the same that the transform function of instance `R` expects. The conversion of instance `P` has already been triggered by transaction `Tx1`. When the transform function of instance `R` attempts to access instance `P`, it is already converted. Thus, it is not on the program version that the transform function expected. This is the same type of error that Figure 3.3 shows. Moreover, it is harder to detect because the multithreaded environment causes this error to be intermittent and unreproducible.

Chapter 4

Related Work

In this chapter, I give an overview of the state of the art on software upgrade systems, describing each approach according to the framework introduced in Section 3.3. Moreover, I describe other work that, albeit not being an upgrade system, are related to the upgrade system that I propose.

4.1 Upgrade Systems for Object-Oriented Systems

The program state in object oriented languages consists of a set of objects. These objects have an internal state—data— and behaviour—code. Each object is an instance of (at least) one class. All the classes are arranged in a hierarchy. Each object can refer to other objects.

Considering all these aspects, we find new problems on upgrading object oriented applications. These problems have been identified by previous work on upgrade systems for object-oriented systems. This section will describe the problems found, as well as the proposed solutions.

4.1.1 PJama

PJama [13] is an experimental prototype that implements Orthogonal Persistence for the Java platform (OPJ). It also provides a tool that supports the evolution of persistent classes and instances.

PJama allows the programmer to identify the classes that must be replaced through a simple change specification language. This language identifies clearly the classes that are to be modified and describes also how the instances should be migrated. Besides this language, PJama's upgrade tool detects also all the classes that must be modified to keep the store consistent after modifying the interface of any class.

Once the upgrade is detected, the upgrade system checks if two versions of the same class—the original in the store and the upgraded—are substitutable. It follows a set of rules to perform this verification. The rules can be found in [13].

PJama's upgrade tool supports a mixture of automatic and custom instance conversion, arranged in three types:

Default Conversion Given a pair of classes, the existing class in the store and its upgrade, this conversion performs a set of simple tasks, such as copying the unchanged fields, default initialization of new fields, and trivial automatic conversions (such as widening primitive conversion, as defined in [16]).

Bulk Conversion This is a kind of custom conversion. It is supposed to be used when all the instances of any upgraded class should be converted in the same way. Other upgrade systems call these transform functions [6, 10]. When using bulk conversion, the programmer may also take advantage of the default conversion, as the upgrade system runs bulk conversion code only after performing the automatic default conversion.

Fully Controlled Custom Conversion Gives the programmer total control about how and in what order the instances are converted. The programmer writes a method `conversionMain` and provides it to the upgrade system. Then, the upgrade system will call this method, after the verification of substitutability, and will ignore any other conversion methods. The programmer gets total freedom and full responsibility for the results of such conversion.

PJama's upgrade tool supports only (at least currently) offline upgrades. All the applications that use an object store must be stopped while the upgrade tool is running. The provided upgrade tool replaces the upgraded classes and converts all their instances.

4.1.2 OODBMS

There are several commercial Object-Oriented Database Management Systems (OODBMS) that have already considered dynamic upgrading the objects contained in the data store that they persist. In his work, Dmitriev [13] presents a survey about these systems. In this section, I summarize the approaches used by each solution, as well as their major limitations.

Objectivity/DB

Objectivity/DB [2] performs custom object conversion through a reflection API to access fields of both the current and the new version of the evolving object. The documentation does not say anything about calling methods of evolving objects and recommends that conversion functions access only the objects being converted. It supports three kinds of object conversion: immediate, lazy, and on-demand, which means eagerly performing previously defined lazy conversion on a subset of objects.

There are some restrictions on custom lazy conversions: it can set primitive fields only, the conversion of an object may be repeated several times if the object is not accessed (and converted) inside an upgrade transaction, and it cannot be combined with upgrades that change the class hierarchy.

Versant Developer Suite

Versant Developer Suite [12] supports lazy conversion, although it is limited to default conversion only. It also supports arbitrary changes (except adding/dropping non-leaf classes) through immediate in a cumbersome fashion. For instance, to change a class C we must follow the following steps:

1. Create a new class D with the same definition, then run a program to create an instance of D per existing instance of C, copy information between them and repair all references of C from other objects. All this process must be coded manually.
2. Delete class C. This also deletes all of its instances.
3. Create a new class C, repeating step 1 to replace D by the new C and perform the required conversion when copying information from old class D.
4. Delete class D.

O_2

O_2 [14] is an OODBMS with one of the most sophisticated upgrade systems. It supports any modification (except adding/removing non-leaf classes) using two approaches:

- Incrementally, through primitives such as adding/deleting fields, described in a DSL.
- By comparison with a new class definition, also written in the DSL.

Conversion of instances may be performed immediately or lazily, with both default and custom conversion functions. O_2 also supports versioning of classes and instances, requiring the programmer to provide forward and backward conversion functions for each version.

4.1.3 Chimera

Chimera [4] is an object-oriented deductive active data model. Besides the concepts commonly ascribed to object-oriented data models it also provides capabilities for defining deductive rules that can be used for a variety of purposes: define views and integrity constraints, formulate queries, or specify methods to compute derived information.

Objects in Chimera can **migrate**—become direct members of a class which is different from the class from which the object has been created. Migrating an instance to a subclass is trivial given that the instance will maintain its interface. When we consider moving an instance to a superclass, consistency problems arise. Another instance may use some fields or methods that were lost. The type-safety of the application is at stake. Chimera uses two approaches to support this kind of migration:

Global Type Modification The object is modified directly, changing its state and deleting some methods or fields. All objects that had a reference to a migrated instance must be notified that the instance is no longer a member of the expected class. Using a programming language with static typing, we can avoid such problem by detecting which classes are affected by this upgrade and accept only upgrades that also make these classes compatible.

Local Type Modification The object is not modified directly, instead a new view of the object is created. In Chimera's model, the local type modification makes sense: the system is creating yet another view of the data. Nevertheless, outside Chimera, this approach adds complexity when reasoning about the upgrade, rendering the upgrade system error prone and vulnerable to data corruption.

4.1.4 OSGi Framework

The OSGi Framework [22] is a Java framework that supports the deployment and management of bundles. A bundle is the unit of modularization specified by the framework and comprises Java classes and other resources. Bundles can share packages among exporter and importer bundles. A bundle can be in one of several states (installed, resolved, starting, active, stopping, or uninstalled). The OSGi framework manages the transition between states and notifies each bundle when it is required to change its state.

The OSGi framework relies on version matching to reason about compatibility between bundles. Each import definition must specify a version (or a version range) that is matched against the version specified in the respective export definition.

The OSGi framework supports partial upgrades at the bundle level. If none of the packages exported by the old version (old exports) are used, they are removed. Otherwise, all old exports must remain available. This is how the OSGi framework supports migration of a bundle to a newer version of that same bundle. There is no bundle state conversion when performing an upgrade. The new version of the bundle initializes its state from scratch and the old version of the bundle keeps operating until all other bundles import the packages exported by the new version.

4.1.5 JavaRebel

JavaRebel [23] is a development-time tool that loads changes detected in the compiled code. It is implemented as a plugin for the JVM. To see the new behaviour, developers using JavaRebel do not have to restart the application when they change the code.

JavaRebel operates at the classloader level (I discuss the JVM classloading mechanism in further detail in Section 4.3.3). It instruments loaded classes and uses extended classloaders to propagate the changes through the application. It supports any change except replacing the superclass and changing the interfaces that a given class implements.

JavaRebel instruments loaded classes and associates them with the corresponding class file. After this step, JavaRebel keeps track of the class file's timestamp to monitor it for changes. When a change is detected, the upgrade is immediately applied and the application features instantly the new behaviour.

When performing upgrades, JavaRebel supports only automatic (default) conversion. The fields existing in both versions are kept and new fields are initialized with its default value (`0` for numerical types, `null` for reference types). This conversion is very limited: if a field changes its type to another compatible type, for instance from `long` to `int`, the old field is discarded and a new one is initialized with the default value.

JavaRebel is a useful tool for development purposes. But, the lack of support for custom instance conversion does not allow to use this upgrade system in a production system. JavaRebel is a proprietary upgrade system, developed and sold by Zero Turnaround [1]. Thus, there are not much details available on how this upgrade system is implemented.

[1]http://www.zeroturnaround.com/javarebel/

4.2 Lazy Modular Upgrades

In their work, Boyapati et al [6] describe how an upgrade system based on transform functions can provide good semantics that let programmers reason about the transform functions locally and run upgrades efficiently, both in space and time.

The authors define an **upgrade** as a set of one or more class-upgrades. A **class-upgrade** identifies a class that will be upgraded, couples it with the new class that will replace it and a transform function for instances of that class.

The upgrade system allows arbitrary modifications to the program. As a consequence, an upgrade may change the interface of a given class incompatibly (for instance, by deleting a method or changing the superclass/implemented interfaces). To maintain the program correctness, all classes that are affected by such incompatible modification must be upgraded as well.

A **complete upgrade** contains class upgrades for all the classes that need to change due to some class-upgrade already in the upgrade. This upgrade system accepts only complete upgrades. Once accepted, the upgrade becomes **installed**. Moreover, conceptually, an installed upgrade has already converted all the objects that it modifies.

This upgrade system supports custom instance conversion using transform functions. A **transform function** converts the instances from the old representation to the new representation introduced by an upgrade.

The semantics of the transform functions is simple and meant to keep it local to the class being upgraded. The programmer must write them as additional methods of each of the old classes, considering the same assumptions he did to write the old methods. Thus, a transform function accesses only the old version of the object and the old version of other instances modified by the upgrade.

The authors suggest executing transform functions lazily to run the upgrades in a time efficient manner. By assuming a transactional system model very similar to what I describe in Chapter 3, the authors deal with multithreaded environments. In their model, applications access objects within atomic transactions and transform functions are also performed inside their own transaction. They define three conditions that the upgrade system must enforce when ordering upgrade and application transactions. These conditions guarantee that transform functions encounter the interfaces that they expect.

1. Conceptually, the upgrade system executes all the transform functions when the upgrade is installed, providing immediate upgrade semantics to the programmer. However, transform functions are executed lazily, as the program flow touches the objects. The first condition states that if an application transaction executes before a conversion transaction, the results must be the same as if the conversion transaction executed before the application transaction. Thus, although the upgrades are executed lazily, the results must be the same as if they were executed immediately.

2. Consider the example shown in Figure 3.3. It shows a situation where one object `R`—an instance of class `Rectangle`—has a reference to another object `P`—an instance of class `Point`. The "immediate semantics order" to execute the transform functions would be: (1) object `R` is converted, because it has the reference to the other object being converted, and (2) object `P` is converted. The second condition states that executing the transform function of object `P` before executing the transform of object `R` must have the same results as running them in the expected order.

3. Considering the previous condition, the last condition is simple to describe. The third condition states the results of executing the transform function of unrelated objects must not depend on the order which the transform functions are executed.

Finally, the authors discuss how such an upgrade system can run upgrades efficiently in space by taking advantage of the modularity of the applications, avoiding versioning unless strictly necessary. However, in my work, I combine the upgrade system with a transactional memory that already uses versioning [9]. Thus, instead of avoiding versioning of instances, I take advantage of that versioning already provided by JVSTM.

4.3 Programming Languages Supporting Dynamic Code Changes

There are some programming languages that provide mechanisms to support dynamic code changes. Two well known examples are the Common LISP Object System (CLOS) and Smalltalk. Besides these languages, there is also a proposal to an extension to the Java programming language that enables dynamic code changes: UpgradeJ [5]. CLOS has an interesting set of mechanisms that I describe below. I shall also present UpgradeJ because it is meant for Java.

4.3.1 CLOS

The Common LISP Object System (CLOS) is an object-oriented programming language with some dynamic upgrade features. The full reference of CLOS can be found in [21], but I summarize here its most relevant dynamic upgrade features.

Upgrades are defined at the class level. The programmer upgrades a class by redefining it. CLOS detects this situation and converts all instances of the class (and all instances of subclasses of the class) without creating new instances nor changing their identity. Thus, partial upgrades are supported at the programming language level. Moreover, CLOS detects which slots were added and which slots were deleted, performing some program comparison as well.

When a programmer defines a class (a completely new class or a redefinition of an existing class) or when he creates new instances, he may also define an initialization for each slot. This automatic initialization can be used when converting instances of redefined classes. CLOS also allows custom instance conversion, giving the programmer access to the instance featuring the new structure, the added slots, and to the deleted ones and their respective value.

The exact moment when instances are converted is left implementation dependent. The standard language specification states only that converting any instance occurs no later than the next time a slot of that instance is read or written. An implementation of CLOS is free to choose between immediate and lazy upgrades. Allowing lazy upgrades introduces some problems. For instance, consider a class C_0 that is redefined twice, C_1 and C_2. Some of the existing instances may keep the structure defined by C_0 even after C_2 has been defined. The programmer must write intermediate conversion code that deals with this problem. As for multithreaded environments, the Common LISP language specification does not discuss them.

4.3.2 UpgradeJ

UpgradeJ is an extension to the Java programming language that provides language support for upgrading a Java application in a variety of ways, whereas checking upgrades for consistency with the running program. Upgrades are defined at the class level and UpgradeJ supports three kinds of upgrades:

New Class Upgrades Add a new class definition to the program

Revision Upgrades Change the method bodies of a class. The signature of all methods must remain the same and the revision class must have the same name and implement the same interfaces as the revised one.

Evolution Upgrades Add methods and/or fields to a class. The evolution must keep all the existing methods and fields, and it must also have the same name as the evolved class.

Any class may have at most one revision and one evolution. This is not a limitation because a revision/evolution is a class itself and can be revised/evolved. A programmer can define instances as upgradable. Such instances use the latest revision upgrades of their class automatically. When creating instances of a class, a programmer may ask for the latest revision of the latest evolution of that class. This way, the program can use the evolution upgrades. As for hierarchy, all upgradable instances also see revisions made to their superclasses.

UpgradeJ does not convert any program state at all. It is focused on adding new classes to the application and performing minor and major upgrades to existing classes. The program has access to all the versions of a class since they were introduced. Combining that with the required explicit version control from the programmer, UpgradeJ can avoid upgrading instances and still achieve type correctness. Nevertheless, requiring the programmer to use explicit versions may pose a big burden.

UpgradeJ may change the behaviour of upgradable instances. Such change is introduced on all upgradable instances instantly. Thus, we can say that UpgradeJ features instantly applied immediate upgrades. As for state conversion, whenever there are evolution upgrades, the developer has to instantiate a newer class, initialize this instance manually, replace the old instance by the new one and repeat this process to every instance he wants to convert.

4.3.3 Java Implementation Mechanisms

The Java platform, consisting of the Java programming language [16], the Java Virtual Machine (JVM) [20], and the standard Java libraries, does not provide native support for upgrading applications dynamically. Nevertheless, this environment has several features that can be used to implement an upgrade system without changing the Java programming language nor the JVM. I discuss such features in this section.

Classloaders

Java supports dynamic class loading using classloaders. In [19], the authors describe in detail the dynamic class loading features of Java. A classloader is an ordinary object that is an instance the class `java.lang.ClassLoader`. Assuming that a class C is loaded by a classloader L, we say that L is the **defining loader** of C. L will also be used to load all classes referenced by C. These references are actually symbolic references that C has to other classes, which are resolved at link time to actual classes.

Classes do not have to be stored in actual files. They can be stored, for instance, on memory buffers or obtained from a network stream. A classloader must be able to obtain a byte array given a binary name, as defined by [16].

The system classes are loaded by a system classloader, directly supported by the Java Virtual Machine (JVM). The class loading mechanism supports delegation: a classloader L_1 may ask another classloader L_2 to load a class C on its behalf. The use of delegating classloaders has several advantages when we consider running several applications inside the same JVM (such as running applets on a web browser).

- A class type is uniquely determined by the combination of the class name and the classloader that loaded that class. This guarantees that all system classes are unique, even if they are shared between different applications.
- Two classes with the same name loaded by different classloaders are considered different types. Thus, we can achieve isolation between applications by using different classloaders.

The classloading mechanisms can be used to implement an upgrade system. Nevertheless, there are some issues identified in [19] for which the Java classloaders do not provide any direct solution. Namely, there is no support for migrating existing instances from one class version to another.

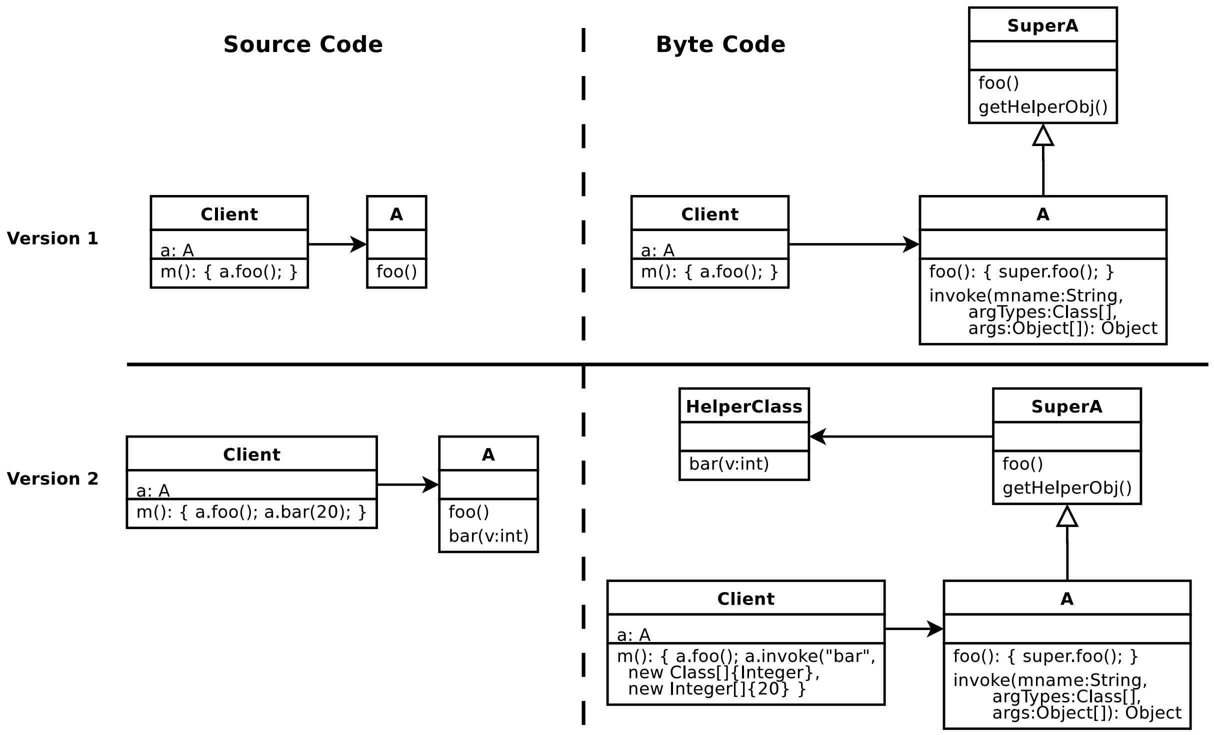
Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA)[1] provides the infrastructure needed to build end-user debugger applications for the Java Platform. The version 1.4 of the Java SDK introduces a JPDA enhancement that may be useful to implement an upgrade system for long running applications: **HotSwap class file replacement**. This feature allows a class to be updated while under the control of a JPDA debugger. Although in a future Java version it may support arbitrary changes to a class, currently the new version of the class must maintain the same signature as the old version. Currently, only method bodies can change.

Binary Refactoring

Using HotSwap technology we could define an upgrade system using the same mechanisms used for a debugger. At first glance, this approach may appear too restrictive, defeating the **flexibility** goal. However, it is possible to overcome this constraint using binary rewriting [18]. Through automated tools that change the bytecode[11, 8], we can introduce a new level of indirection without changing the program semantics. The upgrading process can be explained by the example shown in Figure 4.1.

On version 1, upgradeable class A is transformed by the upgrade system into virtual superclass and proxy. The actual implementation of method foo is defined in the superclass. All the classes that refer to A in the source code (A's clients) will refer to the proxy. Thus, we can say that the proxy keeps the identity of each instance.



Source Code
Byte Code
Version 1
Version 2
Client
a: A
m(): { a.foo(); }
A
foo()
SuperA
foo()
getHelperObj()
foo(): { super.foo(); }
invoke(mname:String, argTypes:Class[], args:Object[]): Object
m(): { a.foo(); a.bar(20); }
bar(v:int)
HelperClass
m(): { a.foo(); a.invoke("bar", new Class[]{Integer}, new Integer[]{20} }

Figure 4.1: An example of how binary refactoring can be used to overcome the HotSwap restrictions.

Then, on version 2, a new method *bar* is inserted into the source code. The upgrade system will create a new HelperClass object that will contain the implementation of *bar*. The superclass has access to all helper objects. Besides, the proxy was featured with an *invoke* method that is able to invoke methods over the helper objects. All invocations of *bar* present in *A*'s client classes are replaced by an invocation to *A*'s invoke method, specifying *bar* as the target of the invocation.

Helper objects can also be used to add new fields. In this case, we can choose from:

- Use a new helper object to contain all fields introduced by each new version
- Use a single class that contains a mapping data structure that represents all the added fields

4.4 Discussion

In this chapter, I described the state of the art on dynamic software upgrade systems. In this section, I shall summarize the state of the art that I described in further detail in this chapter.

Object oriented programming structures the program state in a set of objects that belong to a class and interact with other objects through several types of relations (inheritance, for instance). When designing an upgrade system for object oriented systems, we must deal with the specific characteristics of object oriented programming. Therefore, I described upgrade systems designed for object oriented applications. Among such systems, I described object-oriented database management systems [2, 12, 14]. These systems are able to execute some schema upgrades while the database management system is running, thus providing support for dynamic upgrades for object-oriented applications. The schema upgrade capabilities of such systems also introduces tools for the programmers to express how to migrate instances from the old schema to the new one. Besides OODBMS', I described also the upgrade capabilities of OSGi [22], which is a Java service platform that allows upgrades to happen at the service level (bundle) but without any program state conversion between the two versions of the upgraded service. Finally, I described JavaRebel [23], which is a development time tool that dynamically loads changes detected on the class files. Once again, it does not support any kind of program state conversion between upgrades.

A dynamic software upgrade system is able to upgrade a running program without stopping it, thus avoiding losing non-persisted program state and, potentially, also avoiding long pauses due to the upgrade process. Although the notion of a dynamic upgrade is easy to understand, the notion of an atomic dynamic software upgrade requires more explanation. In their work, Boyapati et al [6] describe what an atomic dynamic software upgrade is. Their upgrade system is based on transform functions, which migrate the program state between upgrades. They describe the conditions that an upgrade system must enforce to enable running the transform functions atomically.

Upgrade systems are one solution to enable the dynamic upgrade of an application. Another solution is to write that application using a programming language that already supports dynamic upgrades. CLOS [21] is an example of such type of language. In CLOS, the programmer may redefine a class, thus upgrading it. He may also migrate the program state between versions, although there is no explicit support for some operations, such as composing several upgrades. Another language that supports dynamic upgrades is UpgradeJ [5]. It is an extension to the Java programming language that provides language support for dynamically upgrading a Java application. It supports three types of upgrades, and the programmer must decompose a program upgrade into the three supported types.

The upgrade system that I shall propose on this document is meant for upgrading a Java application. Although the Java platform does not provide any native support for dynamically upgrading an application, it provides some features that can be used to implement an upgrade system which offers such support. For instance, the JVM supports dynamic class loading using classloaders [19]. The programmer can define their own classloaders, writing classes that inherit from the class `java.lang.Classloader`. Another useful feature for performing dynamic upgrades on the JVM is provided by the Java Platform Debugger Architecture (JPDA): HotSwap class file replacement [1]. This feature allows a class to be upgraded while under the control of a debugger. At the current version of JPDA, only method bodies can change. However, I describe a binary refactoring technique proposed by Kim et al [18] that is able to overcome such restriction and use the HotSwap mechanisms to perform arbitrary modifications to a class.

Chapter 5

Solution

In this chapter, I describe the solution that I propose to the atomic dynamic software upgrade problem. In Section 5.1, I introduce the concept of Software Transactional Memories (STM), and sketch an upgrade system that takes advantage of an STM to perform its task. Then, in Section 5.2, I describe the exact semantics of the upgrade system that I propose. In Section 5.3, I describe how to implement dynamic software upgrades using the limited upgrade capabilities of the JVM. Finally, in Section 5.4, I describe how to integrate the upgrade system that I propose with typical software development practices.

5.1 Software Transactional Memories

In Chapter 3, I described the major components of an upgradable application. But, if we take a closer look, we find an important issue. There are several threads manipulating the shared program state at the same time. Therefore, we require a synchronization mechanism to avoid concurrency related problems and resulting data corruption. There are several ways of doing this. Lately, much work on Software Transactional Memories (STM) propose a new language construct: a transaction. Using it, the developer groups several operations into an unit of work. Transactions are atomic: either they complete successfully and commit, or they abort. Only transactions that commit modify the shared program state.

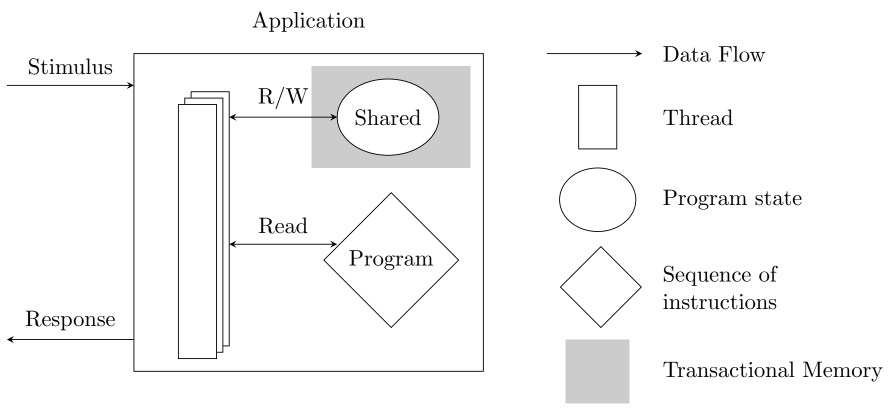


Figure 5.1: Software architecture of an application using an STM to synchronize the shared program state. Each thread executes over a version of the shared program state. The STM ensures that the shared program state version that each thread has is consistent.

Figure 5.1 shows the software architecture of an application using a STM to synchronize the shared program state. Threads access the shared program state through the STM, which is ensures that each thread has a consistent view of the program state. This means that each thread sees the shared program state as it was when the thread started. That thread does not see any modification made to the shared program state by any other thread. Thus, each thread executes over a version of the shared program state.

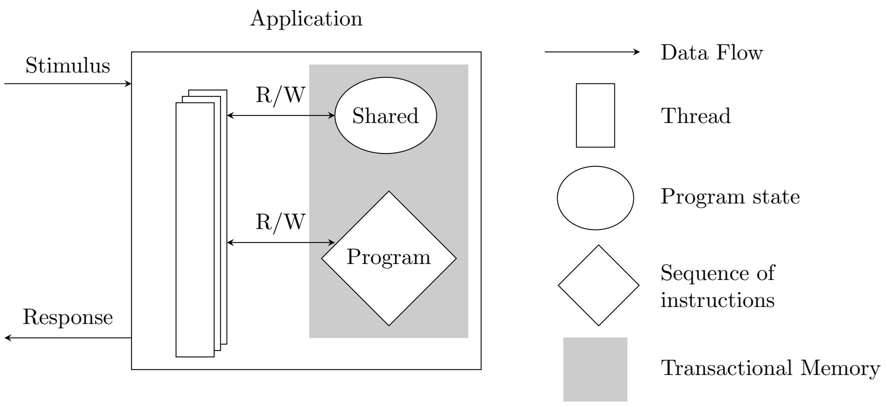


Figure 5.2: Software architecture of an application using a STM to perform dynamic software upgrades. Besides keeping the shared program state, the STM also keeps the program. Thus, each thread executes over a program version, the same way it executes over a shared program state version. The STM ensures that the shared program state version that each thread has is consistent and matches correctly the program version of that same thread.

The program, however, is considered immutable during the lifetime of the application, so it does not require any synchronization mechanism. Each thread can access the program only to read it. If we consider dynamic upgrading a running application, this assumption does not hold anymore. Figure 5.2 roughly explains what I propose in this document: to extend the STM so that it also synchronizes accesses that threads make to the program. This way, each thread executes over a program version, the same way each thread executes over a shared program state version. Besides keeping consistent the view that each thread has over the shared program state, the STM must also keep consistent the view that each thread has over the program.

5.1.1 JVSTM

Among all the different implementations of STMs, my work uses **Java Versioned Software Transactional Memory (JVSTM)** [9]. JVSTM transactions have a version number, which is assigned when they start. This number comes from a global counter that is incremented only when a transaction that modifies some data successfully commits.

JVSTM is based on **versioned boxes** (VBoxes): transactional containers that hold a sequence of tagged values—the **history** of the VBox. Each value in the VBox corresponds to a change made to the box by a successfully committed transaction. The tag of a value is the number of the transaction that made the corresponding modification to the VBox. Each value corresponds to a change made by a committed transaction.

In JVSTM, each transaction `T` has:

T-number A transaction number

T-readSet The set of boxes that were read in the context of `T`

T-writeSet A map that maps boxes that were written in the context of `T` to the values those boxes were set to

There are two operations that a transaction `T` can perform on a VBox `B`:

Read Returns the current value of `B` in the transaction. The current value of `B` in a transaction `T` in the value tagged with the highest number that is less then or equal to the number of `T`. If there is a mapping for `B` in `T-writeSet`, the current value of `B` in `T` is the value stored in that mapping.

Write Sets the value of B in the transaction. JVSTM keeps modifications made to `B` local to the transaction `T` until `T` commits. Instead of storing the new value directly on `B`, it is added as a mapping from `B` the new value on `T-writeSet`.

When `T` commits, JVSTM checks if `T` conflicts with other transactions. `T` can be classified as:

Read-Only `T` does not modify any box. `T` always sees a consistent view of the shared program state because all boxes read return the value they had when `T` started. Thus, `T` can be safely committed. To the rest of the application, `T` appears to commit immediately after starting.

Read-Write `T` reads and modifies boxes. JVSTM must ensure that none of the boxes that `T` reads are changed by another committed transaction after `T` started. If `T` does not conflict with other transactions, `T` is **valid**.

Write-only `T` does not read any box. Thus, `T` is always valid because there can be no conflicts with other transactions.

When `T` commits successfully, its transaction number is incremented and the value associated with each modified box is added to the respective box. Please note that boxes are changed only at this point.

5.2 Upgrade Semantics

As I described in Section 3.2, there is a tight coupling of the program with the state that it manipulates. Thus, each upgrade has a transform function that initializes the new state based on the current one. The new state features the modifications that the upgrade introduces. In Section 3.3.3, I described two possible moments when the upgrade system may execute a transform function, without restarting the application. Either we have an immediate upgrade or a lazy upgrade.

Immediate upgrades offer the programmer good semantics because he knows exactly when the transform functions are executed. Moreover, each upgrade introduces a new, distinct version of the program and its state. Thus, the only version-aware code that the programmer writes is the transform functions. These functions deal with only two consecutive versions of the program. But this comes with a cost: Converting completely the program state may pause the application for a long time because the program state can be quite large.

Lazy upgrades, on the other hand, avoid the long-pausing conversion. Instead, they convert the program state as the natural flow of the program touches it. But the programmer never knows exactly when a transform function will be executed. After installing several upgrades, it is possible that different instances are in different versions of the program state. This happens with CLOS, as I described in Section 4.3.1. In such scenario, the programmer may have to write conversion code that deals with all the previous versions of the program.

5.2.1 Notation

Throughout this section, I shall use figures to illustrate the upgrade system's semantics. Those figures follow a specific notation. Figure 5.3 is an example of such figures.

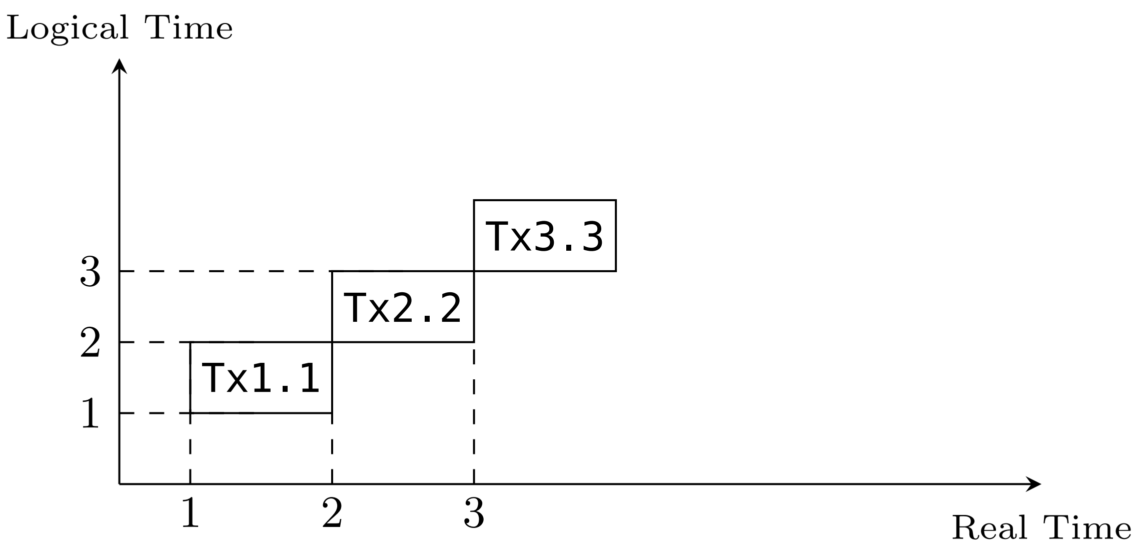


Figure 5.3: Example of the transaction notation used in this section. The horizontal axis represents real time. The vertical axis represents logical time. Each box represents a transaction.

Figure 5.3 has two axis: the **real time**—horizontal—axis and the **logical time**—vertical—axis. In Section 5.1.1 I explained how JVSTM assigns numbers to each transaction, according to the instant when it starts and when it is serialized. This numbering is the logical time. On the other hand, the real time axis denotes the time that each transaction spends executing.

Figure 5.3 also has three boxes: `Tx1.1`, `Tx2.2`, and `Tx3.3`. Each box represents a transaction. I assume that all transactions are read-write transactions. Otherwise, instead of boxes, each figure would represent transactions using horizontal lines, since the remaining types of transactions—read-only and write-only—serialize either at the instant when they started, or at the instant when they committed. For simplicity, each transaction is named `Tx X.Y`, where `X` denotes the instant in logical time when it started and `Y` denotes that same instant, in real time.

In this example, the three transactions execute in series by the order `Tx1.1`, `Tx2.2`, and `Tx3.3`, both in logical and real time. When the behaviour is the same in both axis, I will collapse those axis into a single one simply name **time**. Figure 5.4 shows this simplification applied to Figure 5.3.

However, there are situations where the behaviour in both axis differ, as Figure 5.5 shows. It starts with the same scenario as Figure 5.3 shows, but transaction `Tx3.3` is suspended before it commits. While it is suspended, transaction `Tx1.4` executes. However, transaction `Tx1.4` starts its execution in the same logical timespan as transaction `Tx1.1`. Thus, it does not see the changes made by the committed

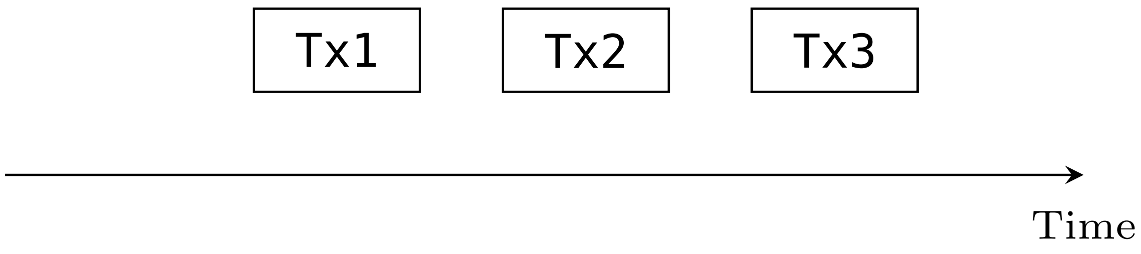


Figure 5.4: Example of the collapsed transaction notation used in this section. The only axis represents both logical and real time. Thus, this notation can be used only when the behaviour is the same when viewed either from the real time point of view or from the logical time point of view.

transactions `Tx1.1` and `Tx2.2`. Transaction `Tx1.4` would never see the changes made by transaction `Tx3.3` because it has not committed yet. When transaction `Tx1.4` commits, in real instant 5, transaction `Tx3.3` resumes and commits.

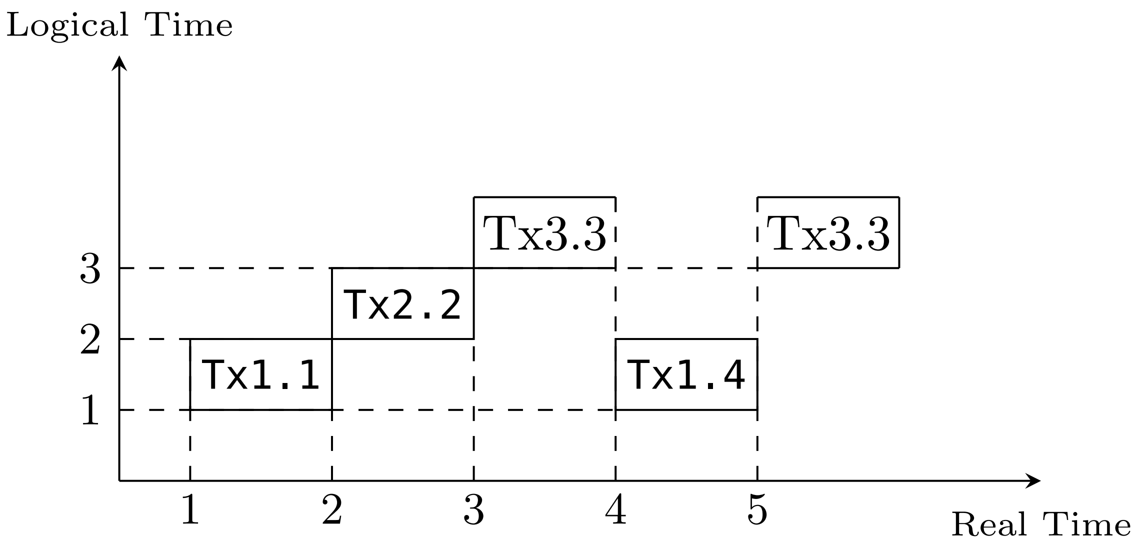


Figure 5.5: Example of the transaction notation in which the behaviour in both axis differ. Although transaction `Tx1.4` executes in the middle of transaction `Tx3.3` in real time, in logical time it executes at the same time of transaction `Tx1.1`. In this case, we cannot use the collapsed transaction notation.

Transaction `Tx1.4` may modify values that transaction `Tx2.2` read. Logically, transaction `Tx1.4` could be executed between real instants 0 and 2. However, it would result in possible different results in transaction `Tx2.2`. This is what I call a **paradox**. I shall present them in further detail in Section 5.2.4.

5.2.2 Expected Upgrade Semantics

Let us recall the example shown in Figure 2.1. It features a geometric application that manipulates points and rectangles. Rectangles are represented using two points: the upper left vertex, and the lower right vertex.

Consider now the sequence of events shown in Figure 5.6. The application starts, creates a rectangle (and respective points) in transaction `Tx1`, and is upgraded.

With immediate upgrade semantics, the programmer writes the program assuming that all the state

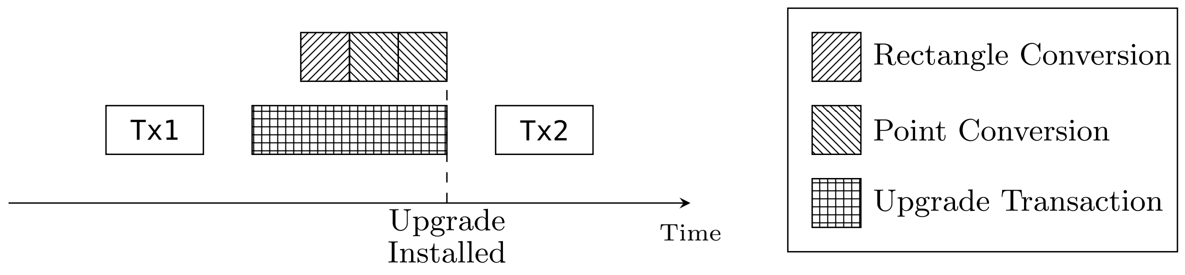


Figure 5.6: Upgrade semantics expected by the programmer. The programmer expects that all the program state is converted when the upgrade is installed. To the programmer, transaction `Tx2` will see only the converted program state.

is converted when the upgrade is installed. When transaction `Tx2` executes, it expects to find instances of the new version of the program only. The upgrade system that I propose exposes this simple semantics to the programmer that is writing the new program version and the transform functions.

5.2.3 Lazy Upgrade Semantics

The immediate upgrades semantics is quite simple to use and to understand. But a complete immediate conversion of the program state may be very costly and introduce a long pause while the upgrade system converts all the program state.

So, I propose a different approach. Conceptually, every instance is converted when the upgrade is installed. Yet, there is no way to check if the instance was really converted, unless by manipulating that instance. By the time the program is able to manipulate any instance, that instance must be already converted. The upgrade system does not have to convert that instance any time sooner. This is a key observation: the upgrade system is free to delay the conversion of any instance to the last possible moment.

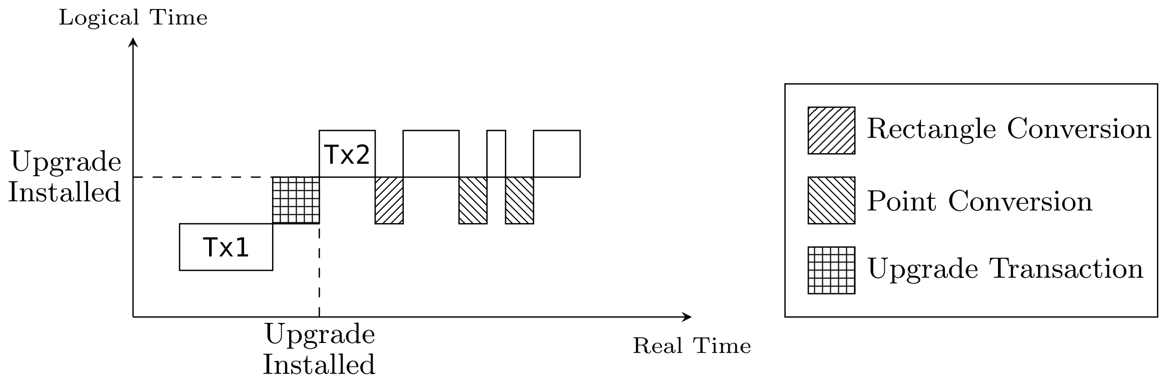


Figure 5.7: Lazy upgrade semantics. When the upgrade is installed, the upgrade system does not convert any program state. Instead, it converts the program state lazily, as it is touched by the natural flow of execution of the program.

Let us consider the same sequence of events described in Section 5.2.2. Figure 5.7 shows those events. We can see that, logically, all transform functions are executed when the upgrade is installed. But, in reality, each transform function executes only when a transaction attempts to manipulate an instance that is not yet converted. The transaction that manipulates that instance is suspended before executing the transform function. The original transaction resumes as soon as the transform function finishes.

5.2.4 Paradoxes

The mechanism shown above hides the lazy nature of the upgrades away from the perception of the programmer. But, unfortunately, it fails to make the lazy upgrades completely transparent to the programmer.

For instance, consider that the geometric application keeps a global list of all the points existing in the application. From one version to another, the programmer decides to implement the rectangle with all the four vertexes, instead of just two. The transform function is quite straightforward; assuming immediate upgrade semantics, it computes the new vertexes and adds them to the new rectangle and to the global point list.

The problem with this upgrade, however, is that it fails to keep the immediate upgrade semantics. Figure 5.8 shows why.

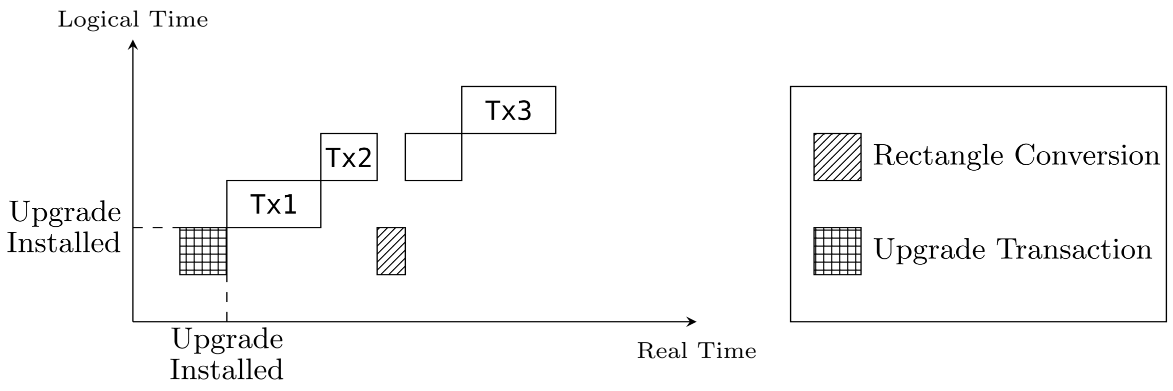


Figure 5.8: Example of a paradox. The upgrade installs a new version of class `Rectangle`, which is represented by four vertexes instead of two. Transactions `Tx1` and `Tx3` print all the existing points. Transaction `Tx2` prints a rectangle, triggering its conversion. Transaction `Tx3` will display extra points that transaction `Tx1` does not display. This is a paradox.

Assume that transactions `Tx1` and `Tx3` print all the points existing in the global point list, and that transaction `Tx2` prints a rectangle. The upgrade transaction installs a new version of class `Rectangle`. When transaction `Tx2` touches the rectangle to print it, it triggers its conversion. In this scenario, and with the lazy approach described before, transaction `Tx3` will display more points than transaction `Tx1`. Yet they should display the same points, because no rectangle was created between their execution. This happens because transaction `Tx2` creates points when it converts a rectangle.

I call this phenomenon a **paradox**. It happens because the upgrade system is postponing the execution of transform functions, creating a mismatch between what the programmer expects and what the upgrade system performs.

Paradoxes can only occur in specific conditions: the transform function modifies some part of the state that is not exclusivelly accessible through the object being upgraded. I shall discuss means to detect paradoxes on Section .

5.2.5 Atomicity

The upgrade system that I propose is based on a transactional memory system that provides strict serializable semantics. All operations are performed inside an atomic block. This means that each transaction always has a consistent view of the program state. An upgrade is an operation that modifies the program state. Thus, the upgrade is also atomic because it is executed inside its own transaction.

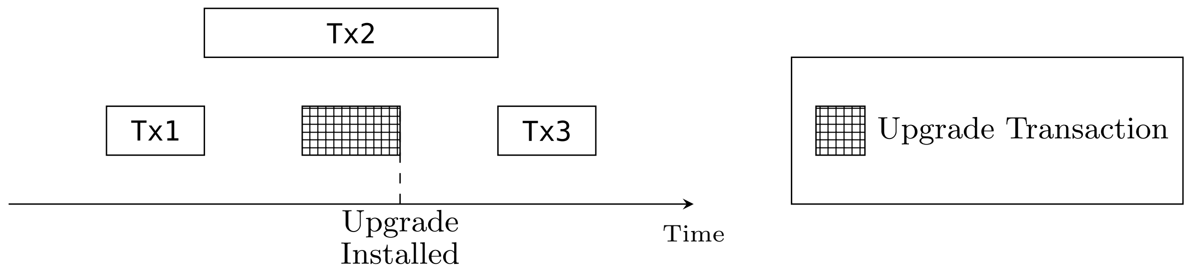


Figure 5.9: Possible execution of application transactions and an upgrade. Transaction `Tx1` executes and commits before the upgrade, thus in the old program version. On the other hand, transaction `Tx3` executes and commits after the upgrade, thus in the new program version. Transaction `Tx2`, however, overlaps the upgrade.

In Figure 5.9 we see an upgrade transaction and three application transactions. Transaction `Tx1` starts and finishes before the upgrade has started—it is executed in the old program version. On the other hand, transaction `Tx3` starts and finishes after the upgrade has finished—it is executed in the new program version. Transaction `Tx2` is different. It starts before the upgrade has started and finishes after the upgrade has finished.

Taking into account the atomic semantics of each operation, we see that there are two possible outcomes for transaction `Tx2`: either it serializes after the upgrade occurs, or it serializes before the upgrade occurs. When the upgrade transaction starts, transaction `Tx2` may already have read some instances that will be logically converted by the upgrade. So, if `Tx2` serializes after the upgrade is performed, it has to be restarted and re-read the new version of the program state. But if `Tx2` serializes before the upgrade is performed, it does not have to be restarted just because an upgrade was performed.

5.3 Dynamic Upgrades

The Java Runtime Environment has very limited upgrade capabilities, as we saw in Section 4.3.3. To overcome such limitations, the upgrade system that I propose creates a new Java class at each class upgrade. For instance, in the geometric application example shown in Section 2.2, we would start with class `Point$1` and upgrade it, generating class `Point$2`. There is a limitation to this approach: the programmer must be able to write conversion code that initializes an instance of class `Point$2` given an instance of class `Point$1`. I call such conversion code a **transform function**, following the terminology introduced by Boyapati et al[6].

With this approach, programmers can modify any part of class `Point` between upgrades. Nevertheless, if we choose to break the interface of class `Point`, we must also change all client classes of class `Point`. A **class upgrade**, as defined by Boyapati et al [6], is an upgrade intended for a single class. They define an upgrade as a set of class upgrades. They also introduce the notion of a **complete upgrade**, which contains class upgrades for all the classes that need to change due to some class-upgrade already in the upgrade. The upgrade system that I propose accepts only complete upgrades. In the example shown in Section 2.2.4, an upgrade containing the new versions of classes `Rectangle` and `Point` is a complete upgrade.

The upgrade system that I propose keeps an **upgrade version number** for the running program. When the program starts with the initial version of its classes, it is in upgrade version 1. Each upgrade increments the upgrade version number of the program.

5.3.1 Handles

Object oriented applications work by manipulating **objects**. When the programmer is writing them, he reasons about objects and how they interact with each other. Objects exist conceptually and are useful tools to the programmer. They have a specific type, dictated by the class they belong to. But, when the program is executing, those objects are mapped to instances. An **instance** is a runtime manifestation of a class. Typically, the mapping between objects and instances is very simple: An object is also an instance.

To introduce a new Java class per class upgrade has a major problem: An object is now represented by different instances across class upgrades. Consider that object `rectangle` refers to object `point` of class `Point`. On runtime, instance `rectangle` refers to an instance `point$1` of class `Point$1`. When the upgrade system installs a new program version, it installs the new version of class `Point`: Class `Point$2`. In this new version, object `point` is represented by an instance `point$2` of class `Point$2`. Instances `point$1` and `point$2` represent the same object, in different upgrade versions of class `Point`. Thus, the identity of object `point` is kept by a different instance at each version of class `Point`. But instance `rectangle` has a reference to instance `point$1`, even after the new version of class `Point` is installed. Normal Java references do not solve this problem.

We could use the transform function to fix the reference on `rectangle`, making it refer to instance `point$2`. This is a naive solution that does not work. Figure 5.10 show why. When the program upgrade is installed, two instances of class `Rectangle` (`r1` and `r2`) refer to the same instance of class `Point` (`p$1`). When converting instance `r1`, the transform function creates an instance `p$2` of class `Point$2`, migrates instance `p$1` to `p$2`, and fixes the reference on instance `r1`, so it refers instance `p$2`. Converting instance `r2` is very similar: the transform function creates an instance `p$2.1` of class `Point$2` and repeats the whole process. However, when both instances `r1` and `r2` are converted, instance `r1` refers to a different instance of class `Point` than instance `r2`. Clearly, the resulting program state is not equivalent to the starting one.

The upgrade system that I propose uses an extra level of indirection to keep the identities of the objects across upgrades—the **handle**. Handles refer to one instance and, at each upgrade, change the instance that they refer to the instance that represents the object in the new upgrade version of the program. Thus, instead of referring directly to instance `point$1`, object `rectangle` refers to an handle `H` that represents the identity of object `point`, which in turn refers to the current instance that represents object `point`: `point$1` on the first upgrade version, and `point$2` after the upgrade is installed. Handles are completely transparent to the programmer because they are generated automatically by the upgrade system.

Handles are also a natural place to trigger instance conversion because they intermediate all object manipulation. When a transaction accesses an instance that was not converted yet, it must do so using an handle. If that instance was not converted yet, the handle detects that the instance must be converted,

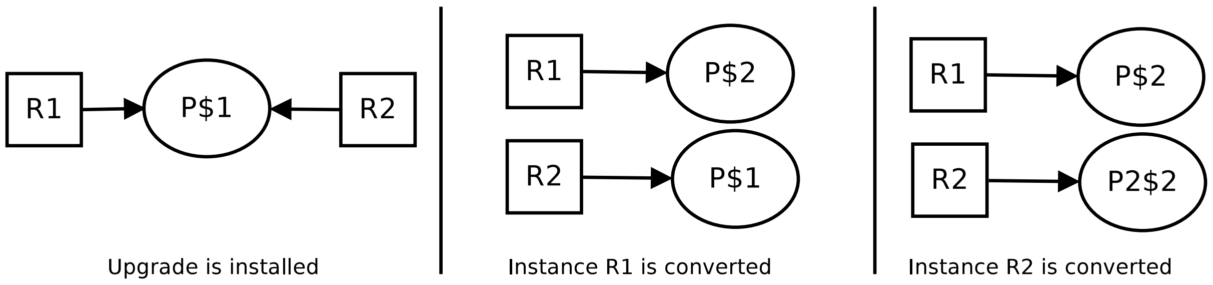


Figure 5.10: Using transform functions to fix references across upgrades. When the upgrade is installed, two instances of class `Rectangle` (`R1` and `R2`) refer to the same instance of class `Point` (`P$1`). Then, instance `R1` is converted and its transform function creates a new instance of class `Point` (`P$2`). Then, instance `R2` is converted and its transform function creates a new instance of class `Point` (`P2$2`). Clearly, the resulting program state is not equivalent to the starting one.

suspends the transaction that requires the object, instantiates the new class, executes the transform function, and resumes the transaction, returning the expected instance.

5.3.2 Structural Modification

All instances of upgradable types must be accessed through handles, to keep the identity of the object they represent across upgrades. This means that the type of all fields with upgradable type—upgradable fields—must change to an handle type. This is the first solution: change only the types of the upgradable fields, keeping the types of the method arguments and local variables the same. Upgradable objects must be accessed through handles when reading them from fields. Then, they are passed as direct references. If they are stored in a field again, the upgrade system can generate a back-reference from the object to its handle to store it correctly.

Although simple, this solution does not work in all cases. Consider that an upgradable instance is passed back and manipulated directly by upgradable types. By doing so, we are defeating the handle mechanism and exposing the upgrade system to the problems that handles solved in the first place.

There is another reason against passing direct references to instances of upgradable types. Consider that an instance of an upgradable type is used by a non-upgradable type. Even though the upgrade system installs several upgrades for that upgradable type, the non-upgradable type that directly manipulates the instance will never use those upgrades.

To avoid these scenario, I propose to replace all types of field, method arguments, and local variables that are upgradable by handles. Therefore, the program always manipulates upgradable instances using the respective handle. Moreover, situations where direct references to upgradable instances are passed to non-upgradable types cease to exist. Instead, the non-upgradable types receive handles to the upgradable instances. Passing these handles back to upgradable types does not pose any problem. Figure 5.11 shows how the application described in Section 2.2.1 looks like after performing the structural modification.

5.3.3 Inheritance

In the example shown in Section 2.2.1, consider that classes `Point` and `Rectangle` are non-upgradable types, but that class `Square` is an upgradable type. Figure 5.12 illustrates this scenario. Consider, also,

```java
class Point {
  private double rho, theta;

  static double distance (Handle p1, Handle p2) { ... }

  String toString() { ... }
}

class Rectangle {
  private Handle topLeft, topRight, botLeft;

  double area() { ... }
}
```

Figure 5.11: Structure of the geometric application example after replacing the upgradable types by handles.

that class `Rectangle` has a new method that computes the rectangle resulting of the intersection with another rectangle. The Java signature of the new method is: `Rectangle intersection(Rectangle r)`. After performing the program transformation described in the previous section, everytime this method is called with an instance of `Square`, it is really being called with an handle to that instance. That handle is not a subclass of class `Rectangle`, generating runtime errors.

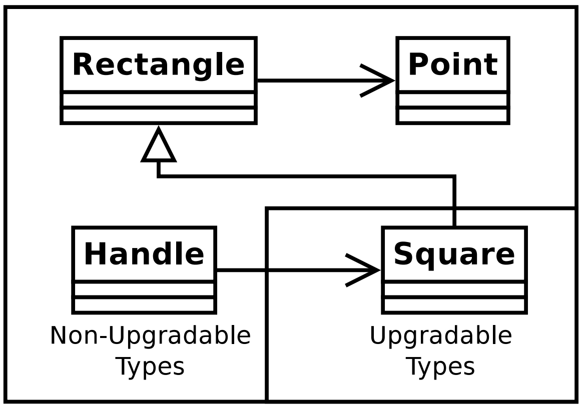


Figure 5.12: Basic scenario that shows the upgradable type inheritance problem: The upgradable type `Square` inherits directly from the non-upgradable parent type `Rectangle`. When methods on type `Rectangle` are called with instances of `Square`, they are being called with handles instead of direct references. This results in runtime errors.

The solution that I propose follows a different path. Instead of having one generic handle type for all upgradable types, the upgrade system generates a specific handle type for each upgradable type. In this case, it would generate an handle `Handle$Square`. With this solution, I am splitting an object into two instances. The rational behind this is simple: there is an instance that keeps the upgradable state of the object, and another that keeps the non-upgradable state. The handle does not change during the lifetime of the application. Thus, it is non-upgradable and is used to keep the non-upgradable state of the object.

When an upgradable type is a subclass of a non-upgradable type, the upgrade system uses the handle to store the inherited state and behaviour. That handle is a subclass of the superclass of the upgradable type, and also implements all non-upgradable interfaces that the upgradable type implements. In the example that we have been following, class `Handle$Square` is a subclass of class `Rectangle`. Conceptually, class `Handle$Square` is also a subtype of class `Handle`. Figure 5.13 illustrates this scenario.

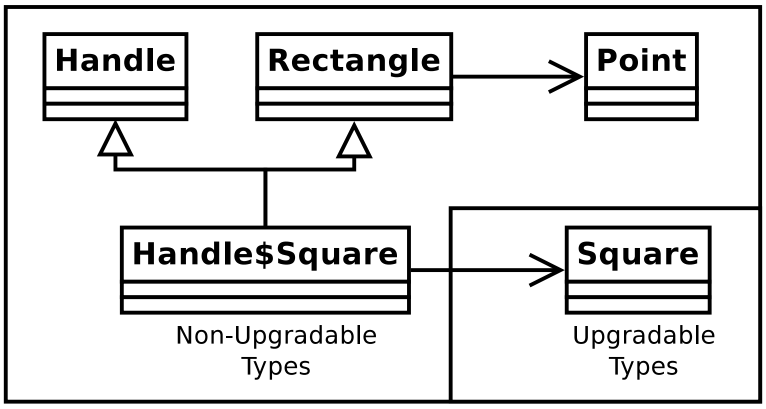


Figure 5.13: First approach to the upgradable type inheritance problem: Upgradable types inherit only from upgradable types. When an upgradable type inherits from a non-upgradable type, the handle is used to keep that inheritance relationship. The inherited methods on the handle dispatch the execution to the upgradable instance.

The superclass of class `Square` is class `java.lang.Object`. Thus, there are no duplicated fields between the upgradable type and its handle. Duplicated fields would be troublesome because upgradable types would manipulate fields existing in class `Square` and non-upgradable types would manipulate fields existing in class `Handle$Square`.

The handles are non-upgradable types and are tightly coupled with the upgradable classes that they represent. Thus, refactoring upgradable types with a non-upgradable superclass as I propose limits the range of possible upgrades that the upgrade system supports, introducing two limitations:

1. An upgradable type which superclass is non-upgradable must always have that superclass during the following dynamic upgrades.
2. An upgradable type which superclass is upgradable must always have an upgradable superclass during the following dynamic upgrades. Upgradable types with an upgradable superclass may change that superclass on an upgrade, but the new superclass must be upgradable as well.

Please note that I have been using the term *dynamic upgrades* to refer the upgrades that the upgrade system supports without restarting the application. Of course, we could restart the application and regenerate the handles. By doing so, the upgrade system can supports any possible modification.

So far I have shown how to refactor upgradable types to support direct inheritance from non-upgradable types. But an upgradable type may have a non-upgradable type as an indirect parent class. For instance, in the example that we have been following, consider that we introduce class `Pixel`, which conceptually is a square with unitary side length. Class `Pixel` is a subclass of class `Square` and has a non-upgradable type as an indirect parent class, class `Rectangle`, which is the superclass of class `Square`. Figure 5.14 shows a possible solution to generate the handles of upgradable types with indirect non-upgradable classes: Class `Handle$Pixel` is a subclass of class `Handle$Square`.

The solution shown by Figure 5.14 introduces a major limitation on the possible range of upgrades that the upgrade system supports: Class `Pixel` must always be a subclass of class `Square`. To overcome such limitation, I propose that the generated handle for class `Pixel`—class `Handle$Pixel`—inherits directly from class `Rectangle`. Figure 5.15 shows that approach. This approach still has a limitation: Any class with a direct or indirect non-upgradable parent class must keep that parent class, at least indirectly, in

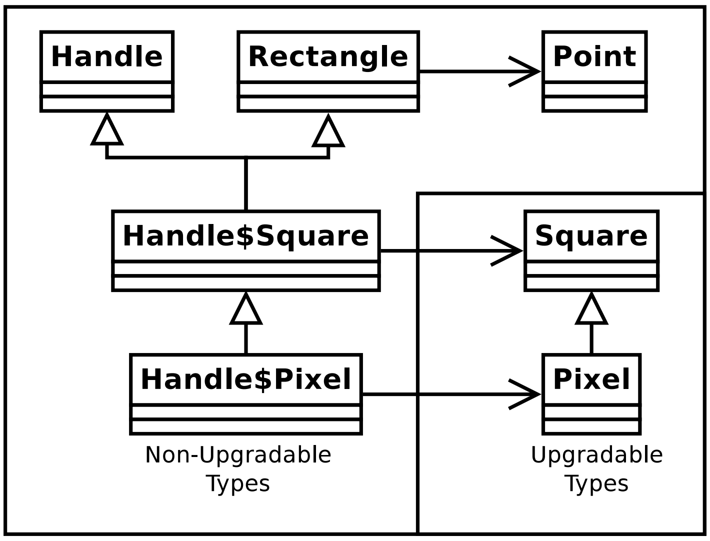


Figure 5.14: Basic solution for the inheritance problem on upgradable types that have an indirect non-upgradable parent type. Class `Pixel` is an example of such types. In this solution, both the upgradable type and its handle inherit directly from the upgradable direct parent type.

all upgrades. This means that classes `Square` and `Pixel` must always have class `Rectangle` as a parent class, either directly or indirectly.

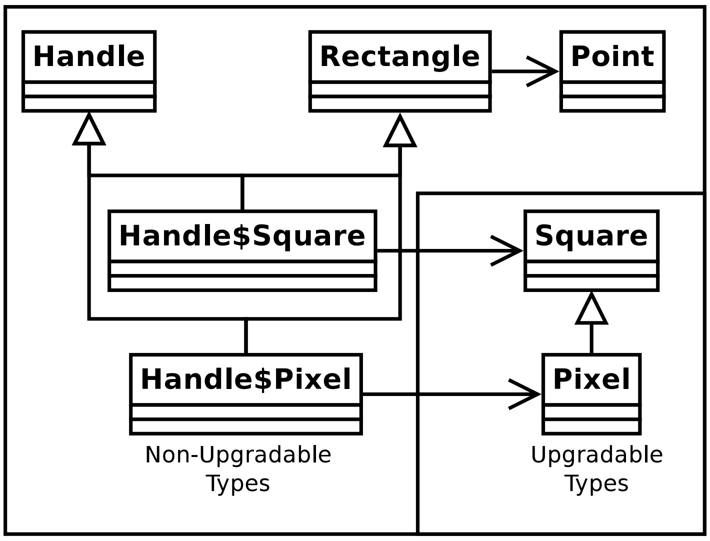


Figure 5.15: Less restrictive solution for the inheritance problem on upgradable types that have an indirect non-upgradable parent type. Class `Pixel` is an example of such types. The upgradable type `Pixel` inherits from the upgradable direct parent type and the non-upgradable handle inherits from the non-upgradable indirect parent type.

If we take a closer look at handle classes `Handle$Square` and `Handle$Pixel`, we can see that they are redundant. The purpose of these classes is to enable the non-upgradable types to use upgradable types. Both classes `Square` and `Pixel` are subclasses of class `Rectangle`. The non-upgradable types do not know the upgradable types, thus these two upgradable types are the same as far as the non-upgradable types are concerned. Therefore, instead of having one handle type per upgradable type, the upgrade systems needs to generate one handle type per each inheritance relationship in which an upgradable type inherits from a non-upgradable type. In this case, the upgrade system generates handle class `Handle$Rectangle`. Figure 5.16 shows this solution.

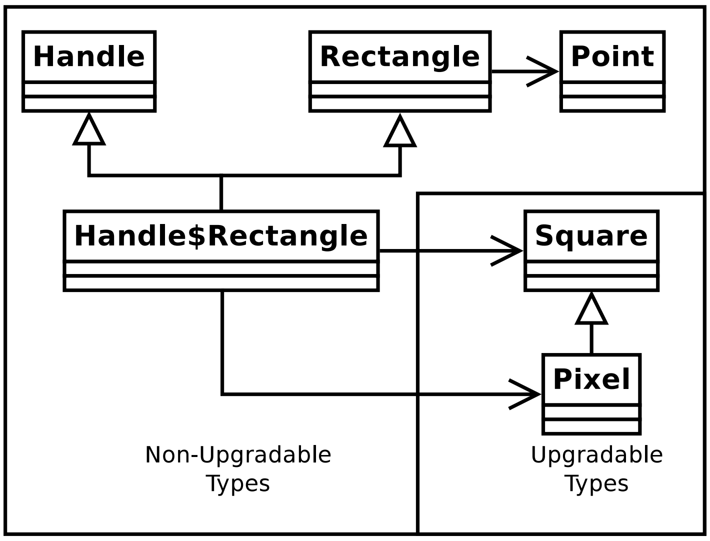


Figure 5.16: Simplification of the solution for the upgradable type inheritance problem. In Figure 5.15, handle classes `Handle$Square` and `Handle$Pixel` are exactly equal. So, instead of generating an handle class per each upgradable type that inherits (directly or not) from a non-upgradable type, the upgrade system generates an handle class per each non-upgradable type that is the parent type of, at least, one upgradable type.

Please note that Java does not support multiple inheritance. However, I have used it in my solution. The implementation is quite simple, though: Each handle class is generated starting from a copy of class `Handle`. Also, I have not shown the implementation of inherited methods on the generated handle class, nor how upgradable types are modified to cope with handles. I shall describe those implementation details in further detail in Section 6.3.1.

Handles and Overloaded Methods

Using a single handle type to represent the identity of several upgradable types may be troublesome if the programmer uses method overloading. An overloaded method is a method that has the same name as another method. They differ on the number, type, and position of the arguments they receive.

```
class Util {
  static void draw(Square s);
  static void draw(Pixel p);
}
```

Figure 5.17: Example of method overloading.

Figure 5.17 shows an extension of the basic geometric application, described in Section 2.2.1. It introduces a new class `Util` that is able to print, for instance in a canvas, a geometric figure manipulated by the application. The method `draw` performs such computation.

In the example that we have been following, types `Square` and `Pixel` are upgradable, and that type `Rectangle` is non-upgradable. Figure 5.18 shows the result of replacing upgradable types by handles, as I have proposed so far. The overload of the method `Util.draw` is now indistinguishable.

Method overloading is resolved at compile time. Exploring that fact, the upgrade system can change the names of the overloaded methods. In the geometric application example, it would generate different

```
class Util {
  static void draw(Handle$Rectangle s);
  static void draw(Handle$Rectangle p);
}
```

Figure 5.18: Example on how replacing upgradable types by handle breaks method overloading. Both methods `Util` were distinguish by their types. They are now indistinguishable because both methods receive an handle as the single argument.

names for each method `draw`, as Figure 5.19 shows.

```
class Util {
  static void drawSquare(Handle$Rectangle s);
  static void drawPixel(Handle$Rectangle p);
}
```

Figure 5.19: Changing overloading methods name to deal with the type replacement problem. The upgrade system changes the overloaded methods name to distinguish them when the upgradable types are replaced by handles. This solution does not work with constructors.

Although simple, this solution does not work with overloaded constructors because they must have the same name. The upgrade system can introduce a new argument, just to differentiate the two overloaded methods. This extra argument would have a completely synthetic type generated by the upgrade system. We may see an example of this solution in Figure 5.20.

```
class Util {
  static void draw(Handle$Rectangle s,Util$1 o);
  static void draw(Handle$Rectangle p,Util$2 o);
}
```

Figure 5.20: Generating an extra dummy argument to overloaded methods to deal with the type replacement problem. This dummy argument, which type is different for each overloaded method, clearly distinguishes all the overloaded methods.

Although this solution works, there is a simpler solution to this problem. So far, I have argued that the upgrade system needs to generate different handles only per each non-upgradable type that is a parent type of any upgradable type. But, if the upgrade system uses a different type of handles for each upgradable type, the problem of the overloaded methods is trivially solved. Therefore, instead of generating just one handle type for classes `Square` and `Pixel`, if the upgrade system generates two different handle types, `Handle$Square` and `Handle$Pixel`, the method overload problem is naturally solved, as Figure 5.21 shows.

As far as the non-upgradable types are concerned, both handle types `Handle$Square` and `Handle$Pixel` are subclasses of the non-upgradable type `Rectangle`. Outside the upgradable types, instances of classes `Square` and `Pixel` may still be used as instances of their parent class `Rectangle`.

5.4 Integration with the Development Process

Typically, developers use a revision control system to keep track of the source code evolution. Revision control systems have a notion of version that differs from the upgrade system's notion. For a revision control system, a version is a piece of source code that was submitted latter than a previous one and that differs from it in some way. Such notion of version is commonly know as a **revision**.

```
class Util {
  static void draw(Handle$Square s);
  static void draw(Handle$Pixel p);
}
```

Figure 5.21: Generating an handle type per each upgradable type is a trivial solution to the method overloading problem.

On other hand, for the upgrade system, an **upgradable version** is a portion of executable code that redefines a subset of the program currently in execution. Besides the definition of the new application behaviour, an upgradable version also has transform functions that enable the upgrade system to convert the program state used by the immediately previous upgradable version to an equivalent program state.

How do upgradable versions relate to revisions? Not every revision can be an upgradable version because revisions may contain inconsistent code. An upgradable version is a revision with two non-trivial properties (trivially, it must compile and display a correct behaviour).

Completeness If the interface exported by some upgradable class `C` changes, all of `C`'s client classes must also have a new version.

Transform Functions Every transform function must be able to fully initialize an instance of any class present in the upgrade, given an existing instance in the previous version.

The developer may mark a revision as an upgradable version. When he does that, the upgrade system creates a new revision by removing the transform functions. This revision is the basis of the future upgradable version of the program. To write a new upgradable version, the programmer writes the modifications and adds the appropriate conversion code to the transform functions.

5.5 Discussion

This chapter described how the upgrade system that I propose uses a Software Transactional Memory to perform atomic upgrades. The particular STM that the upgrade system that I propose uses is JVSTM.

This chapter also described the semantics of the upgrade system that I propose. The upgrade system follows lazy upgrades to avoid the long pausing state conversion imposed by immediate upgrades. Yet, the programmer writes the transform functions expecting immediate upgrades because they provide strong semantics that allow the programmer to write modular transform functions. To write conversion code, the programmer has to consider two version of the program only: the current version and the previous one.

However, this solutions may suffer from mismatches between what the programmer expects and what the upgrade system actually performs. I name such mismatches as **paradoxes**. Paradoxes can occur in specific conditions: When the transform function modifies some part of the program state that is not exclusively accessible through the object being upgraded.

This chapter also described how to overcome the limited upgrade capabilities of the Java Runtime Environment. I proposed to introduce a new Java class per class upgrade. The programmer then defines a **transform function** per upgraded class to migrate instances from the old class to the new one.

Although simple, this approach introduced a new problem: An object is now represented by different instances across class upgrades. I described how to use an extra level of indirection to manipulate objects of upgradable types—the **handle**. Handles are responsible for keeping the identity of upgradable objects across upgrades, referring to the correct instance that represent that object in each program version. Handles are also a natural place to trigger the lazy conversion of upgradable objects because it intermediates all upgradable object accesses.

If a non-upgradable type attempts to use an upgradable type, it is actually using an handle to that type. However, the non-upgradable type must compile without the upgradable type because they are in different layers, as Figure 3.1 shows. Therefore, the non-upgradable type can use upgradable types only if they are subclasses of non-upgradable types. In this chapter, I proposed to split upgradable objects into two instances to solve that problem. One of this instances contains the non-upgradable part of the object and is a subclass of the non-upgradable parent type. This instance is the handle of that specific upgradable type. The other instance contains the upgradable part of the object and changes from upgrade to upgrade.

Finally, this chapter described how to integrate the upgrade system in the typical development process. It described how to integrate the notion of upgradable version, from the upgrade system's point of view, with the notion of revision, from the revision control system's point of view. The developer may signal the upgrade system if a revision is an upgradable version. Then, the upgrade system cleans the revision by removing the transform functions, so it can be the basis of the following upgradable version. This way, the program always evolves from upgradable version to upgradable version. However, each upgradable version is able to convert the state that existed on the immediately previous upgradable version only. This means that the programmer cannot skip installing upgradable versions, or else he breaks this conversion chain and renders the upgrade system unable to migrate older instances.

Chapter 6

Implementation

This chapter describes the implementation of the upgrade system that I proposed in the previous chapter. Throughout this chapter I shall use examples to illustrate some implementation details of the upgrade system. Such examples start from the basic geometric application, described in detail in Section 2.2.1.

6.1 Structural Classes

The upgrade system that I propose is based on two main classes: (1) the `UpgradeSystem` class, and (2) the `Handle` class. I call these classes as **structural classes** and I shall describe their outline on this section. The interaction of these two structural classes dictates the behaviour of the upgrade system. For instance, the class `Handle` must interact with the class `UpgradeSystem` to check if an instance is outdated and should be migrated to the new program version. Thus, this section also describes the processes of installing, executing and composing upgrades as a result of the interaction of the two structural classes `UpgradeSystem` and `Handle`.

6.1.1 The `UpgradeSystem` Class

The class `UpgradeSystem` is the central point of decision in the implementation of the upgrade system. It is responsible for keeping the upgrade version number, mapping it to JVSTM transaction numbers, and installing new upgrades. Its outline is depicted in Figure 6.1.

```java
class UpgradeSystem {
  VBox<Integer> upgradeVersion;
  ArrayList<Integer> map;

  void installNewVersion(String sourceBasePath,
                         String packageName,
                         String binaryPath,
                         String oldClassesPath) { ... }
}
```

Figure 6.1: Outline of the structural class `UpgradeSystem`.

The last upgrade version number is stored using a VBox. The field `map` maps the upgrade version numbers to JVSTM transaction numbers. Let us assume that an upgrade transaction starts at instant 30 and finishes at instant 31, in terms of the JVSTM version numbering. During this time, the upgrade version number is incremented from, lets say, 1 to 2. The class `UpgradeSystem` adds a new mapping to its field `map`: mapping from the JVSTM version number, 30, to the upgrade version number, 2.

The method `installNewVersion` installs a new version of the code. The root of the new class files hierarchy is located in the path specified by the argument `sourceBasePath`. The upgrade is meant for the package specified by the argument `packageName`. After detecting the class files, the upgrade system modifies the bytecode, as we shall see in Section 6.3, and stores the modified class files in the path specified by the argument `binaryPath`. The final argument, `oldClassesPath`, specifies where the old classes should be stored. We shall see the role of these classes on Section 6.2.2.

6.1.2 The `Handle` Class

The handle plays a central role in the upgrade system that I propose, as we saw in Section 5.3.1. It is responsible for keeping the identities of the objects across upgrades. Its outline is depicted in Figure 6.2.

```
class Handle {
  VBox<Object> object;
  VBox<Integer> version;

  static Object getObject(Handle h) {
    if (UpgradeSystem.getInstance.getVersion() > h.version.get())
      h.convert();

    return h.object.get();
  }

  private void convert() { ... }
}
```

Figure 6.2: Outline of the structural class `Handle`.

The current instance of each object and its upgrade version number are both stored in VBoxes, because they will be changed atomically in the conversion process.

For accessing the currently stored instance we have the `getObject` method. This method is also responsible for converting the object before returning it, if there is a new program version. The conversion is performed by the method `convert`, which I shall describe on the following section.

6.1.3 Installing and Executing Upgrades

The process of installing a new upgrade is very simple: it modifies the bytecode of the upgrade, as we shall see in Section 6.3, it increments the upgrade version number kept by the class `UpgradeSystem`, and it adds a new mapping to the `map` field of class `UpgradeSystem`. For instance, consider the example shown in Figure 6.3. The upgrade transaction starts at instant 30 and finishes at instant 31, in terms of the JVSTM version numbering. During this time, the upgrade version number is incremented from, lets say, 1 to 2 and the class `UpgradeSystem` adds a new mapping to its field `map`: mapping from the JVSTM version number, 30, to the upgrade version number, 2.

From now on, when new transactions call the method `Handle.getObject` with instances that have not been converted yet, the method detects it and executes the method `convert` on the handle. For instance, in Figure 6.3 assume that transaction `Tx1` starts with JVSTM version number 31, after the upgrade transaction commits, and then at instant `A`, it tries to manipulate an instance that was converted yet, resulting in an invocation to the method `Handle.convert`. As we saw in Section 5.2.3, this method suspends transaction `Tx1`, launches a new **conversion transaction**, and then resumes the transaction `Tx1` on instant `B`.

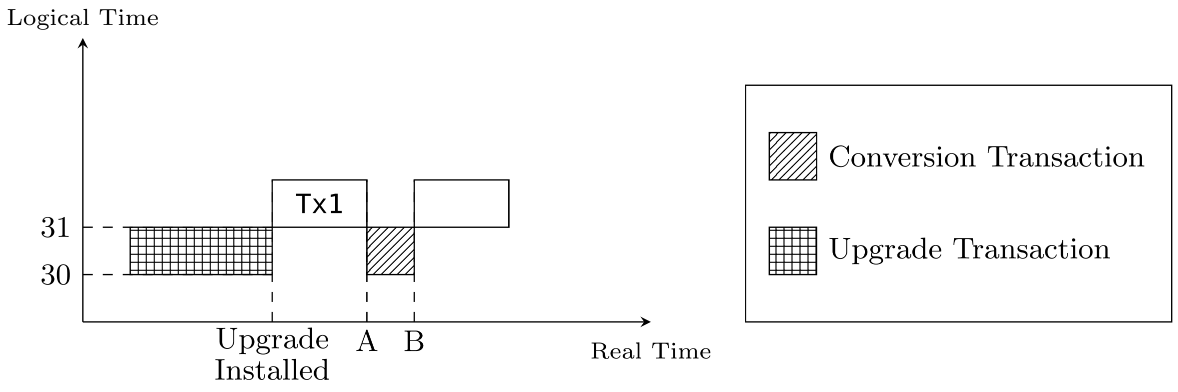


Figure 6.3: Simple example of the conversion of an outdated instance. Transaction `Tx1` attempts to manipulate an outdated instance on instant A, thus triggering its conversion. Therefore, transaction `Tx1` is suspended and the upgrade system creates a conversion transaction, which occurs logically at the same time as the upgrade was installed. When the instance is converted, transaction `Tx1` resumes with the correct version of the instance.

A **conversion transaction** differs from a regular read-write transaction. It starts on the past logical time, according to the mapping stored in the field `UpgradeSystem.map`. Thus, conceptually, all conversion transactions execute at the same time that the upgrade transaction. In this case, it starts at logical instant 30 and finishes on logical instant 31. All the handles that are manipulated during the conversion transaction return the instance that they stored in instant 30 because they store those instances using VBoxes. Thus, the conversion transaction has access only to the state that existed when the upgrade transaction started.

When the conversion transaction commits, it performs a set of verifications different from what read-write transactions perform. These verifications detect paradoxes, described in Section 5.2.4, instead of detecting conflicts, as read-write transactions do. The write set of the conversion transaction must have only objects that were created during the conversion transaction itself. This rule has two major implications: (1) no other transaction was able to see the objects present in the write-set before the upgrade transaction, thus there are no possible paradoxes, and (2) there are no conflicts with any other transaction because all objects in the write-set were created inside the conversion transaction.

6.1.4 Composing Upgrades

When the method `Handle.getObject` executes inside a conversion transaction, the test that it makes may fail if the instance that the handle keeps was not converted on the previous upgrade. On such case, the upgrade system executes method `Handle.convert`. The execution of this method suspends the current conversion transaction, launches a new one, and resumes it when the new conversion transactions commits.

The object that the handle keeps may be older than the previous program version. On such case, the same process repeats itself until the upgrade system launches the conversion transaction that is on the same upgrade version than the instance that the handles keeps.

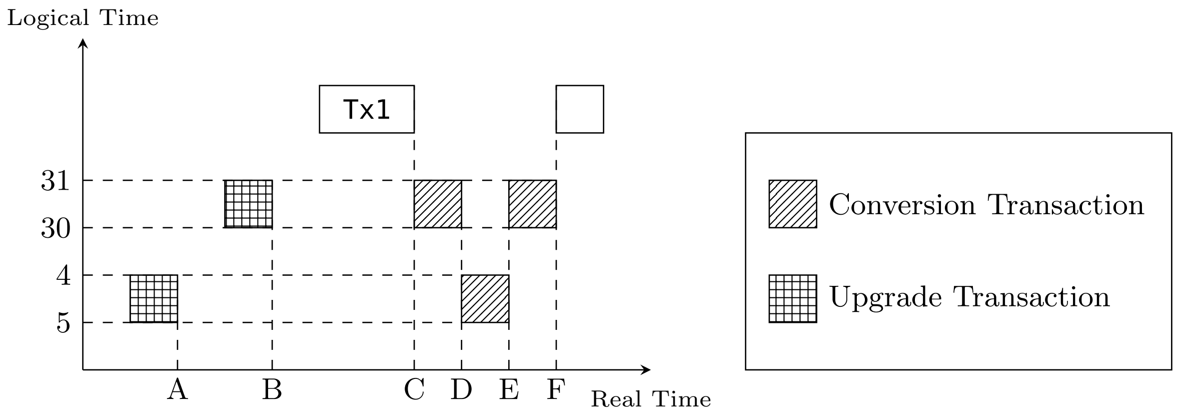


Figure 6.4: Compositions of conversion transactions. At instant C, transaction `Tx1` attempts to use an instance that has not been converted in the last 2 upgrades. Thus, it triggers a conversion transaction that is not able to convert the instance because it is in an older program version. Therefore, the conversion transaction triggers another conversion transaction, which is able to convert the instance.

Figure 6.4 shows a possible composition of conversion transactions. At instant `A` and `B`, the upgrade system installs a new program version. Then, transaction `Tx1` uses an handle which refers to an outdated object, spawning a conversion transaction at instant `C`. However, the object that the handle refers is older than the immediately previous version. Thus, its conversion spawns yet another conversion transaction at instant `D` that converts the object to the previous program version. The first spawned conversion transaction resumes when the second conversion transaction commits, at instant `E`, and transaction `Tx1` resumes when the first conversion transaction commits, at instant `F`.

6.2 Transform Functions

Transform functions enable the programmer to migrate the program state from the current version to the new one that a program upgrade introduces. These functions are defined on the new class definition but must be able to access the old program state to migrate the state when converting an instance. In this section, I describe the syntax of the transform functions and how the programmer is able to access the previous version of the program state when writing them. I also describe some modifications at the bytecode level that the upgradable types must suffer after compilation to enable the correct operation of the transform functions.

6.2.1 Conversion Code

The upgrade system that I propose enables the conversion of the program state from one program version to the following using **transform functions**. A transform function is a method defined on the new version of any type `Class` with the name and signature: `static void convert(old.Class oldI,Class newI)`. The type of the first argument `oldI` belongs to the **old classes**. The upgrade system generates this classes to enable the programmer to refer the old state of the application when converting the program state. The old classes are described in further detail in the following section.

Consider that the programmer wants to modify the internal representation of class `Point` shown in Figure 2.1. He wants to change from rectangular coordinates to polar coordinates, as shown by Figure 2.3. Figure 6.5 shows the transform function that he could write for this upgrade.

```
class Point {
  private double rho,theta;
  ...
  public static convert(old.Point oldI,Point newI) {
    newI.rho = sqrt( square(oldI.x) + square(oldI.x) );
    newI.theta = arctan( oldI.y / oldI.x );
  }
}

class Rectangle {
  ...
  public static convert(old.Rectangle oldI,Rectangle newI) {
    newI.topLeft  = oldI.topLeft.convert();
    newI.topRight = oldI.topRight.convert();
    newI.botLeft  = oldI.botLeft.convert();
  }
}
```

Figure 6.5: Example of a transform function. On class `Point`, the internal representation changes from rectangular to polar coordinates. On the other hand, class `Rectangle` is left unchanged.

6.2.2 Old Classes

The programmer must be able to refer to the old version of the program when he is writing the conversion code. The upgrade system generates the **old classes** to enable the programmer to do so. The old classes are the skeleton of the program classes with some modifications: all fields are public and all methods return a dummy value (`0` or `null`, for instance). Old classes are not meant to be executed. Their purpose is to enable the compilation of the conversion code that refers to the old program state.

Consider that the programmer modifies the structure of the geometric application shown by Figure 2.1. The old classes that the upgrade system generates for this example are shown by Figure 6.6.

As may be seen, all old classes have a method named `convert`. Although the classes that were used to generate the old classes do not have this method, it is needed to satisfy the type checker. For instance, consider that, when converting the class `Rectangle`, the programmer wants to keep the same value of the field `topLeft`. The most natural expression—`newI.topLeft = oldI.topLeft`—fails to compile because the type of `oldI.topLeft`—`old.Point`—is completely unrelated and, therefore, incompatible with the type of `newI.topLeft`—`Point`. The method `convert` provides a bridge that connects these two namespaces. Using it, the programmer writes `newI.topLeft = oldI.topLeft.convert()`. The conversion code displayed in Figure 6.5 shows the usage of the method `convert`.

6.2.3 Post Processing Conversion Code

The application is not ready to be put into execution immediately after being compiled. For instance, the bytecode resulting of the compilation stage still has some references to old classes on the `convert` methods. Although useful for compilation, these classes are not meant to be executed and such references

```
class old.Point {
  public double x, y;

  double distance (old.Point p) { return 0.0 }

  String toString() { return  ""; }

  public Point convert() { return null; }
}

class old.Rectangle {
  public old.Point topLeft, botRight;

  double area() { return  0.0; }

  public Rectangle convert() {return null; }
}

class old.Square extends old.Rectangle {
  public Square convert() { return null; }
}
```

Figure 6.6: Old classes generated for the basic geometric application example.

must be removed. Thus, there is an extra stage located after compilation and before execution. In this stage, the upgrade system modifies the bytecode of the program that will be installed.

In this section, we shall see how the upgrade system prepares the conversion code to be put into execution. Please not that there are more modifications that the upgrade system performs during this stage that shall be addressed on the following section. In this section, I shall describe only the modifications that are directly related to the transform functions. Such modifications are:

Generating Field Accessors Enable that the future transform functions can access to the internal state of the current program state.

Removing Old Classes Removes old classes references that the `convert` methods still have, because such references are meant for the compilation stage only.

Field Accessors

Transform functions are defined on the new class definition, but the programmer needs to have access to the internal state of the old instance to initialize the new version of the instance based on the old instance. However, the visibility rules that the programmer defined for some portions of the old class may prevent its direct access from the new class. For instance, private fields of the old class are not accessible directly from the new class.

Executing this code directly results in a runtime error. There are three possible solutions to this problem:

1. Rewrite all field declarations, making all fields public. The disadvantage of this option is that, by changing the visibility of the fields, the upgrade system may modify the semantics of the program.

2. Use reflection to access the private fields. This option may lead to poor performance when executing the transform functions.

3. Generate new public methods (accessors) that access the private fields.

I follow the last approach: the upgrade system generates an **accessor method** for each field of an upgradable class. This is a public method that returns the value of that field. It is named after the originating field preceded by a `$` character. Figure 6.7 shows the accessor methods generated to for classes `Point` and `Rectangle` and their usage.

```
//First version of class Point
class Point {
  private double x, y;

  public static double $x(Point p) { return p.x; }
  public static double $y(Point p) { return p.y; }
  ...
}

//Second version of class Point
class Point {
  private double rho,theta;

  public static double $rho(Point p) { return p.rho; }
  public static double $theta(Point p) { return p.theta; }

  public static convert(old.Point oldI,Point newI) {
    newI.rho = sqrt( square(old.Point.$x(oldI)) + square(old.Point.$y(oldI)) );
    newI.theta = arctan( old.Point.$y(oldI) / old.Point.$x(oldI) );
  }
  ...
}
```

Figure 6.7: Field accessors generated for the basic geometric application example. These methods are named after the original fields, preceded by a “$” character. This code also illustrates their usage on the method `convert` present in the second version of class `Point`.

The accessor methods are static to allow the JVM just-in-time compiler to generate optimized code. All automatically generated methods are static for the same reason.

The accessor methods are flagged as synthetic. Modern JVMs expect that all class members that do not appear in the source code are flagged with the synthetic attribute. This attribute is useful to increase the transparency of the generated methods. The JVM may, for instance, skip the synthetic entries when displaying a stack trace.

Removing Old Classes

The upgrade system generates the old classes to enable the compilation of the conversion code. Past this stage, they are no longer needed and must be removed. Consider that the programmer writes an upgrade to class `Point`. According to the nomenclature introduced in Section 5.3, currently the program has class `Point$1` installed. The upgrade contains class `Point$2`. When the programmer refers to class `old.Point`, he is actually referring to class `Point$1`. The upgrade system replaces all references to class `old.Point` to references to class `Point$1`.

Also, the invocations to the methods `convert`, belonging to the old classes, are no longer needed. So, the upgrade system must remove them from the new class definition.

6.3 Implementing Handles

Upgradable types must be transformed to work with the upgrade system before they are loaded by the application. This transformation is done by the upgrade system and must be performed after compile time and before load time.

Throughout this section, we shall see examples based on the geometric application described in Section 2.2.1. Although the classes are modified at the bytecode level, we shall see the examples at source code level, because source code is easier to understand. Some modifications however are not possible to express using source code. In such cases, I shall use bytecode.

The bytecode modification is performed using ASM [8]—an all purpose Java bytecode manipulation and analysis framework.

6.3.1 Handle Generation

Before the bytecode is modified, the upgrade system must generate the required handle types based on the class hierarchy. As I described in Section 5.3.1, each upgradable type uses an handle type of its own. In the following, consider that classes `Point` and `Rectangle` are non-upgradable types, and that class `Square` is the only upgradable type.

Multiple Inheritance

Conceptually, handles use multiple inheritance, as Figure 5.13 shows. Unfortunately, Java does not support multiple inheritance. To generate each handle type, the upgrade system uses a different strategy. It starts by generating a copy of the general `Handle` class per each non-upgradable inherited type. In this example, class `Rectangle` is the only non-upgradable class that needs an handle. The upgrade system generates a copy of the class `Handle` and names it `Handle$Square` because this handle type will be used by the upgradable type `Square`. The handle is a non-upgradable class, thus it will not change in future versions of classes that inherit from class `Rectangle`. Thus the handle must override all inherited methods, to enable that future versions of class `Square` can override methods of the parent class `Rectangle` that the current version does not.

Implementing Inherited Methods

Figure 6.8 shows the implementation of inherited methods on `Handle$Square`. Its constructor is invoked from upgradable code, when creating instances of class `Square`. Thus, its implementation invokes the contructor of the superclass `Rectangle`. The remaining (non-constructor) methods are invoked from non-upgradable code, when using instances of class `Square` as instances of class `Rectangle`. Thus, their implementation invokes the respective method on class `$Rectangle`. At this point, we must recall that upgradable class `Square` no longer inherits from non-upgradable class `Rectangle`. Instead, the handle class `Handle$Square` keeps that inheritance relationship for class `Square`. Thus, the upgrade system must generate an interface `$Rectangle` that matches the interface of the non-upgradable class `Rectangle`. Upgradable class `Square` implements that interface, enabling that the upgrade system invokes methods of class `Rectangle` on instances of class `Square`. I shall discuss this approach in further detail in the following section.

```
 1  class Handle$Square extends Rectangle {
 2    ...
 3    public Handle$Square(Object rectangle,Point topLeft,Point botRight) {
 4      super(topLeft,botRight);
 5      this.object = rectangle;
 6    }
 7
 8    public double area$super() { return super.area(); }
 9
10    public double area() { return (($Rectangle) (Handle$Square.get(this)).area(); }
```

Figure 6.8: Implementation of inherited methods on handle class `Rectangle`. Type `$Rectangle` is the interface of class `Rectangle`, generated by the upgrade system.

There are some methods, such as `double area()`, that `Square` does not override. Moreover, `Square` may invoke `Rectangle`'s methods. To handle these situations properly, the upgrade system generates extra methods on `Handle$Square` and overrides all inherited methods on `Square`. Figure 6.8 illustrates this solution. For each method on class `Rectangle`, class `Handle$Square` has an extra method that dispatches to `Rectangle`. In this case, the only extra method is named `area$super`.

There is an interesting detail on the implementation of method `Handle$Square.area()`, in Figure 6.8: the handle uses itself to access the upgradable type instance that it keeps. To understand why, we must take into account that handles are responsible for triggering object conversion. If a new version of `Square` is installed and `Handle$Square` manipulates an old instance while using the new program, it may miss the conversion and generate runtime errors.

Generating Interfaces

Taking a closer look at the implementation of method `Handle$Square.area()`, in Figure 6.8, we see that the upgrade system performs a cast on the expression `Handle$Square.get(this)`, on the end of line 9. The upgrade system intends to dispatch the execution to the method `Square.area()`. It cannot cast the result directly to class `Square` because it may change and class `Handle$Square` cannot. If class `Handle$Square` refers directly to class `Square`, the later ceases to be upgradable.

The upgrade system cannot cast an instance of class `Square` to class `Rectangle`. This solutions seems valid because, if we recall the geometric application example, described in Section 2.2.1, class `Square` is a subclass of class `Rectangle`. However, the upgrade system modified class `Square` because it inherits from a non-upgradable type, as Section 5.3.3 explains. As a result, class `Square` is not a subclass of class `Rectangle` anymore. Thus, this approach does not work either.

Although class `Square` exports the interface of class `Rectangle`, the JVM does not know that. The JVM must have that information to enable using instances of class `Square` through the interface of class `Rectangle`. The upgrade system informs explicitly the JVM by generating an interface `$Rectangle` that matches the interface of class `Rectangle`, and that class `Square` implements. Now, the upgrade system can cast instances of class Square to $Rectangle, which in turn allows the upgrade system to invoke methods of class Rectangle on instances of class Square. Figure 6.9 shows the generated interface $Rectangle.

```
interface $Rectangle {
  double area();
}
```

Figure 6.9: Interface that the upgrade system generates for non-upgradable class Rectangle. This interface enables invoking methods of class Rectangle on upgradable types that inherit it, after the upgradable types are modified as Section 5.3.3 describes.

6.3.2 Dereferencing Handles

Handles are responsible for representing the identity of all objects belonging to upgradable types. Thus, all fields, method parameters, local variables, and return types that are upgradable must be replaced by handles. This modification is described in detail in Section 5.3.2. Figure 5.11 shows the structural changes that the upgrade system performs to upgradable types Point and Rectangle. Besides this structural modification, some behavioural modifications are in order on the methods of upgradable types.

In this section, I shall describe behavioural modifications in bytecode, instead of Java source code. In bytecode, behavioural modifications are very simple: they detect a bytecode pattern and replace it by a different bytecode sequence. The direct mapping to source code may be too verbose or too complex to show such simple modification.

Consider the invocation of the method Point.distance that is made by method Rectangle.area in the example shown in Figure 2.1. After the upgrade system performs the structural modification, parameters p1 and p2 are instances of class Handle$Point instead of class Point. So, the direct execution of the body of method Point.distance would result in a runtime error.

There are several operations that require a direct reference to the instance instead of a reference to the handle. For instance, invoking methods and manipulating fields are operations that fail if they are performed on instances of unexpected types. To solve this, the upgrade system dereferences the handle whenever it detects an operation of this sort.

All the examples shown throughout this section start from the example shown in Figure 2.1 and after the modification shown in Figure 5.11 is performed. I shall also use the following notation to display the contents of the JVM stack: `item1 - item2 - item3`, where item1 is at the top of the stack and item3 is at the bottom.

Reading Fields

The instruction used for reading a field of an instance—getfield—expects a direct reference to that instance on top of the stack. After the upgrade system replaces all upgradable types by handles, this instruction finds an handle instead of a direct reference. Thus, the upgrade system must add some instructions to dereference the handle before every getfield instruction. Figure 6.10 shows that modification, in the context of the method Rectangle.area.

There is an interesting detail in this modification. The resulting type of the original instruction was an upgradable type. Therefore, the resulting type of the modified instruction must be an handle to be

```
1 //Original Bytecode Sequence
2 GETFIELD Rectangle.topLeft
3
4 //Modified Bytecode Sequence
5 INVOKESTATIC Handle$Rectangle.getObject(java.lang.Object)
6 CHECKCAST Rectangle
7 GETFIELD Rectangle.topLeft
```

Figure 6.10: Bytecode modification performed for instruction `getfield`, on the context of upgradable type `Rectangle`. The upgrade system injects the instructions on lines 5 and 6, replacing the handle reference on top of the stack by a direct reference.

consistent with the structural modification shown in Figure 5.11.

Writing Fields

The instruction used for writing a field of an instance, `putfield`,expects the new value of the field, `newval`, and a direct reference to the instance that owns the field, `ref`. The stack order must be: `newval - ref`. The expected direct reference `ref` is a reference to an handle, which must be dereferenced.

There is a possible place where the upgrade system could inject the instructions that dereference the handle: immediately after the instructions that puts the handle in the stack. However, this solution does not work in all cases. For instance, Figure 6.11 shows a segment of the bytecode of a method of an upgradable type `T`. The signature of this method is `T method(T,boolean)`.

```
1 ...
2 ALOAD_1
3 ILOAD_2
4 IFEQ 8
5 ICONST_0
6 PUTFIELD
7 GOTO 9
8 ARETURN
9 ...
```

Figure 6.11: Bytecode example that shows a reference being used on a conditional control structure. The reference that the instruction on line 2 generates may be used by the instructions on line 6 and 8.

After the upgrade system replaces the upgradable types, the signature of the method shown in Figure 6.11 becomes `Handle$T method(Handle$T,boolean)`. The first argument of the `putfield` instruction is put on the stack on line 2. This argument may be also used by the instruction in line 8, which expects an handle. Thus, if the upgrade system injects code to dereference the handle after line 2, it must also inject code to regain the handle before line 8. This is very cumbersome. The implementation that I propose follows a different path.

Instead of injecting the code that dereferences the handle immediately after the instruction that puts that handle on the stack, the upgrade system injects such code immediately before the instruction that expects the direct reference. In the example shown in Figure 6.11, the upgrade system would inject the code before line 6. But the handle that we wish to dereference is not on top of the stack. Thus, the upgrade system must generate code to swap the top two values of the stack to be able to dereference the handle. Figure 6.12 shows that modification, in the context of the constructor of class `Rectangle`.

```
1 //Original Bytecode Sequence
2 PUTFIELD Rectangle.topLeft
3
4 //Modified Bytecode Sequence
5 SWAP
6 INVOKESTATIC Handle$Point.getObject(java.lang.Object)
7 CHECKCAST Point
8 SWAP
9 PUTFIELD Rectangle.topLeft
```

Figure 6.12: Bytecode modification performed for instruction `putfield`, on the context of upgradable type `Rectangle`. The upgrade system injects the instructions on lines 5 through 8, replacing the handle reference on the top second position of the stack by a direct reference.

There are still some cases where this modification does not work. Some instructions expect a data type that occupies two stack slots, for instance a `double` or `long`. The `swap` instruction fails when there is this sort of data on top of the stack. In such cases, the upgrade system generates another modification that results in the same behaviour: bringing the handle reference to the top of the stack, dereferencing it and putting the direct reference on the initial position, where the handle was. Figure 6.13 shows the instructions that are generated for instruction `putfield Point.x`, as well as the stack state in each step of the modification.

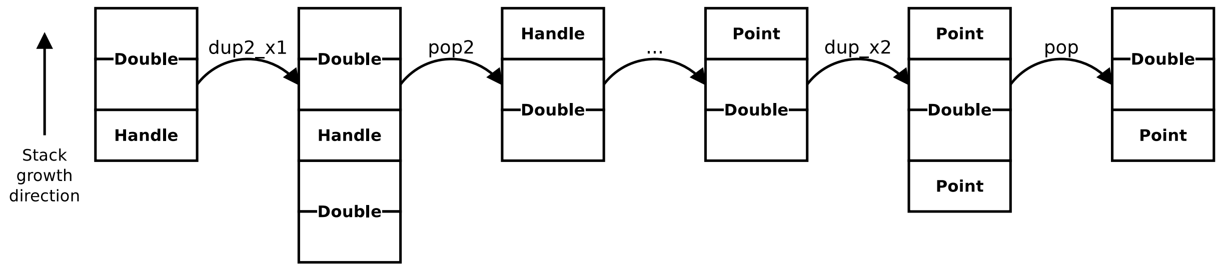


Figure 6.13: Bytecode modification performed for instruction `putfield`, on the context of upgradable type `Point`. Please note that the operand has double precision, thus the handle is on the top third position of the stack. The ellipsis represent the instructions on lines 6 and 7 on Figure 6.12. At the left of the arrow with the ellipsis are the instructions equivalent to the first swap, at the right are the instructions equivalent to the second swap.

The type of the written field is non-upgradable, it is of type `double`. Thus, after the sequence of instruction shown in Figure 6.13, the field is written by the same unmodified initial instruction `putfield Point.x`.

Invoking Methods

There are four instructions that invoke methods in the JVM: `invokespecial`, `invokevirtual`, `invokeinterface` and `invokestatic`. With the exception of the last one, all these instructions expect a direct reference to the **receiver** of the method. The receiver is, conceptually, the object on which the method is invoked. For instance, object `a` is the receiver of method `m` when the programmer writes: `a.m()`. After the upgrade system replaces the upgradable types by handles, as I described in Section 5.3.2, the receiver is an handle instead of a direct reference. So, once more, this handle must be dereferenced before the invoke instruction is executed.

The invoke instructions expect the receiver as the first argument to be put on the stack. All other argument are put after the receiver. Thus, the receiver may appear arbitrarily deep in the stack when the invoke instruction is executed. This invalidates the strategy described in the previous section, where the upgrade system generates instructions that dig through the stack, dereference the handle, and put the direct reference back in the same position on the stack where the handle was.

The solution that I propose uses **trampoline methods** to dereference the handle and call the respective method on the result. Figure 6.14 shows the generated trampolines for classes `Point` and `Rectangle`, described in the geometric application example in Section 2.2.1. The upgrade system performs this modification after replacing all the upgradable types by handles, so the starting point of this example is shown in Figure 5.11. Besides generating the trampoline methods, the upgrade system also replaces every `invokevirtual` and `invokeinterface` instruction to an invocation to the corresponding trampoline. Figure 6.14 shows the usage of trampolines in method `Rectangle.area`, lines 11 to 13.

```
 1  class Point {
 2    double distance (Handle$Point p) {  ... }
 3
 4    double static $distance (Handle$Point rec,Handle$Point p) {
 5      return ( (Point) Handle$Point.getObject(rec) ).distance(p);
 6    }
 7  }
 8
 9  class Rectangle {
10    ...
11    double area() { return  Point.$distance(topLeft,topRight)
12                            *
13                            Point.$distance(topLeft,botLeft);
14    }
15  }
```

Figure 6.14: Trampoline method generated for method `Point.distance`, and respective usage in method `Rectangle.area`. Trampoline methods are used because the handle reference, which must be replaced by a direct reference when invoking methods, may be arbitrarily deep in the stack. Instead of digging through the stack to replace that reference, the upgrade system uses trampoline methods to correctly invoke the method.

Receiver Reference

When the upgrade system is modifying the bytecode, it changes the types of the arguments,local variables, and return types which type is upgradable. The upgrade system changes those types to the respective handles types. If a method passes an upgradable type argument to another method, it is passing an handle instead of a direct reference. If the other method belongs to an upgradable type, it is already expecting an handle. Otherwise, if the other method belongs to a non-upgradable type, the handle acts as a proxy of the upgradable type. Please note that the upgrade system must always pass handles instead of direct references, as Section 5.3.2 explains.

However, there is still a direct reference that has not been modified to an handle: In the Java programming language, all non-static methods have a reference to the receiver of the method—the object in which the method was called. Inside methods, the programmer accesses the receiver using the reference `this`. In bytecode, there is no special instruction to deal with the receiver. Instead, the JVM specification states that it must be present on local variable number zero in non-static methods. At the bytecode level, the method arguments are present on the local variables following the receiver. The remaining local variables are the local variables that the programmer declared on the source code. I shall call this the source code local variables, to distinguish them from the bytecode local variables.

Both these two last groups of local variables, the method arguments and the source code local variables, are handle types. But the receiver reference still is a direct reference to an instance. It must be replaced by a reference to the handle of the receiver object.

A possible implementation of this solution is to add instructions that get the handle from the receiver before every instruction that reads the receiver. However, I propose a simpler solution: to add a preamble to every non-static method that replaces the receiver by its handle. This preamble would be written in Java, if possible, as `this = this.$handle`. The upgrade system generates a field `$handle` on all upgradable types. It is a private field that refers to the handle that represents the identity of the object. Figure 6.15 shows the bytecode preamble that the upgrade system adds to all non-static methods.

```
ALOAD_0
GETFIELD $handle
ASTORE_0
```

Figure 6.15: Bytecode preamble that replaces the receiver reference. The upgrade system injects this preamble at the beginning of every non-static methods.

Upgradable Type Instantiation

Let us recall the distinction made between objects and instances, at the beginning of Section 5.3.1. Programmers write programs that manipulate objects, which exist conceptually at the source code level. When the program is executing, it manipulates instances. Objects are therefore mapped to instances. Typically the mapping is simple: An object is also an instance.

However, the upgrade system that I propose uses a different mapping. Each upgradable object is mapped to two instances: (1) a non-upgradable instance, which is the handle that is responsible for keeping the object's identity between upgrades; and (2) an upgradable instance, which represents that object on the current program version. Thus, the upgrade system must detect when the program is instantiating an upgradable instance and replace those instructions for another set of instructions that instantiate both the handle and the upgradable instance correctly.

At the source code level, an instance is created and initialized at the same time using the keyword `new`. On the other hand, at the bytecode level, there is an instruction to create an instance, `new`, that does not initialize the instance. To do that, the bytecode must invoke explicitly a constructor using the instruction `invokespecial`. Constructors appear as methods named `<init>`. An instance can only be used when it is fully initialized, that is, after invoking the constructor of the superclass `java.lang.Object` on that instance.

Consider the constructor of the upgradable class `Rectangle`, in the basic geometric application example, shown in Section 2.2.1. Figure 6.16 shows the bytecode of the constructor after the upgrade system modifies it. The upgrade system needs to generate instructions to create and initialize the handle on the constructor. Thus, it injects a set of instructions after the instruction that invoke the constructor of the superclass `java.lang.Object`. Then, the receiver reference is replaced, as explained previously. At this point, the rest of the constructor body may be executed normally.

On the upgradable class `Square`, the bytecode modification is much simpler. This class inherits from

```
//Before
ALOAD_0
INVOKESPECIAL java.lang.Object.<init>()
...

//After
ALOAD_0
INVOKESPECIAL java.lang.Object.<init>()
ALOAD_0
NEW Handle$Rectangle
DUP
ALOAD_0
INVOKESPECIAL Handle$Rectangle.<init>(java.lang.Object)
PUTFIELD Rectangle.$handle
...
```

Figure 6.16: Bytecode injected to generate the handle on the constructor of upgradable type `Rectangle`. The upgrade system creates and initializes the handle right after it the invocation of the constructor of the superclass `java.lang.Object`.

another upgradable class, `Rectangle`. Thus, its constructor must invoke the constructor of the superclass, which is already modified so it creates and initializes the handle. Therefore, the upgrade system does not need to inject instructions to generate the handle.

Consider now that classes `Rectangle` and `Point` are non-upgradable types, and that class `Square` is the only upgradable type. If we look at the constructor of class `Square`, on Figure 2.1, we see that it invokes a constructor on the superclass. The same constructor must be invoked when creating the handle `Handle$Square`. When the upgrade system finds this situation, it replaces the invocation of the constructor of the parent non-upgradable type by an invocation to the constructor of class `java.lang.Object`, thus initializing the upgradable instance. Then, when it is initializing the handle, it invokes a constructor of the handle type `Handle$Square`, which in turn will invoke the required constructor on class `Rectangle`. Figure 6.17 shows the relevant portion of bytecode of the class `Square` in this example.

```
//Before
ALOAD_0
//Create arguments for Rectangle’s constructor
INVOKESPECIAL Rectangle.<init>(Point,Point)
...

//After
ALOAD_0
INVOKESPECIAL java.lang.Object.<init>()
ALOAD_0
NEW Handle$Square
DUP
ALOAD_0
//Create arguments for Rectangle’s constructor
INVOKESPECIAL Handle$Square.<init>(java.lang.Object,Point,Point)
PUTFIELD Square.$handle
...
```

Figure 6.17: Bytecode injected to generate the handle on the constructor of upgradable type `Square`. The upgrade system creates and initializes the handle right after it the invocation of the constructor of the superclass `java.lang.Object`.

So far, we have seen examples of two upgradable classes with different inheritance: one inherits directly from a non-upgradable type and the other inherits from an upgradable type. At this point, we can draw some interesting conclusions. The first conclusion is that the upgrade system must inject instructions to generate the handle the constructors of all upgradable types that inherit from a non-upgradable type (`java.lang.Object` at the limit), right after the instruction that invoke the constructor of the non-upgradable parent type. The second conclusion is that the upgrade system does not need to do such task on upgradable types that inherit from another upgradable type.

To better understand the second conclusion, consider that an upgradable type `U` inherits from another upgradable type. All of `U`'s constructors are required to invoke a constructor of the parent class, which in turn is required to do the same until this invocation chain reaches the constructor of class `java.lang.Object`. Therefore, the invocation chain will eventually reach the constructor of an upgradable type that inherits from a non-upgradable type. We have seen that the upgrade system injects instructions to generate the handle on all upgradable types that inherit from non-upgradable types. Thus, after the invocation of the parent constructor, `U`'s constructors may use the handle because some constructor along the invocation chain has generated it correctly.

6.3.3 Renaming Upgradable Types

Running programs are upgraded by loading new versions of their classes and converting the program state from the old classes to the new, instead of evolving existing classes, as we saw in Section 5.3. To the programmer, the new versions of the classes have the same name as the versions currently in execution. However, they are different classes at the JVM level.

In the JVM, it is not possible to have two classes with the same name inside the same classloader. So, to keep the same name, the upgrade mechanism could use different classloaders for different versions of the program. Figure 6.18 shows an example of such solution. The classloader `parent` loads the main classes of the upgrade system and the non-upgradable types. Classloader `CL1` loads the initial version of the program, classloader `CL2` loads the first upgrade of the program, and so forth.

Unfortunately, this classloader organization does not work if we want to run the transform functions. These functions are defined at the new version definition and must manipulate objects of the old version of the program. In the example shown by Figure 6.18, class `Point` loaded by classloader `CL2` must be able to manipulate objects of class `Point` loaded by classloader `CL1`. This is not possible because they are in different branches of the classloader delegation tree. Classloader `parent` is not able to manipulate objects either from classloader `CL1` or classloader `CL2` because classloaders are not able to use classes loaded by child classloaders.

Therefore, I follow a different strategy to enable loading different versions of the same class: the upgrade system renames all classes belonging to an upgrade before the JVM loads them. The naming mechanism, described in Section 5.3, uses the original name of the class followed by an "$" character and the upgrade version number of when that class was introduced to the upgrade system. For instance, the system starts with class `Point$1` and installs an upgrade class `Point$2`.

When debugging the application, the programmer sees the renamed classes. Even though it is different from what he is used to, this is an advantage because the new name still reflects the original name of that class plus the version number existing when that class was installed. Such naming enables the programmer to know exactly in which version is a given instance of any class.

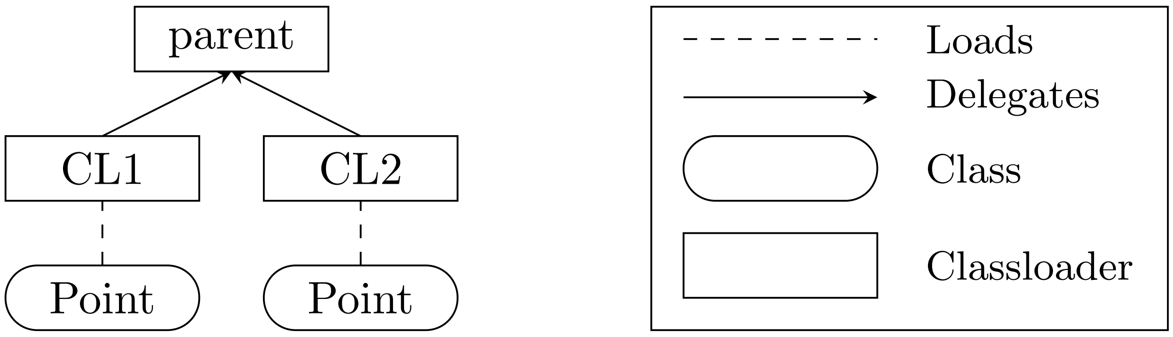


Figure 6.18: Usage of different classloaders for different program versions. The classloader `parent` loads the main classes of the upgrade system and the non-upgradable types. Classloader `CL1` loads the initial version of the program and classloader `CL2` loads the first upgrade of the program.

6.3.4 Transform Functions Arguments

The programmers use the transform function to migrate the instances from the old version of the program to the new one. The signature of a transform function of class `Class` is: `convert(old.Class oldI,Class newI)`. This transform function is defined on the new program version. The upgrade system generates the type `old.Class` to enable the programmer to refer to the old version of the program. On installing an upgrade, the upgrade system removes the old classes and renames the new version of class `Class`, as described in Section 6.3.3. Thus, the signature of the transform function becomes: `convert(Class$1 oldI,Class $2 newI)`. Afterwards, the upgrade system replaces all upgradable types by the respective handle, as described in Section 5.3.2, changing the signature of the transform function to: `convert(Handle$Class oldI,Handle$Class newI)`.

Both the arguments of the transform function have now the same type: the handle that keeps the identity of the object being converted. If we recall that an handle is responsible for keeping the identity of an object across upgrades, we find that both the arguments of the transform function have the same value. That value is the instance of the old version of the program, considering that transform functions execute in the past logical time, as described in Section 6.1.3. The transform function initializes the new instance based on the data existing on the old one. Letting this code to execute results in overwriting the old instance instead of migrating the state to a new one.

To overcome such problem, the upgrade system processes the transform functions as a special case. It explicitly modifies the signature of the transform function to: `convert(Handle$Class oldI,Class$2 newI)`. Therefore, the transform function is able to manipulate both the old program state and the new one. However, the conversion code uses a direct reference to the instance of the new program version. The programmer should not pass this reference to any other place.

6.4 Discussion

This chapter described the implementation details of the upgrade system that I proposed in Chapter 5. I started by describing the main classes of the upgrade system, which I called the structural classes. I also explained how the interaction of the structural classes dictates the behaviour of the upgrade system.

Afterwards, I described the transform functions, which are the tool that the upgrade system provides for the programmer to specify how to migrate the program state between upgrades. I described the syntax of the transform functions, how the programmer manipulates the old program state, and what post processing do the upgradable types require to enable correct operation of the transform functions.

Finally, I described the implementation details of the handles, whose task was described previously in Section 5.3.1. I described how handles are generated before executing the first version of the program and how handles are automatically and transparently inserted on the upgradable types without changing the program semantics.

Chapter 7

Conclusion

In this concluding chapter, I present the validation of this work, the main contributions, the greater limitations that I have found, the major challenges that I had while developping this work, future directions that can complete or improve this work, and some open issues that I have not addressed in this document.

7.1 Validation

This dissertation thesis is that it is possible to implement an upgrade system that can upgrade an application without restarting it, based on programmer defined transform functions that migrate the state between two consecutive versions of the program, that uses a Software Transactional Memory to provide strong atomic semantics to the programmer writing the transform functions, and that can be integrated seamlessly with modern software engineering methodologies.

In Chapters 5 and 6 I described the design and implementation of the upgrade system that I propose. I identified the major problems, proposed solutions, and exemplified how that solution solves the problems identified. Furthermore, I described how to implement such upgrade system, showing that the implementation is possible.

The purpose of this section is to complement the thesis' validation that was given along each proposal.

7.1.1 Prototype

To validate the solution presented in this document, I developed a prototype of the proposed upgrade system. Such prototype represents a proof of concept that demonstrates the feasibility and correction of the proposed upgrade system. The prototype is written in Java and uses JVSTM as its Software Transactional Memory.

Using the upgrade system prototype, the programmer is able to upgrade a running Java application without stopping it nor requiring a long pause to convert all program state at once. The upgrade system converts the program state as the natural control flow of the program touches the instances that are still in a previous version. The semantics of the upgrade system is defined in Section 5.2.

The programmer can define transform functions, which initialize an instance of the new version of the object given an instance of the old version of that same object. Section 6.2 shows examples of transform functions and describes how the programmer can refer to the old program state when writting them.

The prototype does not implement the integration with the revision control system that I describe in Section 5.4.

7.1.2 Goals

In Section 1.2, I established a set of goals that the upgrade system should reach. In this section, I discuss how the prototype that I described in the previous section reaches those goals.

Flexibility

The developed prototype supports all types of upgrades described in Section 2.2. These upgrades are direct modifications of upgradable types. As for the inheritance issues described in Section 5.3.3, the developed prototype supports them on upgradable types only. Upgradable types may freely change their position on the class hierarchy as long as they do not inherit (directly or not) from a non-upgradable type. This limitation occurs because the solution of generating several handle types, suggested in Section 5.3.3, was not implemented. As a result, any non-upgradable type that attempts to use an upgradable type either uses it as an instance of class `java.lang.Object` or this operation results in a runtime error.

Timing

The upgrade system uses a Software Transactional Memory to make every operation atomic. This means that every operation either completes successfully, changing the program state, or aborts without performing any modification to the program state. This also holds for upgrade related operations.

The upgrade system defines two new types of transactions: (1) upgrade transaction, that installs a new program version, and (2) conversion transaction, that converts an instance of an upgradable type. All conversion transactions execute at the same logical time as the upgrade transaction that defines the transform functions that they execute. Thus, the programmer expects clear and strong immediate upgrade semantics. However, the upgrade system executes the conversion transactions lazily, as the control flow touches the instances for the first time after the upgrade is installed. There are situations where what the programmer expects differs from what the upgrade system performs. I name such semantical mismatches paradoxes and pinpoint the exact conditions that enable paradoxes to happen in Section 5.2.4.

A transaction can last long enough to start executing in one version of the program and end in another version. This happens if an upgrade transaction starts and commits while the long lasting transaction is executing. I have shown in Section 5.2.5 that the transactional model of the proposed upgrade system deals with this situation with two possible outcomes.

- If the long lasting transaction is a read-only transaction, it is serialized when it started, before the upgrade was installed. Thus, it commits without conflicting.
- If the long lasting transaction is a read-write transaction, it fails to commit and is retried after the upgrade is installed, in the new version of the program.

Ease of Use

The upgrade system enables the conversion of the program state through transform functions. These functions initialize an instance of the new representation given an instance of the old representation. Other systems, such as PJama [13] and Boyapati's et al work [6], use a similar concept.

Version aware code may be confusing because it deals with several versions of the program. On my work, such code is well contained inside the transform functions. They contain all the version aware code inside an application. The remaining application code remains the same as if there was no upgrade system. Moreover, the version aware code deals with two consecutive versions of the program state only. These characteristics simplify writing and understanding the transform functions.

Transform functions may access other parts of the old program state besides the old representation of the instance. However, they may generate paradoxes by modifying parts of the new representation of the program state. Thus, I strongly suggest that transform functions are used only to initialize the new representation of the instance being converted.

To work correctly, the proposed upgrade system must introduce a level of indirection on the application. I developed tools that perform this task automatically and without changing the semantics of the application. The upgrade system is also able to detect situations when a paradox may occur, and abort the conversion transaction.

7.2 Contributions

The main contributions of this work are:

- Demonstrate that it is possible to implement an upgrade system that can upgrade an application without restarting it, based on programmer defined transform functions that migrate the state between two consecutive versions of the program, that uses a Software Transactional Memory (JVSTM, in this case) to provide strong atomic semantics to the programmer writing the transform functions, and that can be integrated seamlessly with modern software engineering methodologies.
- Propose a refactoring that enables handles to act as transparent proxies to upgradable types when they are used either by upgradable types or non-upgradable types
- Propose a framework for comparing upgrade systems according to how they solve problems common to all upgrade systems
- Extend the work of Boyapati et al [6] with the concept of paradoxes and specify in which exact conditions they can happen

7.3 Limitations

The main goal of my work is to develop an upgrade system that can upgrade the application without restarting it nor losing any part of its state due to the upgrade process. Besides this main goal, I established 3 other goals that the upgrade system must reach: (1) flexibility, (2) timing, and (3) ease of use. These goals are described in detail in Section 1.2. The upgrade system that I propose still has some limitations, regarding those goals. In this section, I describe such limitations.

7.3.1 Inheritance from Non-Upgradable Types

Upgradable types may have a non-upgradable portion. This happens when they inherit from a non-upgradable parent type. To deal with this characteristic, the upgrade system splits the upgradable objects into two instances: the upgradable instance and the handle.

The handle is responsible for keeping the identity of the object across upgrades, referring to the upgradable instance. The handle is a non-upgradable type and is also responsible for keeping the non-upgradable portion of the upgradable type, inheriting from the non-upgradable parent type. This introduces a limitation on the flexibility goal: the upgradable type must always inherit from the non-upgradable parent type because its handle, which is non-upgradable, inherits from that non-upgradable parent type.

7.3.2 Late Paradox Detection

A paradox happens when the lazy upgrade semantics of the transform functions do not match the immediate upgrade semantics that the programmer expects. Transform functions are executed in conversion transactions that differ from regular transactions on the checks that they make at commit time: Conversion transactions detect paradoxes at commit time, aborting the transaction if they detect a paradox. Detecting paradoxes this late is a limitation of the upgrade system that I propose. It means that the erroneous transform function must be installed and executed to detect the paradox. Moreover, in terms of recovering from the paradox, it is not clear what the upgrade system should do when it detects a paradox.

7.4 Future Work

The work described in this document took the first steps to design and implement a dynamic upgrade system that uses JVSTM and that allows the programmer to migrate the state from one version to the following using transform functions with strong atomic semantics. Nevertheless, there are several areas that the proposed solution does not cover, or that the prototype does not implement.

This section describes the limitations of the proposed upgrade system, and proposes solutions to those limitations.

7.4.1 Test Upgrades and Bypass Erroneous Upgrades

When writing software, programmers often introduce bugs inadvertently. Current development practices stress the software in a series of tests to detect (most of) the introduced bugs. Transform functions are not different from the rest of the software. Thus, they may contain bugs as well.

The upgrade system that I propose does not support testing an upgrade before installing it on the running system. Moreover, if an erroneous upgrade is detected, the upgrade system does not provide any support to bypass it with a new upgrade, skipping the erroneous upgrade in the instance conversion chain.

7.4.2 Earlier Paradox Detection

Paradoxes are detected too late, as Section 7.3.2 describes. They would be ideally detected before installing the upgrade, through testing the transform functions. The testing methodology used to perform early paradox detection is left as an extension to the upgrade system that this document describes.

7.4.3 Determine the Implications of the Limitations

The proposed upgrade system suffers from some limitations. For instance, an upgradable type that inherits (directly or not) from a non-upgradable type must always inherit (directly or not) from that non-upgradable type, as Section 7.3.1 describes. However, we need to know what is the significance of such limitations. Are they typical upgrades that the upgradable systems suffer frequently? Or are they rare modifications? The future analysis of the modifications performed at each upgrade on representative systems (such as the FénixEdu [3]) answers these questions.

7.4.4 Prototype

The upgrade system that I propose was validated through a prototype. This prototype is a proof of concept and demonstrates that the upgrade system is possible. However, it has several limitations that can be dealt with. This section explains those limitations.

Implement the Inheritance Solution

In Section 5.3.3, I identify the problem of introducing handles on upgradable types that inherit from non-upgradable ones, and I propose a solution to that problem. However, the prototype used to validate the upgrade system did not implement that solution. The implementation of such solution is a future extension of this work.

Measure the Overhead of the Upgrade System

To work correctly, the upgrade system introduces an extra level of abstraction on the upgradable application. Such indirection comes with an extra overhead cost. That overhead can be divided in two parts: (1) overhead measured after performing an upgrade, when all instances must be converted before being used, and (2) overhead measured when using the application with all instances converted, just accounting the indirection layer overhead.

The current prototype does not allow to measure any type of overhead because it is not able to insert that extra level of indirection on representative applications without changing their semantics. Such applications often have classes, that would be upgradable, that inherit from non-upgradable types.

A fully implemented prototype can be used in the future to measure the overall overhead that the upgrade system introduces.

7.5 Major Challenges

During the development of this work, the major challenges that I have found are as follows.

JVM Support The JVM does not support natively dynamic software upgrades. The upgrade system must be able to work without changing the runtime of the JVM.

Instance Identity The decision to introduce a new java class per class upgrade was not trivial to implement. Now, an object is represented by different instances on different program versions.

Atomicity Running JVSTM transactions in the logical past and understanding the implications (namely paradoxes).

Transparency Designing an upgrade system that does not change the typical software development process.

7.6 Open Issues

The upgrade system that this document describes raises some problems, solves most of them, and suggests future work to solve the remaining problems or improve the resulting upgrade system. However, there are some problems that this document does not approach. Such issues represent a considerable amount of work that deviates from the main goal of this dissertation. This section describes those open issues.

7.6.1 Garbage Collection

Garbage collection is the memory management method used by any JVM. The rational behind this mechanism is very simple: If all references to an object are lost, that object is inaccessible and, therefore, considered as garbage. The JVM reclaims and reuses the memory that that object occupied. This mechanism works effectively as long as the program that the JVM is running does not keep any references to unused objects.

JVSTM deals with garbage collection in its own way. It uses transactional locations (VBoxes) to store objects. When a transaction writes object `obj` on VBox `box`, that modification is not performed immediately on `box`. The VBox is modified only when the transaction commits successfully. Let us assume that the transaction was serialized with number `num`. When it commits, it inserts the pair `(num,obj)` on the VBox `box`. Thus, besides the current object, VBoxes also store an history of older values.

The history of values that each VBox keeps cannot grow indefinitely. To understand how JVSTM handles garbage collection, we must first understand why do VBoxes keep an history of values. Consider the example shown in Figure 7.1. Transaction `Tx1` starts, then transaction `Tx2` starts and commits, and finally transaction `Tx1` commits. Let us assume that transaction `Tx2` modifies a VBox that transaction `Tx1` reads at instant `B`. When transaction `Tx1` reads the VBox, it is not expecting the latest value, the one that transaction Tx2 stored on it. It is expecting the value that that VBox kept on instant A. Thus, the VBox can use the history of values to return the right value to transaction Tx1.

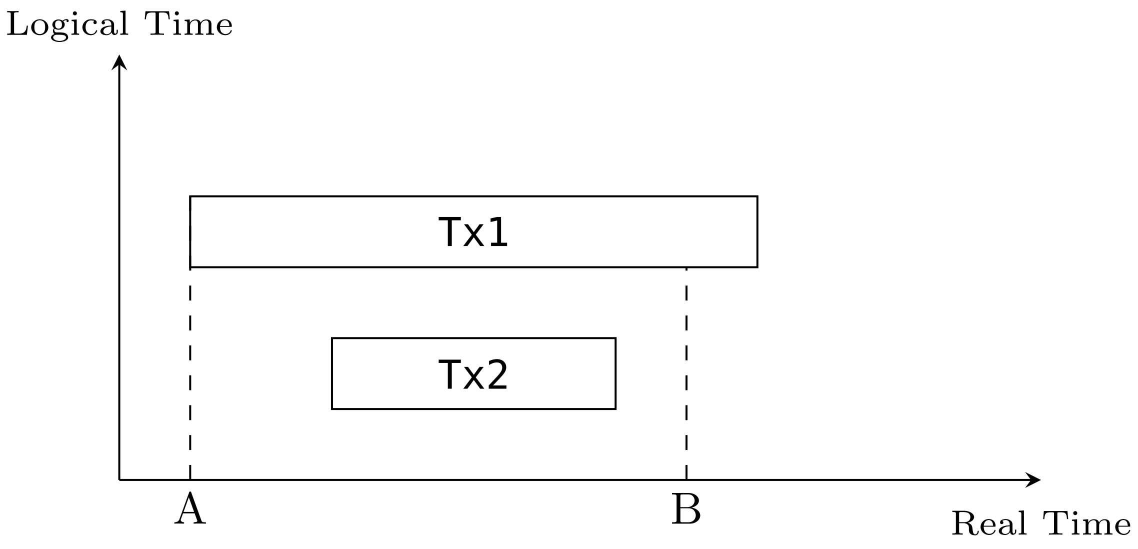


Figure 7.1: Overlapping JVSTM transactions. Transaction Tx2 starts and ends while transaction Tx1 is executing.

Consider that transactions Tx1 and Tx2 are the only transactions running on the system. When transaction Tx1 commits, no other transaction can access any value older than instant A. Thus, all VBoxes have to keep only the values that are more recent than the older transaction in execution. In this case, all boxes can lose all references to values that they keep that are older than instant A.

This assumption does not hold when we consider running upgrade transactions. They start on the logical past, thus they can access values that VBoxes kept at the time that the upgrade was installed. The upgrade system that I propose does not garbage collect the history of values that all VBoxes keep, nor does it discuss how to implement such feature.

7.6.2 Persistence

Typical applications persist some of their state using some persistence technology (for instance, databases). Doing so enables that the persisted program state outlives a single execution of the application. The application can now be stopped and restarted without losing any part of this persisted program state.

The persisted program state, as all types of program state, is tightly coupled with the program code. When reading the persisted program state, the application expects to find it structured in a specific way. However, the structure of the persisted program state may change from one program version to the following. Often, in applications that do not support dynamic software upgrades, the programmer needs to write a small program to migrate the persisted state before running the new program version.

The upgrade system that I describe in this dissertation does not discuss any approach to deal with persisted program state.

Bibliography

[1] Java(tm) platform debugger architecture. http://java.sun.com/javase/6/docs/technotes/guides/jpda/.

[2] Schema evolution in objectivity/db. http://www.objectivity.com/pages/downloads/view-document.asp?doc=/pages/downloads/whitepaper/pdf/SchemaEvolutionWP.pdf.

[3] Fénixedu. http://fenixedu.sourceforge.net, 2005.

[4] E. Bertino, G. Guerrini, and L. Rusca. Object evolution in object databases. In B. Franhoefer and R. Pareschi, editors, *Dynamic Worlds*, pages 219–246. Kluwer Academic Publishers, 1999.

[5] G. Bierman, M. Parkinson, and J. Noble. Upgradej: Incremental typechecking for class upgrades. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 235–259, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] R. Boyapati, B. Liskov, L. Shrira, C. hue Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 403–417, 2003.

[7] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[9] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[10] Y. Cheung. Lazy schema evolution in object-oriented databases. Master's thesis, Massachusetts Institute of Technology, September 2001.

[11] S. Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the European conference on Object-Oriented Programming*, pages 313–336. Springer-Verlag, 2000.

[12] V. Corp. Versant object database fundamentals manual. http://www.versant.com/developer/resources/objectdatabase/documentation, July 2005.

[13] M. Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications.* PhD thesis, University of Glasgow, May 2001.

[14] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the o2 object database system. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 170–181, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[15] M. Fowler. Language workbenches: The killer-app for domain specific languages? May 2005.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[17] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, November 2005.

[18] D. K. Kim and E. Tilevich. Overcoming jvm hotswap constraints via binary rewriting. In *In First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp 2008)*, 2008.

[19] S. Liang and G. Bracha. Dynamic class loading in the java tm virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998.

[20] T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] G. Steele. *Common Lisp the Language.* Digital Press, 2nd edition, June 1990.

[22] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4, Version 4.1. http://www.osgi.org/Download/Release4V41, 2007.

[23] ZeroTurnAround. JavaRebel. http://www.zeroturnaround.com/jrebel/.