View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

Incremental Product Release of Java Applications using Dynamic Updates

Relatori:

Prof. Bo Nørregaard Jørgensen

Prof. Marco Avvenuti

Prof. Alessio Vecchio

Candidato:

Luigi Fortunati

“If you're not failing every now and again, it's a sign you're not doing anything very innovative.”

Woody Allen

Acknowledgments

This dissertation, while an achievement that bears my name, would not have been possible without the help of others, who I would now like to thank.

First of all I want to thank the research team of the Mærsk Mc-Kinney Møller Institute that worked at my side during our collaboration. I would like to express my sincere gratitude to Professor Bo Nørregaard Jørgensen for his supervision and guidance, the Javeleon engineers Allan Raundahl Gregersen and Michael Rasmussen, with which I shared the working space. With their availability on answering my many questions they helped me to understand how Javeleon works and also taught me a lot about programming.

I also have to acknowledge my friend and colleague Andrea Mannocci because he managed to tolerate me during our stay in Denmark; we worked side by side everyday on our project sharing good and bad moments. His moral support and aid both at work and out of the office proved to be essential to me.

I'm indebted with Professor Marco Avvenuti, my home university advisor, for enabling me to go to Denmark to develop my thesis. I must also thank the teaching staff of University of Pisa for all the knowledge that I gained during the years of my university studies.

I also want to thank my relatives and people that are close to my family for the economical and moral support they gave me on undertaking my studies abroad. Moreover I also want to thanks my close friends in Italy and the Erasmus students that I met during my stay in Odense. Being an Erasmus student has been a wonderful experience, both personally and professionally.

Abstract (English)

With the evolution of the software development process, claimed by the new demands of the market, software development has changed considerably since its early days. In order to support the increase in complexity and dimensions of new software product, software engineers have developed new tools and methodologies in order to cope with the market requests. Despite all these advances, a software product can still be in need of changes after the delivery. These modifications are traditionally divided as changes to the functionality of the software that address new unanticipated requirements, changes that allow the software to run on a different environment, changes that fix errors and improvements that can avoid future problems. The maintenance and update process of an application have traditionally involved the classic halt, redeploy and restart scheme. However, this approach cannot be used in every scenario; consider as an example a high availability e-commerce system. For some companies the cost of a system shutdown can be prohibitive in terms of economic outlay, safety and the availability of service.

A Dynamic Software Updating system (DSU) allows overcoming the update problem enabling applications to be updated without recurring to the halt-update-redeploy scheme.. Many DSU systems have been developed since the ‘70s, each of them comes with some peculiar properties defining on-the-run application updateability with a certain level of granularity by allowing only certain subset of modifications to the code.

In this work we examine a software-based DSU system called Javeleon, developed at the University of Southern Denmark – Mærsk Mc-Kinney Møller Institute in collaboration with Sun Microsystems. A novel feature of Javeleon is the support of full redefinition of classes and changes to the type hierarchy. Following the evolution of a case study application we will show how the capability of dynamically updating software with Javeleon impacts on software development process. By working with Javeleon we will also test the transparency of this system towards the programmer.

Abstract (Italian)

La domanda del mercato nell'ambito della produzione di software ha spinto il processo di sviluppo di applicazioni ad evolvere considerevolmente partendo dalle sue origini. Al fine di poter permettere la gestione di prodotti software sempre piú complessi e di dimensioni sempre maggiori gli ingegneri software hanno inventato nuove metodologie per venire incontro alle richieste del mercato. Nonostante i progressi raggiunti, un prodotto software richiede comunque di poter essere modificato in seguito al rilascio. Le modifiche che possono essere apportate durante la fase di manutenzione sono tradizionalmente suddivise in cambiamenti alle funzionalitá dell'applicazione, modifiche che permettono al software di adattarsi a nuovo hardware/software, correzioni di bug o errori dell'applicazione, modifiche all'applicazione che permettono di evitare problemi futuri. Il processo di update inoltre si traduce spesso nel classico schema di spegnimento dell'applicazione, aggiornamento offline e ripristino della operativitá. Questo genere di approccio non puó essere applicato in ogni caso, basti pensare a un sistema con garanzie di alta disponibilitá come un sistema di e-commerce. Per alcune compagnie infatti il costo derivante dallo spegnimento del sistema puó essere proibitivo per motivi economici, di sicurezza o di disponibilitá di un servizio.

Un sistema di Dynamic Software Updating (DSU) permette di aggiornare una applicazione mentre questa é in esecuzione limitando i problemi derivanti dallo shutdown del sistema. Fin dagli anni '70 molti sistemi DSU sono stati sviluppati, ognuno con le proprie caratteristiche e limitazioni in quanto riguarda le modifiche possibili al codice.

In questo lavoro esaminiamo un sistema DSU software-based chiamato Javeleon e sviluppato alla University of Southern Denmark - Mærsk Mc-Kinney Møller Institute in collaborazione con Sun MicroSystems. Javeleon introduce una nuova caratteristica nel campo dei sistemi DSU, la possibilitá di poter ridefinire completamente le classi o modificare intere gerarchie di classi. Mostreremo, attraverso lo sviluppo di una applicazione, in che modo l'integrazione degli aggiornamenti dinamici con Javeleon influenza la fase di sviluppo del software specialmente nel momento in cui viene definita o modificata l'architettura dell'applicazione, testando effettivamente la trasparenza di tale sistema verso il programmatore.

Table of Contents

Acknowledgments iii

Abstract (English) iv

Abstract (Italian) v

Table of Contents vi

List of Figures ix

List of Tables xii

List of Code Snippets xiii

Chapter 1 - Introduction 1

1.1 Problem domain 1

1.2 Problem statement 2

1.3 Research methodology 4

1.4 Report structure 5

1.5 Notation 6

Chapter 2 - State of the art 8

2.1 Characteristics of a DSU system 8

2.2 Javeleon 10

Chapter 3 - Development methodology 12

3.1 Incremental product release 12

3.1.1 Staged model 12

3.1.2 Staged model with dynamic updates 14

3.2 Set-up and approach 16

3.2.1 IDE and Javeleon 16

3.2.2 Application development and Versioning 17

Chapter 4 - The case study application 20
4.1 Overview of the Space Invaders clone game 20
4.2 Description of each iteration 21
4.2.1 First release 21
4.2.2 Second release 22
4.2.3 Third release 23
4.2.4 Fourth release 24
4.2.5 Fifth release 25
4.2.6 Sixth release 25
4.2.7 Seventh release 26
4.3 Expected behavior 27
4.3.1 Updating from release 1 to release 2 29
4.3.2 Updating from release 2 to release 3 29
4.3.3 Updating from release 3 to release 4 29
4.3.4 Updating from release 4 to release 5 30
4.3.5 Updating from release 5 to release 6 30
4.3.6 Updating from release 6 to release 7 30
4.4 Initial design 31
4.4.1 Game logic 32
4.4.2 User input control 37
4.4.3 Representation 38
Chapter 5 - The update experiment 40
5.1 Version 2.0 42
5.1.1 Significant code changes 42
5.1.2 Update test issues and solutions 48
5.2 Version 2.1 61
5.2.1 Significant code changes 61

5.2.2 Update test issues and solutions 64
5.3 Version 3.0 68
5.3.1 Significant code changes 68
5.3.2 Update test issues and solutions 70
5.4 Version 3.1 74
5.4.1 Significant code changes 74
5.4.2 Update test issues and solutions 75
5.5 Version 4.0 77
5.5.1 Significant code changes 78
5.5.2 Update test issues and solutions 95
5.6 Version 5.0 97
5.6.1 Significant code changes 97
5.6.2 Update test issues and solutions 101
Chapter 6 - Good practices for developing dynamically updateable applications with Javeleon 103
6.1 Software evolution considerations 103
6.1.1 Management of code 103
6.1.2 Dynamic update approach 105
6.2 Application design considerations 107
6.2.1 Generic issues 107
6.2.2 Specific issues 120
Chapter 7 - Conclusions 125
Bibliography 127

List of Figures

Figure 1.1- Comparison between the Incremental Product Release approach and the Stepwise Refinement approach ... 3
Figure 1.2 - Research methodology ... 5
Figure 3.1 – Staged Model for the Software Life Cycle ... 14
Figure 3.2 - Move method refactoring ... 16
Figure 3.3 – Example of a software evolution schema ... 18
Figure 3.4 – Comparison between SVN branched and “flat” approach ... 19
Figure 4.1 – First release’s screenshots ... 22
Figure 4.2 – Second release’s screenshot ... 23
Figure 4.3 – Third release’s screenshot ... 23
Figure 4.4 – Fourth release’s screenshot ... 24
Figure 4.5 – Fifth release’s screenshot ... 25
Figure 4.6 – Sixth release’s screenshot ... 25
Figure 4.7 – Seventh release’s screenshot ... 26
Figure 4.8 – UML class diagram of game entities (First release) ... 34
Figure 4.9 –UML class diagram of collections (First release) ... 35
Figure 4.10 - Sequence diagram on the start of the game loop ... 35
Figure 4.11 - Sequence diagram related to the collision between shots and aliens (v1.0) ... 37
Figure 5.1 – Iterations development schema ... 41
Figure 5.2 - Extract superclass EntityCollection ... 42
Figure 5.3 - Collections of entities (v. 2.0 rev. 135) ... 43
Figure 5.4 – Aliens movement’s changes on Alien class ... 44
Figure 5.5 - Aliens movement’s changes on Aliens class ... 45
Figure 5.6 – Introduction of barriers ... 47
Figure 5.7 - Singleton design pattern ... 51
Figure 5.8 – Changes on the structure of the EntityCollection hierarchy tree passing from version 1.0 to version 2.0 rev. 135 ... 53
Figure 5.9 - Alien hierarchy tree (v 1.0) ... 54
Figure 5.10 - Alien hierarchy tree (v 2.0) ... 54

Figure 5.11 - Good design implementation on Alien class (Version 2.0 rev. 166) 55
Figure 5.12 – New design implementation on the Entity hierarchy tree 56
Figure 5.13 – New design implementation on collections hierarchy tree 57
Figure 5.14 - Example of "Push Down Field" refactoring 57
Figure 5.15 - New “Template method” design applied to version 1.0 (rev. 166) 58
Figure 5.16 – Changes on Barrier and Barriers regarding the vulnerability of barriers (v2.0->v2.1) 61
Figure 5.17 - Move method in version 2.0 rev. 167 63
Figure 5.18 - Move methods in version 2.1 rev. 171 64
Figure 5.19 - Incrementing the max health of barriers 65
Figure 5.20 - State mapping of barrier’s health following bad design 66
Figure 5.21 - Decrementing the max health of barriers 66
Figure 5.22 - Score system classes (Version 3.0 Rev. 179) 68
Figure 5.23- Change on Game class passing (v2.1->v3.0) 69
Figure 5.24 - Cascade modifications effect on version 3.0 (rev. 179) 71
Figure 5.25 - Methods changed in class Game (v3.0r179->v3.1r182) 75
Figure 5.26 - Code changes in version 4.0 related to the way entities are drawn on screen 78
Figure 5.27 – Changes to the code related to the introduction of explosions and animations (v4.0) 80
Figure 5.28 – Modifications to the code regarding the visualization and limiter of FPS (v4.0) 81
Figure 5.29 - Use of new interfaces in version 4.0 84
Figure 5.30 - Implementation of move method in version 4. 85
Figure 5.31 - Collision evaluation/handling system implemented in version 4.0 87
Figure 5.32 - Redesign on score framework (first solution) 90
Figure 5.33 - Redesing of score framework (second solution) 91
Figure 5.34 - Differences between v3.1 and v4.0 on shots 92
Figure 5.35 - Changes to the design concerning the introduction of FixedEntityCollection and MovableEntityCollection in version 4.0 94
Figure 5.36- Implementation of the Observer pattern (v.4.0) 94
Figure 5.37 – Modifications on entities hierarchy regarding the introduction of new classes of enemies (v4.0->v5.0) 97
Figure 5.38 - Modifications on collections hierarchy regarding the introduction of new classes of enemies (v4.0->v5.0) 98
Figure 5.39 - Move implementation of interface Collidable down in the hierarchy (v5.0) 99
Figure 6.1 - Revisions journal example 105

Figure 6.2 - Adding reference to Game in Aliens class - change to the constructor (v2.0) 109

Figure 6.3 – Singleton design pattern 110

Figure 6.4 - GameControl class 111

Figure 6.5 - Changing constants value - the barrier example 113

Figure 6.6 - Barrier's health solution with strong coupling of information 114

Figure 6.7 – Implementation of good design in our case study 117

Figure 6.8 – Changing the implementation of methods 119

Figure 6.9 - Program execution example for the "Extract and Inline method" refactoring 120

List of Tables

Table 4.1 - Definitions of valid and invalid dynamic updates 28

Table 5.1 - Fine grain modifications concerning extract superclass refactoring (v1.0->v2.0) 43

Table 5.2 - Fine grain modifications concerning aliens' movement (v1.0->v2.0) 45

Table 5.3 - Fine grain modifications concerning aliens' ability to make fire (v1.0->v2.0) 46

Table 5.4 - Fine grain changes concerning the introduction of barriers 47

Table 5.5 - Fine grain changes on Aliens and Shots classes concerning collision detection and handling (v1.0->v2.0) 48

Table 5.6 - Fine grain changes concerning "push down field" refactoring (v1.0->v2.0) 57

Table 5.7 - Evaluation of aliens speed 59

Table 5.8 - Fine grain changes related to the vulnerability of barriers (v2.0->v2.1) 63

Table 5.9 - Fine grain changes (v3.0->v3.1) 74

Table 5.10 - Fine grain changes related to the introduction of raster images (v3.1->v4.0) 79

Table 5.11 - Fine grain modifications regarding "Extract method" refactoring in class Game (v4.0) 82

Table 5.12 - Differences between Hashtable and Lookup approach with the collision engine 89

List of Code Snippets

Code Snippet 4.1 - Game loop 32
Code Snippet 4.2 - Game class fields 36
Code Snippet 4.3 - User input handling 38
Code Snippet 5.1 - TimerTasks and the infinite loop issue 49
Code Snippet 5.2 - Algorithm that evaluates alien speed 59
Code Snippet 5.3 - New methods of class Barrier (v.2.1) 62
Code Snippet 5.4 - Fine grain changes regarding movement of entities (v2.0->v.2.1) 64
Code Snippet 5.5 - gameLoop method (v2.1) 69
Code Snippet 5.6 - gameLoop method (v3.0) 70
Code Snippet 5.7 - Fine grain changes (v2.1->3.0) 70
Code Snippet 5.8 - Extract method refactoring on notifyDeath and notifyWin methods (v.4.0) 82
Code Snippet 5.9 - GameControl Class (v4.0) 88
Code Snippet 5.10 - Collision handler registration with inline initialization (v.4.0) 88
Code Snippet 5.11 - Version 3.1 EntityCollection class declaration 93
Code Snippet 5.12 - Version 4.0 EntityCollection class declaration 93
Code Snippet 5.13 - Game border classes (v5.0) 100
Code Snippet 5.14 - Initialization of borders in Game class 101
Code Snippet 6.1 - A method that contains an infinite loop 121
Code Snippet 6.2 - Solution provided for the infinite loop problem 121
Code Snippet 6.3 - Twingleton issue's solution on TopComponent class 123
Code Snippet 6.4- Adding abstract modifier to class 123

Chapter 1 - Introduction

In this chapter we start by examining what is the nature of the update problem and how Dynamic Software Updating systems address this problem. In the second part we describe what we want to demonstrate with our work considering a particular scenario and methodology. Finally, we present an outline of how the document is organized and a list of definitions concerning the terminology used in this work.

1.1 Problem domain

Considering the history of software, software engineering has evolved considerably starting from 1940's to our days. Software engineering can be seen as the intelligent application of proven principles, techniques and tools in order to effectively and efficiently create and maintain software that satisfies the user. Software engineers have constantly improved the techniques and the tools they use in order to create software. Just to make some examples, a lot of drivers contributed to the evolution of software engineering techniques or tools: programming languages born in the 50's gave later rise to elements that improved the development of software, like abstractions, modularity or information hiding (Liskov, Data Abstraction and Hierarchy, 1987). The advent of personal computer in the 70's made it possible for hobbyist to get a computer and write software with it; in the 90's the advent of Internet and open-source allowed a wider collaboration between developers. In the last ten years the increasing demand of software in smaller organizations led to the creation of new faster methodologies for developing software from the requirements to the deployment of the software product.

These lightweight methodologies advocate the importance of requirements volatility even late in the development and that is because software development processes that harness change towards customer satisfaction are more competitive on the market (Martin, et al., 2001). The support for rapid change in the requirements of a software and rapid development of new versions raises problems when dealing with programs or systems that have to run continuously without interruption. That is true for mission critical application which is a system critical to the proper running of a business. If this application fails for any length of time you may be out of business. In virtue of their nature these kinds of systems cannot be updated by using the traditional update approach that includes shutdown, update and restart. That approach is not acceptable because it can result in a loss of service for the users, an economic loss or

company image’s loss in reliability, or in the worst case it can compromise safety. Solutions to these kinds of issues are widely deployed today by using a hardware or software solution.

Hardware-based solutions for dynamic updating are based on redundancy of identical systems. In these systems an entire running program can be modified with minimum downtime and maximum flexibility by updating the software on the second system, while the first is still running, and then activate the second one after the update in order to provide the service while the first system is updated. Hardware based solutions are usually costly and difficult to implement. This feature makes the latter solution not suited for smaller organizations but for scenarios where a high level of fault tolerance is needed, like in telecommunication or critical transactional systems.

In order to give a dimension of hardware-based solutions we can consider the VisaNet transactional system. Whenever a purchase is made and paid with a Visa card in one of the available locations, the transaction is transmitted to a VisaNet datacenter. There are four installations of them in the world with a total of 21 IBM supercomputers. When the request is received the datacenter sends the card details to the visa card owner’s bank and receives an approval. Afterwards, an acknowledgment is sent back to the merchant. All of these operations are completed in less than five seconds. The VisaNet transaction engine can hold more than 3000 transaction per second and it’s based on 50 million lines of code which have been modified more than 20,000 times per year allowing though 99,5 percent of availability (Pescovitz, 2000).

Software-based updating systems enable software updates at runtime with less cost and complexity compared to the hardware-based solutions. A software-based approach allows the application to modify its behavior while it is running, dealing with transitioning state. However, these systems today still not permit the programmer to make every possible change to the code, but allow only a subset of update-compliant code-changes and well-accepted program structures. These systems also come with certain characteristics and are distinguished based on the granularity of the modifications that are allowed.

1.2 Problem statement

In this work we test how dynamic updates can be integrated in software evolution by developing several stable releases of a case study application that can be dynamically updated with Javeleon, a novel dynamic update framework developed at the University of Southern Denmark - Mærsk Mc-Kinney Møller Institute in collaboration with Sun Microsystems. Javeleon permits transparent dynamic updates of running Java applications while guaranteeing both type and thread safety with low overhead, high level of flexibility and programming transparency (Gregersen & Jørgensen, 2009) (Gregersen, 2010).

By software evolution we mean the process of developing software initially and then repeatedly update it to cope with requirements volatility. Allowing requirements volatility means that we will need to follow a software life cycle model that permits new requirements to be discovered after the software product is already been deployed. Following the categorization of Lientz and Swanson in (Bennet & Swanson, 1980) there are 4 kinds of maintenance operations that can be done:

- Corrective maintenance: reactive modification of a software product performed after delivery to correct discovered problems;
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability;
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

The model that is taken into consideration is the *Staged Model* explained in (Rajlich & Bennet, 2000). This model admits new requirements of the application to be discovered after the deployment of a release, letting the application grow in functionality version after version. Several releases of the case study application will be developed using an *incremental product release* approach in order to test the dynamic updateability from every release to the subsequent. This approach requires the dynamic update test to be done only from release to release evaluating the impact of dynamic updates considering all of the modifications made during the version development. A release will be ready only when all of the requirements of a version are satisfied by the changes made to the code. That approach is different from the stepwise evolution approach followed in (Mannocci, 2010) where dynamic test updates are made during the development of a single release as it is possible to see from the example below.

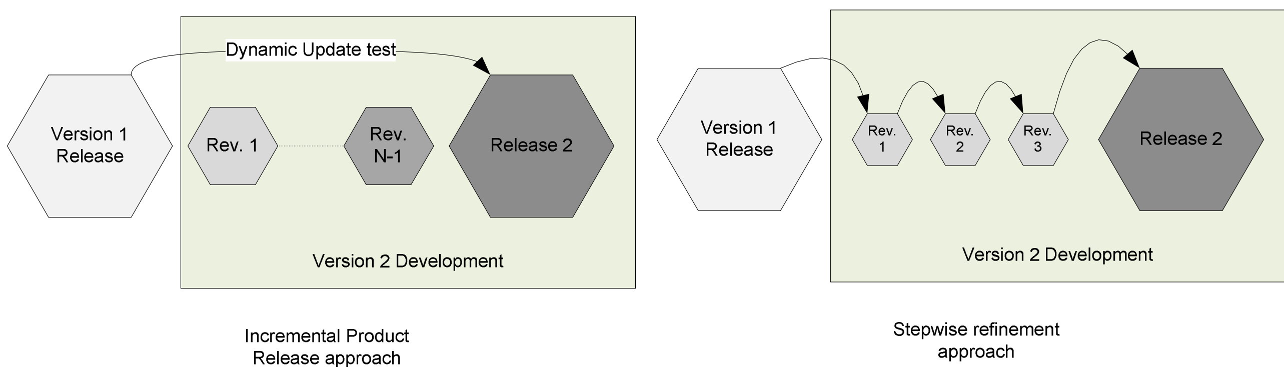


Figure 1.1- Comparison between the Incremental Product Release approach and the Stepwise Refinement approach

Every release will embody at least a new major feature (adaptive modification) and several other changes (corrective, perfective or preventive) to the code. We will explain what consequences come from following the Incremental Product Release model instead of the Stepwise Refinement approach.

Another aim of our work is to test the transparency of Javeleon towards the developer in order to check how it modifies the habits of the software engineer on making design choices for his application. We want to know what modifications to the code are valid while using Javeleon and meanwhile maintaining a correct application behavior with dynamic updates. By examining how Javeleon guide the programmer on taking certain design decisions we will write down a set of good practices to follow in order to develop applications that are dynamically update compliant with this DSU system and see what are the consequences of following these rules.

1.3 Research methodology

In order to test the Javeleon framework we developed 5 stable releases of a game. The application developed is a Java Space Invaders game clone. We chose to base our studies on a game development mainly because a game is usually a stateful application. That way we can easily see directly from the user perspective how data structures, which reside in memory, and the use of dynamic updates with Javeleon affect user experience and the application behavior.

Before starting the development of the game an update plan will count for every release a set of requirements to be implemented. These requirements often refer to the need of new functionalities that will involve adaptive modifications to the software. Moreover, during the development of a release the developer is allowed to make corrective, perfective, preventive modifications to the code. In order to simulate a genuine software evolution experience we also state that when the software engineer has to choose the design for a certain version he can take into consideration no more than the requirements for that version, ignoring the requirements that only future releases should provide. Allowing the developer to make unanticipated preventive, perfective or corrective modifications can also mean permitting him to refactor the code to any degree, potentially twisting the design of the application.

Every time a new software version is released the dynamic update with the previous version is tested. In this phase issues and undesired application behaviors of the application are detected and their causes identified. A solution for the identified issues is then formulated, applied, tested and then generalized. The result of this process should be a collection of good practices to follow when developing dynamically updateable application with Javeleon.

Write down the update plan

Develop the first release of the game following the requirements

Develop another release following the update plan requirements and making perfective/corrective/preventive modifications to the code.

Test dynamic updates between the previous and the latter version

Found issues or strange behavior after the update?

No

Yes

Identify the cause of the issue or the strange application behavior

Formulate a solution as a design rule and implement it

Generalize the solution

Figure 1.2 - Research methodology

1.4 Report structure

The organization of the rest of the thesis is as follows:

- Chapter 2 describes software DSU systems in general by describing the main characteristics that are used to categorize these systems and briefly explain Javeleon and its qualities.
- Chapter 3 gives in the first subchapter a description of the development paradigm followed as a software lifecycle model (staged model) and how dynamic updates impacts this model. In the second subchapter we describe from a more pragmatic point of view what are the methodologies, principles and tools used during the development of the application.

- Chapter 4 shows a first overview of the case study application, the functionalities required on each version and the desired behavior of the application when updating from version to version. The design of the first release of the program is also introduced.
- Chapter 5 deals with the workflow steps that are involved within the dynamic update experiment. For every version after the first a summary of the significant code changes is given. In this chapter we also describe what problems or undesired behavior were encountered during the dynamic update tests, which solutions where found and how these solutions were implemented in our application design.
- Chapter 6 summarize all the practices that were generalized during the dynamic update test phases and shows what are the implications of dynamic updates in a software products.
- Chapter 7 gives a general brief summary of what we set out to investigate, the main problems encountered and how these were solved them telling what the developer community can learn from our findings.

1.5 Notation

Definitions:

- **Software engineering**: application of engineering to the software development process. Is the disciplined application of methodologies that encompasses software design, implementation and testing.
- **Software development**: process of building a program according to given specifications.
- **Software evolution**: process of developing software initially, then repeatedly updating it in order to fulfill some new requirements.
- **Refactoring**: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written" (Fowler, 2000).
- **Dynamic updating**: in general it referes to the ability to change a computer program while in execution.
- **Signature of a method**: Set of properties of a method that identifies it. It comprehends the method's name and the parameter list (number, types and order).
- In this work we use the term **target class** to refer to the most recent version of the class. We use the term **former class** to refer any class version implementation but the target. At the moment of

the update the terms current target and new target class refer respectively to the target class before and after the update.

- **Instance Object State**: represent the value of the instance variables stored in memory.
- **Application state**: represent the union of all the application objects states.
- In-line initialization: initialization of a class field contextual to its declaration.
- **Iteration**: span of time during the development of the software when the application stakeholders can decide which new requirements the application needs.
- **On-line program change** and **dynamic update of a program** are terms used interchangeably
- **Module** or **component**: an entity that contains resources and classes, usually organized in packages. It provides and API that dependent modules can use, explicitly establishing dependencies between modules.
- The terms **object** or **class instance** are used interchangeably
- UML class diagrams are used in order to describe the modifications made to the application design. For the sake of simplicity in these diagrams some classes can partially or entirely hide information about their attributes, methods or relationships with other classes.
- A few code snippets are provided in order to show in a simplified fashion the program main algorithms.

Chapter 2 - State of the art

In this chapter we examine the characteristics of software DSU systems by first giving a theoretical description of what a DSU system is and then we introduce Javeleon as a general purpose dynamic update approach in this field.

2.1 Characteristics of a DSU system

In order to describe the characteristics of a framework that performs on-line changes to a program we can start by defining from a theoretical point of view the problem of dynamic updating following the definition in (Gupta, Jalote, & Barua, 1996). Before defining what is an online change to a program we should describe what a process is: a program in execution made up by code and state. The code is the application algorithm that in a dynamic-update free scenario should remain the same throughout the whole life cycle of the process. State is the complete characterization of the process and it starts from an initial state and evolves through program execution also thanks to the interaction between the program and the environment. Given a certain instant of time, state can be seen like a picture of the process that includes program counter, data structures in memory and also threads state. A state is said to be reachable for an application if the process, executing the program code, can let the state evolve to that condition at a certain time and considering particular environment interaction.

Systems for dynamic update of software allow swapping program code while the process is in execution, changing the behavior of the application at runtime and reducing the period of service unavailability typical of the shutdown-update-redeploy scheme. Changing the program of a process in execution introduces a problem of state compatibility. In fact, in order to obtain a valid update, the process state that is present after the on-line program change should be a reachable and correct state for the new program code, otherwise a reachable state should be reached by the process in a reasonable finite amount of time. In the ideal scenario we want the process to behave after the update like if it was executing the new program code from start (and like executing the old program before the update).

In order to produce state, which is compliant with the new program that the process is going to execute after the update, we should provide a state mapping function. This function should map the state of the process, right before the code is swapped, into a new state. This mapping function is usually provided by

the programmer because he is the only one that can define a relation between states of the application having knowledge of the two versions of the program.

We now give a list of the main goals of a DSU system:

- **Transparency and Flexibility**
  Transparency and Flexibility of a system for dynamic updates are two sides of the same coin.
  The transparency of a DSU system towards the programmer can be measured with the level of awareness that the programmer has of the underlying dynamic update mechanism. We say that the programmer, in order to perform dynamic updates of a program, has to produce a state mapping function so we can state that, considering the integration of dynamic updates in software development, he will be always conditioned by the underlay on-line program change framework.
  The way the programmer is conditioned by the DSU system impacts on flexibility. In an ideal DSU system the programmer can do any modifications to the code but normally real systems, in order to obtain expected behavior and absence of faulty conditions after the update, support a certain set of modifications. The programmer could be constrained to write dynamic patches like in (Hicks & Nettles, 2005) or comply with certain design rules or methodologies during development.
  Flexibility and transparency collaborate on making the success of a DSU system as stated by the rule stated by Segal in (Segal, 2002): the more a DSU system is transparent and flexible for the developer, the more likely it will be used.
- **Efficiency**
  Considering an ideal scenario, a DSU system should perform dynamic updates with no downtime. That is clearly not feasible in a realistic scenario because in order perform the program change and state mapping some overhead will be introduced. By measuring the delay introduced with a dynamic update solution in several working conditions we can evaluate the qualities of that solution. As the performance of the DSU system affects the availability of the service, which is provided by the application that is updated, it also affects the willingness of the people to use that solution.
- **Robustness**
  The on-line program change framework should strive for correct updates. Incorrect updates should not cause the application to crash or to behave inconsistently for an indefinite amount of time.
  In order to preserve correctness of program execution the DSU system can, for example, provide tools for detecting program state that are suitable for dynamic update or code verifiers that check

if the developer has made changes compliant to the design rules advocated by the on-line program change framework.

In addition to these main goals for a DSU system there are also other characteristics:

- **Concurrency**
  Multithreaded programs should be supported without introducing deadlocks or faulty conditions after the on-line program change is performed.
- **Configurability**
  DSU systems should support different update policies in order to allow greater flexibility on how the updateability is provided and on the set of coding conventions required. For example a solution can provide a way to select a correct state mapping function depending on the application execution state.
- **Roll-back**
  A DSU system could detect faulty conditions after the update and provide some means to command a roll-back to the previous version of the program. This characteristic fall in the robustness main goal.

2.2 Javeleon

Javeleon is a DSU system that aims to improve the limitations of already present works for statically-typed, class-based object-oriented languages. It is a dynamic software update approach that strives for flexibility, transparency and robustness allowing full redefinition of classes in Java by working on the application level. In order to provide dynamic updates Javeleon does not introduce any new construct in the Java Language nor make modifications to the JVM.

As stated in (Gregersen & Jørgensen, 2009), rich client platforms like NetBeans or Eclipse, even allowing reload of new components, lack general support for dynamic update of already running components because Java 6 does not support full dynamic reload of already loaded classes. Java does offer support for redefinition of classes thanks to the implementation proposed by Dmitriev’s class reloading (Dmitriev, 2001). However the modifications allowed to the code in order to not break the binary compatibility of classes are limited. As a matter of fact a class redefinition cannot change the class interface or the inheritance hierarchy, it can change the body of methods.

In order to overcome the version barrier problem (Sato & Chiba, 2005) between different versions of a module Javeleon makes use of in-place generated code at the application level providing a layer of

indirection and enabling a proxy behavior for classes. Whenever a new version of a module is loaded direct forwarding to the new version cannot happen. The In-place proxification mechanism provided by Gregersen makes use of Java Reflection API in order to communicate from the former component to the target.

With Javeleon the updates occur at the granularity of modules allowing entire modules to be updated at runtime. Regarding class modifications, changes that violate binary compatibility are allowed: classes can add fields or methods, methods signatures can be changed as also the hierarchy structure of classes. Changes to the modules that do not break the modules API compatibility have no impact beyond the module itself. Hence, the assumption that Javeleon makes is that in order for the dynamic update to succeed, whenever a module is modified, it doesn't have to break the binary compatibility with client modules. That aspect, seen from a software evolution perspective, seems to follow the common practice for standard development. If we consider a case were an application is made up by several modules, whenever a module is modified breaking binary compatibility with client modules also the depending modules have to be changed conforming them to the modified module's API, ensuring correctness of the application.

By allowing a greater set of changes for redefinition of classes and by hiding the underlying update mechanisms to the developer Javeleon provides programmer transparency and flexibility. Support for concurrent applications is also provided within a deadlock free environment thanks to a lazy update scheme. Regarding configurability Javeleon does not offer the possibility to define update policies explicitly.

Javeleon's actual implementation focus on development and it comes as a plug-in for Netbeans. Engineers can develop applications on top of Netbeans platform and easily test dynamic updateability on them. The programmer can modify and test dynamic updates while the application is running enabling fast verification process for on-line program change validity and facilitating implementation of update compliant code changes. In the future Javeleon can be extended in order to support dynamic updates of application developed outside of the Netbeans platform.

For a detailed description of Javeleon characteristics and an insight on mechanisms and implementation refer to (Gregersen & Jørgensen, 2009).

Chapter 3 - Development methodology

This chapter deals with the software development process.

In the first subchapter we examine the problem of developing applications when we have volatile requirements. We introduce the staged model as a software lifecycle for addressing the constant change of software requirements and start to give an insight of what could be the consequences of introducing dynamic updates in a software project.

In the second subchapter we go deeper on describing the methodologies and tools used during the development.

3.1 Incremental product release

3.1.1 Staged model

We consider a software lifecycle model that takes into concern the fact that it is impossible to anticipate every future change to the software. Unveiling all the requirements of a software product in the initial phase of the project requires a divinatory ability that is not available to the developer nor to the customer; thus software evolution is forced to cope with requirements volatility (Rajlich V. , 2006). In fact, one of the major critics to the waterfall model for software development is that clients may not be aware of exactly what requirements they need before reviewing a working prototype and commenting on it; they may change their requirements constantly. Designers and programmers may have little control over this. If clients change their requirements after the design is finalized, the design must be modified to accommodate the new requirements. This effectively means invalidating a good deal of working hours, which means increased cost, especially if a large amount of the projects resources has already been invested in design choices.

Considering, for example, what Cusumano and Selby discovered in a study about Microsoft software development (Cusumano & Selby, 1997), 30% of new requirements on Microsoft project emerged during the development phase, probably because of an increased developer's learning. That means that design choices based on volatile requirements have to be considered as temporary.

The iterative software development paradigm takes into account these remarks by defining the concept of iteration. We define "Iteration" as a span of time during the development of the software when the

application stakeholders can decide which new requirements the application needs. These requirements are then implemented during that iteration and, when the latter ends, the development process must have produced a working program (iterative release) so that stakeholders can again make an evaluation of the product and consider new requirements or features for the next iterative release, proceeding in a cyclical way.

We decided to follow a software lifecycle paradigm based on iterative releases. According to that pattern the developer has to implement a working program for each iteration, considering only the requirements for the currently developed iterative release. These requirements are the result of observations on the precedent iterative release and can concern the implementation of new features in the application, fixes for bugs or refactoring on the code.

Our software lifecycle paradigm follows the staged model that comprehends 5 phases as stated in (Rajlich & Bennet, 2000):

1. Initial development (“Alpha stage”): Engineers develop the software first functioning version release.
2. Evolution (“Growth stage”): Engineers bring iterative changes to the software while discovering new requirements, adding new features to the software product.
3. Servicing (“Saturation stage”): When the code begins to decay with design aging or the management decides that it’s time to stop adding new features the software evolution enters in this stage where changes to the software are limited to patches for minor software deficiencies.
4. Phase-out (“Decline stage”): No more servicing is provided but users continue to work with it working around its deficiencies.
5. Closedown: The software is shut down and users are directed to a replacement product, if one exists.

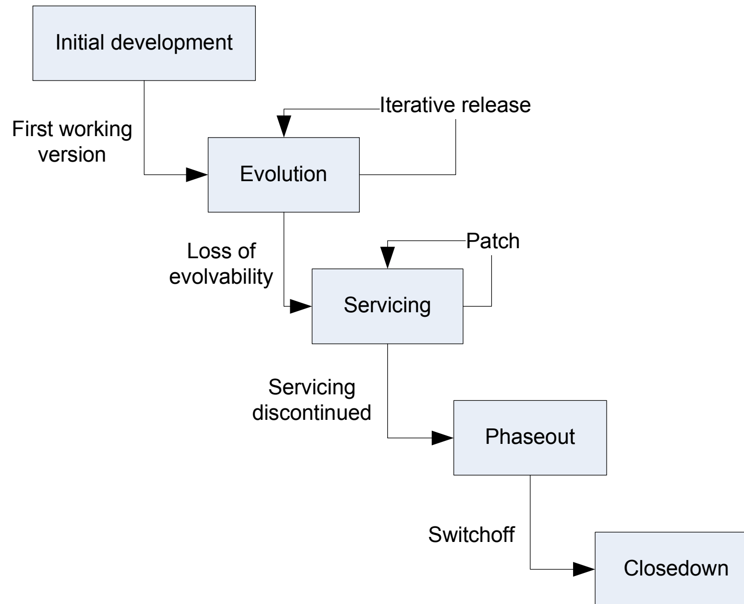


Figure 3.1 – Staged Model for the Software Life Cycle

3.1.2 Staged model with dynamic updates

In our case study we consider the first two phases of this paradigm: Initial development and Evolution.

Working with dynamic updates the *Iteration* phase gets more complicated. During the development of a new iterative release the developer have to test dynamic updates between former releases and the currently developed one identifying the issues that come up from the presence of the dynamic updates and trying to formulate a solution for these issues.

In fact after the development of an iteration release the dynamic update is tested from the previous iterative release to the latter in order to see if the behavior is as expected. In this test phase strange application behavior or runtime errors can be discovered; in both cases the cause is identified and a solution for the issue is formulated. That is traduced in a new requirement for the current iteration and implemented in the currently developed iterative release. After the effectiveness of the solution has been proved the dynamic update is tested again for new requirements. The development can then proceed to the next iterative release when dynamic update tests don't produce any other requirement.

It is clear that enforcing the application to follow a particular behavior can't always be possible by modifying only the code of the latter iterative release. In these cases an update barrier is declared, which is a point in the software evolution where it is impossible to update from a former version to the latter without modifying the design of the former application.

In case of an update barrier there are two options:

1. Declare that the dynamic update is not attainable starting from the previous iteration release to the currently developed one.
2. Do a cascade modification, which is a modification that affects all previous iterations code in order to let the update works from former releases to the latter one preserving latter design choices.

We can clearly state that the second option, while being a reasonable choice in our case for research purposes, is hardly available in a real software project because the former iterative releases can be already on the market. In that case the software engineer is compelled to the first option.

Normally a good design should be worked out before the coding phase begins; however during the development of iterative releases the code will be modified several times, adding new discovered functionalities by modifying methods, working with class hierarchies and implementing new design patterns possibly replacing old ones. As defined in (Fowler, 2000): “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure”. Without refactoring the changes made in order to apply new features on each iteration will let the code naturally degrade to a level of complexity such as it becomes impossible to debug or make further changes easily. This issue can also let the entire project fail or force the development process step into the servicing phase where no new features can be added. Thus refactorings are needed in order to let the software project stay in the *Iteration* phase of the staged development model.

With refactorings a bad application design can be reworked into a well-designed code applying several small modifications that in a cumulative effect radically improve the design making a stand to the code natural decay tendency. These changes can be classified in a catalog, as Fowler partially did in (Fowler, 2000), that tells when and how to apply in a safe manner a certain refactoring. Moreover a refactoring can also be seen as a set of fine grain changes to the code. Following the classification of fine-grain code changes provided by (Gustavsson, 2003) we try to create 1-to-N relationships between refactorings and fine grain code changes when a generalization is possible. These code changes could affect the validity of a dynamic update therefore the developer has to pay attention to which refactorings are made during the iteration. When a certain change to the code introduce issues using the Javeleon framework in a dynamic update test the refactorings related to that code change could be listed as refactorings that are not update-friendly with Javeleon.

For instance consider the refactoring “Move method”. That refactoring state that in a condition where a method is, or will be, using or used by more features of another class than the class on which it is defined

then that method should be moved to the class it uses most. The old method can become a delegate or removed entirely. This refactoring implies these changes:

- Instance method removed from class (optional)
- Instance method added to class

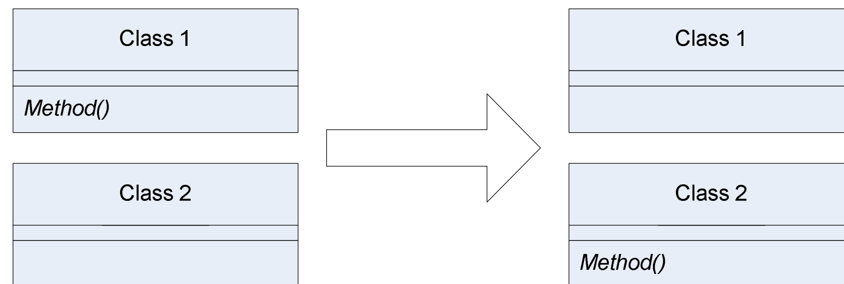


Figure 3.2 - Move method refactoring

In order to follow the Incremental Product release Approach we decided to test the update only after a set of new features and refactorings were applied in the new iterative release, testing dynamic updates only as the last phase of an iteration. The update test then can shows several issues, the application can cast an exception and crash or behave unexpectedly for a limited or undefined time span. All of the issues and their causes are then identified and resolved one by one producing a working and dynamically updateable release at last.

3.2 Set-up and approach

The application was developed using the Netbeans IDE (ver. 6.7 and 6.8) and SVN (Subversion) as the revision control system. In order to achieve module updateability the application was developed as a module built on top of Netbeans platform 6.7.1. The Java language was the compulsory choice for the programming language as Javeleon works with java bytecode transformations.

3.2.1 IDE and Javeleon

Javeleon exploit the Netbeans reload feature of the Netbeans rich client platform (RCP). A rich client platform is a software platform that can be used by programmers to build applications using a modular approach. Instead of writing an application from scratch the developer exploits code reuse benefitting from proven and tested features of the framework provided by the platform. This approach, which brings faster application development and system integration, has some examples in RCPs for Java: Eclipse, NetBeans and Spring Framework. As Gregersen and Jørgensen state in (Gregersen & Jørgensen, Module Reload through Dynamic Update - The Case of NetBeans, 2008) today the increasing number of developers relying on the benefits provided by RCPs justifies the research on dynamic software update of components. That is

the reason why the community of programmers developed class-reloading facilities within Eclipse and NetBeans in order to support dynamic module behavior for the applications built on those platforms, overcoming the lack of support for class-reloading in the JVM at the application level.

The NetBeans reload functionality allows on-the-fly component hot-swap providing support for dynamic module reloads of an arbitrary application written on top of Netbeans RCP. However the approach used by NetBeans on reloading classes is limited by the version barrier problem. This functionality allows types shared between reloaded modules to be type compatible but it does not support state migration, leaving all the work to the developer. The In-Place Proxification technique provided by the Javeleon framework overcomes this deficiency of the reloading feature in Netbeans by making dynamic update of running modules possible removing the burden of handling directly state transfer from the application developer.

The application was therefore built as a single module on top of the Netbeans Platform following some of the guidelines present in (Boudreau, Tulach, & Wielenga, 2007) for developing RCP applications on NetBeans platform.

3.2.2 Application development and Versioning

During the development of the application several modifications to the code were made as a consequence of requirements present in the initial update plan or found during the coding phase. Moreover, as we had to deal with software evolution project, several versions of the applications were developed and the code of each version was organized using the branching functionality of the revision control system.

In order to see how fine grain code changes impacts on dynamic update validity we had to document thoroughly all the code changes and differences between code revisions in the subversion repository. We kept a journal, generated thanks to the information present in the code repository, that shows for each revision which modification were made and how the development proceeded when working on several versions of the same program.

In order to have a clearer high-level view of code repository changes from version to version we designed a simplified schema of the entire application development process.

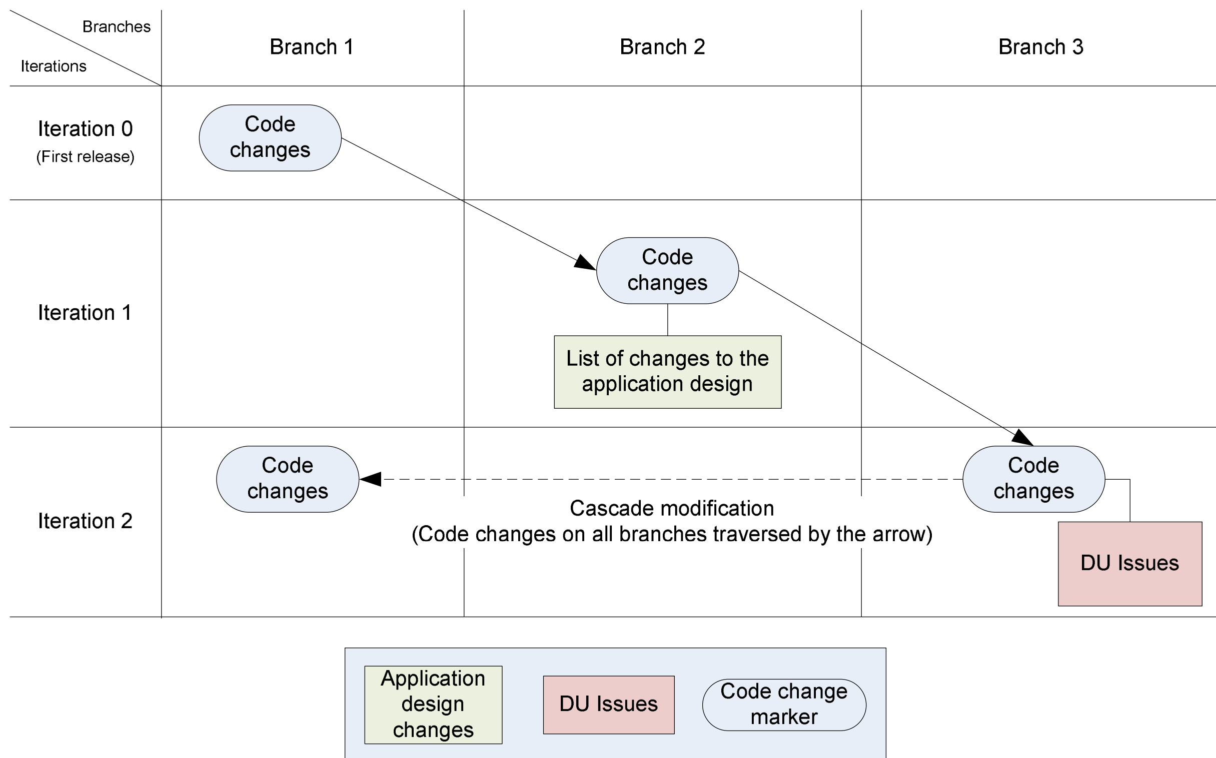


Figure 3.3 – Example of a software evolution schema

In the example schema in Figure 3.3 – Example of a software evolution schema code branches and iterations are distinguished because a branch identifies a place where the code resides, an iteration represents a span of time. Every iteration brings requirements that should be implemented on the iterative release, which is the final revision produced within the iteration. We used a blue marker to state that in a particular iteration we made modifications to the code of a certain branch. Revision numbers or a brief description of the changes can be included inside these blue markers. The development of the application moves through these markers passing from iteration to iteration and creating new branches when they are needed.

During the iteration development refactorings or changes to the code that does change the program design without adding features to the program can be implemented. Those kinds of changes are grouped into green rectangles linked to the branch and iteration in which they are applied.

As the last phase of each iteration we test dynamic update with the previous iterative release. When the dynamic update test reveals an issue that implies an update barrier that subject is named as “Core Issue” and reported on the diagram list in red rectangles. The presence of a core issue in program design can be ignored, and its drawbacks accepted, or imply cascade modification of application code of former iterative releases. Moreover the developer can decide to implement an intermediate version of the application,

which can be seen as an intermediate update. Intermediate updates can allow to overcome core issues with the use of update specific code. As this approach is not considered to be transparent to the developer we decided to avoid it, focusing our attention on design choices that enable the application behavior to be flexible. In the example is shown a case where a cascade modification is applied and affects all the previous branches.

As it should be clear at this point the use of branching in the versioning control system helps the developer to separate code of different versions of the application. As branches can be modified during different iterations, using a single branch to develop all the application releases is considered a bad choice in our case. Consider a scenario where the developer has to do a cascade modification. Following a flat approach when performing cascade modifications, the developer should revert the code to a certain revision number, which indicate the final version code of a previous release, modify that code with the changes needed, commit the changes and iterate the process for all of the previous versions. Using the branching system the developer can easily jump from a version code to another just by switching the branch. Moreover he doesn't need to keep track of the relationship between revision numbers and versions of the application because that information is embedded in the branches.

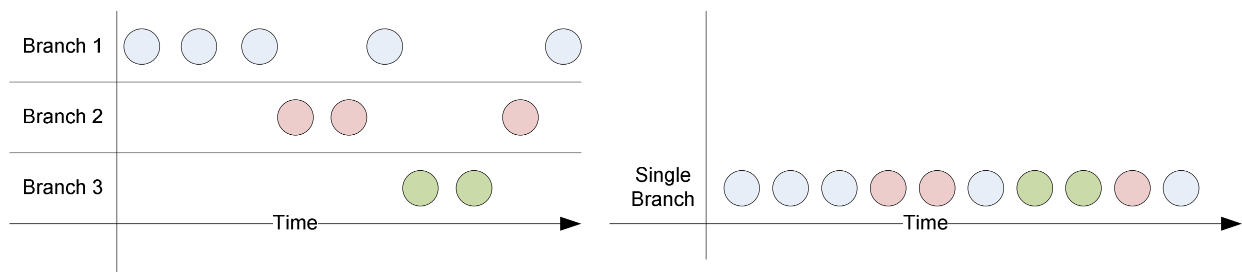


Figure 3.4 – Comparison between SVN branched and "flat" approach

Chapter 4 - The case study application

In this chapter we first describe the application developed listing the functionalities that it has to implement on each iterative release. In the second part we state how we do expect the application to behave standing to our paradigm of valid dynamic update. In the last subchapter we give an insight on the initial design of the program.

4.1 Overview of the Space Invaders clone game

In order to evaluate the flexibility and transparency of Javeleon we decided to create a simple Space Invaders game clone.

In this subchapter we describe the test application seen by the application designer's point of view before he started to develop the application. At this moment of the development we know that the application will comprehend several releases where each one will add a, at least, a new functionality.

At this level our description defines the features that will be added on each iterative release without considering changes to the software code that will be introduced because of dynamic updates or particular developer choices about the application design. We should consider that normally new features are discovered after an iterative release is deployed. In our case we planned first all the release features, trying to balance the programming effort between versions. Moreover, we should also consider that some requirements are dependent on the coding phase. For instance, during the development of an iterative release, the developer can realize that the application design is decaying therefore refactorings are needed.

Screenshots of the developed implementation will be shown for each version in order to give a clearer description of the functionalities.

4.2 Description of each iteration

The first step of the application development consisted of a brainstorming phase aimed at finding features that could be included in a hypothetical Space Invaders game project. We considered different features, chosen some of them and then arranged the result in groups. After this step we created a schedule that described for each iteration the linked group of features to be implemented.

4.2.1 First release

The first release of the game has to be simple and a good starting point for future releases. This release has to include only few functionalities of the original game. The basic idea behind this choice is to implement all the functionalities of the original game and new features in future releases in order to find the issues related to the iterative software evolution during dynamic test updates.

According to the iteration plan the first release of the game should present a welcome screen after loading. When the user presses a button during the welcome screen the game should start and the user should be able to control the spaceship with the keyboard. The only actions permitted to the user during this game phase are about the control of the spaceship (horizontal movement and shoot). The user should also be able to put the game on pause by pressing a key on the keyboard. Aliens are non-moving entities arranged as a matrix on the top of the game screen. The aliens pose no threat to the player's spaceship as they will not make fire or move. Every alien should disappear from game after being hit one time. The objective of the game in this first version is to kill all the aliens by shooting at them. When the user completes a game a proper screen is shown and then the game can restart.

Graphics elements in this first release should be really simple and no raster images should be used to represent game entities like shots, aliens and the spaceship. No sound effects or music has to be present in this release.

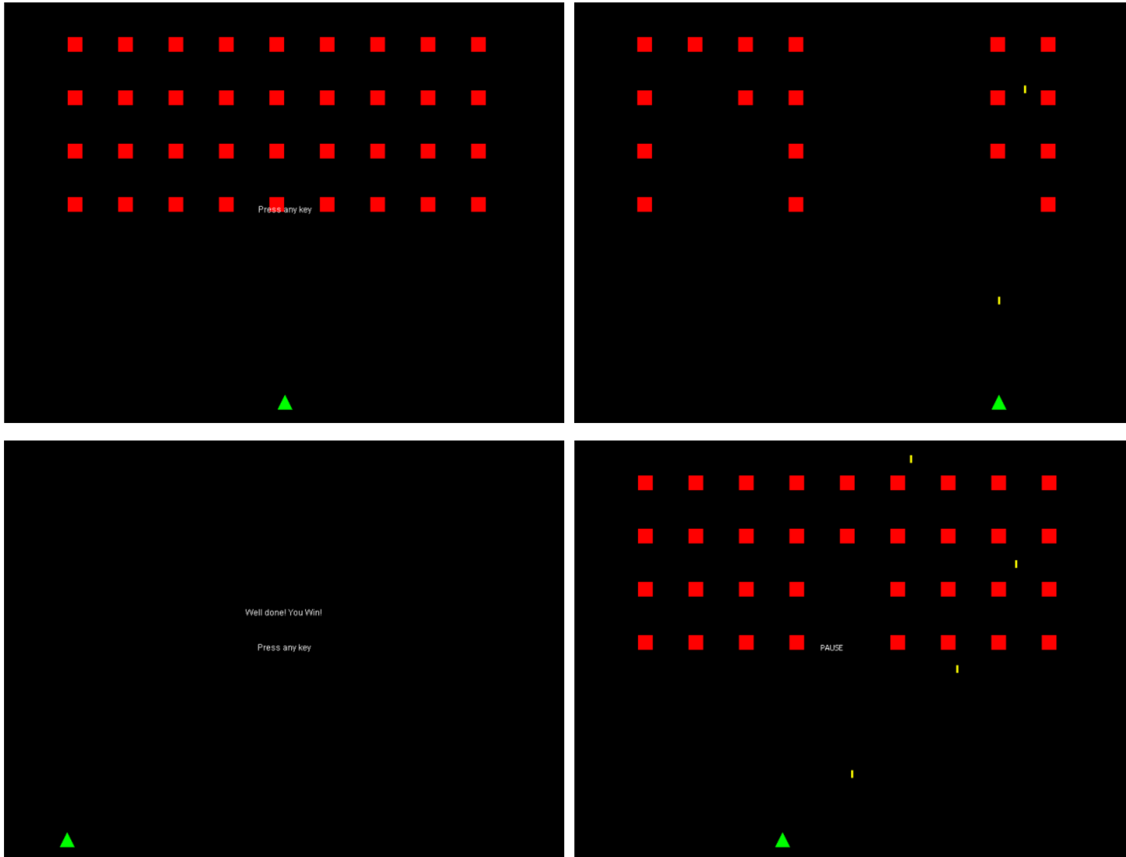


Figure 4.1 – First release's screenshots

4.2.2 Second release

The second release has to introduce few modifications to the gameplay. The new features are described in the list below:

- **The aliens should be able to move.**
  The alien group can move horizontally with a limited speed. The horizontal speed should be determined by the number of aliens that are still alive: the more aliens are present in game the slower they move on the screen. When the group of aliens reach the side borders of the screen the aliens should perform a step down. When the alien group touches the bottom border of the screen or an alien hit the spaceship the user has lost the game and a proper screen is shown to the player.
- **The aliens should be able to make fire.**
  Aliens should be able to make fire at a random but limited pace. Only the aliens in the first line should be able to make fire. If the player spaceship is hit once during the game the user has lost the game and a proper screen should be shown to the player.

- **Invulnerable barriers should appear at the bottom of the screen**.

  These barriers should be invulnerable to aliens' shots and spaceship's shots. Every shot that hits a barrier should disappear from the game.

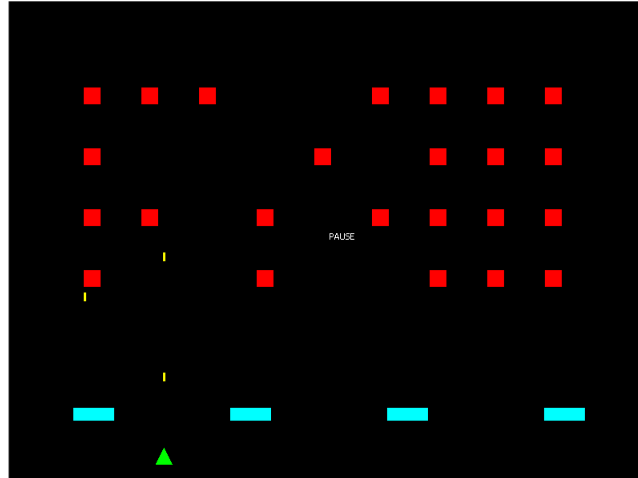


Figure 4.2 – Second release's screenshot

4.2.3 Third release

In the third release the barriers have to lose the invulnerable property and become vulnerable to shots either fired from the spaceship or the aliens. A max number of sustainable hits should be defined for every barrier. When that number of hits is reached for a barrier that barrier should disappear from game. A transparency effect can be used to represent the state of a barrier that was hit.

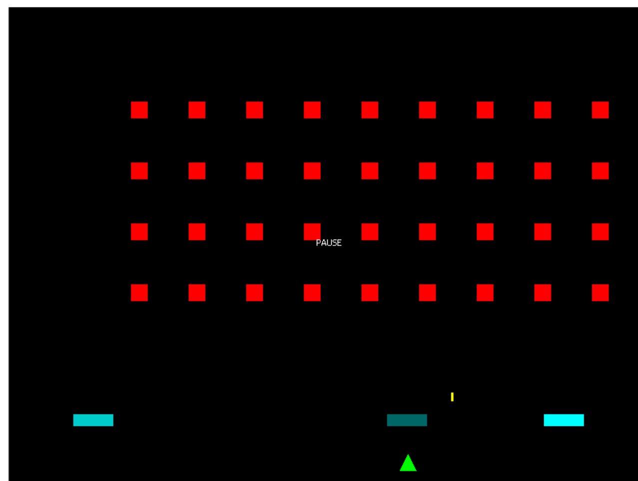


Figure 4.3 – Third release's screenshot

4.2.4 Fourth release

In this release a score system has to be added to the game. The score should be evaluated following these rules:

- During the game the score is proportional to the number of aliens killed, bonus time points are not considered in this phase.
- At the end of the game if the user completed the game successfully the bonus time is evaluated by subtracting the elapsed time from the fixed stage time and the result of the subtraction is multiplied by a bonus time multiplier. If the user didn't complete the level the bonus time is not taken into consideration and the total points are equal to the number of points taken for killing the aliens.

In this release points should be shown only after the game ends. After a game the user should also be prompted for entering the name initials in order to put the score in the high score table. The high score table must be saved on permanent storage and persist through different executions of the application. After the user has entered the name initials the high score table is shown, the application wait for the user to press a key in order to start a new game.

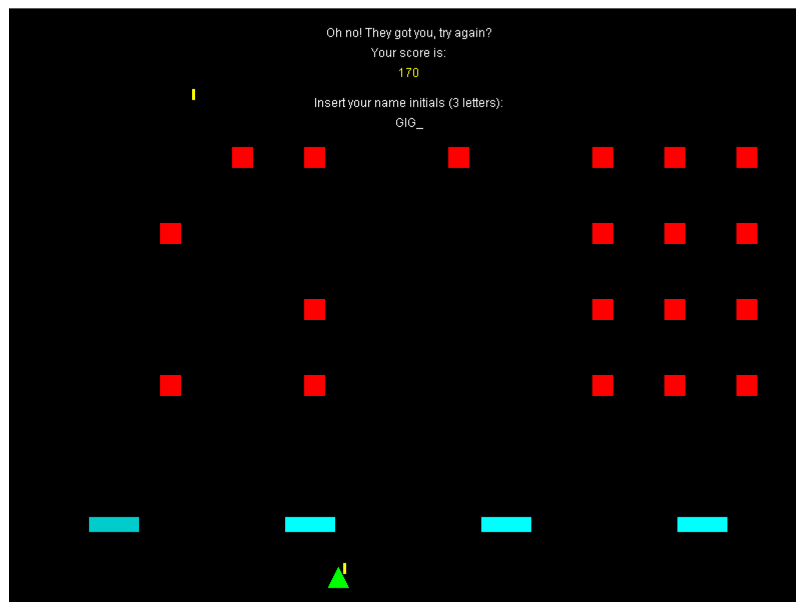


Figure 4.4 – Fourth release's screenshot

4.2.5 Fifth release

A minor requirement should be introduced in this release: the points scored during the game must be shown while the user is playing. The amount of points evaluated while game is in progress must not take into consideration the time bonus, which is to be evaluated at the end of the game.

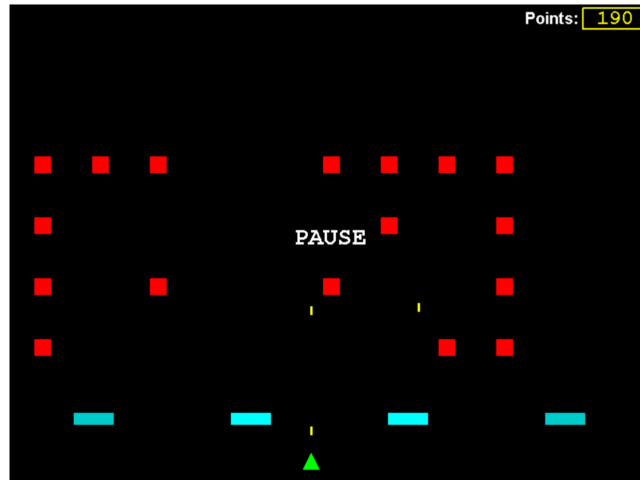


Figure 4.5 – Fifth release's screenshot

4.2.6 Sixth release

In this release the user experience should be improved with the use of raster images instead of simple colored shapes to represent entities of the game like shots, aliens, barriers and the ship. Moreover the spaceship and the aliens should generate an explosion animation when hit.

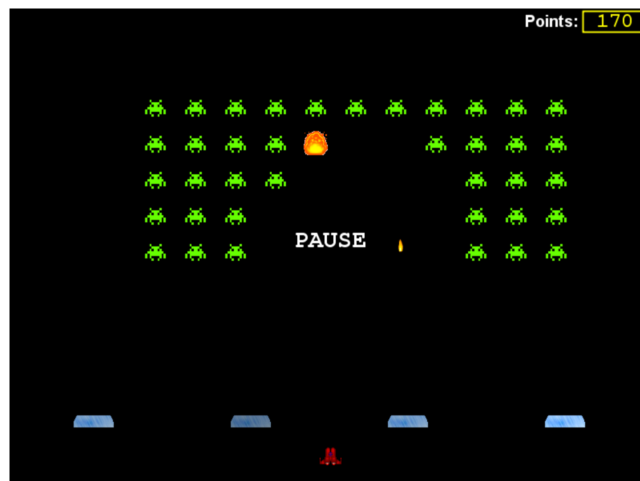


Figure 4.6 – Sixth release's screenshot

4.2.7 Seventh release

In this release new gameplay elements should be introduced. New classes of enemies should appear in game:

- Tough aliens: an alien that has to be hit two times before disappear. These aliens should be arranged on the back lines of the matrix with the standard aliens on the front lines. These aliens should be distinguishable from the others simple aliens by the color. A transparency effect can be used to represent the status (number of time the alien has been hit).
- Mothership: a special unique alien that has to be hit multiple times before die. The mothership should be unique and placed behind the matrix of the aliens. The mothership should also be able to shot lasers instead of simple shots. The laser fired by the mothership should be able to destroy barriers outright. A transparency effect can be used to represent the status of the mothership.

As the new game elements are introduced the enemies should be arranged in a different way. Aliens that are contained in the matrix should free some space in order to let the Mothership make fire.

The player has still to destroy all the aliens on the screen to complete the game.

A background image should be added in this release.

The standard aliens and the tough aliens have to show a simple animation.

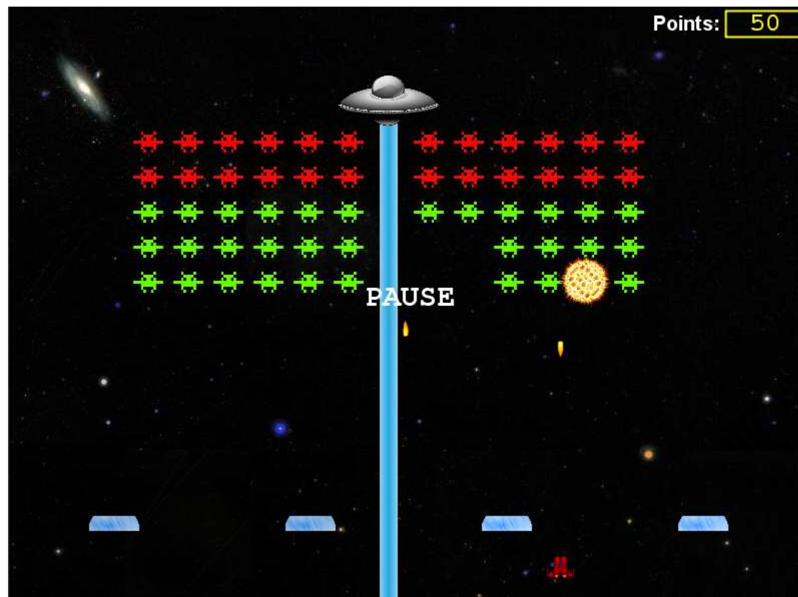


Figure 4.7 – Seventh release's screenshot

4.3 Expected behavior

In this section, we define what is the desirable behavior of the application when dynamically update from an iterative release to another. The statements made in this chapter refers only to the functionalities added on each release and do not take into consideration the implementation of the game (design choices, data structures, algorithms, etc.). We follow the formal framework developed by Gupta et al. in order to make our considerations (Gupta, Jalote, & Barua, 1996).

The basic goal of dynamic updating is to change the behavior of an application to some other acceptable behavior during its execution. Normally a program in execution, that is a process, is composed of a program π and state, which start from an initial condition s_{π^0} end evolve with every statement of the procedural language. State can be seen as the complete characterization of the process at a certain point in process lifetime. State evolves also thanks to the interaction between the process and the environment, which can include the user.

The behavior that we can obtain when dynamically updating a process' program π to π' will not be, in general, the same behavior we can obtain from executing the updated program π' from the beginning. The best that we can expect is that the application behaves like a process executing the new program after the update. If we want the application to behave consistently after the update, regardless of the past evolution of the process state, we should guarantee a reachable state for the updated program π' after the dynamic update. A state is said to be reachable for an application if the process executing the program code can let the state evolve from the initial state to that condition at a certain time and considering particular environment/user interaction.

However, expecting the application to switch its behavior instantly may be unrealistic in some situations because of the state present at the moment of the dynamic update. If the state is not correctly mapped into new state for the new program code the application can show an undesired behavior for a certain period of time. In (Gupta, Jalote, & Barua, 1996) a dynamic update is still considered to be a valid "if after a certain transition period after the change, the process starts behaving as if it had been executing the newer version of the program since the beginning of its initial state".

Therefore, at best we want the process state present after the update to be reachable state for the new updated program. If we strive for state consistency with new program code after any update, we should be able to avoid undesired application behavior or process crash.

Expecting the application to behave like it was started with the new program from the beginning of the process execution seems impossible because it would require a transformation of the past evolution of the application state (also considering environment and user interaction) into state that is valid for the new program at the instant where the on-line program change take place. Therefore we should consider that option only valid on a theoretical basis.

Expecting the application to behave like the process was started with the initial state of the updated program and try to obtain any reachable state for the updated program after the update are two very different things.

The table below summarizes the definitions that can be used when considering the validity of a dynamic update.

<table>
<tr><td rowspan="2">Valid Dynamic Update</td><td>After the dynamic update the state of the process is a reachable state for the updated program π'. The application starts to behave accordingly to the new program right after the online program change.</td><td>1</td></tr>
<tr><td>After the dynamic update the process reaches a reachable state for the update program π' after a reasonable amount of time. The application starts to behave accordingly to the new program after a transition period.</td><td>2</td></tr>
<tr><td>Invalid Dynamic update</td><td>After the dynamic update the process doesn't reach a reachable state for the update program π' or crash.</td><td>3</td></tr>
</table>

Table 4.1 - Definitions of valid and invalid dynamic updates

Following this table we can choose for each version step what the desired behavior is, accordingly to one definition of valid on-line program change. Pretending the process to behave like it was started from the beginning with the new program implies having knowledge of the entire past evolution of process state at the moment of the update. Moreover, it is not possible to decide how the user could have interacted with the process if it was executing new updated program since the beginning. In our work, instead of defining a correct behavior accordingly to one definition we will just define a desired behavior that is at least valid for the third definition.

It is possible that a certain desirable behavior cannot be obtained using a particular design choice. We will see in chapter 5 how an implementation choice brings the program to show undesired behavior.

4.3.1 Updating from release 1 to release 2

In release 2 the aliens gain the ability to move and shot. Invulnerable barriers also appear in game. We now consider the desired behavior when updating the application starting from each one of the screens present in the former version.

If the update occurs while the user is playing:

- The aliens that are still alive should begin to move from their current position toward the standard initial horizontal direction. The speed of the aliens should be evaluated on the number of aliens killed from the start of the game (the number or aliens doesn't change from the previous version).
- The aliens group should start to make fire instantly.
- The barriers should appear instantly.

If the update occurs during the execution of the pause screen the modification to the gameplay has to be noticed by the user instantly. The user should see during the pause screen the barriers appear. It is impossible to let the user notice the other two new features.

If the update occurs during a waiting screen that is the screen between the end of a game and the beginning of a new one we expect the modifications to be noticed instantly. The user should only see the barriers appear. The other features cannot be noticed during the waiting screen.

4.3.2 Updating from release 2 to release 3

In the latter release the barriers lose the invulnerable property.

If the update occurs during any screen the barriers should start with the maximum health.

As the appearance of barriers change only when those are hit it is impossible for the user to notice the difference if a game is not running. So the user will not be aware of the update during the pause or waiting screen.

4.3.3 Updating from release 3 to release 4

In the fourth release the scoring system is introduced. The game evaluates points scored at the end of the game and shows new screens that weren't present in previous releases.

When updating during a game and after the game ended:

- If the player has won the game the bonus points that come as a time bonus should be evaluated correctly and added to the points taken from killing aliens from the start of the game. The score of the actual game must be saved on the high score table after the user is prompted for initials.

- If the player loses the game only the points taken for destroying aliens from the start of the game should be taken in consideration. The prompt for name initials and the high score table screen should be shown for the current game.

If updating during a waiting screen or a pause screen the user can’t notice the new version changes unless he finishes the current game after a pause period or starts a new game after a waiting screen.

4.3.4 Updating from release 4 to release 5

In the fifth release a minor modification is introduced: the user should be able to see the points scored during the game.

If the update occurs during the game the user should be able to see the point box with the correct amount of points appear on the screen starting from the current game.

If updating during the pause screen the user should notice the appearance of the game points on screen.

If the update occurs during a wait screen the user should not notice the points’ box appear on screen as in the latter version the application start to draw the points’ box once a game is started.

4.3.5 Updating from release 5 to release 6

In the sixth release better graphics and explosion animations are added to the game.

If the update occurs during the game:

- The representation of the game elements should change instantly. The user should see the new representation of the aliens, shots, ship and barriers in game change according to the new version.
- After the update if an alien or the player spaceship dies it should generate an animation explosion starting from the current game.

If the update occurs during pause screen, the wait screen or the score screen the user should notice the change of the representation of the game elements during the pause screen.

4.3.6 Updating from release 6 to release 7

In the latter release new enemy classes, a new kind of shot, a background image, a change on the disposition of the aliens and aliens’ animations are introduced.

If updating during a game:

- A few back lines of the alien matrix should become tough aliens effectively changing their class.

- The mothership should appear on top of the matrix and the column of aliens directly beneath it should disappear to left space for the mothership laser. Mothership should be able to make fire starting from the current game following the normal pace of fire.
- The background should change from black to the background image instantly
- The aliens animations should start from the current game

If the update occurs during wait screen, pause screen the user should notice the appearance of the mothership and the changes relative to the aliens' matrix and the background image.

If the update occurs during score screen the user should not see the aliens change their class or disposition or see the mothership appear. The user should only see the background image appear.

4.4 Initial design

In this subchapter we expose briefly the initial design of the game by describing classes, data structures and algorithms used in order to implement the Space Invaders game clone.

In order to describe each iteration changes to the code and the issues encountered during the dynamic update test phase we start by briefly describing the structure of the game starting from the first release. No particular design or pattern was followed during the creation of this first version of the game. The only requirements taken into consideration in this phase were the features of the first of the game.

In the first working version of the game the user should be able to control the spaceship movement using the keyboard in order to shoot to the aliens that are fixed on the screen. The user cannot lose the game as the aliens aren't able to make fire or move. After the player has killed all of the aliens the game can start again after a brief victory screen. Game graphics is realized using simple shapes and no sound effects are present.

The first version comprehends several java class files collected in a few packages. Packages and the most important files contained in them are shown in the following list:

- `org.lufor.Invaders`
    - **`InvadersTopComponent`** class
      Class that extends the **`TopComponent`** class that is the embeddable visual component to be displayed in NetBeans. **`TopComponent`** is the basic unit of display and windows should not be created directly when developing applications on top of the Netbeans platform, but rather extend this class. At program launch an instance of **`InvadersTopComponent`** is

created and, within its constructor, a **Game** class instance (extends **JComponent** class) is attached to it.

  - **Game** class
    The central hook of the game, responsible for the game logic, the control of the user input and the representation (graphics and sounds).

- `org.lufor.Invaders.Entities`
  Include several classes that represents entities of the game like the ship, shots and alien.
- `org.lufor.Invaders.Collections`
  Include some classes that manage collections of entities, like the matrix of aliens and the shots fired by the ship. These classes include methods with logic that involve all the objects in the collection.

As the application in the first version is really simple no Model-View-Controller pattern was used in order to separate domain logic from representation. We now explain how the **Game** class handles those three main domains of a game application.

4.4.1 Game logic

The game logic consists of several classes related to game entities and the gameloop method of the **Game** class.

Every game during the execution should repeat some simple actions in an infinite loop. In the code below is shown a typical game loop (the actions can also be arranged in a different order).

```
While(gameRunning){
      moveEntities();
      evaluateCollisions();
      updateLogic();
      draw();
      handleUserInput();
      sleep(TIME_INTERVAL)
}
```

Code Snippet 4.1 - Game loop

The ***gameLoop*** method of class **Game** implements this general behavior. Every cycle it does these steps:

- Move the entities in game according to their speed
- Evaluate the collision between pair of entities. If a collision is detected and an action is needed, that action is performed by calling some methods.
- Update the logic of the game (like removing from data structures the dead entities)

- Draw the game elements on screen
- Check if the user has pressed a key during this cycle and set data structures accordingly (like setting the speed of the spaceship or add a new **Shot** object to the list of shots in game when the user pressed a key)
- Wait for a certain fixed interval

All of these actions are done every cycle if the user is actually playing the game. If the game is on a waiting screen the loop should only draw elements on the screen waiting for the user to press a key.

When the user presses the pause key the game loop is terminated by changing the value of the Boolean that controls the while cycle and a new thread, responsible of handling the pause screen, is launched. When the user presses again the pause key the pause thread is terminated and a new thread that executes the game loop is launched, effectively restarting the game logic execution.

The game loop also evaluates every cycle the time that has passed from the execution of the last loop in order to move the objects accordingly to their speed.

Game entities and collections

The **Game** class works with entities. Entities are game elements that are drawable and can participate in collisions. Entities also have a position and a dimension on the screen. The **Entity** abstract class represents the general entity of the game; it has numeric fields for the position and the dimension, getter methods for position and size, one method used for collision detection between entities and an abstract method that concrete subclasses have to provide in order to draw the entity on a graphic context given as a parameter.

Entity concrete classes are defined for the objects in game, like **Ship**, **Alien** and **Shot**. All these classes are collected under the same package (`org.lufor.Invaders.Entities`).

Shot and **Ship** entities must also be able to move in the game. For that purpose a **MovingEntity** abstract class (extends **Entity** class) has been created. This class represents an entity that can move on screen and thus have a vertical and horizontal speed, represented by Double fields `dx` and `dy`. **Ship** and **Shot** classes extend this class and implement a ***move*** method.

The relationship between entities is represented in the Figure 4.8 – UML class diagram of game entities (First release) where some details of methods and attributes of **Shot**, **Ship** and **Alien** classes are hidden for simplicity.

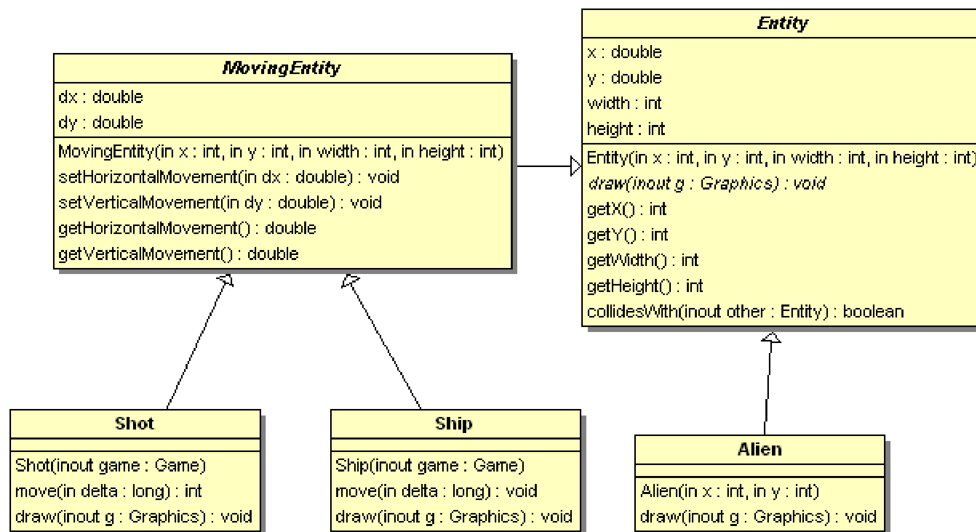


Figure 4.8 – UML class diagram of game entities (First release)

The *draw* method of **`Entity`** class is abstract, the implementation of that method is provided in **`Shot`**, **`Ship`** and **`Alien`** classes.

A collection of entities is a class that gathers entity instances of the same type (like a group of shots) and provides methods that works on all of the entities controlled. Collections maintain alive or dead entities separated using parameterized java.util.ArrayList instances. Some of the methods, like the ***`draw`*** or ***`move`*** methods have the same name of methods of the entities that the collections contain. The logic of these methods is just to call the function with the same name and parameters list on all of the instances controlled.

Collisions in this version are handled inside of collections method ***`collideWith`***. In this version the possible collisions are between shots fired by the ship and aliens therefore only **`Ships`** class implements that method.

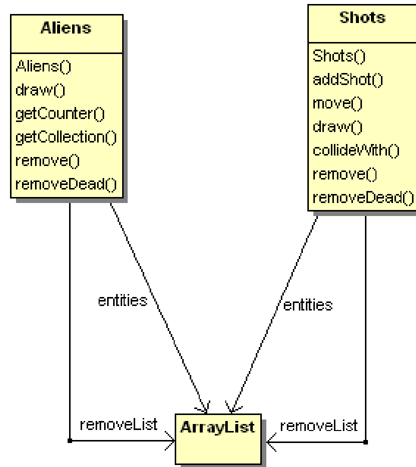


Figure 4.9 –UML class diagram of collections (First release)

With the use of collections and entities the behavior of the game elements is hidden from the game class. The logic of the group of elements is encapsulated in collection classes' methods and entity instances are manipulated only thanks to their classes' methods.

Brief explanation of how the gameloop works

When the application is started the `InvadersTopComponent` object creates a new `Game` instance (whose constructor initializes all the entities needed) and after the `InvadersTopComponent` is shown a new thread that runs the game loop is started. The application at this point is executing repeatedly the game loop.

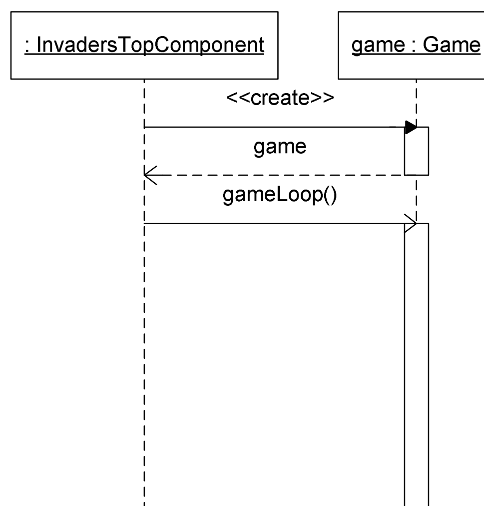


Figure 4.10 - Sequence diagram on the start of the game loop

During the game loop a Boolean variable *waitingForKeypress,* which tells if the application is on a wait screen, is checked and the right set of instructions is executed accordingly. On the first execution this variable is set to true so the game loop only draws a welcome screen waiting for the user to press a key and start the game.

The **Game** instance holds references to single entities (like a **Ship** object) or collections of entities (like **Aliens** and **Shots**) as class attributes and works by calling methods on these instances in the game loop.

```
Public class Game extends JComponent {
	Private Aliens aliens;
	Private Shots shots;
	Private Ship ship;
	[class attributes, methods and inner classes]
}
```

Code Snippet 4.2 - Game class fields

For instance when the game loop move entities it calls the ***move*** method on the **Ship** instance and on **Shots** instance. The ***move*** method of the collections is also responsible for scheduling the removal of entities that were moved out of the game screen, like shots.

After moving the elements on screen it checks for collisions; the ***collideWith*** method of the **Shots** instance is called passing the collection of aliens as a parameter. That method checks for collision between every shot and every alien in the collections. If a collision is detected between an alien and a shot both of them are scheduled for removal by calling the ***remove*** method on **Aliens** and **Shots** instances; additionally the game instance is notified of the death of an alien with the purpose of checking if the game is ended. Class **Shots** has to hold a reference to the Game instance in order to call its methods like ***notifyAlienKilled*** that checks if all the aliens are all gone and the game is ended.

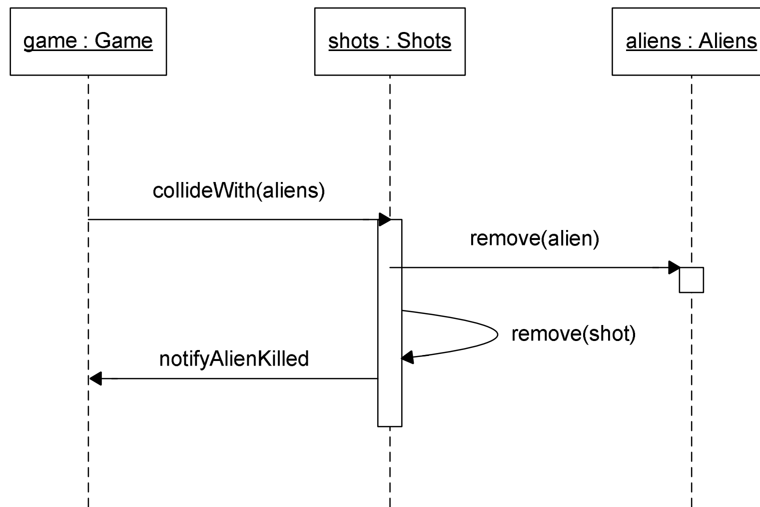


Figure 4.11 - Sequence diagram related to the collision between shots and aliens (v1.0)

After the collision evaluation phase the game logic is updated. In this stage the gameloop remove from memory dead entities inside of collections' arraylists by calling the ***removeDead*** method of each collection instance referenced by the **Game** instance.

After removing dead entities the gameloop method calls the ***draw*** method on **Ship**, **Aliens** and **Shots** instances. The drawing technique implemented is shown in the representation subsection.

The game loop then check the user input watching at some boolean variables that indicate if the user has pressed particular keys. The speed of the spaceship is set accordingly to the user input so that in the next cycle it will be able to move. If the user has pressed the fire key a shot is generated from the **Ship** instance by calling its ***fire*** method, which returns a **Shot** object that is added to the collection of shots referenced by the **Game** class.

After handling the user input the cycle is paused for a fixed amount of milliseconds.

4.4.2 User input control

The user input control is handled by the **Game** class that registers on the application **TopComponent** a **KeyAdapter** subclass implemented as an inner class of **Game** named **KeyInputHandler**. That class provides an implementation for the methods of **KeyAdapter** (KeyPressed, KeyReleased, KeyTyped). For example when the user press the left, right or space key on the keyboard these methods are called and variables of the **Game** instance are set to a certain value as a result to the key pressed.

The game loop will consequently watch at these variables every cycle and call methods on the entities accordingly.

```
public class Game extends JComponent {
    […]
    public Game(JComponent container) {
        […]
        //The keylistener is added to the top component
        container.addKeyListener(new KeyInputHandler());
    }
    //This class methods modifies boolean variables of Game class
    //that tells to the game loop which keys the user pressed
    protected class KeyInputHandler extends KeyAdapter {
        public void keyPressed(KeyEvent e) {
            [code that handle keyPressed event type]
        }
        public void keyReleased(KeyEvent e) {
            [code that handle keyReleased event type]
        }
        public void keyTyped(KeyEvent e) {
            [code that handle keyTyped event type]
        }
    }
}
```

Code Snippet 4.3 - User input handling

4.4.3 Representation

When the application is started an **`InvadersTopComponent`** instance is created an, within its constructor, a **`Game`** instance is produced and attached to the container as a **`Component`**. In this initialization phase the **`KeyAdapter`** is also registered within the **`InvadersTopComponent`** instance. When the **`InvadersTopComponent`** instance is shown, a new thread that executes the game loop method is launched effectively starting to draw the game elements with the game cycle.

Every entity is responsible to draw itself on a graphic context by providing a method ***`draw`***. The logic that handles the drawing of the elements in the gameloop uses the double buffering technique in order to eliminate flickering effects. That practice considers the use of a buffer that is simply an off-screen image in memory. When double buffering is implemented, instead of drawing directly on the screen, the drawing process is first done into a back buffer and then, when all the elements are drawn, the buffer content is copied to the screen.

Following this guideline the gameloop, when it comes to draw elements, first creates a graphic context from the buffer with the same dimension of the screen context, then ask all of the entities in game to draw themselves on that context and, as a last step, it copies the buffer image on the screen.

This algorithm is used every cycle in the game loop, which, in the first version, affects the wait screen and the game screen. Thus every time the application is showing the wait screen the game loop is executed by the application thread and every entity is drawing itself on the buffer image.

This algorithm is not used when it comes to draw the pause screen. Instead of ask every entity to draw itself on screen, as all the entities are known to be still in game, we decided an implementation where less drawing is made. Every cycle of the pause loop the algorithm takes the last drawn buffer image, where all the entities are drawn in their last position, and draw the pause message on top of it, finally flipping the resulting image on the screen. By working that way we save a lot of useless processing.

Though we will see how this approach influences the user’s dynamic update awareness.

The representation used for the game elements are simple color-filled shapes implemented using the java AWT library.

Chapter 5 - The update experiment

In this chapter we look at the iterations development. Each section will describe an iteration explaining:

- The changes made in order to implement new requirements listed in the update plan
- Refactorings and changes to the code driven by decisions made during coding phase
- The dynamic update test with the previous iteration release
    - The issues encountered
    - Unexpected behavior of the application
    - Solutions given

In Figure 5.1 is represented the iteration development schema for the case study application. The first release and all the implemented releases are shown in that schema. The diagram offers a simplified view of how the incremental changes where implemented and how the dynamic updates influenced the developing process enforcing, only in few cases, cascade modifications with the introduction of update barriers.

In order to describe the evolution steps of the application, several time pointers have been chosen. The revision numbers are used as time indicators that describe the state of all the branches present in that moment in the repository.

For example, starting from the first release, we state that the first working program code is the one that refers to revision number 131. The successive picture of the application development is taken at revision 135, where a new iterative release (and branch) is created (v2.0). The update was tested between branches v1.0 and v2.0 at revision 135 and that brought to a cascade modification which involved several changes to the branches v1.0 and v2.0 and ended at revision 166.

Some of the most important changes to the design of the application and the issues that came out from the dynamic update tests are listed. These elements will be described in detail in next subchapters.

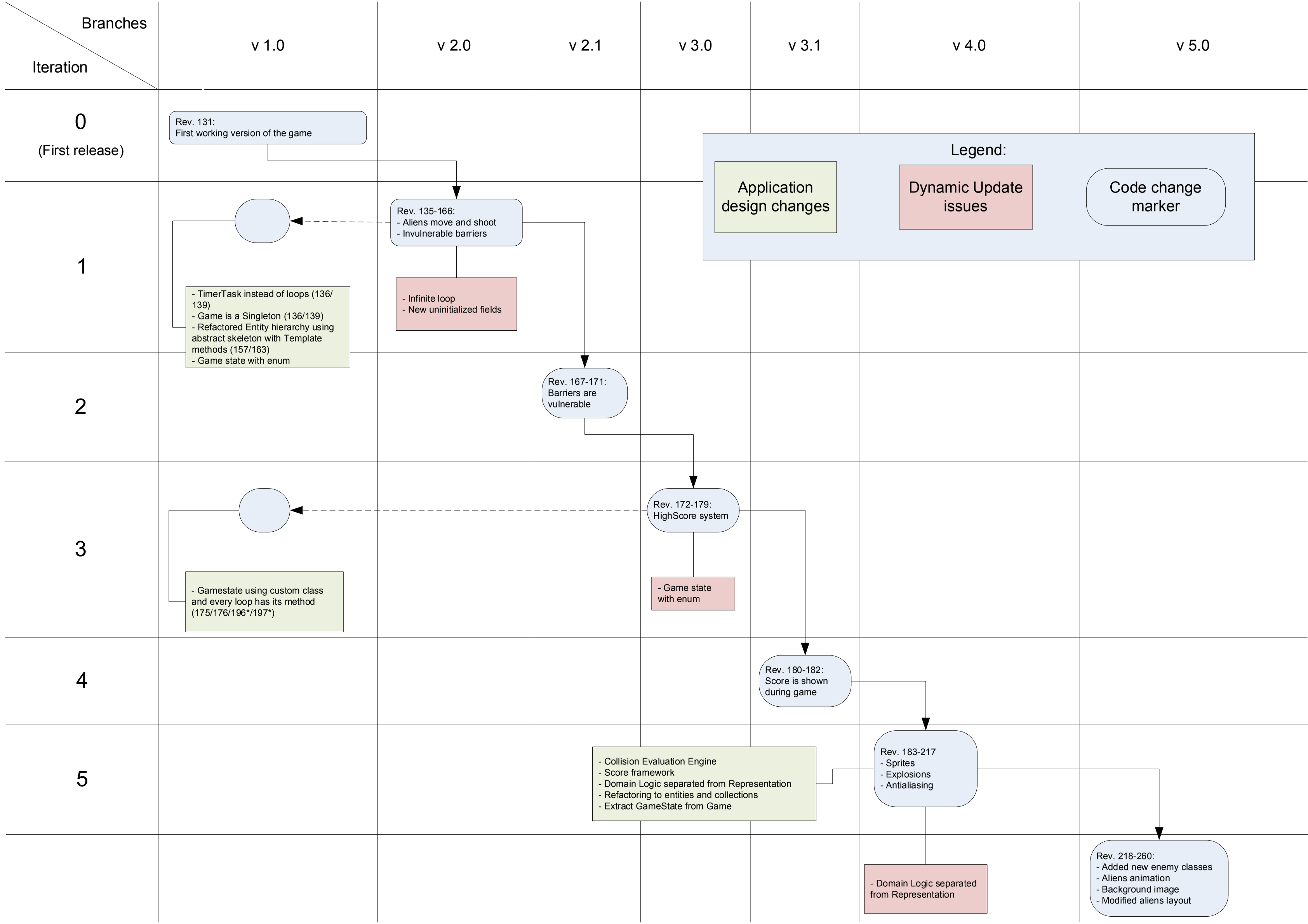


Figure 5.1 – Iterations development schema

5.1 Version 2.0

The requirements of this version are:

- Aliens move on the screen when a new game is started
- Using random intervals an alien from the first line of the enemies group is selected to make fire
- Invulnerable barriers appear on screen

We also implemented some refactorings.

5.1.1 Significant code changes

Refactoring – Extract Superclass

Aliens and Shots classes are similar and share similar features. A super class **`EntityCollection`** with common features of the two classes was created in this version.

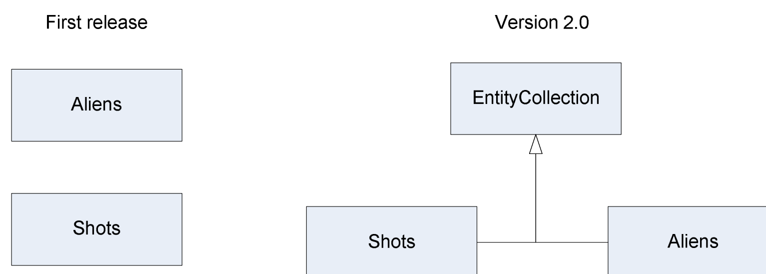


Figure 5.2 - Extract superclass EntityCollection

As it's shown in the UML class diagram of Figure 5.2 the references to array lists that in the former version were present as fields in the **`Aliens`** and **`Shots`** classes are moved in the **`EntityCollection`** class. Common methods were also moved to the super class.

While in the first version collections worked with object of the concrete type in this new version the collections work with objects of type Entity.

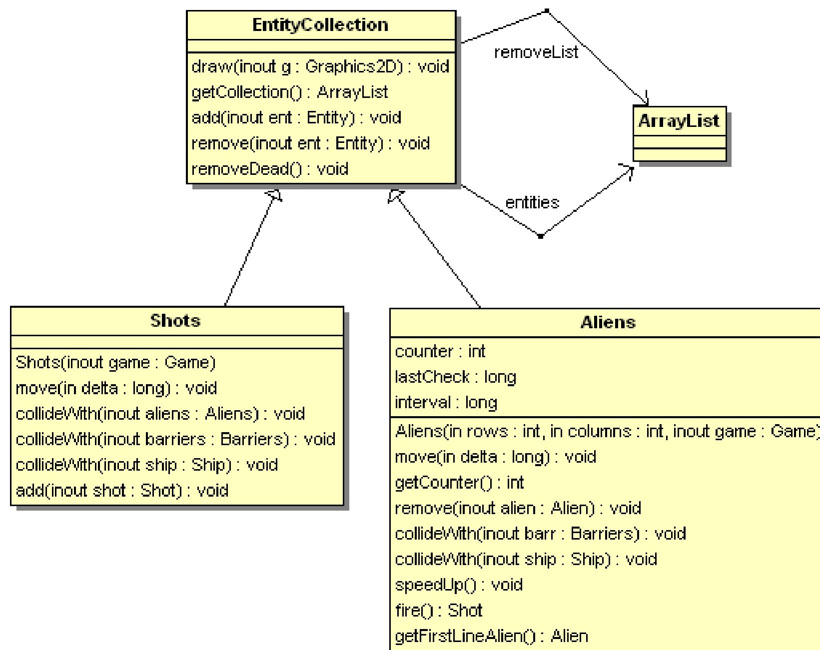


Figure 5.3 - Collections of entities (v. 2.0 rev. 135)

The fine grain changes made to the code for this refactoring are listed below:

Finer grain changes to the code	**Where**
Class added	Added class `EntityCollection` with common methods and implementation of data structures (arraylists)
Super class of class changed	`Aliens` and `Shots` extends `EntityCollection`
Instance method removed from class	Removed common methods from `Aliens` and `Shots` classes
Instance field removed from class	Removed common fields from `Aliens` and `Shots` classes

Table 5.1 - Fine grain modifications concerning extract superclass refactoring (v1.0->v2.0)

Aliens movement

In order to let the aliens move a change to the type hierarchy was needed. As in the former release the **`Alien`** class was extending the **`Entity`** class in the new version it extends the **`MovingEntity`** class.

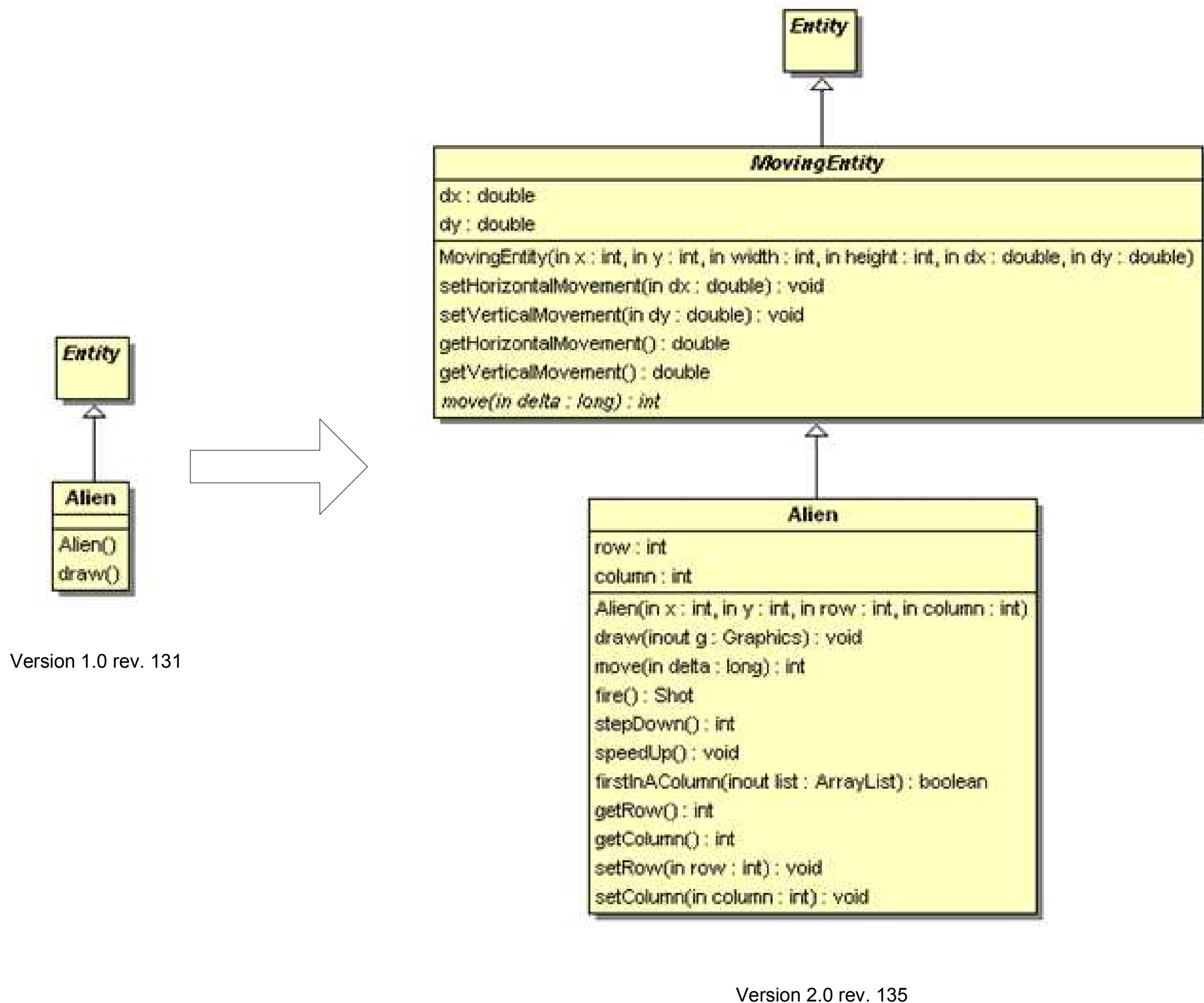


Figure 5.4 – Aliens movement's changes on Alien class

A ***`move`*** abstract method is also added to the **`MovingEntity`** class forcing all non abstract subclasses to provide an implementation for that method.

Several methods were added to Alien class in order to perform a movement:

- ***`move`*** method: move the entity horizontally, performing a step down when the block of aliens touched the side borders of the screen. It also checks if the block of aliens moved to the bottom of the screen triggering the end of the game.

- ***speedUp*** method: increase the horizontal speed of the alien. Is called whenever an alien dies on every instance of Alien gradually increasing the difficulty of the game.
- ***stepDown*** method: move the entity down toward the spaceship performing a step.

The **Game** class handles the movement of the aliens by calling the **move** method on the collection instance in the game loop.

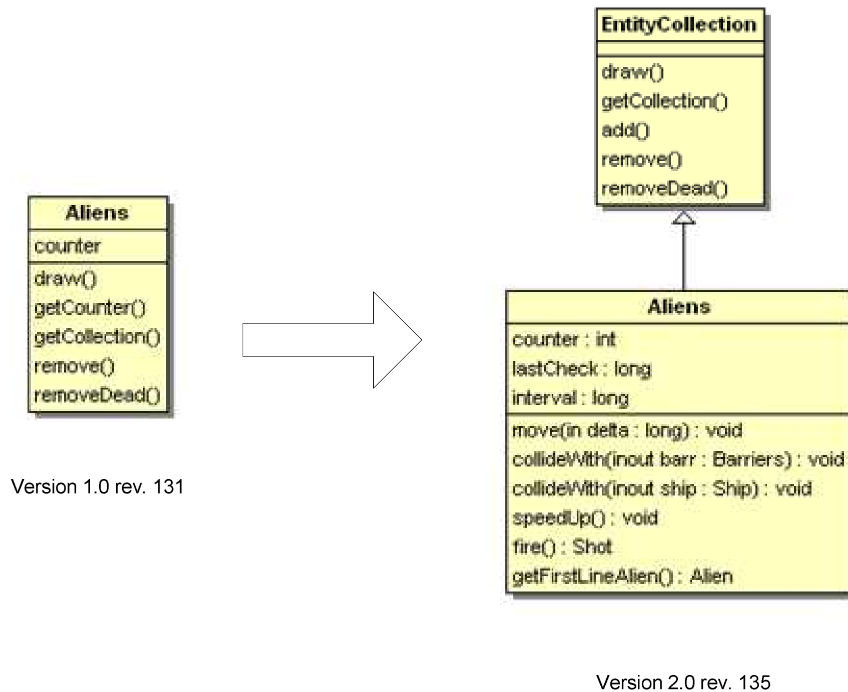


Figure 5.5 - Aliens movement's changes on Aliens class

The changes listed up to now can be seen at a fine grain level as:

Finer grain changes to the code	**Where**
Super class of class changed	**Alien** class extends **MovingEntity** instead of **Entity**
Instance method added to class	Several methods added to class **Alien** and **Aliens.** Also added ***move*** abstract method in **MovingEntity**.
Instance method implementation changed in class	Modified the method class **Game** that handles the movement of the entities so that now it also moves the aliens calling method ***move*** on the **Aliens** instance field.

Table 5.2 - Fine grain modifications concerning aliens' movement (v1.0->v2.0)

Aliens ability to fire

In order to let the aliens make fire a method ***fire*** is added to the **Alien** and to the **Aliens** collection classes. This method produces a **Shot** entity with the correct starting position evaluated watching at the position of the **Alien** instance on which this method was called. The ***fire*** method of the **Aliens** class is a bit more complicated because it has to check first if a correct amount of time has passed since the last shot and then select an alien from the front line of the aliens group in order to return a **Shot** entity fired from that alien. If not enough time has passed it returns a *null* value. In order to implement the logic that keep tracks of time some variables were introduced in **Aliens** class (lastCheck and interval initialized to 0).

The ***getFirstLineAlien*** private method returns an **Alien** instance chosen randomly from the front line. This method calls the function ***isFirstLineAlien*** of every alien object in the collection passing the alien collection as a parameter. In order to see if an alien is on the first line of the group the information of the row and the column of the alien was added to the **Alien** class as two integer fields. These fields are set when the **Alien** constructor is called and where not present in the previous version.

The game loop calls the method ***fire*** on the aliens' collection every cycle and adds the shot produced, only if returned, to a new collection of shots, introduced in this version as a **Shots** field of class game, that collects shots fired by the aliens.

Finer grain changes to the code	**Where**
Instance method added to class	Added methods ***fire***, ***firstInAColumn*** and getter/setter methods for row and column fields to class **Alien**.
Instance method added to class	Added methods ***fire*** and ***getFirstLineAlien*** to class **Aliens**
Instance field added to class	Added a new **Shots** collection field in class **Game** to contain shots fired by the aliens.
Instance field added to class	Added integer fields that hold information about the position of the aliens on the matrix in **Alien** class.
Instance field added to class	Added fields lastCheck and interval in class **Aliens**
Instance method implementation changed in class	Modified the constructor of **Game** so now it initializes the new introduced field for aliens' shots.
Instance method implementation changed in class	Modified the game loop method of class **Game** so that now it let the aliens make fire.

Table 5.3 - Fine grain modifications concerning aliens' ability to make fire (v1.0->v2.0)

Barriers

In order to introduce the barriers a new entity class **`Barrier`** was created. This class extends the **`Entity`** class as the relative game element is not moving.

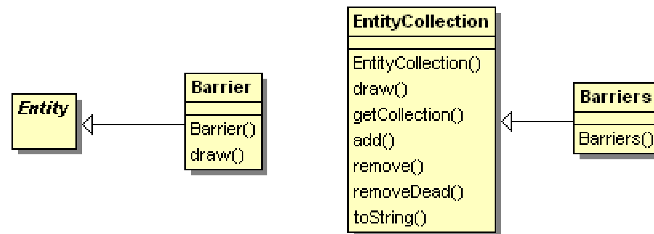


Figure 5.6 – Introduction of barriers

A class representing a collection of barriers was also added in order to handle the operations related to that group of entities. A new field of type **`Barriers`** was added to class **`Game`**. The game loop method was modified in order to draw barriers.

Finer grain changes to the code	**Where**
Class added	Added class **`Barrier`** and **`Barriers`**
Instance field added to class	Added a new **`Barriers`** collection field in class **`Game`**
Instance method implementation changed in class	Changed the method implementation of class **`Game`** that initialize entities adding the instantiation of a new **`Barriers`** object.

Table 5.4 - Fine grain changes concerning the introduction of barriers

Collision detection and handling

As the gameplay changed and new elements were added kind the collision detection phase of the gameloop was also subject to modifications. Starting from this version, the aliens can collide with the spaceship or the barriers and the same thing happen to the shots; that happens because now an instance of the **`Shot`** class can represent both a shot fired by the spaceship or an alien. The logic that detects the collision and takes action after a collision is discovered is placed in collection classes methods named ***`collideWith`***. **`Aliens`** and **`Shots`** classes define a new method for each elements with which they can collide (see Figure 5.3). Moreover this method needs a reference to the **`Game`** instance in order to call methods like ***`notifyDeath`*** that tells the game that it has to end consequently to a collision between the spaceship and an alien. As the **`Aliens`** class instances didn't have a reference to the **`Game`** object in the

former version an attribute that refer to that instance was added. When **Game** class creates a new instance of **Aliens** the reference to the **Game** object is passed in the constructor parameters list.

Finer grain changes to the code	**Where**
Instance field added to class	Added **Game** reference field to **Aliens**
Instance method added to class	Added ***collideWith*** methods to **Aliens** class for collisions with **Ship** and **Barriers**
Instance method added to class	Added ***collideWith*** method to Shots class for collisions with **Ship** and **Barriers**

Table 5.5 - Fine grain changes on Aliens and Shots classes concerning collision detection and handling (v1.0->v2.0)

5.1.2 Update test issues and solutions

The dynamic update test made at rev. 135 between version 1.0 and version 2.0 revealed several issues because of major modifications to the application design. The issues encountered where:

- Infinite loop problem
- Uninitialized fields
- Extract class refactoring
- Super class of class changed and Push Down Field refactoring
- Miscalculation of aliens speed

Infinite loop issue

The first issue found during the dynamic update test is well known and also cited in (Gregersen & Jørgensen, Dynamic update of Java applications - balancing change flexibility vs programming transparency, 2009). This problem concerns the update of a component that has a running thread executing a method of an updateable class. In case the update occurs while a thread is executing a method, we consider that method to be active. The thread that is executing an active method will continue to execute the instructions of the old method till that method becomes inactive. This can result in an inconsistency in case of an infinite loop or a long lived loop because the application will run old code for a long interval. The old code can also call other methods. When calling functions Javeleon looks for a new version of that function and, if one is found, that one is called instead of the old one. Considering these hypotheses the executing thread will continue to execute both old code and new code for a long time. A design solution that deals with this problem tailored on our application is given.

We know that the ***gameLoop*** method of **Game** class contains a long lived while loop. If the application thread is executing this loop (that happens while the user is playing or a “wait screen“ is visualized) the

thread will continue to execute the old code until the thread quit from the while loop and ends the execution of that method. This condition occurs only when the user press the pause key. After the user returns from pause the active thread ends and a new thread executing the game loop method is launched. At this point the code of the ***gameloop*** function is updated and there's no inconsistency as all the threads of the application are running new code.

In order to solve this issue we considered the structure of the while loop. That loop executes some logic and then holds the running thread for a fixed amount of time every cycle. To avoid the infinite loop issue our solution makes use of java **Timer** and **TimerTask** classes.

```
public class Game extends JComponent {
    […]
    private enum GameState {onGame, onPause};
    private GameState state = GameState.onGame;
    private synchronized void gameLoop() {
        [body of the former while cycle]
    }
    public void startGameLoop() {
     loopTimer = new Timer("Game Loop Timer");
     loopTimer.scheduleAtFixedRate(new GameLoopRunner(), 0, LOOP_INTERVAL);
            state = GameState.onGame;
    }
    public void endLoop() {
     loopTimer.cancel();
    }
    private class GameLoopRunner extends TimerTask {
     @Override
     public void run() {
         gameLoop();
   }
}
```

Code Snippet 5.1 - TimerTasks and the infinite loop issue

As it is possible to see from Code Snippet 5.1 the **Game** class uses a **Timer** object to schedule a task for repeated fixed-rate execution. For each long lived loop a class that extends the **TimerTask** class is created as an inner class of **Game**. These classes implement a ***run*** method that calls a synchronized function of the **Game** class, which contains the body of the loop. When the application needs to launch a loop a "loop start" method, like ***startGameLoop,*** is called. That method initializes a new **Timer** and schedule the execution of a **TimerTask** for fixed-rate execution. If an execution is delayed for any reason two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period. Using this solution permits the running main thread of the application to be able to update the game loop body every cycle. When the ***endLoop*** method is called all the scheduled

timer tasks on the timer are cancelled and removed from the queue, allowing the last currently running timer task to end gracefully.

The methods of class **Game** that are called by the timer tasks have to declare the synchronized keyword otherwise it could be possible for two timer tasks to work at the same time on the same **Game** instance.

The **Game** class also keeps track of the currently executing loop using an enumerated variable. This information about the state of the application was represented in the former version by few boolean variables which values determined the screen that the game is actually showing. This information is important when the user press a key on the keyboard because whenever that happens a new thread that executes **KeyAdapter's** method is started. These threads don't know which screen the application is currently showing unless the **Game** class keeps track of it. In this implementation we still chose to merge two semantically separated loops together, the game loop and the loop that shows the wait screen (win or lose screen and the initial title screen). We will see in the next iteration how the separation of the logic inside of the ***gameloop*** into two different timer tasks bodies will condition the application behavior with dynamic update.

The infinite loop issue posed an update barrier. We decided to solve this update barrier by making cascade modifications on the previous release (the first release) and on branch v2.0.

Uninitialized fields (Introduction of state)

Another well known problem is related to the introduction of new fields in an updateable class. When a new instance field is added into a class definition and no initialization is provided within the declaration of the new field we know that:

- If the field type is primitive the value is initialized to the default value (e.g.: for integer variable value 0 is used);
- If the field is a reference to a class instance the value assigned is *null.*

New fields in classes were introduced in these cases:

- A reference to an instance of **Barriers** and **Shots** (for aliens' shots) in **Game** class;
- A reference to the **Game** instance in **Aliens** class, initialized in the constructor;
- Two integer fields regarding the position of aliens in the matrix in **Alien** class (row and column).

Whenever the introduced field is a reference to an object and that reference is used after the update the program generates a *NullPointerException* and, if not handled, the program crashes. This behavior applies

to the first two cases. However the developer can assign a new initialized object to the new reference field using inline initialization and a constructor of the object.

First case

Javeleon allows the programmer to introduce new class instances with the dynamic update by checking in-line initialization of class fields, which is the initialization provided with the declaration of the field, and code changes on constructors of classes. If a new field is found and an in-line initialization with a constructor call or a method that returns an object is provided, that is called and the instance is assigned correctly to the reference avoiding the null pointer exception.

In the **`Barriers`** reference's case the developer knows that, even if the game is being started as standalone with version 2.0 or is being updated from the first version, the reference to the Barriers instance should be initialized using the constructor of **`Barriers`**. Therefore, we solved the problem by using inline initialization of **`Barriers`** field in **`Game`** class.

Second case

The same solution doesn't work with the second case where the reference to the **`Game`** object was assigned to the **`Aliens`** instance using a parameterized constructor. In order to instantiate an **`Aliens`** object the constructor of **`Aliens`** needed a **`Game`** reference, like **`Shots`** class in first version code. In fact in this case the **`Aliens`** instance could be already instantiated when the dynamic update occurs. That means that calling again the constructor of Aliens we would lose all the state regarding **`Aliens`** instances running at the moment of the update. This issue was resolved using the *Singleton* pattern.

A Singleton is a class that has only one instance and provides a global point of access.

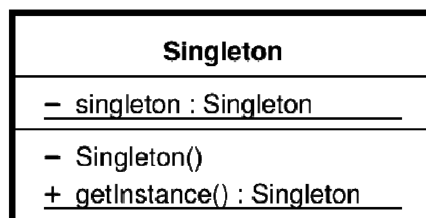


Figure 5.7 - Singleton design pattern

The singleton is a class that provides a static getter method that returns the unique instance of that class. The getter methods checks if the static singleton field already holds a reference to a Singleton object and in

positive case returns that object. Otherwise, the object is created using the private constructor and returned.

Implementing the singleton pattern within the **`Game`** class lets the application design get rid of all the references to the **`Game`** instance. All the objects can get a reference to the singleton instance using the global method ***`getInstance`***.

This modification introduced an update barrier because if the **`Game`** class is transformed into a singleton there is no way to assign to the static reference of the singleton the running **`Game`** instance.

The solution shows that using parameterized initialization of objects is not a good pattern to follow. Otherwise having a global interface for accessing objects seems to resolve the problems related to the introduction of relationships between objects. Whenever a new relation between, let's say, class A and class B is introduced, and an instance of class A needs a reference of an object of class B, it uses a global interface to obtain that reference.

Third case

When primitive fields are introduced in a class and inline initialization is not used, the default value for that particular type is assigned to the field. An object takes in memory a certain amount of space where it maintains its state (the value of its instance variables), which can change during the execution of the program. The program state can be seen as the union of all the objects state and some other information regarding process execution and threads.

The initial state for class instances is provided by the constructor of the class and inline explicit initialization. The developer knows what the initial state of the application should be, in fact he provides constructor for classes and default values for primitive type fields, but he doesn't know how the state will evolve during any execution of the program.

When dealing with new introduced state in dynamically updateable application the developer has to consider if initializing new variables using the initial state will give the desired behavior or not after the update has taken place. We consider that the update can happen at any time during the execution of the program.

When dealing with the reference to the **`Barriers`** object in the **`Game`** instance we used the default initial state to initialize the reference, creating a new **`Barriers`** object that was not present in previous version. That object is created using a constructor that accepts no parameters.

In this third case the in-line initialization cannot be used to set the value for `row` and `column` variables of the already created **`Alien`** instances. That information is set using the parameterized constructor of **`Alien`** during the execution of the **`Aliens`** constructor. Differently from the case of the introduction of uninitialized references to class instances when dealing with primitive types no *NullPointerException* are casted. The application in these cases will present an undesired behavior and eventually cast exceptions (like arithmetic exceptions) that force the application to crash.

After the update the value found in each **`Alien`** instance row and column field is zero. Consequently the ***`isFirstLineAlien`*** method of **`Alien`** class, because of the algorithm implemented, will return always true, causing all the aliens in game to be eligible for shooting.

This problem was solved avoiding the introduction of new state into the application. The algorithm implemented in ***`isFirstLineAlien`***, which is the only method of **`Alien`** class that make use of the information about the cell coordinates of the alien in the matrix, was changed. In the new implementation that method uses the already present information about the position (x and y coordinates) in order to know whose alien are on the front line. .

Extract class refactoring

Moving from the first release code to version 2.0 a class was extracted from collection classes **`Aliens`** and **`Shots`**. This class is also extended in the new version by the **`Barriers`** class.

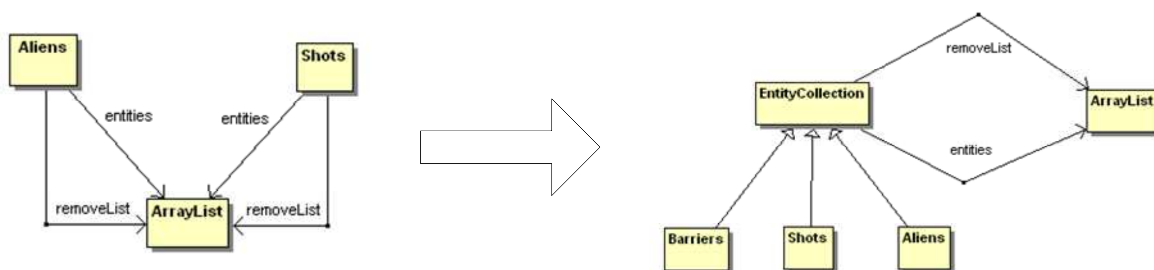


Figure 5.8 – Changes on the structure of the EntityCollection hierarchy tree passing from version 1.0 to version 2.0 rev. 135

In **`EntityCollection`** the array list fields are inline initialized with empty array lists.

This refactoring was tested and the result showed that the lists of entities after the update were reinitialized to empty arraylists. That happens because the array list fields in **`EntityCollection`** class are considered new fields by Javeleon and consequently the constructor for these fields is called. As the name of the fields didn't change from the last version, the application after the update lose all the entities that were present in the old array list instances.

In order to solve this problem we decided to include the change on **`Aliens`** and **`Shots`** classes on first release code by making a cascade modification.

Javeleon allows state to be moved upwards in the hierarchy. That means that it is possible to move a field from a subclass to a superclass preserving the state of that field. In order to obtain the preservation of state however the field has to maintain the same name and type. Though in our case, as we provided in-line initialization in superclass, the instances are initialized.

Superclass of class changed and push down field refactoring

In version 2.0 the class **`Alien`** that inherited **`Entity`** class on the former version becomes a subclass of **`MovingEntity`**. Being a subclass of **`MovingEntity`** implies having two new double fields that holds information about the speed of the entity (variables *dx* and *dy*). Those fields after the update are initialized to the default value (that is 0) because no in-line initialization was provided.

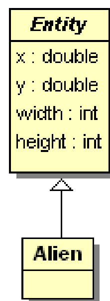


Figure 5.9 - Alien hierarchy tree (v 1.0)

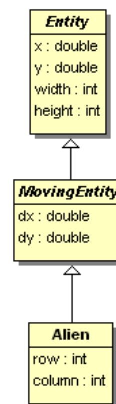


Figure 5.10 - Alien hierarchy tree (v 2.0)

Using in-line initialization in this case makes no sense because the **`MovingEntity`** class is a super class of several other classes (**`Shot`** and **`Ship`**).

In order to solve this and the previous issues a new design was developed following the recommendations of (Johnson & Foote, 1991),(Liskov, 1987) and (Snyder, 1986). These works advocates the importance of data abstraction as a method for developing applications that are easier to modify and maintain. Moreover they investigate the relationship between data abstraction and inheritance and how these two techniques should be used properly in object-oriented programming by exploiting polymorphism.

With the new design all the implementation was moved to the leaves of the tree hierarchy using a skeleton of abstract super classes.

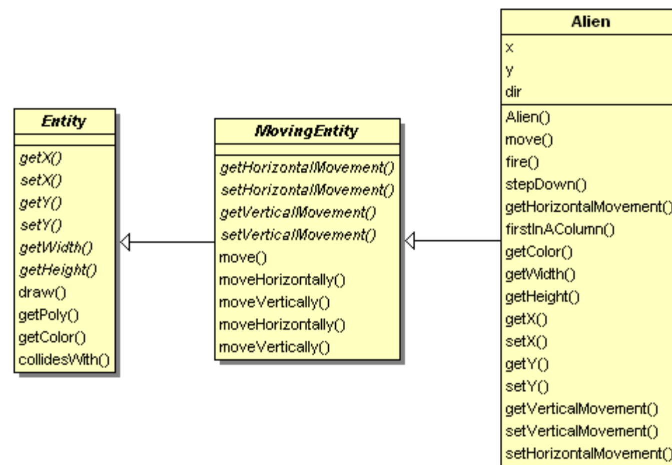


Figure 5.11 - Good design implementation on Alien class (Version 2.0 rev. 166)

As it is possible to see from Figure 5.11 all the fields were moved from super classes to the child class. Abstract super classes define a protocol, a set of messages that can be sent to objects and define a default implementation for some methods like `draw` or `move`. The default implementation of these methods recovers data about the instance using properties (getter and setter methods) exploiting polymorphism and inheritance. Concrete subclasses provide an implementation for getter and setter methods declared as abstract methods in super classes.

This model follows the "*Template Method*" pattern. With that model the behavior of objects is defined by the means of algorithms implemented in abstract super classes, which defer some steps to subclasses. This design pattern lets subclasses redefine certain steps of an algorithm without rewriting the whole method in the subclass. The use of abstract getter/setter methods introduce data abstraction and polymorphism, which allows to obtain variability in the behavior of subclasses.

The "Template Method" pattern was applied for all the classes of the Entity hierarchy tree, the result of which is shown in Figure 5.12.

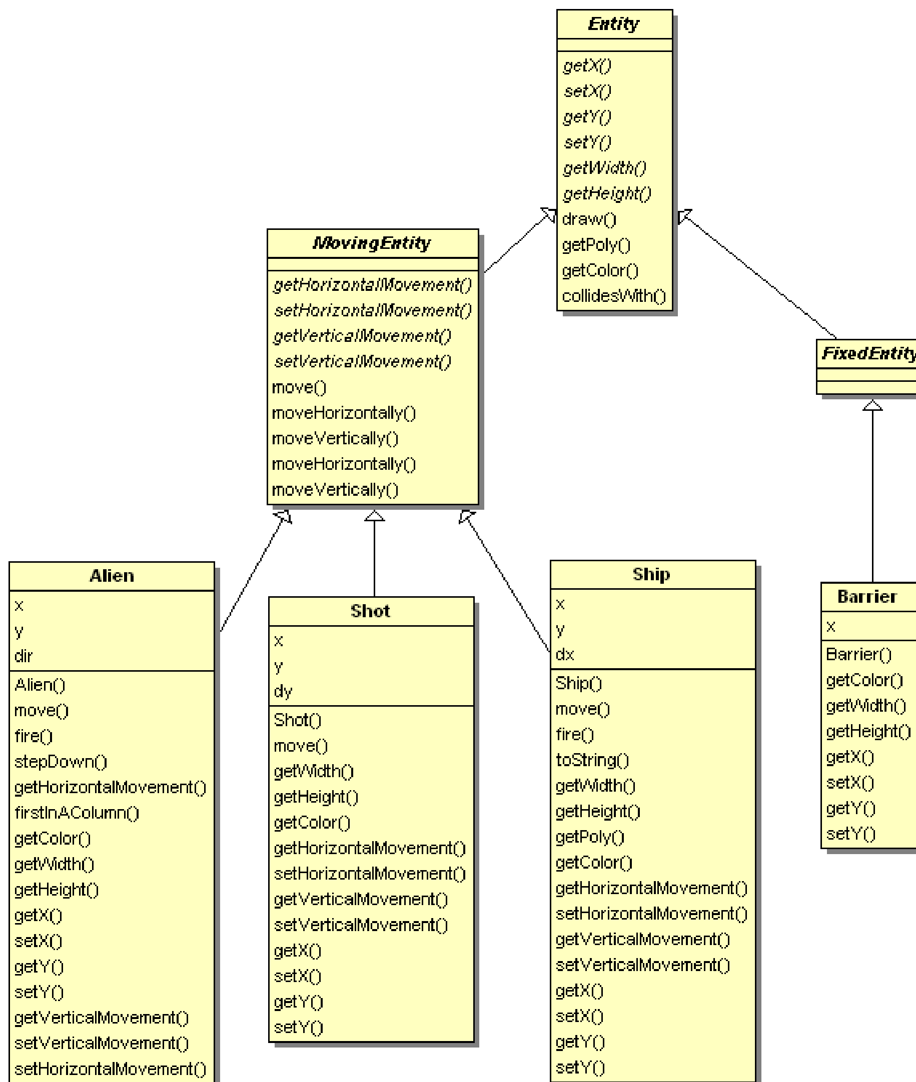


Figure 5.12 – New design implementation on the Entity hierarchy tree

The Entity hierarchy tree was also transformed in a balanced tree with the introduction of the `FixedEntity` abstract class.

Collection classes were also transformed following this pattern. The array lists that in version 2.0 branch rev. 135 were placed in the `EntityCollection` are now in placed in the concrete subclasses (`Aliens`, `Barriers`, `Shots`).

In order to exploit the advantages of the template pattern in collections generics were used. That way the collection knows what kind of objects the arraylists are containing (instances of the `Entity` class) and can call methods on these instances using `Entity` interface without use casting.

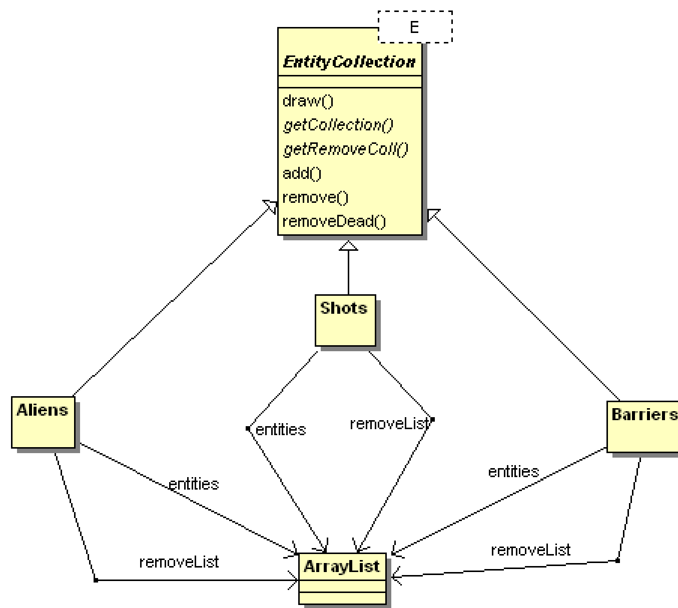


Figure 5.13 – New design implementation on collections hierarchy tree

After implementing the new design in version 2.0 branch we observed the changes made on the entities hierarchy tree between the first release and the version with the new design and we noticed that the implementation of the new design brought to the refactoring "Push down field". An example is given in Figure 5.14.

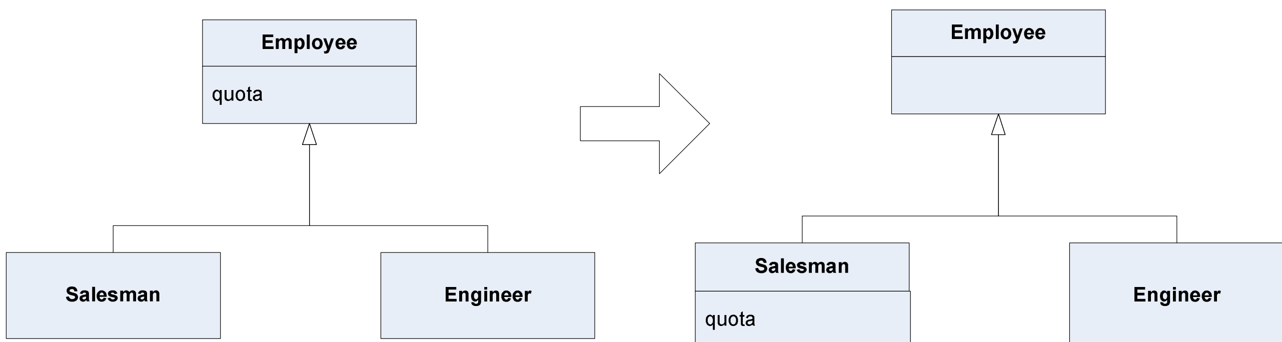


Figure 5.14 - Example of "Push Down Field" refactoring

This refactoring implies these finer grain modifications:

Finer grain changes to the code	**Where**
Instance field removed from class	Removed fields in super classes (`Entity` and `MovingEntity`)
Instance field added to class	Added fields to the leaves of the hierarchy tree (`Alien`, `Ship`, `Shot`, `Barrier`)

Table 5.6 - Fine grain changes concerning "push down field" refactoring (v1.0->v2.0)

We tried to test the effect of this refactoring and noticed that for all the fields that were moved the state was lost. That is because the new fields are treated by Javeleon like new fields thus the old state is not recovered.

The use of the new design introduced an update barrier so we decided to include this design change with backward modifications. The first version branch code was therefore modified accordingly to that pattern as shown in Figure 5.15 where the methods of concrete classes are hidden for simplicity.

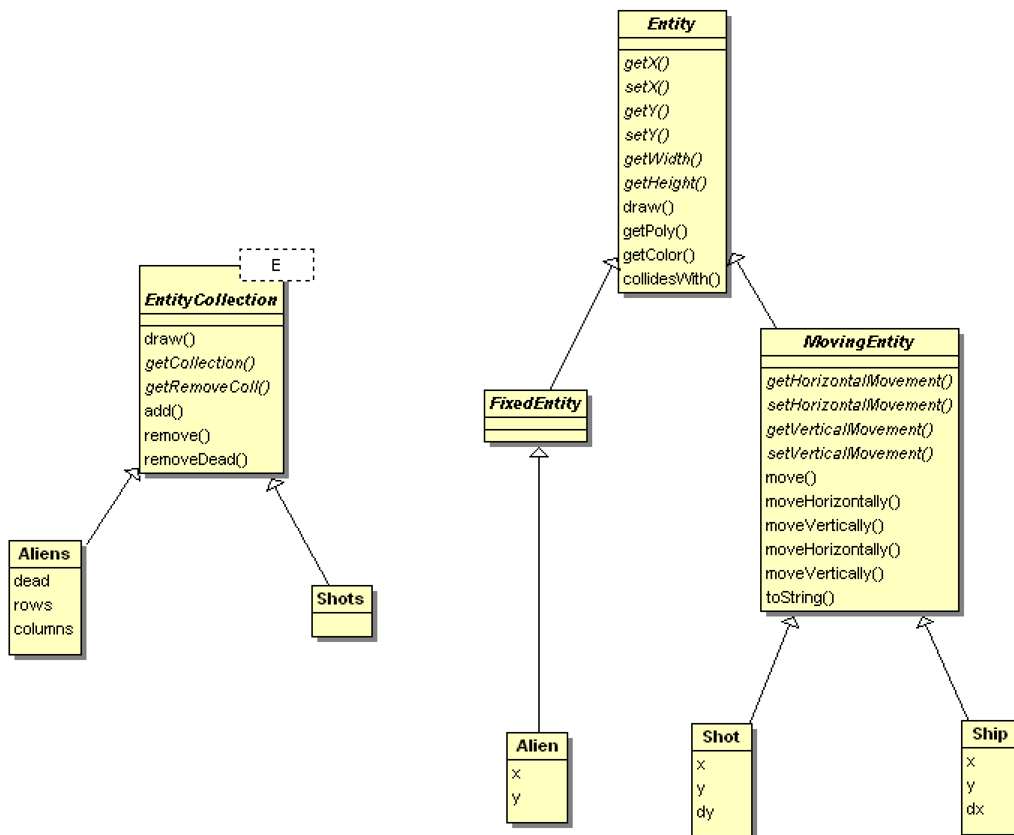


Figure 5.15 - New "Template method" design applied to version 1.0 (rev. 166)

Miscalculation of aliens speed

At this point, after solving all previous issues, the application could successfully update without casting unhandled exceptions. After several tests we decided to change the number of aliens loaded at the start of the game in version 2.0 code.

When the update was triggered during a game we noticed that the number of aliens in game did not change but the speed of the aliens was not reflecting the actual number live enemies. The speed of the alien group is calculated in method ***`getSpeed`*** of **`Aliens`** as reported in the listing below.

```
public double getSpeed() {
    return Alien.ALIEN_SPEED + Alien.ALIEN_SPEED_INC *
           (ALIEN_ROWS*ALIEN_COLUMNS-counter);
}
```

Code Snippet 5.2 - Algorithm that evaluates alien speed

The fields in capital letters are constants declared as `final static` fields. The speed multiplier is multiplied by the number of dead aliens evaluated by subtracting from the amount of starting aliens the number of dead aliens. The counter is incremented each time an alien is removed from the game.

The change in the number of initial aliens affected the constants `ALIEN_ROWS` and `ALIEN_COLUMNS` that are declared in **`Aliens`** class using `static final int` keywords. These fields are updated with values of the new code when the update occurs consequently causing the miscalculation on the aliens speed.

In the table below are shown some cases generated by this issue considering `ALIEN_SPEED` constant equal to 70 and a `SPEED_INCREMENT` equal to 5.

Initial amount of enemies in version 1.0	Initial amount of enemies in version 2.0	Dead aliens	Speed evaluated in version 2.0 after update	Correct speed	Effect
100	20	10	70 + 5 * (20 – 10) = 120	70 + 5 * (100-10) = 520	Speed underestimated
100	20	50	70 + 5 * (20- 50) = -80	70 + 5 * (100 - 50) = 320	Change of direction of movement
20	100	10	70 + 5 * (100 - 10) = 520	70 + 5 * (20 – 10) = 120	Speed overestimated

Table 5.7 - Evaluation of aliens speed

Moreover if the update occurs while the game thread is executing the ***`move`*** method of **`Aliens`** class (and did not finished yet to move all the aliens within the game cycle) some enemies will move with a different speed in that round scrambling the matrix structure of the aliens.

However this issue will last only for the game that is played during the execution of the update so the developer can also consider accepting this kind of issue leaving the unwanted behavior for the transitory period that will last until a new game is started.

This issue is caused by the fact that the application doesn't keep track of the initial state of class **`Aliens`** and this information can't be recovered during the evolution of the game. The assumption that the constants `ALIEN_ROWS` and `ALIEN_COLUMNS` will always reflect the number of initial aliens is wrong when dealing with dynamic updates because these fields are updated while the state, represented by instances of aliens in memory, remains the same. Static final fields are directly in-lined by the java compiler into the class files. Basically the java compiler substitutes the field name with the value whenever it finds an instance in the code. Constants used in this way are not really constants because they can change passing from a version to another so the programmer should avoid their use and rely on encapsulation of fields instead. With the latter approach static final constants can be easily substituted with getter methods (class properties) that return values. This approach does not solve the problem of inconsistencies between state and "fixed" values, which the programmer should address directly, but solves the problem that comes from substituting directly the constants into the bytecode. In fact, if we consider the use of static final constants, only class files that are recompiled will be aware of new values of "constants" with an update. As we said, the problem of inconsistencies between state and "constants" still remains even using encapsulation. The state present in memory and the constants returned by class properties could break invariants when the new program comes into execution. We would encounter this kind of problem even if we use a serialize-update-deserialize schema. Javeleon doesn't offer the programmer explicit ways to modify the state in memory. The only state mapping function that Javeleon offers is the identity function, so the programmer has to find some workaround using data abstraction or abiding transition periods or exceptions.

As the first version of the game was modified because of the issues already explained, this problem was resolved by keeping track from the first version of the starting number of aliens using variables in **`Aliens`** class.

Drawing during pause screen

As said in the subchapter 3.2.3 the pause screen is drawn starting from the last buffer image where all the entities of the game are already drawn in their last position. When updating during a pause period we noticed that the barriers do not appear immediately but they appear only after the player quit from pause and returning to the game.

The reason of this unexpected behavior can be explained by watching at how the pause screen is drawn. The pause loop uses the last drawn image of the gameloop to draw the pause screen. Basically it flips the last drawn buffer image on the screen context and draw on top of the latter the pause message. What happens after the update is that the buffer image remains unchanged. Therefore, even if the barriers are initialized correctly, there is no representation of them on the screen because they didn't draw themselves on the image buffer once. That problem is due to the fact that some state (the image buffer) is not updated in order to reflect changes after the update process has taken place. However, the state represented by the buffer image depends strongly on the entities implementation (the ***draw*** method), which in this case is updated because it's code inside of a function. If we want the dynamic update to produce immediately the change on the behavior of the application, avoiding any transition period, we have to let the elements draw themselves every pause loop cycle, abandoning any performance optimization.

This behavior is not considered to be an issue, however a solution to this problem will be given in version 4.0.

5.2 Version 2.1

In this version we changed the barrier game element in order to make it vulnerable to shots.

5.2.1 Significant code changes

Barriers become vulnerable

The changes in the new branch (version 2.1) created for this iteration affected **Barriers** and **Barrier** classes.

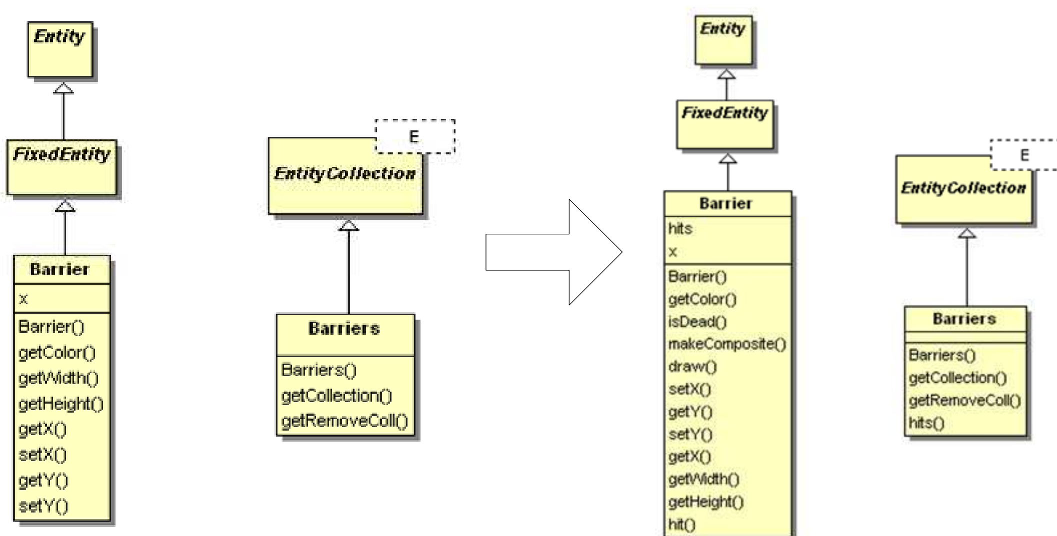


Figure 5.16 – Changes on Barrier and Barriers regarding the vulnerability of barriers (v2.0->v2.1)

Particular attention was given to the algorithm that manages the status of the barriers. Two solutions that differ on the value initially assigned to `hits` field were possible:

1. Initialize `hits` field to the value of max number of hits the barrier can sustain. Whenever the barrier is hit, decrement that value.
2. Initialize the `hits` field to 0 and increment it every time the barrier is hit, keeps track of the max number of hits in a constant (`static final`) field (that is updateable) or using a property on the barrier (getter method that returns a constant value, which is updateable because depends on code). That was the approach used.

The logic that is executed after a collision is detected between shots and barriers was changed in order to modify the status of the barrier hit. This modification occurred in the **`Shots`** class because the collision handling logic is placed in that class. This new requirement unveiled a design flaw. A change in the behavior of a game element resulted in a modification to other game element classes. That design flaw will be resolved in the iteration of version 4.0 with the development of a collision system package.

Whenever a collision between a shot and a barrier is detected the method hits is called on the **`Barriers`** reference passing the collided barrier object as a parameter. This function calls the ***`hit`*** method on the **`Barrier`** instance and checks if it should be removed from the game.

In the listing below is shown the logic of the method ***`isDead`*** and ***`hit`*** according to the choice made about the initialization of the new state (`hits` field).

```
public void hit() {
      hits++;
}

public boolean isDead(){
      if (hits >= BARRIER_HEALTH) return true;
      else return false;
}
```

Code Snippet 5.3 - New methods of class Barrier (v.2.1)

In order to draw the barrier transparency effect the draw method was modified too.

The modifications made in order to implement the new behavior are:

Finer grain changes to the code	**Where**
Instance field added to class	Added field hits to **`Barrier`** (used in line

	initialization to 0).
Instance methods added to class	Added several methods to **`Barrier`** class (like ***`hit`***, ***`isDead`***) and **`Barriers`** (***`hits`*** method).
Instance method implementation changed in class	Changed implementation of methods in **`Barrier`** and **`Shots`** classes.

Table 5.8 - Fine grain changes related to the vulnerability of barriers (v2.0->v2.1)

Moving method implementation through the hierarchy

In version 2.1 a refactoring on **`MovingEntity`** class and concrete subclasses removed the implementation of the ***`move`*** method from **`MovingEntity`** making that an abstract method. Moreover, many methods that complicated that class were removed, delegating all the implementation of the move logic to concrete subclasses. Below are shown some UML class diagrams that show partial information about the classes of interest.

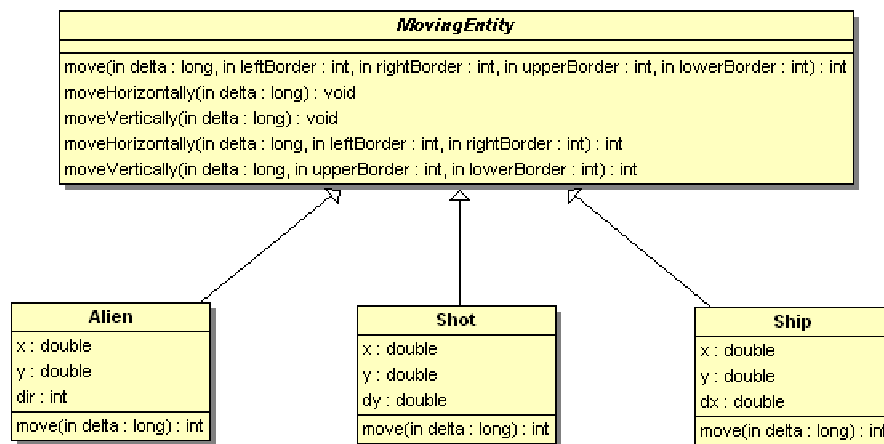


Figure 5.17 - Move method in version 2.0 rev. 167

Some move methods defined in **`MovingEntity`** were removed because they introduced unwanted complexity in the logic of the move operation. We decided to totally remove the logic of the move operation from **`MovingEntity`** removing all the methods declared and adding an abstract method ***`move`*** that subclasses have to implement.

`Move` methods defined in concrete classes only makes use of getter/setter methods to move the entity, no longer using the methods that were declared on **`MovingEntity`** like ***`moveHorizontally`*** or ***`moveVertically`***.

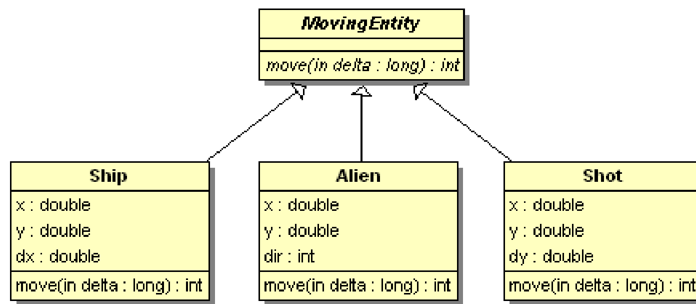


Figure 5.18 - Move methods in version 2.1 rev. 171

Finer grain changes to the code	**Where**
Instance method removed from class	Removed all methods that handle movement of the entity from **`MovingEntity`** class
Abstract method added to class	Added method ***`move`*** to **`MovingEntity`**
Instance method implementation changed in class	Changed the implementation of move methods in **`Alien`**, **`Ship`** and **`Shot`** classes so that they do not use anymore methods of **`MovingEntity`** that were dealing with movement of entities.

Code Snippet 5.4 - Fine grain changes regarding movement of entities (v2.0->v.2.1)

5.2.2 Update test issues and solutions

Testing the update from version 2.0 to version 2.1 succeeded without errors or unexpected behavior.

Barriers become vulnerable

When the update executes and the new version is loaded for each **`Barrier`** object a new uninitialized one is created. When a field of **`Barrier`** instance is accessed the state of that field is transferred to the new instance. If the field is new the value that comes with in-line initialization is assigned to that field in the newly created object.

Instead of using an health field in **`Barrier`** class and working on hit by decrements, we decided to use a field `hits` initialized to 0 in order to be able to update the max health of the barrier during the game with dynamic updates. If the constant that keeps track of that information changes in a future version the hits value, which is state, doesn't change because it represents the number of time a barrier has been hit.

Consider the example shown below.

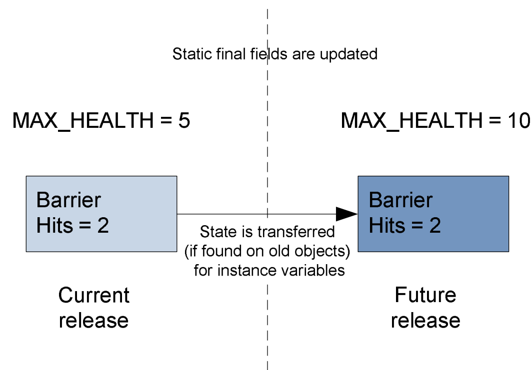


Figure 5.19 - Incrementing the max health of barriers

The constants gets updated because in Java according to the Java Language Specification, any `static final` field initialized with an expression that can be evaluated at compile time must be compiled to byte code that "inlines" the field value. Constants values (declared as `static final` variables) are directly hardwired into bytecode and therefore get updated after the on-line program change.

The same effects can be obtained using getter methods that return values. Even in this case the new value is used because after the update every call to the getter method will be forwarded to the new implementation of that method.

By introducing new state in the program the `hits` field was initialized to 0. We decided to take this approach in order to let the application modify the max number of hits of barriers dynamically with updates.

We could have initialized `hits` field to the max number of hits, and worked on it by decrements, avoiding the use of the constant that keeps track of the max hits. However, with the latter approach, we would have coupled the information about max hits (that we assume dependent on the version) and the number of hits that the barrier has received into a variable, which is state that does not change within the updates. Therefore, as the example below shows, it would be difficult for the programmer to manage dynamic behavior of the application by using the latter solution.

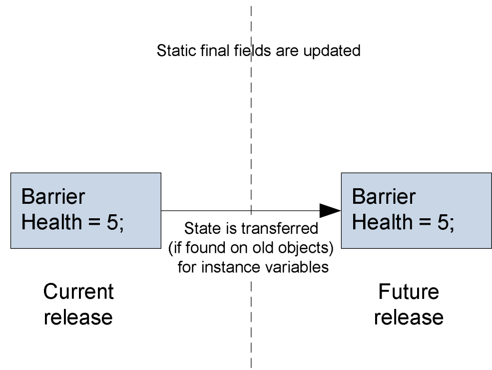


Figure 5.20 - State mapping of barrier's health following bad design

However, even with our solution, there are still problems caused by the relationship between state and environment that in this case is code. In fact, following our solution, there is still a problem that affects evolvability of the state with changes to the constant MAX_HITS. Considering our solution in Figure 5.19, in case were MAX_HEALTH can only grow no issues arise, because we maintain true the invariant that hits is always lower than MAX_HITS. Otherwise, if in a future release we decide to lower the max hits to a value that is lower than the actual value of hits at the moment of the update we break the invariant. This action can lead to inconsistent state after the on-line program change is performed. The inconsistent state can bring two consequences in our case:

- Arithmetic exceptions can be thrown because of the algorithm that evaluates the level of transparency basing its calculation on hits and `MAX_HITS`.
- The application checks if a barrier is dead only when it gets hit, therefore barriers that should be dead after the update because of the decrement on `MAX_HEALTH` value still remain in game and participate in collisions with other entities.

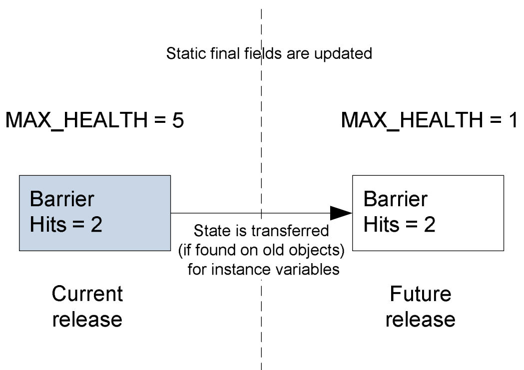


Figure 5.21 - Decrementing the max health of barriers

The first issue is critical because it makes the dynamic update invalid, the second is acceptable because it brings to a transition period that is resolved in consistent and reachable state for the updated program in a finite amount of time.

This example showed that dealing with state transformation is not an easy task for the developer. In fact, as Javeleon offers only the identity function as state mapping for existent state, the programmer has to develop a workaround for issues that are affected by relationship between state and the environment. While transition periods can be accepted when dealing with dynamic updates, exceptions and application crashes have to be avoided. In order to avoid those problems the programmer should mask the data using encapsulation of variables and enforcing the use of properties instead of direct access to variables (state). Properties mix updateable code and state so they can be used to enforce policies on the values returned in order to avoid exception casting.

Moving method implementation through the hierarchy

In order to explain why the update succeed considering the code change that affected **`MovingEntity`** and its subclasses we should think about which method the game thread is executing while the update is performed. This method is considered to be *active* and therefore can't be updated on the run; the new implementation of that method will be used next time a method call is found.

About the removal of methods from **`MovingEntity`** that were used by move method of **`Alien`**, **`Ship`**, **`Shot`** we have considered two cases:

- ***`move`*** method of **`Alien`**, **`Ship`** or **`Shot`** is active when the update is performed. A call to a removed method of **`MovingEntity`** is found. For that method call the old implementation of that method is used.
- ***`move`*** method is not active. When a ***`move`*** method call is found the new implementation is used.

In both cases the move action is performed correctly.

5.3 Version 3.0

In application version 3.0 the score system is introduced. The game calculates point at the end of a game evaluating the number of aliens killed and, only if the player ends the game successfully, adding the time bonus. A fixed time for completing the single stage is assigned. The score is saved on permanent storage with the initials of the player.

5.3.1 Significant code changes

In order to implement the new feature a few classes were created and new state was introduced within the `Game` class.

The `HighScoreManager` is a singleton responsible of managing the high score table using serialization to load and save high score table. It holds a reference to an array list of objects of type `Score`. `Score` instances holds information about the name of the player (initials) and the score. That class also has some fields that are used to calculate the total points as the sum of alien points and time bonus points.

`Game` singleton holds a reference to an instance of `Score` (initialized inline using default constructor) and when a new game starts the ***`startTimer`*** method is called. A fixed amount of points is added to that instance of `Score` each time the `Game` object is notified of the death of an alien. When the game ends the timer is stopped and time bonus is evaluated, depending on the player success of completing the game, and added to the total score using method ***`evaluateTimePoints`***. The timer of that method is implemented using timestamps (retrieved with ***`System.currentTimeMillis()`***).

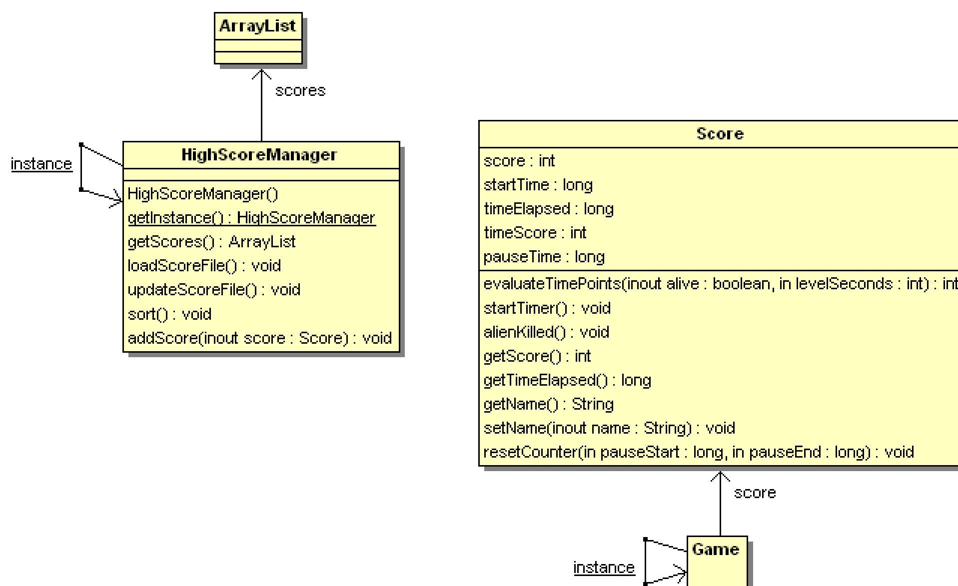


Figure 5.22 - Score system classes (Version 3.0 Rev. 179)

The **Game** class provides a new score screen that gets the initials of the player and shows the high score table at the end of each game. The enumerated type defined in **Game** class for representing all the possible screens of the game was therefore modified as shown in Figure 5.23. Recall that the **Game** instance holds a reference to an instance of **GameState** which value represents the actual screen.

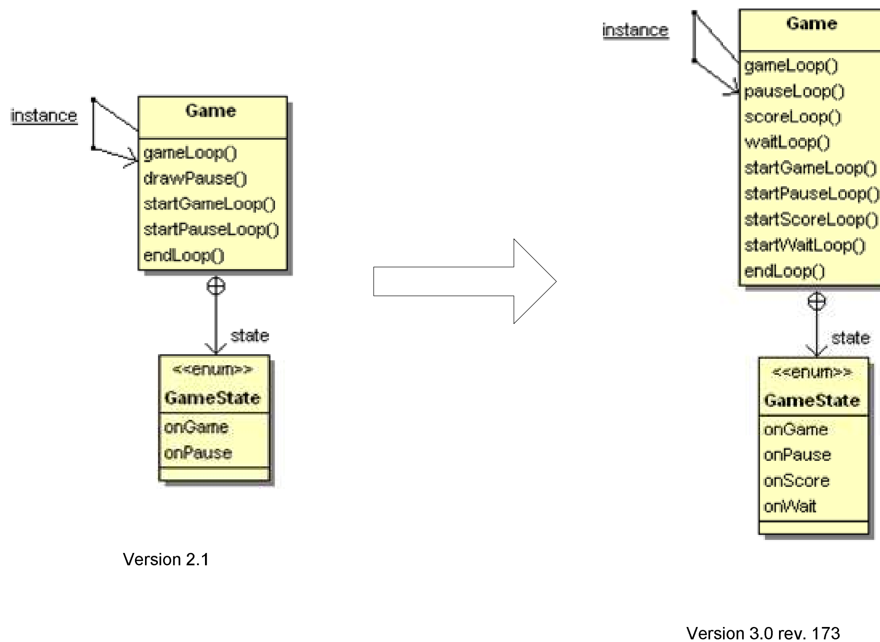


Figure 5.23- Change on Game class passing (v2.1->v3.0)

We also decided to split the game loop in two methods: one method for the wait screen and one for the game screen. The changes on **Game** methods are also represented in Figure 5.23.

In the simplified code snippets below are represented the differences on the implementation of the loop methods.

```
private void gameLoop(){
    If (waitingForKeyPress == TRUE){
        //Move entities, evaluate collisions, update logic, draw game elements on
          screen and handle user input
        […]
    } else {
        //Draw the wait screen
        […]
    }
}
```

Code Snippet 5.5 - gameLoop method (v2.1)

```
private void gameLoop(){
      //Move entities, evaluate collisions, update logic, draw game elements on
        screen and handle user input
      […]
}
Private void waitLoop(){
      //Draw the wait screen
      […]
}
```

Code Snippet 5.6 - gameLoop method (v3.0)

The finer grain changes made in this version were:

Finer grain changes to the code	**Where**
Class added	Added classes **`HighScoreManager`**, **`Score`**
Instance field added to class	Added field of type **`Score`** on Game class (initialized using the default constructor of **`Score`**).
Instance method implementation changed in class	The implementation of method ***`notifyAlienKilled`*** in class **`Game`** changed in order to add points
Instance method added to class	Added methods for score loop in **`Game`** class: ***`startScoreLoop`***, ***`scoreLoop`***
Instance method removed from class	Remove ***`drawPause`*** method from **`Game`** class
Instance method added to class	Added method ***`pauseLoop`*** to **`Game`** class
Instance method implementation changed	Modified implementation of gameloop method. Removed the logic that handled the wait screen.
Instance method added to class	Added methods for wait loop in Game class: ***`startWaitLoop`***, ***`waitLoop.`***
Instance field type changed in class	Changed the `enum` type **`GameState`** in **`Game`** class.

Code Snippet 5.7 - Fine grain changes (v2.1->3.0)

5.3.2 Update test issues and solutions

Enumerated type issue

The first issue encountered was about the use of enumerated types. Enumerated types were poorly supported with the version of Javeleon used so it was impossible to change the number of constants in the definition of **`GameState`** enum type.

Testing the update from previous version (V. 2.1 r.172 -> V. 3.0 r. 174) we noticed after the new module has been loaded that:

- Whenever the player ends a game the “input initials” screen is shown (score screen)
- The value assigned to the variable of the **Game** instance, after assigning the value of `onScore` constant, is still equal to the `onGame` constant; therefore the user input is not handled correctly by the **KeyInputHandler** because it uses the set of actions that apply to the game screen when showing the score screen

In order to solve this issue we raised an update barrier and decided to use a class instead of an enumerated type. The enum type and the field of that type were therefore removed and a static inner class was defined in **Game** class as represented in UML diagram of Figure 5.24.

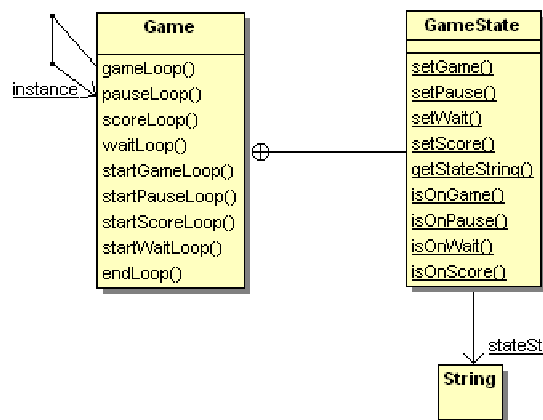


Figure 5.24 - Cascade modifications effect on version 3.0 (rev. 179)

GameState inner static class represents the current screen through the use of the **String** variable `stateStr`. The value assigned to that variable in the default constructor is an empty string, the **Game** class set a value to it using setter methods when the **Game** constructor is called.

With the use of the new inner static class the old state contained in the enum type variable is lost so a cascade modification is necessary because of the loss of state.

Split the game loop into two loops (Changed the specification of a method)

The refactoring that dealt with the game loop implied a change on the specification of the ***gameLoop*** method. As we split that loop in two parts the specification of that method changed. In previous version code the gameloop method provided logic for the wait and the game screen while in the new version code it only handles the game loop logic.

Testing the update from version 2.1 (rev. 176) to version 3.0 (rev. 178) gave different results depending on the value of the state variable that tells which part of the game loop is executing:

- If the update occurs while the game thread is executing the game loop (showing the wait screen) a new game starts unexpectedly.
- If the update occurs while the game thread is executing the game loop (showing the game screen) the behavior is as expected.

In the first case, after the update occurs, the active thread is still executing the game loop but that method in the new version code simply handles the game logic thus starting the game automatically. At that point the value of the state field and the executing loop are consistent with each other so the application can continue.

We solved this issue by modifying the previous versions gameloop body within the cascade modification.

User points are not evaluated correctly

Testing the update from version 2.1 (rev. 176) to version 3.0 (rev. 178) showed an unexpected behavior regarding the calculation of points.

The points that come from aliens killed are not evaluated correctly because the game logic starts to add the points for every alien killed only after the update has taken place and the method ***`notifyAlienKilled`*** has been changed.

Considering the case where the update occurs during a **`Game`**, if the player succeeds on eliminating all the aliens the time bonus is not evaluated correctly. The reason of that unexpected behavior is inherent to the algorithm used to evaluate bonus time and to the introduction of new state (inside of the newly created **`Score`** instance).

When the update occurs during the game a new instance of **`Score`** is created because Javeleon finds the initialization of **`Score`** in **`Game`** constructor. The constructor set all the fields of the **`Score`** instance to the default value but doesn't start the timer. In fact, in the new program code the game starting time is registered only when a new game begins. Using inline initialization is not a solution in this case. It is also impossible to recover the information about when the game started, even considering the use of a state transform function, because that information is not present with the application state representation. Consequently the field remains initialized to 0 (default initialization for long primitive type variables) and the ***`evaluateTimePoints`*** method, which is called when a game ends, cannot evaluate the correct amount of bonus.

In this case we decided to provide method ***`evaluateTimePoints`*** with an implementation that tests the value of the starting time and if it's 0 (that means that the timer was not started) does nothing.

Another approach suggests starting the timer of the `Score` object within the constructor call. Following this approach, if the update occurs while playing a game, the timer gets a timestamp when the `Score` object is instantiated, which is after the game has effectively started. That produces at the end of the game an overestimated amount of bonus points. Considering that the high scores are saved on permanent storage we preferred the approach where the score is underestimated.

Normally when we consider the expected behavior we also define what should be the state after the dynamic update; we do so by considering the several definitions of dynamic update provided in Chapter 4.3.

Consider the introduction of movement capability for the aliens, which was introduced in version 2.0. Consider a case where the user has started a game and has waited a certain amount of time moving the spaceship and firing at the aliens. After this amount of time a dynamic update occurs and, at this point, several states are feasible for the application:

- If we consider the updated program as executed from the start of the application the aliens should be moved accordingly to that consideration. Therefore the block of aliens should not be in the initial starting position after the update. Moreover, since now we consider the aliens as moving entities in the process execution before the update, probably some of them should not be dead, and some of them should be, because of their different interaction with shots fired by the ship. This hypothesis is completely theoretical and does not fit in the real case because it seems impossible to transform the past evolution of the process (when it was executing old program code) to a state that is valid for the new program code, also considering environment interaction.
- We only need a reachable state for the updated program after the on-line program change, so the aliens can start to move from the initial position. The time passed since the beginning of the game is not taken into consideration in this case.

Considering the scenario regarding the user point evaluation, as it seems impossible to pretend that the process will execute, starting from the dynamic update, as if it was executing new program since the beginning of the process, it also seems impossible to recover the time when the user started the game, given the state present at the moment of the update; there is no state mapping function that can manage the retrieval of that information.

5.4 Version 3.1

Version 3.1 presents only a small new feature about the score. The score in this version is drawn on screen during the game. The score that is evaluated during the game reflect the number of aliens killed in that game.

5.4.1 Significant code changes

Visualization of score during game

In version 3.0 the code was already modified in order to keep track of the score of the player during the game. That in-game score is memorized in a field of the **`Score`** instance referenced by the **`Game`** singleton and can be retrieved using method ***`getScore`***.

The implementation of the game loop method of **`Game`** class was modified in order to draw every cycle a box in the upper right side of the screen with the score points retrieved from the **`Score`** instance. In the implementation provided we used only method of the java AWT library in order to draw new elements on screen thus no new classes were created for that purpose.

Change to a method parameters list

In this version we tested a change on the number of parameters of an already existing utility method of **`Game`** class used to draw horizontally centered strings on screen. We added a new parameter to the parameter list of method ***`drawHorCenteredString`***.

Method made less accessible in class

In this version we tested a change on the visibility of methods of class **`Game`**.

The methods of **`Game`** class that were affected by changes in this version are shown in Figure 5.25.

Finer grain changes to the code	**Where**
Instance method implementation changed in class	Changed implementation of game loop method in order to draw the points on screen every cycle
Instance method parameter list changed in class	Added parameter of type **`Font`** to method ***`drawHorCenteredString`*** in **`Game`** class
Method made less accessible in class	Changed visibility of several methods in Game class from `public` to `protected`

Table 5.9 - Fine grain changes (v3.0->v3.1)

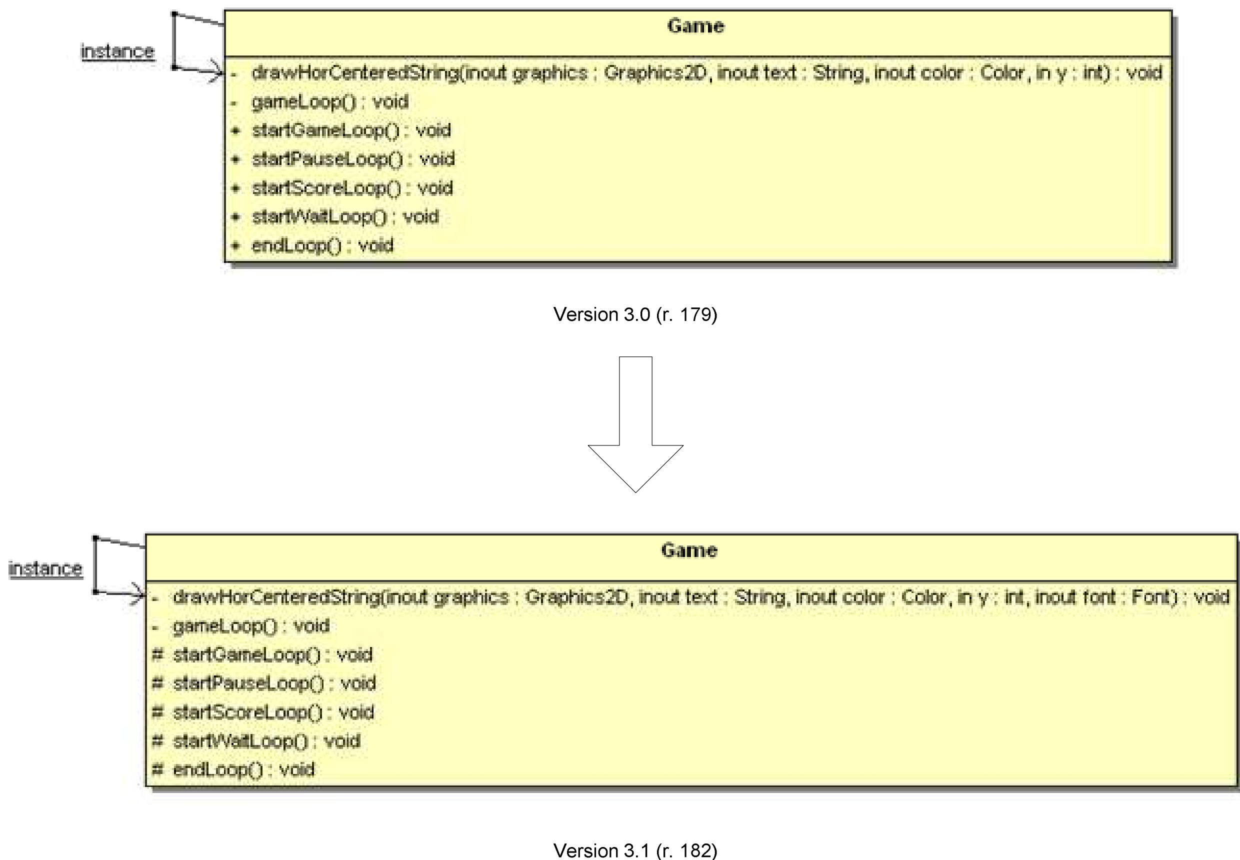


Figure 5.25 - Methods changed in class Game (v3.0r179->v3.1r182)

5.4.2 Update test issues and solutions

Visualization of score during game (Implementation of method changed)

Regarding the visualization of the score on screen during the game, the score box is drawn on screen as soon as the game thread ends the execution of the old game loop code and starts the execution of the game loop again, effectively loading the new version code of that method. Therefore the score box will be visible right after the new module is loaded when the game is showing the game screen. If the user is playing, the effect will be visible after, at most, the duration of one ***gameloop*** cycle.

The effect will not be visible immediately to the user when executing the pause or wait loop because in these loops the last gameLoop's drawn buffer image is used to draw their respective screens.

Change to a method parameters list (remove and add method)

Considering the modifications made to the parameters list we should consider that as a modification to the method's signature. In Javeleon whenever a change to a method affects its name, parameters (order, number, type) or return type the method is considered as a new method. Practically we can see the

modification made to the ***drawHorCenteredString*** in version 3.1 code as the removal of the old method and the addition of a new one. When the update is triggered and if the running game thread is executing an old method version (like the ***gameloop***), which makes use of the removed method, the method call is forwarded to the old method implementation preserving the application execution.

Simple changes that doesn't modify the signature of a method, like the one made in this version to the implementation of ***gameLoop*** method, works as well. If we consider a scenario where an update as been triggered and an active thread is still executing old code where is present a methods call to a function that has only changed its implementation, that methods call is forwarded to the newest implementation of that function.

The only unexpected behavior present when updating from version 3.0 to 3.1 is relative to the user update awareness already discussed in the chapter regarding version 2.0. When updating during pause screen the user will not see the points' box appear immediately because of the way the pause loop draws elements on screen.

Method made less accessible in class

The modification made to the visibility of several methods in **Game** class didn't change the signature neither break the interface with caller and so it produced no unexpected behavior within the dynamic update.

5.5 Version 4.0

The new main features of this version are:

- Game entities are represented with raster images;
- Explosions animations.

In addition to these features we made several modifications to the code implementing minor features, fixing bugs and testing some refactorings:

- Minor features:
  - Antialiasing;
  - Frame-per-second visualization and fps limit control;
  - Modifications to gameplay:
    - Incremented the starting number of aliens;
    - Aliens that hit barriers die.
- Bug fixes:
  - Check if the score is allowed to be on highscore before saving it
  - Changed the format of the score file (no retro compatibility with the old format).
- Refactorings:
  - Moving classes in different packages;
  - New classes and interfaces;
  - New default implementation of method in abstract super class;
  - Refactoring of the collision evaluation/handling system;
  - Refactoring of the score system;
  - Refactoring of the entities and collections hierarchies;
  - Split game logic from control and presentation (drastic refactoring);
  - Moved (and inline) method.

We tried to implement all of these changes to the code and then test the update with the previous release. Even if the set of modifications was numerous and heterogeneous, most of the code changes showed a satisfactory application behavior. That means that the experience gained with past versions helped thus improving the updateability of the application through the selection of good design choices.

The only refactoring that raised an update barrier was the refactoring about the separation of domain logic from control and presentation. We will see, after the explanation of the code changes, why that refactoring posed an update barrier.

5.5.1 Significant code changes

Starting from the new functionalities let's see what are the related code changes.

Code changes related to new main features

Use of images to represent entities in game

In order to let every object represent itself with a sprite the default implementation of ***draw*** method that is provided in **Entity** super class in classes **Alien**, **Ship** and **Shot** was overridden, exploiting the Liskov Substitution Principle. We also added the new interface **Drawable** to the implementation list of class **Entity**.

The new implementation of the ***draw*** method recovers a **Sprite** object (that represent an image) using the ***getSprite*** method on the singleton **SpriteStore** by passing a string with the name of the image. The **SpriteStore** class simply caches all the images that are requested so that multiple requests on the same image do not require multiple accesses to hard disk. The method ***draw*** of the recovered ***Sprite*** instance is consequently called passing parameters about the position and the maximum dimension of the image.

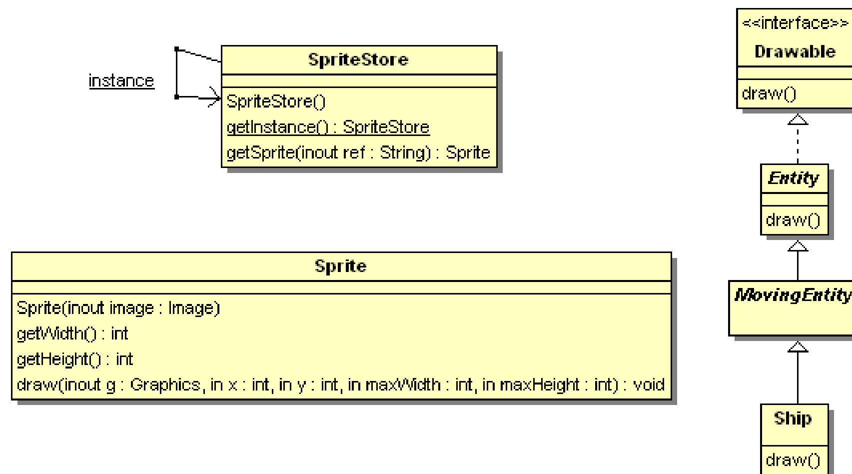


Figure 5.26 - Code changes in version 4.0 related to the way entities are drawn on screen

Finer grain changes to the code	Where
Class added	Added classes **SpriteStore** (singleton) and **Sprite**
Interface added	Added interface **Drawable** that declare method ***draw***

Interface added to implementation list of class	Added interface **`Drawable`** to implementation list of **`Entity`** class
Instance method added to class	Added method draw class to classes **`Alien`**, **`Ship`**, **`Shot`**
Instance method implementation changed in class	Changed implementation of method draw in **`Barrier`** class

Table 5.10 - Fine grain changes related to the introduction of raster images (v3.1->v4.0)

Explosions and animations

With the implementation of explosions we have introduced few more abstractions applying refactorings to the code:

- **`Collidable`** interface: declare a method that returns a Rectangle object used by the collision checking system to evaluate collisions between entities. Null value is admitted as a return type.
- **`Animated`** interface: interfaced implemented by entities that shows an animation. Declare a method animate that performs an animation step considering the time interval passed as a parameter.
- **`PauseAffected`** interface: Interface implemented by all classes that are affected by pause periods and should be notified when a pause ends. Elements that use timers like Ship and Aliens (for idle fire time) implements that interface. Animated entities also implements that interface
- **`MovableEntityCollection`** and **`FixedEntityCollection`**: two abstract classes that defines the behavior of collections of fixed entities or movable entities reflecting the design used in the entity hierarchy tree.

The finer grain changes that were influenced by the introduction of these new abstractions will be discussed in detail in the refactoring section that deals with new classes and interfaces.

The changes made to the design relatively to the implementation of explosions are shown in Figure 5.27.

Game singleton keeps a reference to an instance of **`Explosions`**, which is a moving collection of explosions. The explosions reference of **`Game`** is new state initialized (using in-line initialization) with a default empty set of explosions (**`Explosions`** default constructor). Whenever a collision is found and an alien or the ship die an **`Explosion`** object is created and added to the collection. **`Game`** class now also manages to move, draw and animate the explosions every cycle in the game loop by calling the appropriate methods on the collections.

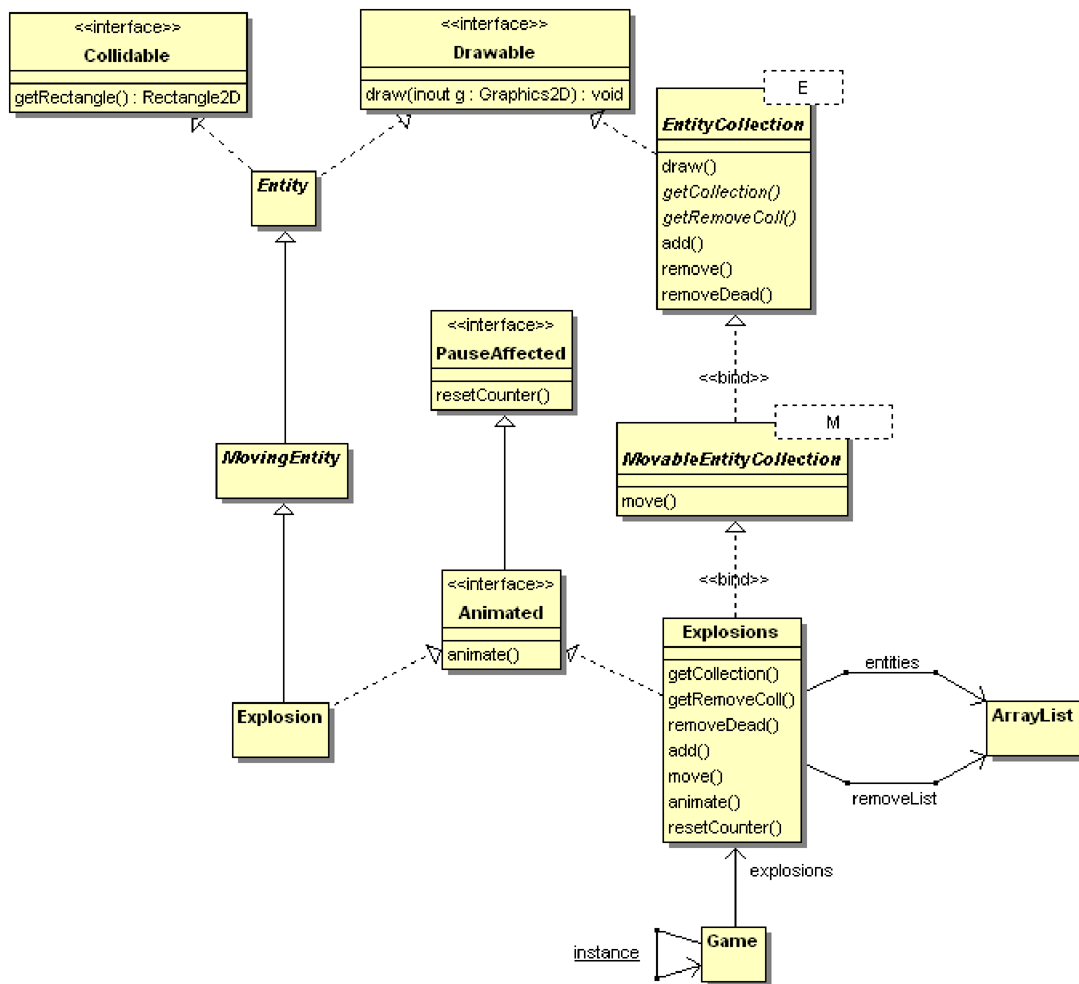


Figure 5.27 – Changes to the code related to the introduction of explosions and animations (v4.0)

Code changes related to new minor features

Antialiasing

In order to add antialiasing effect in the game a new boolean field `antialiasing` was introduced in **`Game`** class. This field is inline initialized to true. The method that draws the game elements on screen was modified in order to check that value every cycle and enable or disable the relative rendering effects.

With this modification state was introduced correctly and initialized to the default value (antialiasing enabled)

Frame per second limiter and visualization

In order to implement this feature a new class **`FPSDisplayer`** was created. That class implements drawable and so provides a method ***`draw`*** that draws the fps, provided a graphic context.

Like in the previous case a Boolean field was added to Game class (inline initialized to false). The method that draws entities of class Game was modified in order to evaluate the fps, check if by drawing on that cycle is surpassing the max fps limit and draws the fps value on screen by creating a new FPSDisplayer on the fly and calling the draw method on that.

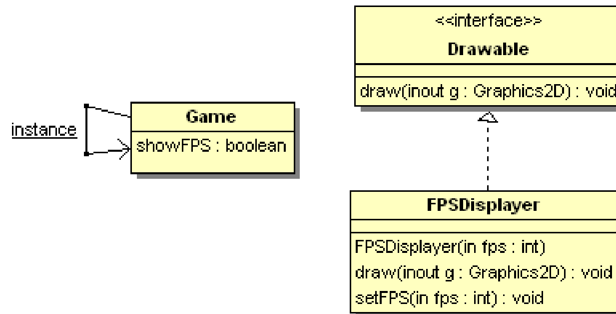


Figure 5.28 – Modifications to the code regarding the visualization and limiter of FPS (v4.0)

Modifications to the gameplay

In this version we decided to raise the difficulty level of the game by adding more aliens and making the aliens' matrix thicker. In order to do that we had to modify **`Aliens`** class by making these code changes.

- Changed the constants `DEFAULT_ROWS` and `DEFAULT_COLUMNS` in **`Aliens`** class that tells how many rows and columns the aliens' matrix should have when an **`Aliens`** instance is created.
- Modified the constructor of **`Aliens`** in order to create a thicker matrix

Recall that the **`Aliens`** collection instance is created whenever the **`Game`** constructor is called and at the start of a new game.

The modification that regarded the interaction between aliens and barrier affect the collision detections system. That part of the application was completely redesigned in this version so we will describe that later when dealing about that refactoring.

Bug fixes

Check for high score and "Extract method" refactoring

In order to check if a score can be on the high score table we made these modifications to the code:

Finer grain changes to the code	**Where**
Instance method added to class	Added a method that tells if a **`Score`** object passed as a parameter is going to be on high score in **`HighScoreManager`** class
Instance method added to class	Added procedure ***`endGame`*** in **`Game`** class that handles some

	common statements of methods ***notifyWin*** and ***notifyDeath*** and checks if the score of the user is going on the high score, taking subsequent action (calls the right loop).
Instance method implementation changed	Removed statement that are common in methods ***notifyWin***() and ***notifyDeath***()

Table 5.11 - Fine grain modifications regarding "Extract method" refactoring in class Game (v4.0)

```
public void notifyDeath() {
 message = "Oh no! They got you, try again?";
 score.evaluateTimePoints(false, 60);
 endGame();
}
public void notifyWin() {
 message = "Well done! You Win!";
 score.evaluateTimePoints(true, LEVEL_TIME);
 endGame();
}
private void endGame() {
 waitingForKeyPress = false;
 endLoop();

 if (SimpleHighScoreManager.getInstance().isInHighScore(score)) {
     startScoreLoop();
 } else {
     startWaitLoop();
 }
}
```

Code Snippet 5.8 - Extract method refactoring on notifyDeath and notifyWin methods (v.4.0)

The program checks the validity of the score in the ***endGame*** method.

With these three modifications we practically extracted a common procedure from 2 methods of the same class and moved that logic into a separate method. This refactoring can be called at higher granularity level as “**Extract method**”.

Modification to the way scores are saved to disk

This refactoring is explained in the section that deals with the refactoring of the score hierarchy tree.

Refactorings

Moving classes in different packages

As in this version we’ve introduced several abstractions and made some classes reusable we decided to move some of the classes that are part of a reusable framework into separated packages.

- EntityCollection class from Invaders.Collection to GameLogic.Collections
- Entity FixedEntity and MovingEntity classes from Invaders.Entities to GameLogic.Collections

- HighScoreManager and ScoreComparator classes from Invaders.ScoreSystem to GameLogic.ScoreSystem

Moving a class from a package to another result in changing the full name of that class; hence the moved class is considered as completely new in the latter version of the program. Therefore this action can cause problems with classes that hold state. When a class that holds state is moved into a new package the classes is considered as new and the state is not preserved for instances of that class. However moving abstract classes that only defines behavior and relies on properties implemented by concrete subclasses is possible and does not cause problems as with the EntityCollection, FixedEntity and MovingEntity classes. The loss of state caused by moving concrete class HighScoreManager is tolerated. ScoreComparator is an utility class that provides a method for comparing score so it doesn't hold state and can be moved.

New classes and interfaces

As already told when dealing with the introduction of new major features this version introduced new interfaces, abstract classes and concrete classes.

New interfaces

The new interfaces are:

- PauseAffected
- Animated
- Drawable

The PauseAffected is implemented by all the classes that are sensitive to pause periods because they work with timers, like the **`Ship`** class. These classes have to be notified of pause periods in order to keep track of how much time the pause period lasts. Recall that whenever the Ship method fire is called that method checks if the ship instance is qualified to fire in order to maintain a certain fire interval.

Some classes like **`Aliens`** and **`Ship`** already provided a method for resetting a pause counter but the name of that method is different from the one in the new interface, so in these cases the modification implied the removal of the old method and the addition of the new one.

The introduction of this new interface affected also new classes or interfaces (**`Animated`** interface and **`TimedScore`** class) that, respectively, extends or implements it.

The **`Animated`** interface is implemented by entities classes that perform animations.

The **Drawable** interface is implemented by all the objects that can have a representation on screen, entities or not. The *draw* method declared in this interface accepts a **Graphics2D** parameter. All the drawable elements classes must implement this interface. In the former version some drawable elements had a draw method that accepted a **Graphics** parameter. We had to change all of these methods to accept the **Graphics2D** instead of the **Graphics** object. This operation, like said before, is equal to the removal of the old method and the addition of the new one.

The application scenario of these new interfaces is shown in the image below.

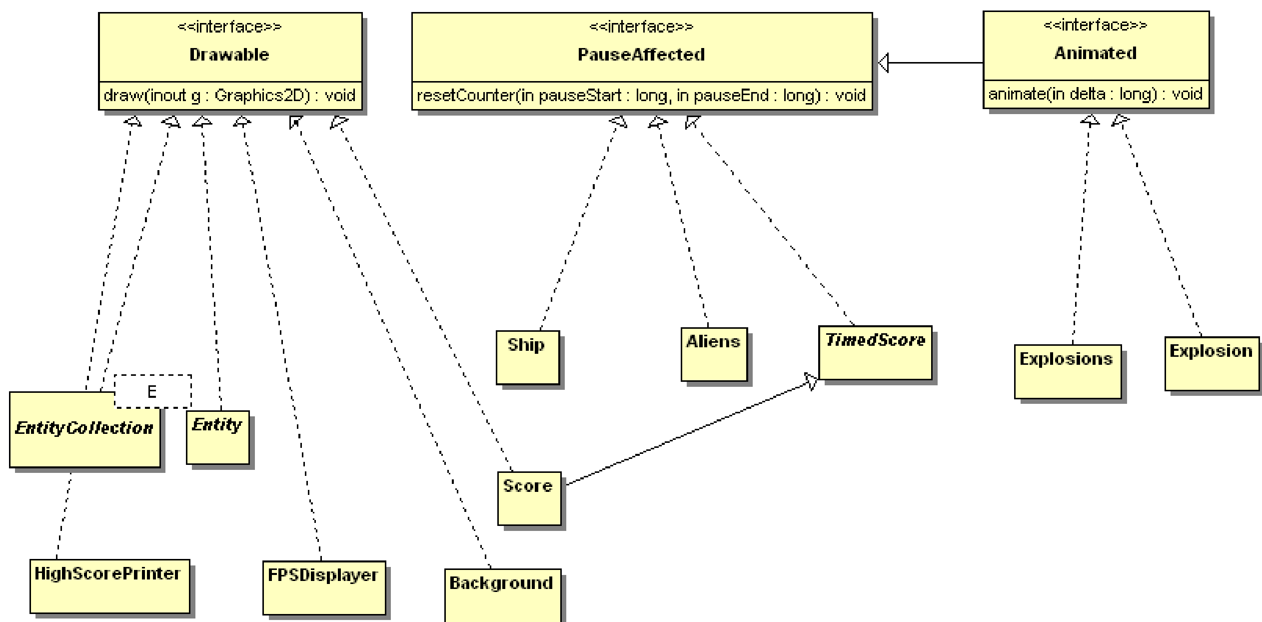


Figure 5.29 - Use of new interfaces in version 4.0

New concrete classes

We moved the code inside of the game loop of **Game** class that draw the high score on screen into the draw method of a new class **HighScorePrinter** that implements the **drawable** interface.

We moved the method that draws strings ***drawHorCenteredString*** from **Game** class to a new class **StringToolkit**. That new method is static in the new class.

This kind of refactoring can be called at a higher level of granularity as “**Move method**”.

New default implementation of method in abstract super class

We’ve introduced a default implementation for the ***move*** method in **MovingEntity** abstract class. All the classes that in the former version were subclasses of **MovingEntity** still have their custom

implementation for this method. Only new class Explosion (that is a **`MovingEntity`**) does not redefine the default behavior of that method so it relies on the implementation of **`MovingEntity`**.

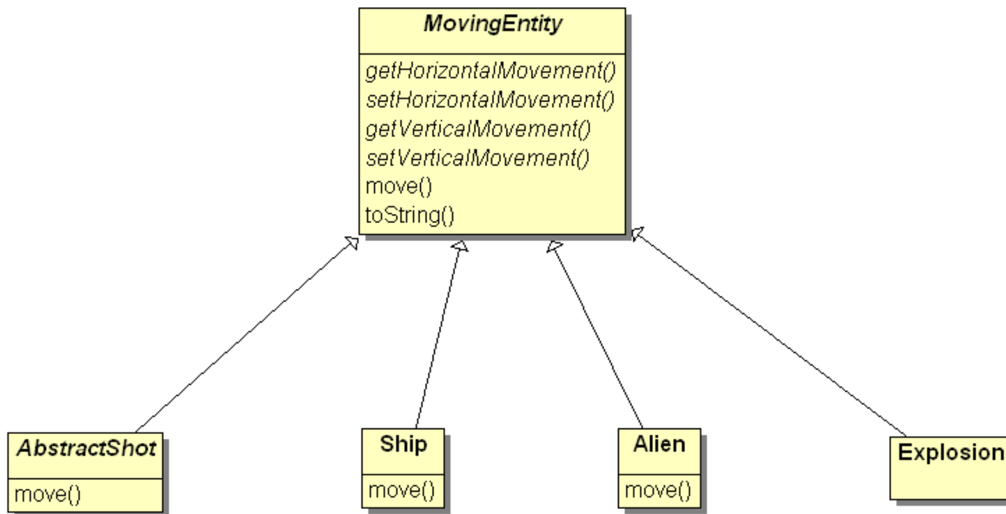


Figure 5.30 - Implementation of move method in version 4.

Refactoring of the collision evaluation/handling system

In this version we did a refactoring on the part of the application that handles collisions between objects using a component based approach.

In the former version the entities collections had a method used to check if instances of type **`Entity`** collide with other instances of type **`Entity`**. This approach does not follow a good design in my opinion because the collision checking duty can be easily delegated to a separate reusable component. Whenever an operation involves two or more instances of the same type why place that operation inside a method of that class? How do we choose on which of the instances call the method? Programs should be developed using reusable components instead of application specific code.

First of all we made a distinction between collision detection and handling. Collision detection can be executed by a single module and when a collision is detected, the collision can be sent to another module that knows how to handle it.

We implemented an interface **`Collidable`** for objects that can collide with other **`Collidable`** objects. This interface declares a method that returns a **`Rectangle2D`** object used for collision checking. The **`Entity`** class naturally implements that interface returning a rectangle with the dimension and position of the entity retrieved by using the getter methods of the **`Entity`** class.

`CollisionEngine` is an abstract class that defines the standard behavior of a collision checking engines. A collision engine usually keeps track of a list of associations between collision types and handlers.

This abstract class is structured using the template design pattern, deferring the implementation of the associations list to concrete subclasses. It defines some methods that checks collisions on `Collidable` objects or lists. When one of these methods is called the collision engine checks collisions on every object passed as a parameter and call appropriate handlers for the detected collision. The way the collision engine retrieves the handler for the detected collision depends on a specific implementation.

On a first instance we've adopted an implementation that makes use of a hash table in order to store the association between collision types and handlers. In that implementation the Game class needs to register itself to the collision engine for all the collisions that it requires controlling. If we initialize the collision engine field with an empty hash collision engine when the Game class instance call methods of the collision engine that checks for collisions the collisions can be detected but no action is taken because no handlers are registered.

The registration of handlers is made possible by the means of a factory class that produce for the Game instance a custom `HashCollisionEngine` with all the registrations already set.
With this implementation the update works passing from version 3.1 to version 4 because a new collision engine is created using the factory method. What is not desirable in this approach is the future possible behavior of the application when facing new collision types.

The UML class diagram of Figure 5.31 shows the relationships between classes related to the collision framework.

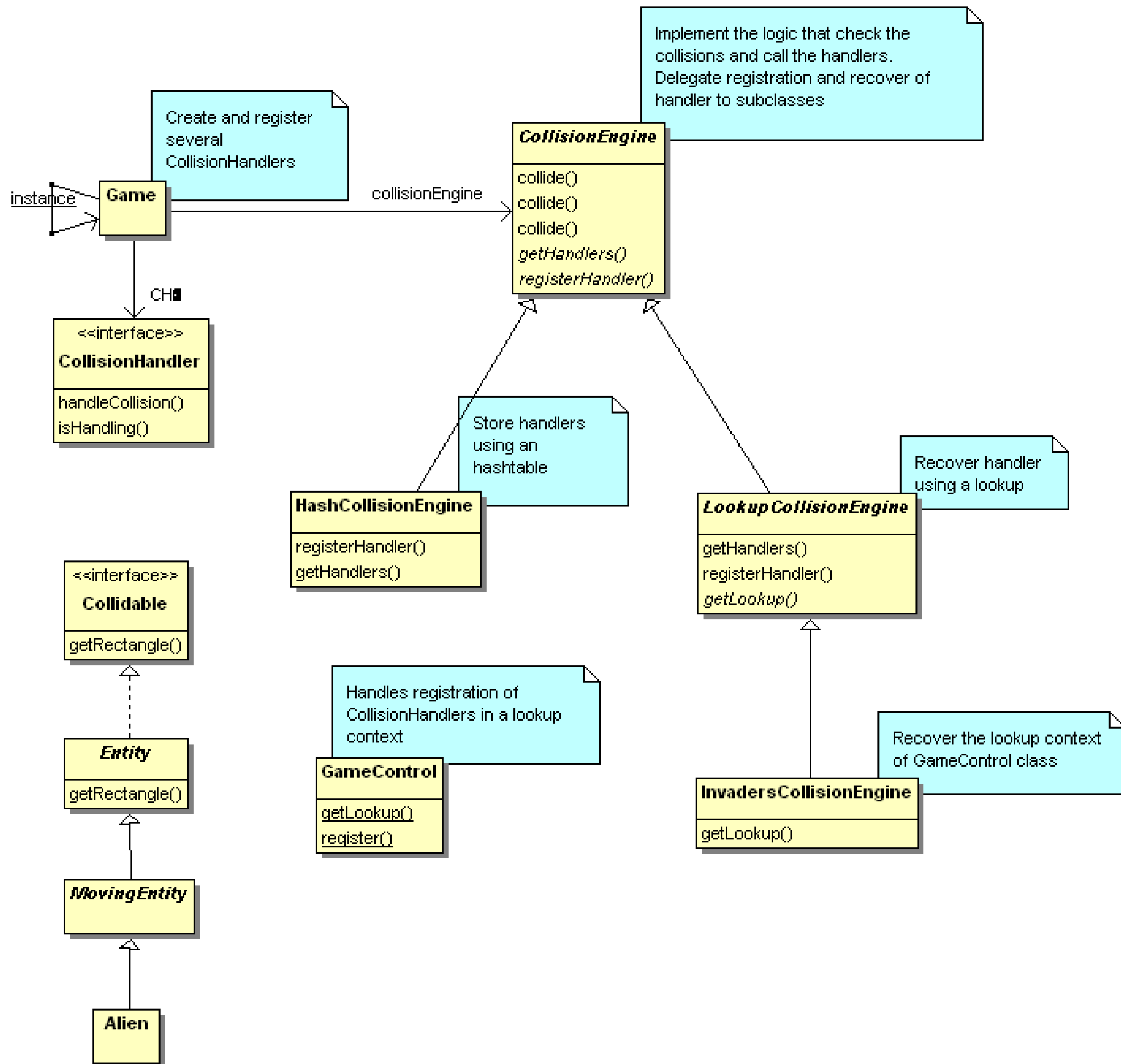


Figure 5.31 - Collision evaluation/handling system implemented in version 4.0

If a new version of the application will need new registrations to the collision engine how can the programmer introduce new state without letting the application being affected by transitory period of undetected collisions?

If in a future version we'll need to introduce new collisions, even if we modify the factory method that produce the collision engine, this method will not be called unless a new game restarts (***startGame*** method of class **Game**). That is because Javeleon cannot check for differences on the body of the factory method that returns the collision engine's instance; therefore a new collision engine (with all the new associations) is not created.

By working this way, before the game restarts, the new collision types will remain undetected showing a transitory period where the game behaves strangely and new collision types are not handled.

The problem in this case resides in the parameterization of the collision engine. Following the actual implementation that module needs explicit initialization to work correctly.

The associations between collision types and handlers have to be considered as new state, new data in memory. In order to resolve this issue we've decided to use a loosely coupled communication mechanism provided by the `Lookup` library.

Before implementing the collision engine we created a custom lookup context for objects used by the application game. This context is accessible using static methods of class **`GameControl`**.

```
public class GameControl {
    private static final InstanceContent content = new InstanceContent();
    private static final Lookup lookup = new AbstractLookup(content);

    public static final Lookup getLookup() {
     return lookup;
    }
    public static <T> T register(T obj) {
     content.add(obj);
     return obj;
    }
}
```

Code Snippet 5.9 - GameControl Class (v4.0)

Subsequently we created a new concrete implementation of the collision engine that uses lookups in order to retrieve collision handlers. Class **`Game`** defines several handlers as inner classes, one for every kind of collision. **`Game`** class uses field initialization to create and register in the custom lookup context one of each of these classes. In that way whenever a new collision handler is needed the programmer needs only to create a new inner class of **`Game`** that implements the **`CollisionHandler`** interface, implement the methods and use field initialization in class game in order to be sure that one instance of that class is registered in the lookup context.

```
private CollisionHandler CH1 = GameControl.register(new Alien_BarrierCollHandler());
private CollisionHandler CH2 = GameControl.register(new Alien_ShipCollHandler());
private CollisionHandler CH3 = GameControl.register(new Alien_ShipShotCollHandler());
```

Code Snippet 5.10 - Collision handler registration with inline initialization (v.4.0)

At this point we have several collision handlers but the collision engine is still not using them. The lookup collision engine uses the template design pattern.

When the Game class commands a collision check on several entities the collision engine checks these collisions and whenever it finds one it calls the method that recovers a handler for that specific kind of collision. That method execute a lookup in order to find all the classes that implement the collision handler interface and check which of these are able to handle the collision by calling the method *isHandling* (of the collision handler interface) of each one of them. When this method finishes we have a list of handlers for the detected collision so we can call the *handleCollision* method on each one of them.

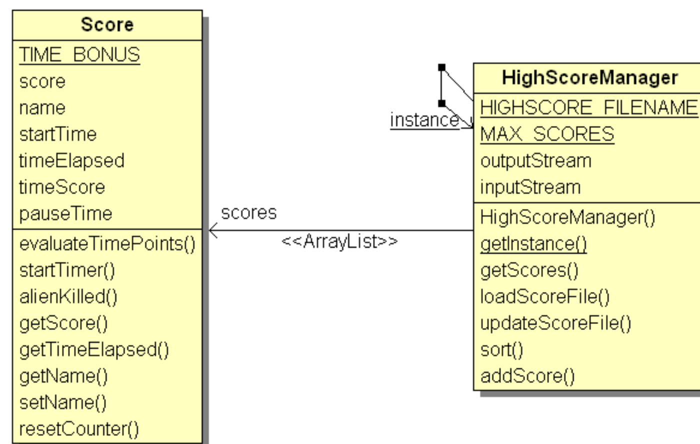
Thanks to the use of loosely coupled communication we avoid the issue with the initialization of the collision engine. Parameterization of an instance of a class (in this case the **`HashCollisionEngine`**) during an update is not possible. Instead of using a hash table to store the information about the availability of the handlers and create dependencies between objects (Collision handlers and the collision engine) we create instances of collision handlers and we let the collision engine finds them. Instead of registering the information of which kind of collisions a certain handler manage directly in the hash table we put that information directly in the code with the *isHandling* method.

	Hashtable approach	Lookup approach
How to register handlers	Parameterization of the collision engine	Registration in the lookup context
How to recover handlers	Recover the handlers from the hash table	Recover the handlers from the lookup
How do the engine knows which handlers are needed for a certain collision	Query the hash table to return handler for a specific collision	Get all the handlers and then ask each handler if it can handle the collision

Table 5.12 - Differences between Hashtable and Lookup approach with the collision engine

Refactoring of the score system

The score system was subject to many modifications from the previous version. With reusability of design in mind we've made a redesign of the score system that involves **`Score`** and **`HighScoreManager`** hierarchy trees. The changes on the hierarchies are represented in the diagrams of Figure 5.32.



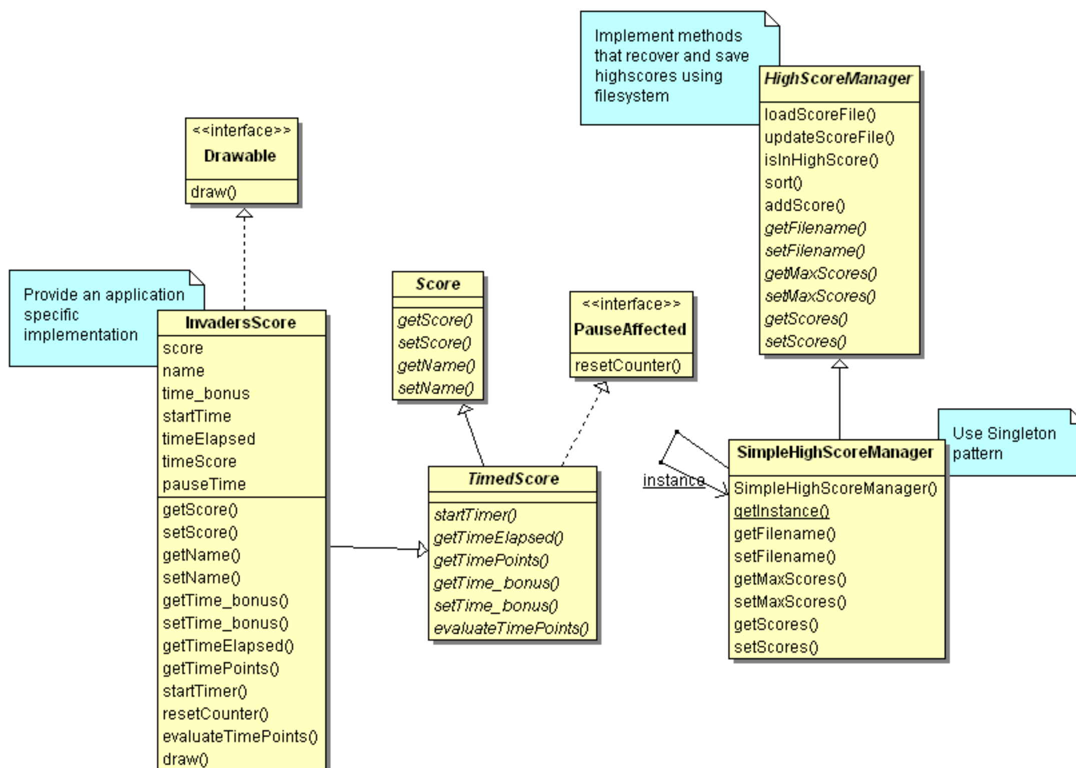


Figure 5.32 - Redesign on score framework (first solution)

As it is possible to see from the UML class diagram the **`HighScoreManager`** is structured using the T*emplate* pattern so all the implementation is moved to the leaves. All the implementation is moved to the leaf also for the **`Score`** hierarchy (see class **`InvadersScore`**).

Examining the change in the **`Score`** tree hierarchy we can easily see that the **`Score`** class became an abstract class.

As the Game class has a reference to a `Score` object through the reference named `score` when updating from the version 3.1 to version 4 Javeleon tries to create a new `Score` object to perform the state transfer. This operation results in an *instantiation exception* and cause the application to crash. The object creation and state transfer occurs whenever an object reference is used.

By using the design represented in the diagram of Figure 5.32 and changing the variable name (*score*), which keeps a reference to the score, we made two changes to the code in `Game` class:

- Instance field removed from class
- Instance field added to class

With these changes we lose the state because a new instance of `InvadersScore` is created.

A solution that should solve the issue with the update introduces a constraint on classes naming as it is possible to see from the UML diagram below.

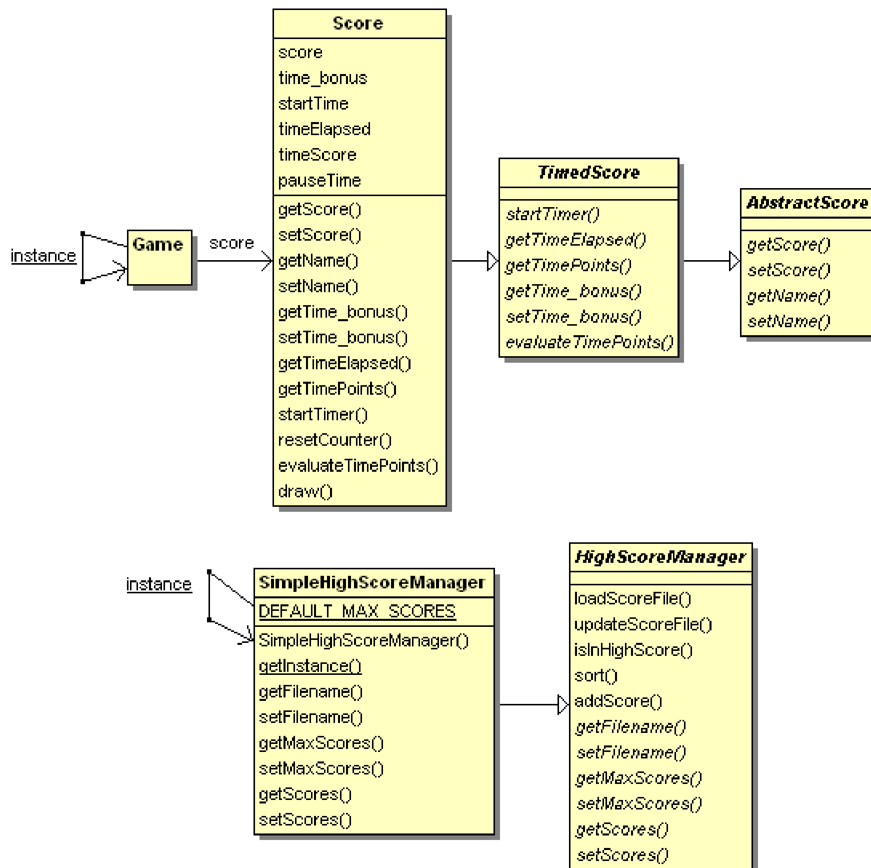


Figure 5.33 - Redesing of score framework (second solution)

Keeping the concrete class with the same name allows recovering the old state of score because the class does not change internal representation, hence the state is transferred correctly with the dynamic update.

One other thing that changed from the former version to the latter is the introduction of modifier *transient* for several fields of the class `Score`.

The high score manager class saves scores on disk using serialization. During the development of the latter version we realized that saving all the fields contained inside the score class is useless, the only fields that the game needs are the score value and the name of the player for each score entry. That is the reason why we introduced the *transient* modifier on the other fields of class `Score`.

No exceptions are thrown after the new module load, but the former game high score file saved on disk is not compatible with the new version. When a problem on loading scores from disk occurs the high score manager create and empty list of scores. After the user name input the manager saves the list of high scores in memory to the disk, overwriting the old file.

This is an example that shows how code changes involve a state change in the environment outside of the program.

Refactoring of the entities and collections hierarchies

In the latter version, with the introduction of the collision engine we had to discriminate alien shot and ship shot entities. We had to create two distinct classes for these objects as the collision evaluation systems works with class types.

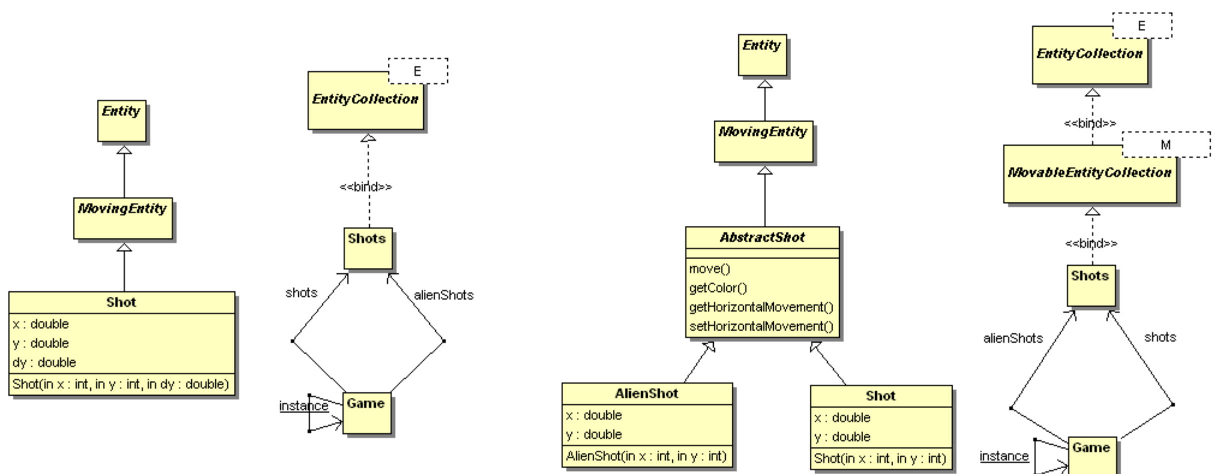


Figure 5.34 - Differences between v3.1 and v4.0 on shots

We introduced an abstract class **AbstractShot** to represent the behavior of the generic shot. In order to maintain the state of **Shot** entities we decided to represent alien shots with objects of class **AlienShot** and ship shots with objects of class Shot. This example however is trickier than in the previous **Score** case. In this case an already existent class that represents more concepts in the previous version (alien and ship shots) is replaced by two classes with the use of polymorphism. After the update whenever the program will access a reference to Shot object Javeleon will try to create a new Shot object (using the definition of the latter version) and execute the state transfer from the old object to the new create instance. All of the former **Shot** instances that were representing shots of aliens will become shot fired by the ship. Moreover these "transformed" former alien shots will remain referenced in the old container used for alien shots.

A transitory period can be observed by executing the update:

- All the shots will continue to move on the screen but as the shots fired by the aliens become shots fired by the ship these shots will move up. That happens because the implementation of the move method of new instances is taken from the **Shot** class instead of the **AlienShot** class.
- No collision will be detected when a "transformed" former alien shot will hit an alien because the **Game** class will never check collision between list of entities contained in the container for alien shots and the entities contained in the alien container.

Regarding collections We introduced two main modifications in the collections package:

1. New level of abstractions.
   Introduced a new level of abstraction for collection of moving and fixed entities.
2. Template binding
   In the previous version classes **Barriers**, **Aliens** and **Shots** simply inherited **EntityCollection** without binding the template type E. In the latter version all the leaf classes inherit the superclass with a binding on the specific application type. See the code differences below.

```
public class Shots extends EntityCollection
```

Code Snippet 5.11 - Version 3.1 EntityCollection class declaration

```
public class Shots extends MovableEntityCollection<AbstractShot>
```

Code Snippet 5.12 - Version 4.0 EntityCollection class declaration

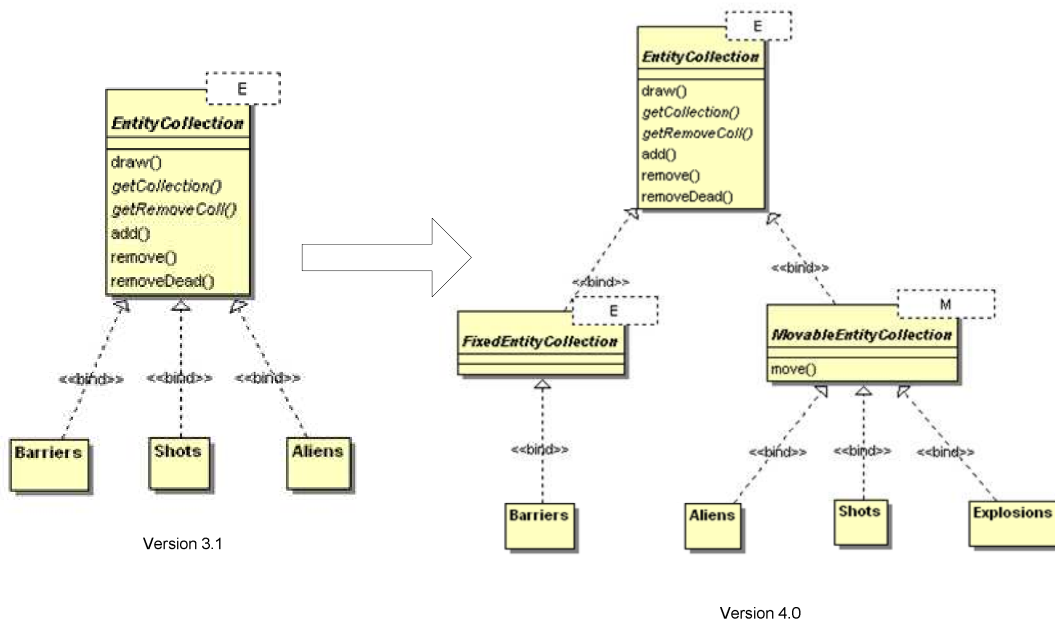


Figure 5.35 - Changes to the design concerning the introduction of FixedEntityCollection and MovableEntityCollection in version 4.0

Separating game logic from presentation and control

Representation, domain logic and control are mixed in the same class (class **`Game`**) in release 3.1 code. Many application developers tend to avoid this approach for long lived applications because it roughly binds three aspects of the application that should be separated. That approach has an impact on interdependency between those three aspects, flexibility of design and therefore affects the maintainability of the architecture. A good design in this case should involve the usage of three different classes with the *Model-View-Controller Pattern* (that use *Observer Pattern*).

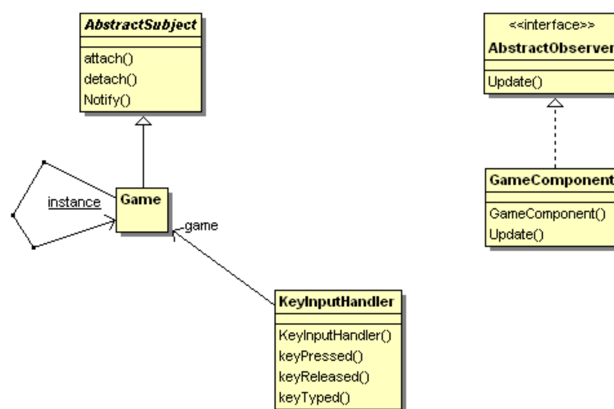


Figure 5.36- Implementation of the Observer pattern (v.4.0)

We partially implemented a *MVC pattern* using the *Observer pattern* with the **`GameComponent`** class and the **`Game`** class that manage the game logic. **`GameComponent`** is a class that inherits the **`Component`** class

so it can be added to the **TopComponent** as the class **Game** in the former version. With the *Observer pattern* an object, called the subject, maintains a list of dependants, called observers. Whenever the subject needs to notify the observer because something changed in its state it notifies all of them. The subject is the class that manages the game logic and the observers are the classes that handle the representation.

This refactoring cause the application to crash during the dynamic update test. That happens because the refactoring completely changed the hierarchy tree for the **Game** class as in the former version it inherited class **Component**. **Game** class in the latter version is s subclass of **AbstractSubject** instead of **Component**. Whenever the **TopComponent** tries to access the **Game** instance it tries to cast that object to a **Component** therefore casting a **ClassCastException** because of the new definition of class **Game**. Avoid this issue means losing the state of class Game, which is practically the state of the entire application. That is why we made rollback and decided to not implement this feature.

Move (and inline) method

Move the logic that checks if an alien is on first line from **Alien** class to **Aliens** class.

Code changes:

- Instance method removed from class
  Removed method that tells if an **Alien** is on the first line
- Instance method implementation changed
  Added logic that looks if an alien is on the first line in method ***getFirstLineAlien***() of **Aliens**

This refactoring can be called at a higher granule level as "**Inline method**". This kind of refactoring is update-compliant because it doesn't involve movement of state between classes.

5.5.2 Update test issues and solutions

Soft issue (no exceptions) with the change of aliens dimension and stepdown

Under certain circumstances the modification made to the width of the aliens can bring to undesired behavior and transition period. If the user is playing and the alien come to right extremity of the game field when the update occurs, the growth in width of the aliens can bring the elements on the right side to be suddenly out of bound. When aliens are found to be out of bounds the game logic tries to bring the whole matrix inside of the game fields by, first letting the matrix have a step down and then moving all the aliens in the opposite direction by the same amount of space of the last movement. That algorithm clearly cannot work in our case because it expect the dimension of the aliens to be fixed. In fact after the game logic has

moved the block of aliens in the opposite direction it is possible that the elements on the right side of the matrix are still out of bound because of their width growth. In this case the matrix of aliens will rapidly perform multiple stepdowns forcing the game to end.

This problem is caused by the poor architecture choices that were taken during the development of the first release regarding the representation of aliens' state. Chapter 6 will describe a few practices that can avoid this problem by separating information into data atoms allowing constant to vary from release to release while avoiding transition periods with dynamic updates.

Increased the number of aliens

The variability on the behavior of the aliens is also noticeable considering the change on the number of aliens. Since the **`Aliens`** collection is generated only each time a new game start and whenever the constructor of **`Game`** class is called, the number of aliens in memory is not updated immediately with the online program change.

5.6 Version 5.0

In this iteration we focused on enhancing the game play by adding new classes of enemies. A new class of alien that has health was added, therefore the player has to hit this kind of alien multiple times to destroy it. Moreover, we added the spaceship, a completely different kind of enemy that fire laser instead of simple shots. Aliens' animations and some minor features regarding graphics (background image and better explosions) were also introduced during the development of this iteration.

In addition to the new features, we made some refactorings:

- Removed the dependency between moving entities' classes and Game class
- Push down method (interface)

5.6.1 Significant code changes

New alien types

In order to introduce the new types of aliens we created a new level in the hierarchy represented by the `AbstractAlien` class. Version 4 `Alien` class is now a subclass of `AbstractAlien`, together with new classes `ToughAlien` and `Mothership`. Using the same name for concrete class Alien in the former and in the latter version ensures the preservation of state for the instances of that class.

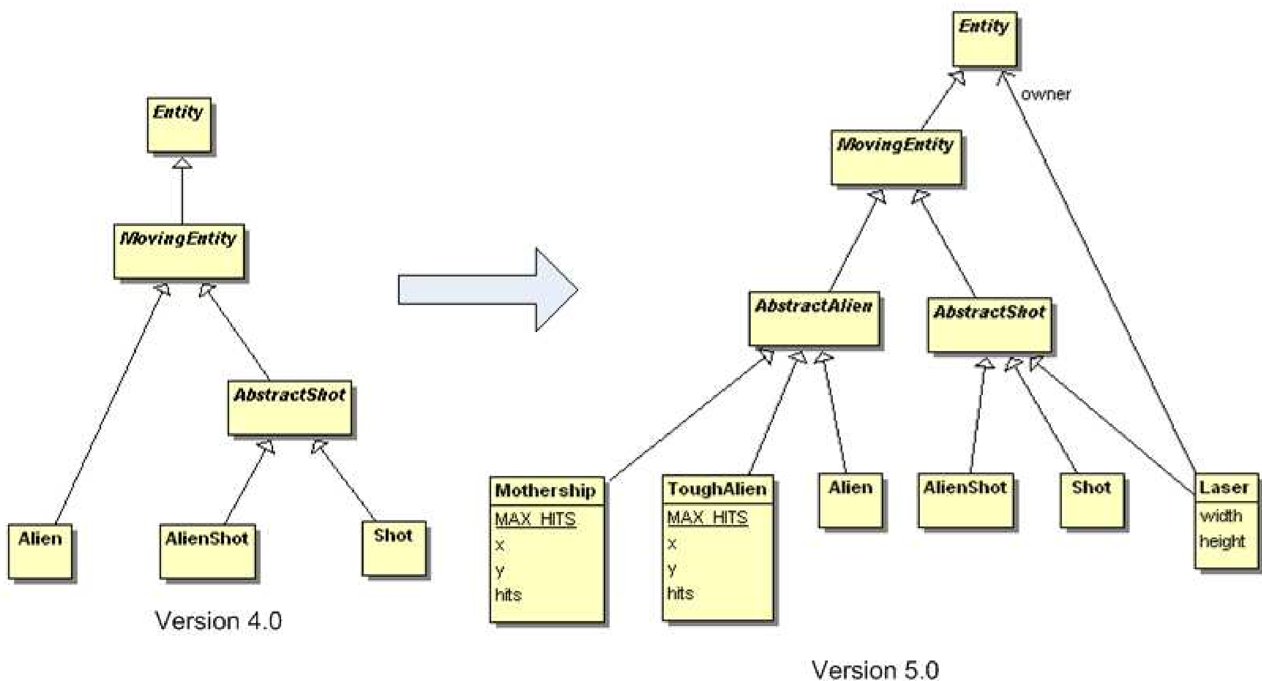


Figure 5.37 – Modifications on entities hierarchy regarding the introduction of new classes of enemies (v4.0->v5.0)

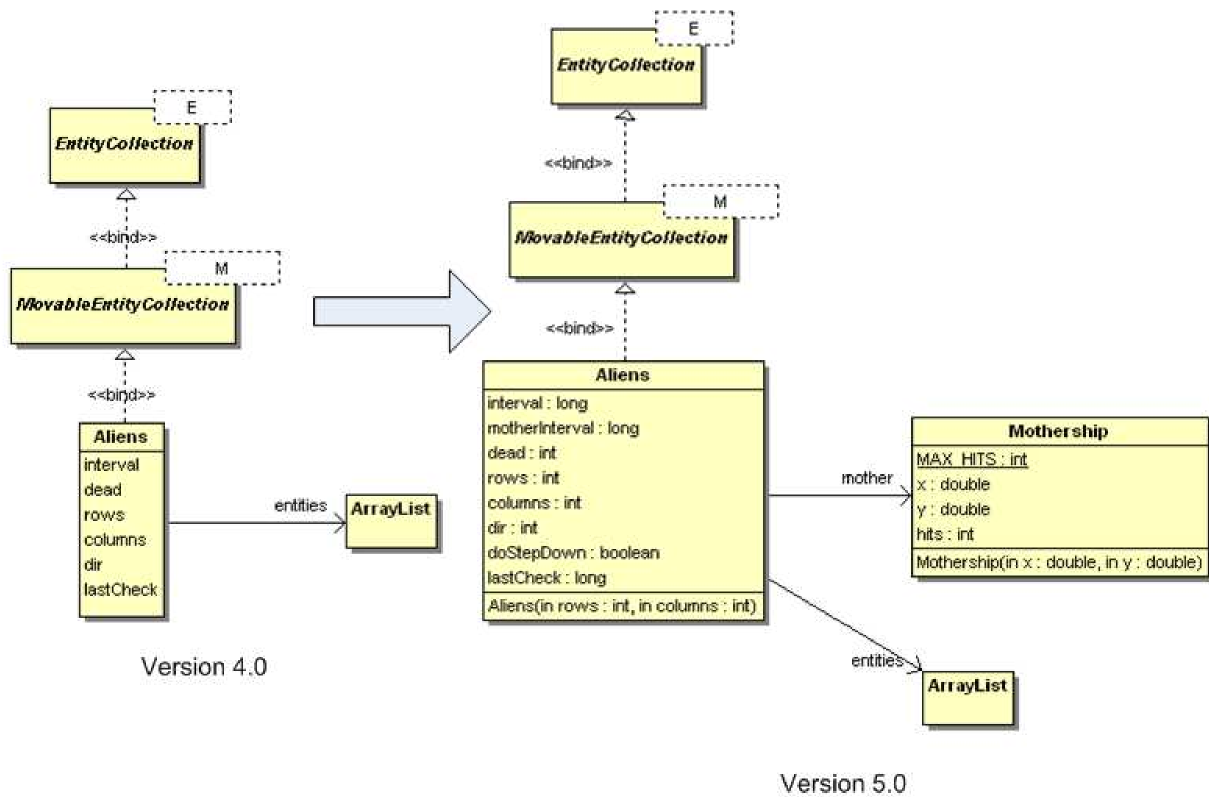


Figure 5.38 - Modifications on collections hierarchy regarding the introduction of new classes of enemies (v4.0->v5.0)

Class **`Aliens`** still manage in version 5 the collection of aliens by creating within its constructor the set of aliens needed for the game. A reference to a **`Mothership`** instance is also added and initialized in the **`Aliens`** class using default constructor, which create a **`Mothership`** by placing it in the default initial position. The constructor of **`Aliens`**'s class in the latter release creates a matrix of aliens with tough aliens on the top two rows and with normal aliens in the other rows. It also removes the middle column in order to make space for the laser of the mothership.

Push down method (interface) refactoring

In release 4 code the Entity class implements **`Collidable`** interface. However not every subclass of **`Entity`**, like **`Explosion`** class, really need to implement that interface, because instances of those subclasses do not participate in collisions. We decided to move the implementation of interface **`Collidable`** on subclasses instead of keeping it on the root class.

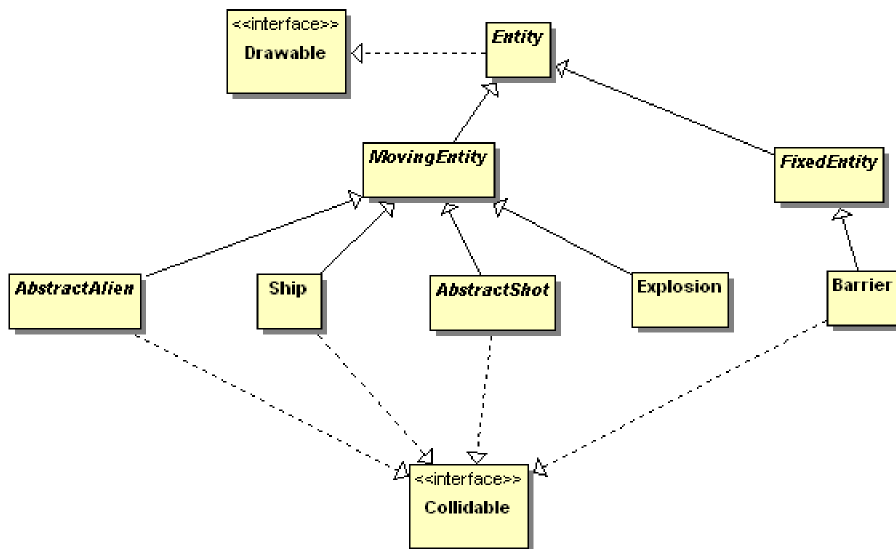


Figure 5.39 - Move implementation of interface Collidable down in the hierarchy (v5.0)

As it is possible to see from Figure 5.39, the implementation of **`Collidable`** is only used for subclasses whose instances participate in collisions. The **`Entity`** class implements only the **`Drawable`** interface.

Pushing down an interface's implementation means pushing down methods. We saw in previous iterations that this kind of modification doesn't introduce any issue or unexpected behavior with dynamic updates as we are not changing the semantic of any method neither moving state across the hierarchy.

Changing the implementation of move method

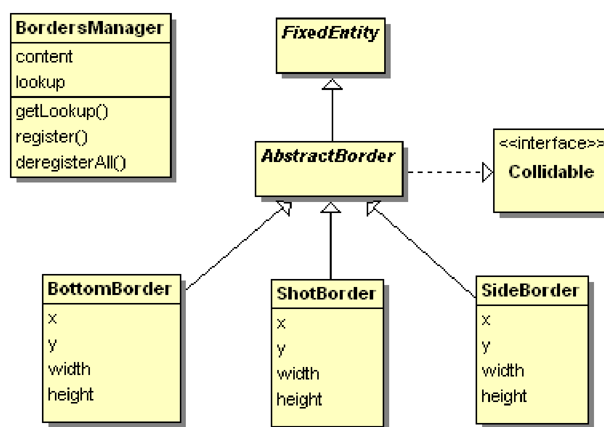
`Move` method, declared in **`MovingEntity`** class and defined in subclasses of **`MovingEntity`**, in release 4 performs the movement of the entity and checks for out of bounds condition. That implementation introduced a dependency between the class that implements the ***`move`*** method and the **`Game`** class that holds information on game field borders.

In version 5 we wanted to modify that method so that it only manages the movement of the object without caring about out of bound conditions. Pragmatically, we want to change what that method does, altering its implementation and giving the collision system the responsibility to check for out of bounds, effectively removing the dependency with **`Game`** class.

We first removed the ***`move`*** method implementation in every subclass of **`MovingEntity`** and left the implementation defined in **`MovingEntity`** class. In previous version we had different implementations of ***`move`*** method on **`MovingEntity`**'s subclasses because each different class had different bounds.

The return type of ***move*** method was changed from *integer* to *void*, changing the signature of the method. With this modification Javeleon will identify the method as a new method; therefore the program after the dynamic update will use the implementation of the old method when a move method call with the integer return type is encountered.

In order to check the out of bound condition for every entity we decided to exploit the collision engine adding invisible borders to the game and checking collisions between the borders and the game entities. The solution proved the scalability of the collision system with dynamic updates, attesting the correctness of the design choices made with the implementation of the collision framework.



Code Snippet 5.13 - Game border classes (v5.0)

We created new classes for borders and a class **BordersManager** that collect all the instances of borders in one place. **Game** class maintains a reference to an instance of **BordersManager** and uses inline initialization to initialize and register instances of borders, using the parameterized constructor of **AbstractBorder**'s subclasses as shown in Code Snippet 5.14.

```
public final class Game extends JComponent {
    […]

    private BordersManager gameBordersMgr = new BordersManager();
    private BordersManager outsideBordersMgr = new BordersManager();
    private AbstractBorder bBottom = gameBordersMgr.register(new BottomBorder(0,
RES_HEIGHT - BORDER_DOWN, RES_WIDTH, BORDER_DOWN));
    private AbstractBorder bLeft = gameBordersMgr.register(new SideBorder(0, 0,
BORDER_LEFT, RES_HEIGHT));
    private AbstractBorder bRight = gameBordersMgr.register(new SideBorder(RES_WIDTH -
BORDER_RIGHT, 0, BORDER_RIGHT, RES_HEIGHT));
    private AbstractBorder obTop = outsideBordersMgr.register(new ShotBorder(0, -100,
RES_WIDTH, 20));
```

```
    private AbstractBorder obBottom = outsideBordersMgr.register(new ShotBorder(0,
RES_HEIGHT + 100, RES_WIDTH, 20));

   […]
}
```

Code Snippet 5.14 - Initialization of borders in Game class

New collision handlers were defined as inner classes of `Game` for collisions between entities and borders, defining the logic that has to be executed when an entity goes out of bound.

5.6.2 Update test issues and solutions

New alien types

The update test phase showed a transitory period regarding the modifications made to the alien types. If the update occurs while the player is playing a game the mothership instantly appear on its default initial position ignoring the position of the matrix of aliens. Moreover the matrix of aliens doesn't change instantly during the game because the initialization of that matrix is done within the constructor of Aliens class that is called only at the start of the game. The player has therefore to wait for the start of the new game in order to see the new layout for the aliens.

The problem for this issue is caused by state and representation of information with our application architecture. An instant change of the layout of the aliens was not possible in this case because of the design choices that were made in the past. In fact, by introducing the mothership we have no means to recover the correct position where to place it. Every `Alien` instance has its own position initialized at the start of the game. However the position of an alien is data that derive from other data, like the position and the dimension of the matrix, the relative distance between contiguous aliens, etc. Having the information on the position of each entity, instead of recovering each time that data, means caching.

Moreover our architecture does not allow dynamic change of class type for aliens so we cannot change the behavior of a simple alien to a tough alien with the dynamic update.

Push down method refactoring

The refactoring that affected the implementation of `Collidable` interface on several classes didn't cause issues or strange application behaviors. That proved another time that moving methods across a class hierarchy, without changing the semantic of those methods, is an update compliant modification for Javeleon.

Changing the implementation of move method

The changes that regarded the move method practically removed the old ***move*** method and added a new one. In fact the signature of that method was modified because the return type was changed.

If the dynamic update occurs while a method that calls the old ***move*** function is active, function calls to ***move*** are forwarded to the old implementation, effectively moving entities and checking out-of-bound conditions, like in release 4.0. In other cases, when a new ***move*** method call is found, that method call is forwarded to the new implementation; the position of the entities will be checked right after having moved all the entities so no unexpected behavior or faulty conditions are generated.

Chapter 6 - Good practices for developing dynamically updateable applications with Javeleon

In this chapter we review the findings that were discovered during the development of the case study application. This chapter is divided into two parts:

In the first section we make some considerations regarding how dynamic updates impact on software evolution, what methodologies should be followed in order to keep track of changes and what tools the developer can use in order to manage dynamic updateability of code.

In the second section we draw up a list of good practices inherent to application design and coding choices by reviewing code changes related issues that we encountered in our case study.

6.1 Software evolution considerations

In this chapter we point out some considerations regarding the software development without taking into consideration problems that came out from specific code changes discovered during the dynamic update test. The developer can find benefit from the use of some of these practices even when not working with dynamic updates.

6.1.1 Management of code

Having to deal with a software project that develops with several releases pose the problem of code management. As we said in chapter 3.2, we chose to use SVN as the revision control system and the branching system to separate code of each iterative release.

The use of branching within the revision control system is greatly recommended for complex projects that include dynamic updates because it allows switching with ease from one version to another without dealing directly with revision numbers. Our methodology for dynamic updating states that the on-line program change should be tested only considering head revision of branches, which are the latest code versions of each branch. Following this methodology and thanks to the use of branches, whenever we had to test a dynamic update between two versions, we had to start the program using the head of a certain branch, then switch to the head of another branch and reload the program. That scheme really simplifies the whole update test phase allowing fast application test and debugging.

Considering the management of code we provided for each revision committed to the repository supplementary information regarding the nature of the code changes included in that revision. We used the categorization provided by Lientz and Swanson in (Bennet & Swanson, 1980) about 4 kind of maintenance:

- Corrective maintenance: reactive modification of a software product performed after delivery to correct discovered problems;
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability;
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

This categorization helped to understand the relationship between types of modifications and issues that can come up after dynamic updates. We found out that there is no relation between the type of the modification and the probability that that modification will cause issues or unexpected behavior with the dynamic update. Refactorings, provided the definition in (Fowler, 2000), should not alter the external behavior of the application, however we have proven in several cases that some refactorings can cause a loss of state and let the application manifest an unwanted behavior or, in the worst case, crash with exceptions. Therefore the use of this categorization is, in our opinion, not necessary or helpful for identifying issues with changes.

In most of the cases we found out that in order to easily identify changes to the code and, most importantly, changes to the code that are known to be source of issues with dynamic updates, the only things that the developer has to keep track of are changes to state and changes to the semantic of methods. These two domains were the two main sources of problems in our case study.

Instead of keeping track of changes in those two domains we wrote down a journal of changes to the code, which can be automatically generated taking the information present in the repository commit messages. An example of that journal is shown in Figure 6.1. As it is explained in the next subsection regarding the dynamic update approach, the usefulness of that document is inversely proportional to the average number of modifications made to the code on each commit. While this document turned out to be helpful for identifying changes in a certain point of the development it was completely useless for getting the whole idea of how the software artifact evolved from iterative release to iterative release. In order to have

a clear high level idea of the software evolution we used the software evolution schema. In that diagram we can clearly see the features that are implemented in each release, the modifications that caused issues with dynamic updates, refactorings and application redesigns, cascade modifications.

Revision number:	Version number:	Modifications:
1-10	0	First tests on Javeleon
11	1.0	Start development of iterative release 1
12	1.0	Implemented feature 1 on
13	2.0	Start development of iterative release 2
14	2.0	Implemented feature 1 on version 2
15	2.0	Cascade modification on version 2.0, implementing new design in order to let the feature 1 work with dynamic update
16	1.0	Cascade modification on version 1.0, implementing new design in order to let the feature 1 work with dynamic update
17	2.0	Make refactoring X to the code that makes the code looks better
18	1.0	Make refactoring X to the code that makes the code looks better

Colour	Meaning
(green)	Modification to previous iterative releases (not mandatory for the application to work correctly)
(grey)	Undocumented revisions
(yellow)	Marker for the start of the new development release
(red)	Critical cascade modification
(white)	Modification to the latter release in order to implement new features, fixing bugs, or make refactorings

Figure 6.1 - Revisions journal example

6.1.2 Dynamic update approach

In our line of work the methodology used for testing dynamic updates demanded to test on-line program changes only at the end iterative release coding phase. That means that the developer has to implement all the features, bug fixes and refactorings requested for the current iteration, before testing the update with the previous iterative release. The dynamic update test will then reveal issues related to changes to the code or design choices inherent to the coding phase of the previous and of the currently developed iterative release. These issues will be resolved by modifying the code of the latter iterative release or resorting to cascade modifications.

Most of the iterations prove that working by testing updates this way is not a good way of dealing with dynamic updates. If a version brings many new functionalities, bug fixes and refactorings to the code all in once some of them will be update-compliant and some will not, causing the application to crash or behave unexpectedly. If only one modification is not update-compliant hence the dynamic update cannot be considered as valid. In that case the developer has to track the code changes that make that version not dynamically updateable and call off those modifications. This latter process is affected by the number of modifications that an iterative undergoes, the way these modifications are applied to the code and the quality of the design.

When lots of modifications are made to an iterative release, like in version 4.0 of our case study application, and one of them results non update-compliant, the developer has to track down in the revisions journal the revision where he applied that particular change to the code and revert to that code version. Reverting the branch to a previous revision means losing at the same time all the changes to the code that were made from that revision to the head of the branch. This kind of situation can slow down the development and dynamic update test phase considerably. Therefore as a general approach we state that is better to test dynamic updates with micro code changes and solve issues immediately. The latter approach is called stepwise refinement and is followed by Mannocci in (Mannocci, 2010).

Moreover there are some good practices that should be kept in mind when managing the code within the code repository.

Some considerations regard the temporal locality of changes, which is the information on the degree of how a code change is applied relatively to time that can be measured with revision numbers. When it comes for the developer to implement a set of changes, it is always a good idea for him to have a clear view of the modifications he wants to apply. Before starting to code, a good principle is to separate as much as possible these code changes semantically. For instance, we can state that one code change should implement a certain feature, another one can imply the implementation of a certain refactoring to the code, etc. When all the code changes to implement are semantically divided it is better to not join the implementation of several modifications in the same revision. Moreover it is preferable to not spread one modification across several revisions during the iteration. These practices affect the readability of the revision journal and help the developer to identify where a particular modification to the code was done.

Another domain of interest is about the spatial locality of changes. A change to the code that enables a new functionality into the program can affect several classes, when the application architecture comes with many classes interdependencies. At the class level the *Open-Closed Principle (OCP)*, advocated by Martin in

(Martin R. C., 1996), states that software entities should be open for extension, but closed for modifications. When a single change to a software entity results in a modification of other entities the program code shows a bad design. Modules should be designed in a way so that they do not change. When a new requirement is given, this requirement should be implemented by adding code rather than modifying code that already exists and works. Proper use of abstractions and inheritance support this principle as stated in the work of Liskov (Liskov, Data Abstraction and Hierarchy, 1987): "Abstraction when supported by specifications and encapsulation provides locality". The OCP allows the developer to easily implement and identify code changes and "yields the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability". At a higher level, the package level, we have the *Common Closure Principle* (CCP) (Martin R. C., Granularity, 1996) : "Classes in a package should be closed together against the same kind of changes. A change that affects a package affects all the classes in that package". Classes that collaborate with each other and that are part of a reusable abstraction should be grouped together in a package to achieve the CCP. Moreover, packages can be seen as modules that have dependencies with each other. It is important that the graph of dependencies between packages to be an acyclic directed graph. In fact, in the latter case, we prevent changes to a package to cause cascade modifications on all the other packages. The NetBeans Platform allows developers to write programs by providing NetBeans modules, which can be seen as plugins for the NetBeans Rich Client Platform. The developer can choose to deliver each functionality of his application with a distinct module. A module represents a group of java classes that deliver functionality and offers a certain interface. The benefit that comes from using the module system provided by the NetBeans platform is about organization of code and reuse. In fact, only modules that have explicitly declared dependencies on each other are able to use code from each other's exposed packages. A programmer that wants to use a certain service of a module has only to know its API in order to use it, without worrying about the implementation. Javeleon works at the granularity of NetBeans modules, which means that when a reload action is called on a module all the classes of that module are reloaded.

6.2 Application design considerations

In this subchapter we synthesize our findings regarding solutions for issues regarding dynamic updates found during the development of the case study application.

6.2.1 Generic issues

During the development of the application the majority of problems that we encountered raised from two domains: state handling and semantic modifications.

The problem of state

According to the definition of state in (Gupta, Jalote, & Barua, 1996): “state is the complete characterization of a point in the lifetime of the process”. Whenever a dynamic update occurs and the program is updated to a different version it is often desirable to map the state of the application into new state valid for the updated program. That state should be reachable state for the new program. For example, the new program could need a new variable added in the updated version, some instances could be removed or field values modified using a specific algorithm, etc. Obviously, when a state transform function has to be written, a meaningful state transfer function can only be specified by the developer that has close knowledge of the two versions of the program. When no state mapping function is needed the state can be migrated as it is, applying the identity state mapping. Javeleon does not offer tools to the developer for writing state transform functions, it simply apply the identity state mapping on the existent state when an on-line program change is called. However Javeleon allows the programmer to add new state (primitive type fields or references) and initialize it safely with in-line initialization. For that reason we say that Javeleon is transparent to the programmer, because the programmer does not have to write specific dynamic update code neither has to know the underlay mechanisms that perform the update. Therefore the problem of creating state mapping function in Javeleon is translated into a problem of choosing the right design for our application. The developer should strive for design choices that allow the state mapping functionalities provided by Javeleon to be sufficient for enabling valid dynamic updates. Basically Javeleon emulates and enrich the offline schema “serialize-update-deserialize”; that means that most of the problems that can be encountered by modifying the design and updating the program with the offline schema will be also present with Javeleon. The effort of finding good designs for the sake of dynamic updateability often resulted on pointing the developer to apply design patterns and coding practices that are today well known and recommended in the field of object oriented programming. So, in a certain way, we can state that Javeleon educates the programmer to follow good programming practices in the field of Object Oriented Programming.

Introduction of state

Starting from introduction of state we examined several cases in our work. We classified introduction of state into three classes, each one poses a different problem and has different solution:

1. New state is known at coding time
2. A state mapping function can be inferred
3. A state mapping function cannot be inferred

New state is known at coding time

The first point refers to the case where new field is added to a class and the developer knows at coding time which is the value that should be assigned to that field. An example for this case is the introduction of barriers in version 2.0. In version 1.0 **`Game`** class didn't have a field for collection of barriers, as barriers weren't required in that version. In version 2.0 code we introduce a field for the collection of barriers in **`Game`** class, initializing it using in-line initialization with default constructor of **`Barriers`**. The constructor of **`Barriers`** collection creates the default set of barriers adding four **`Barrier`** instances. All the information that is needed for initializing the new objects and fields are all known at coding time so this kind of introduction of state is not problematic. The developer can add new objects references to a class definition or new fields and initialize them with in-line initialization safely.

A state mapping function can be inferred

New reference type fields

Problems arise when the new state that is introduced depends on information that is already present inside the logic of the game. Starting from the case of adding reference type fields consider the case of the introduction of the reference to **`Game`** instance in **`Aliens`** class in version 2.0. We introduced that dependency because in that version the aliens started to move and as the collision was handled inside of collections classes, the **`Aliens`** class needed a reference to the **`Game`** instance in order to access its interface. As it is possible to see from Figure 6.2 we passed the reference to the Game instance using the parameterized constructor. This approach turned out to be a bad design choice because we cannot call the constructor of **`Aliens`**, otherwise we lose **`Aliens`** instance's state that is present at the moment of the update by creating a brand new object. We cannot even use in-line initialization because the reference has to be passed to the **`Aliens`** instance somehow.

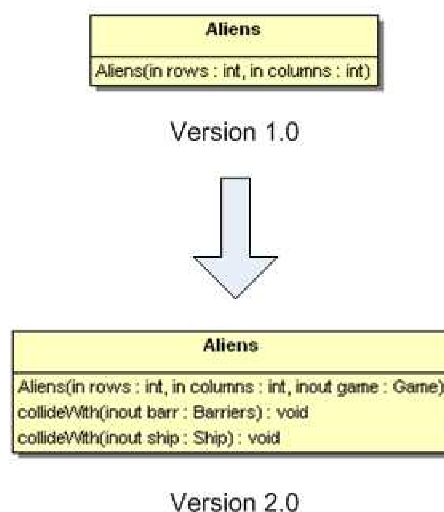


Figure 6.2 - Adding reference to Game in Aliens class - change to the constructor (v2.0)

A solution found for this problem consist on providing a global interface for accessing objects, in our case the `Game` instance. The presence of a global interface prevents the use of parameterized initialization at the cost of global visibility of objects. We provide two general solutions:

- Singleton pattern
- Lookup

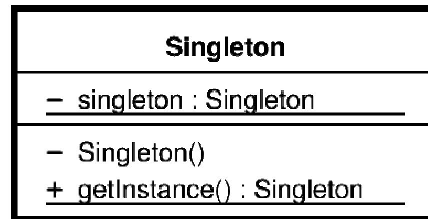


Figure 6.3 – Singleton design pattern

The singleton is a class that provides a static getter method that returns the unique instance of that class. The getter methods checks if the static singleton field already holds a reference to a Singleton object and, in positive case, returns that object. Otherwise the object is created using the private constructor and returned.

Implementing the singleton pattern within the `Game` class let the `Aliens` class getting rid of all the references to the `Game` instance. All the objects can get a reference to the singleton instance using the global method ***`getInstance`***. This pattern, however, is valid only for classes that we know to have an unique instance at runtime; when the singleton pattern is implemented for a class it is impossible to have two instances of that class. The main concept of the Singleton that helps us to solve our problems with reference type fields introduction is the presence of a global interface, so many variants can be derived from the singleton.

We have another scenario when the developer has to deal with collections of instances and he wants to be able to add new instances to a collection with dynamic updates. The solution in that case can make use of the NetBeans Lookup Library in turn to obtain decoupling. Whenever a new object has to be registered in the application logic the programmer can introduce a new reference field of that object. The in-line initialization uses a registration method of a class, which interface is globally visible, that register that object in a lookup context and consequently returns the object registered. We used this approach in order

to register collision handlers (Code Snippet 5.10) using the lookup context provided by class **GameControl** (Figure 6.4).

Figure 6.4 - GameControl class

The use of lookups and the registration mechanism introduce a new problem with Javeleon concerning the removal of objects. While it is really simple to extend the state registering instances to a lookup context there is no explicit mean for removing those instances with a dynamic update.

Primitive type fields

Up to now we have only examined the second case of introduction of state relatively to reference type fields. When primitive type fields are introduced into a class definition and the value of that variables depends on the value of the state that is already present at the moment of the dynamic update other issues arise. As Javeleon doesn't offer to the developer any mean to specify state transfer function for transforming old state into new state, design choices have to be taken in order to avoid inconsistency of state in the new version. An example for this scenario is the case of new fields `row` and `column` in **Alien** class, introduced in version 2.0. Inline initialization cannot be used in this situation because the developer cannot know the indexes of every instance of **Alien** class at coding time. In the first implementation those indexes were assigned to each instance of **Alien** within the constructor of **Aliens**. However this implementation turned out to be invalid because creating a new set of **Aliens** when the update occur means losing all the old state regarding that collection. We believe that in order to avoid this kind of issues an elaborate planning of state is needed.

State, or primitive type fields in general, represent information about the application domain that evolve with process execution. Whenever a new field is introduced in a class it can represent a group of information about that class instances. Consider, for example, the information about the position of aliens in our game. Starting from the first release we used integer fields to represent coordinates for every alien. However we didn't consider that the information on each alien position can be inferred from other data:

- Initial position of the block of aliens;
- Number of step down performed by the aliens block;

- Step length
- Horizontal offset of the matrix
- Row and column indexes of the alien inside of the aliens matrix
- Relative distance between contiguous aliens

So the information regarding the position of each alien can be inferred from a set of other bits of information. By decomposing each data into small bits, the developer can later decide which of them depends on software execution (state) and which are fixed during process life (constants). In every case the developer should encapsulate all those data using properties (getter/setter methods). Using properties instead of direct access to variables enable a level of indirection and data abstraction. Properties that represent constants will simply return a value while properties that represent state will hide the internal representation. Going back to our alien's example we could have used an implementation where the **`Alien`** class has a property for x and y coordinates and, in the properties implementation, it evaluates that information by means of all the small bits of data that are needed. So, instead of using a single field to represent data of interest we divide that information into little pieces and build the data every time it is needed. This implementation implies the minimization of caching, because the application has to recalculate composite data every time it's needed, and therefore cause a degradation of the application performance. The advantage that we get from this design involves better responsiveness of the application on showing the behavior of the new program with dynamic update. Static final fields are simply values substituted directly in program bytecode, therefore when an on-line program change is performed the values of constants can change, unlike state that in Javeleon is simply migrated as it is. The way of handling static final constants by the java compiler can pose problems when several java classes depend on a constant declared in another class. If only the class, which contains the static final field, gets recompiled with the module reload, the other class files will not notice the change and will maintain the previous in-lined field value. This problem can be solved by replacing the use of static final fields with class properties that return values. Even using the latter approach we will fall in inconsistencies between state and "constants".

Changing constants in order to change the behavior of the application however can bring to faulty conditions because of informations coupling. Consider the case related to the introduction of barrier's health in version 2.1. We decided to use an implementation that introduced a new integer field `hits` in **`Barrier`** class and a constant `MAX_HITS` to keep track of the maximum health of a barrier. Without dynamic updates we assume the assumption $hits < MAX_HITS$ to be always valid. With dynamic updates the constant `MAX_HITS` can change with the new updated program breaking the assumption. This is the

case where a change to a constant causes an invalidation of state. In an environment where state mapping function definition was possible we would have removed all those barriers whose number of hits exceeds the new value of MAX_HITS. In order to solve the problems related to the inconsistence between hits and MAX_HITS what the developer can do is to provide encapsulation for hits field and return correct value for this data relatively to the value of MAX_HITS, maintaining the assumption still valid. Moreover, it is also possible to implement directly the state mapping function inside of the property in order to modify the state when inconsistent values are found. However, even using this approach, it is impossible for the program to remove barriers that should be eliminated at the moment of the update. In fact the logic that cares about the removal of barrier instances from **Barriers** collection is called only when a **Barrier** is hit, therefore those barriers will remain anyhow on the screen participating in collisions until a new collision between barriers and a shot occurs.

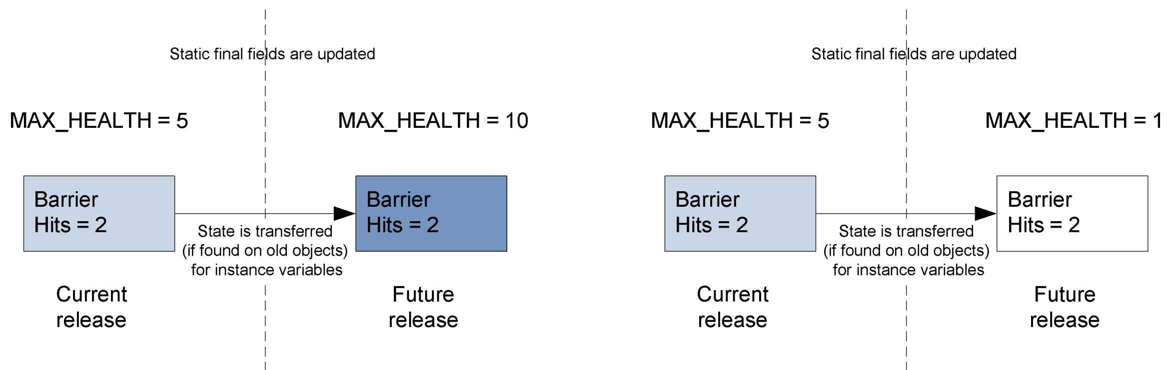


Figure 6.5 - Changing constants value - the barrier example

Information coupling can bring to unreachable state for the new updated program. When unreachable state is found but the process can reach a reachable state for the new program in a finite amount of time, the update is still considered to be valid, as stated in Table 4.1. However, during our work, we found two different kinds of information coupling. The first form is represented by the solution found for barriers' health of Figure 6.5. As we said, by using this solution, we separated the concepts of number of hits, strictly related to the process execution and the interaction with the user, and the max health of a barrier, which is constant during game execution. These two bits of information are still coupled by the assumption that the number of hits sustained by a barrier cannot exceed the max health. The second form of information coupling is strong coupling (or direct coupling). Consider the solution for the problem of barriers health represented in Figure 6.6.

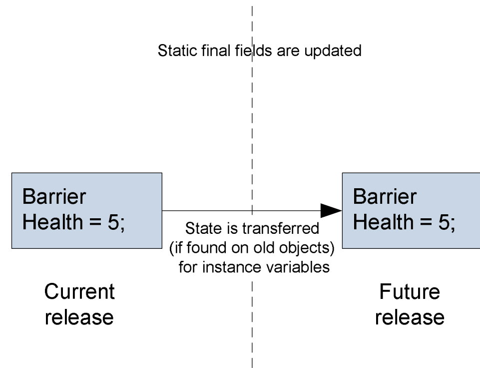


Figure 6.6 - Barrier's health solution with strong coupling of information

With the latter solution we use a variable to represent health, strongly coupling the two concepts into one variable. The variable is initialized to a certain value within the constructor of the barrier. As Javeleon uses the identity function to transfer state between old and new instances, it turns out to be impossible to change immediately with the update the maximum health of a barrier. This solution turns out to be worse than the first form of concepts coupling from a perspective of application behavior. When concepts are strongly coupled into state, the application loses the capability of reaching a valid state for the new program with dynamic updates. It will be impossible to change the behavior of barriers immediately, considering every possible change to the max number of hits. With the first form of information coupling we still have an assumption that bounds the two concepts, however it is possible for the process, assuming a certain set of modifications to the constant `MAX_HITS`, to reach a valid state for the new program right after the update. If the number of maximum hits is incremented, the state of the application right after the update is considered to be valid for the new program code. This degree of application reactivity towards the update on showing the new behavior is not achievable with the solution that involves direct coupling. In fact, we can state that the use of strong coupling implies the presence of non-valid state when a concept, on which the coupling is built, changes within a dynamic update. This can eventually bring to a transition period, which makes the update still valid, or to the application's crash.

Considering a more general perspective we can state that the problems that come from state are due to relationships between concepts and their implementation in our program code. When two concepts are strongly coupled (direct coupling) it seems to be impossible to obtain a valid state for the new program right after the dynamic update. Examples of direct coupling can be caused by caching, by duplicating data on memory (copying variables from a class to another) or by holding references of objects inside of classes. Caching is a technique that aims at improving the performance of the application exploiting an internal perspective on the code. With caching some data is retrieved and stored in state, on which the application can work faster, without evaluating if some of the concepts behind where changed. A class that retrieves

images from the hard drive and store a copy of each image in memory can be seen as cache. When the images are loaded in memory and a dynamic update change the image files, the cache will continue to retain old images. The state will be inconsistent and the assumption that the cache reflects the image files will be broken. Therefore it is clear that the application will not show immediately the new program behavior after the dynamic update because it will rely on cache, which is state copied as it is at the end of former program execution.

When the relationship between concepts is loosened in the program code, the assumption that bounds the concepts still remains. This is the case of indirect coupling. With the latter approach we enable our application to present a valid state for the new program right after the update, assuming a certain set of changes to the concepts, which are bounded by an assumption. If the change to the concepts doesn't break the assumption the application will show valid state and correct behavior, according to the new program, right after the dynamic update.

Some relationship between concepts can be subtle. Relatively to the case of barriers' health we showed that, using indirect coupling for hits and maximum health, we can break the assumption by decrementing the maximum health to a value that is less than the value of hits on an instance of **`Barrier`**. Therefore that barrier will remain in game until hit again by a shot. This strange behavior is caused by breaking the assumption which states that when a **`Barrier`** instance has its number of hits greater than the number of max hits, it should not be present in the collection of barriers.

Indirect coupling allows the application to show better reactivity on showing the new program behavior right after the update. If the application design allows the state of the process to be consistent with new program execution we fall into the first definition of valid dynamic update according to Table 4.1. This result is never achievable when direct coupling is present. Using indirect coupling however has some weak spots. When a lot of concepts are bounded together by an assumption it can be difficult to evaluate if the assumption is broken when one or more concepts change. Moreover, it will be easier for an assumption to be broken when it depends on many concepts. Additionally, eliminating references in classes or direct copy of variables between instances of classes, while using loosely coupled communication, introduce many interdependencies between classes. If we consider the example, given in this chapter, related to a different representation of aliens' position, we can see that a change on the step length can bring the aliens to be out of the lower game's bounds. Moreover, in order to let the aliens instances evaluate their positions, we have to create dependencies between the **`Alien`** class and the classes that hold the bits of information required for that computation.

State mapping function cannot be inferred

The third scenario regarding introduction of state is the most problematic. An example that can give the dimension of the problem is the one relative to the introduction of score and time bonus in version 3.0. In that version we introduced the **`Score`** class. That class keeps hold of the information when the game started in order to evaluate, at the successful end of the game, the time bonus to assign to the player. The field that keeps track of that information is simply a long primitive field initialized by the method ***`startTimer`*** which is called by the **`Game`** class when a new game starts.

If the update occurs during a game, there is no state mapping function that can retrieve the correct timestamp relative to the starting instant of the game, which happened while executing the former version. A solution could be to initialize the variable with the timestamp of the instant when the update occurred, but in that case the bonus points will be overestimated. In the worst case the overestimation of the player score will remain on the high score table on the hard disk permanently, influencing future execution of the application with high score persistence. Another solution, which is the solution that we suggest, could be to leave the field uninitialized, create a property to encapsulate the field and deal with invalid values within the property implementation.

State and hierarchies

We proved in the first releases of the game that moving state from a class to another can bring to a loss of state, which should generally be avoided. In order to not move state we defined an architectural design for hierarchies of classes, which make use of abstractions and the "template method" design pattern, exploiting encapsulation and inheritance. Our solution is influenced by the guidelines found in (Snyder, 1986), (Liskov, Data Abstraction and Hierarchy, 1987) and (Gamma, Helm, Johnson, & Vlissides, 1995).

The idea behind the design is to have the definition of the behavior in abstract classes that forms the skeleton of the hierarchy, and, at the last level of the class tree, the definition of the implementation details. That means that for every concept that should be represented with a class an abstract class defining the behavior of that concept should be present. Moreover the implementation, which means the data structures, used for that concept should be placed on a concrete subclass of the classes that define the behavior, hiding the implementation details by providing properties. The "Template Method" design pattern allows to "define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure". In our case the algorithm is defined in abstract classes' methods which defer the part of the algorithm concerning the access to data to concrete subclasses.

By using this design the developer can add new abstract classes in the hierarchy, new concrete classes that brings new state. Moreover is it possible to take advantage of inheritance by defining default behavior for a set of concepts by implementing that behavior up in the hierarchy. Any subclass that will need a specific behavior can still define its own by implementing it in subclasses. For example, considering Figure 6.7, the **`Entity`** class define a standard implementation of the ***`draw`*** method that draws simple geometrical shapes using methods ***`getPoly`*** and ***`getColor`*** to retrieve the specific shape and color for an entity. The Ship class redefines the draw method implementation drawing a particular raster image for the **`Ship`**. Whenever a call to method ***`draw`*** is used on a reference of type **`Entity`** that points to a **`Ship`** entity, that call will be forwarded to the **`Ship`** class' ***`draw`*** implementation, following the Liskov Substitution Principle.

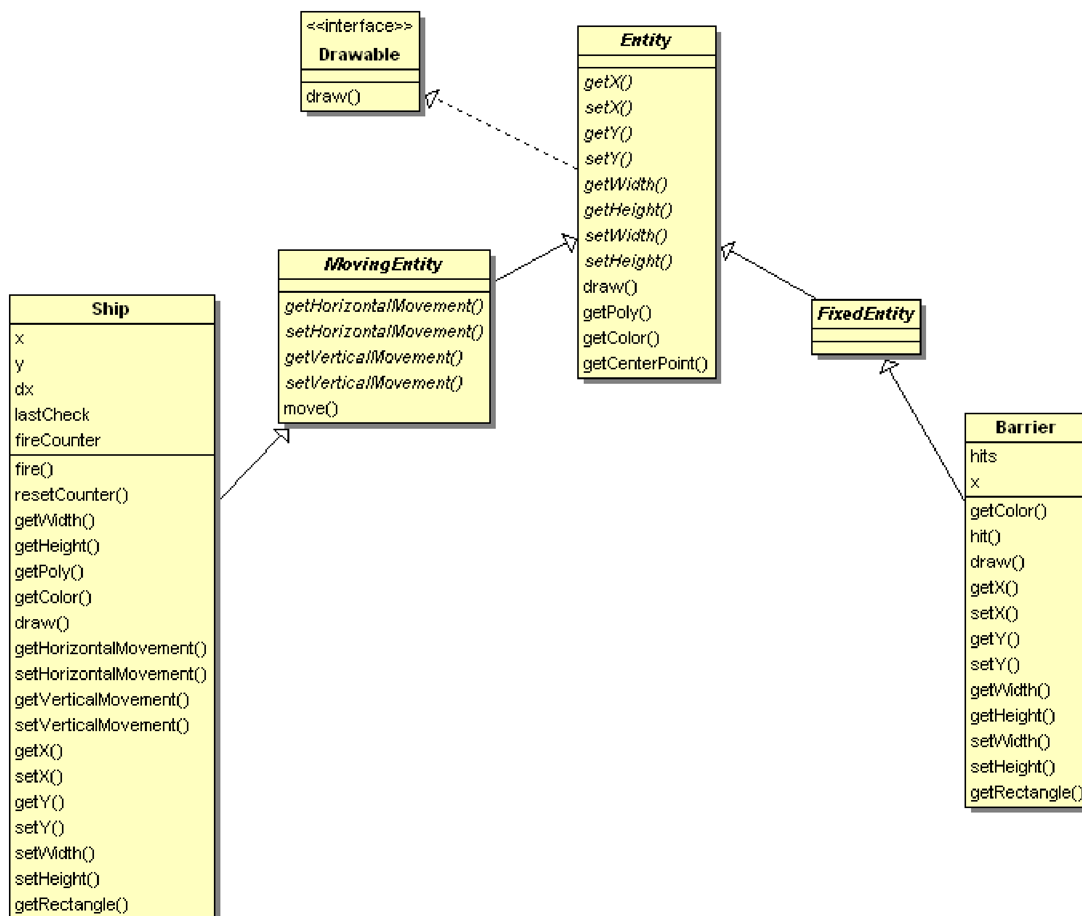


Figure 6.7 – Implementation of good design in our case study

This design however still has drawbacks regarding its evolution. In version 4.0 we decided to split the generic concept of shot, which was represented by class **`Shot`** in former versions in two different concepts: shots fired by the ship and shots fired by the aliens. That turned out to be a semantic modification of classes. In this case we introduced a new abstract level represented by the abstract class **`AbstractShot`**

and, in order to maintain state for already existent shots, we used the former class Shot that contained the implementation of generic shots as the implementation class for shots fired by the ship. Moreover we created a new implementation class, which is a subclass of `AbstractShot` along with `Shot` class, for alien shots. In Figure 5.34 - Differences between v3.1 and v4.0 on shots" is shown the modification to the `Shot` hierarchy and collections. Performing semantic modifications on concrete subclasses imply an implicit transformation of state with this kind of modification. In fact, after the update, the shots that were representing shots fired by the alien became to all intent and purposes shots fired by the ship. Therefore changing the concept represented by a concrete class resulted in unexpected behavior that, however, in our case, expires in a finite amount of time. This issue showed that it is impossible to turn instances of one class into instances of several classes with a dynamic update. However, even if we used a "serialize, update and deserialize" schema we would have obtained the same problem. Javeleon virtually implements and enriches that schema by hiding all the logic from the developer. This kind of problem can only be handled by including an intermediate update or by letting the developer implement design choices that allows this kind of behavior modification.

Variables life span

Let's now consider the problem of drawing elements on screen. In our case study we decided to let all the entities define within a draw method what is their representation. Whenever the game logic has to draw the game elements on screen it calls the draw method on each entity passing the graphic context as a parameter. That action is done every game cycle and practically refreshes the whole screen. An approach counts more on performance can draw only a subset of the elements on the screen; for instance it can draw only the elements that changed their representation since the last cycle. Mannocci followed this approach in his case study (Mannocci, 2010); the differences between the two ways of dealing with drawing shown a peculiarity regarding variables and objects life span relatively to dynamic updates.

The buffer where the graphics are drawn is considered to be cache in a scenario that involves dynamic updates. That is true because, whenever a dynamic update occur, the representation of entities, hardwired directly in the code as a series of statements that draw elements on screen, can change. While the methods that define the representation of entities change with on-line program modifications, the cache, represented by the graphic buffer, remains the same, as it is migrated by Javeleon to the new state as it is. In order to achieve better responsiveness of the application on showing the correct behavior after the update we have to eliminate caching or, at best, making sure that cache is updated as frequently as possible. Natural consequences of the latter statement involve performances considerations. More the

developer wants his application to have responsive dynamic behavior with dynamic updates, the more he has to degrade the performance of his application by removing caching.

Program execution issues

State is not the only matter when dealing with Javeleon. Considering the update schema of Javeleon, whenever a dynamic update occurs, all the threads that were executing a method continue to execute that method implementation until the end. Unless the method that is being executed is a private method, method calls found within that method implementation can be forwarded to new method implementations if these are provided with the new program version. This approach poses several problems when semantic modifications are done to methods.

In our case study we didn't encountered many problems related to this field because we relied on strong specification of methods. We therefore avoided changing methods semantics with program evolution. However we can show a simple example where semantic modifications are performed and what are the consequences on statements execution.

Changing the implementation of methods

Changing method implementation or moving statements from a method to another can be tricky. Here's an example:

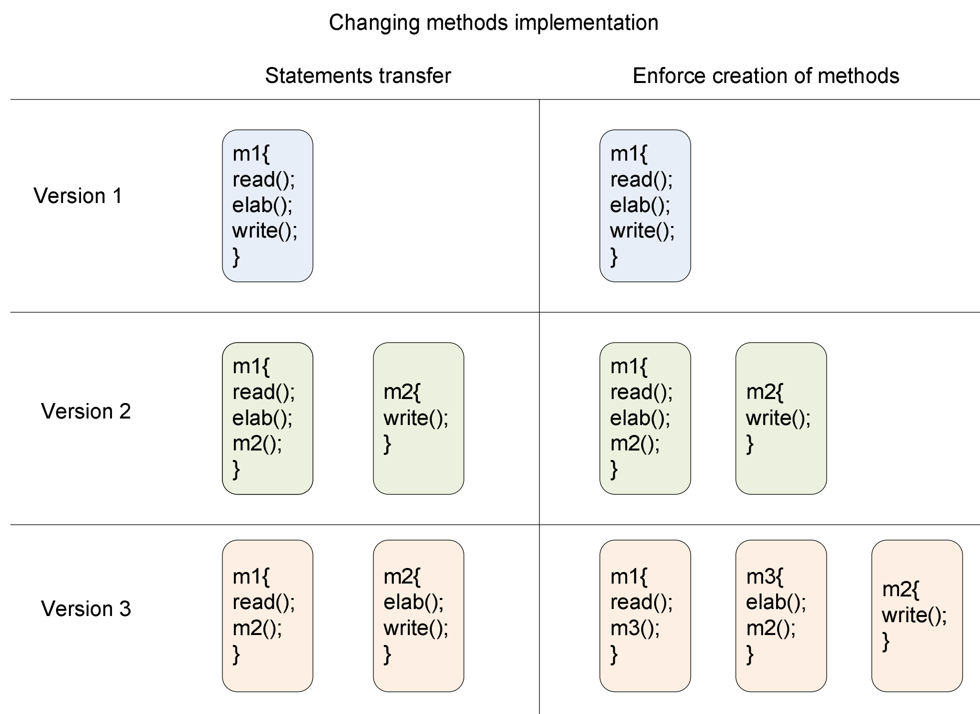


Figure 6.8 – Changing the implementation of methods

In the example we have a procedure that does three simple tasks. Consider now the changes of the first column “Statements transfer”. From the first to the second release, we moved the write statements group into another method, without changing the specification of method ***m1***. By doing this, we also created a new method with its own specification. Passing to version 3, we modified both methods moving the ***elab*** statements group to ***m2***. This modification turns out to be a semantic change to method ***m2***; in fact we changed the specification of that method.

Now consider the execution example reported in Figure 6.9. If the update from version 2 to 3 occurs while executing ***m1*** code, before the ***m2*** method call, the process will execute the ***elab*** statements group two times. If the dynamic update is done from version 3 to version 2 the ***elab*** statements group is not executed at all. That happens because we changed the specification of method ***m2*** passing from version 2 to version 3. The update from version 1 to version 2 doesn’t introduce any issues with program execution both in upgrade and downgrade case.

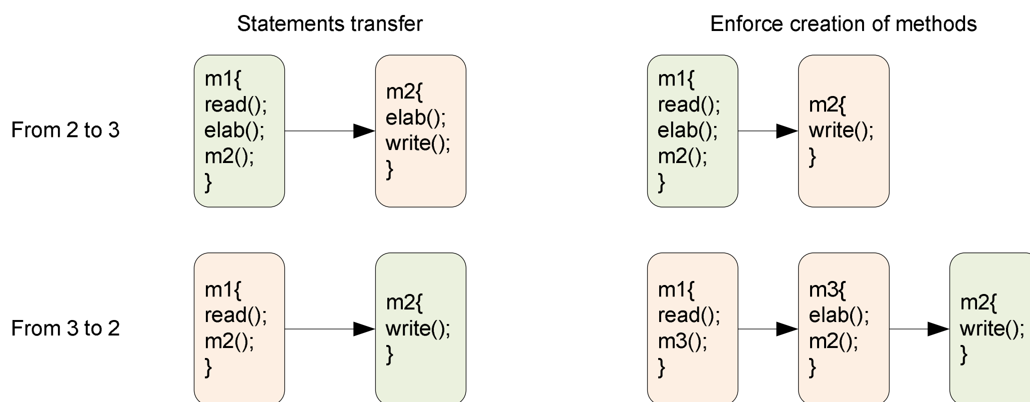


Figure 6.9 - Program execution example for the "Extract and Inline method" refactoring

In order to avoid the change of specification of a method we could use the solution reported in the second column of Figure 6.8. The execution example is also reported in Figure 6.9.

In order to avoid problems with methods execution the developer should first define a strong specification of the methods.

6.2.2 Specific issues

Infinite loops

Recalling what we said about Javeleon’s active methods update scheme, there are some considerations that should be made about methods that contain long-lived loops. In case the update occurs while a thread is executing a method, we consider that method to be active. The thread that is executing an active method

will continue to execute the instructions of the old method till that method becomes inactive. This can result in an inconsistency in case of an infinite loop or a long lived loop because the application will continue to run old code for a long interval, ignoring the new implementation.

Consider the case shown in the code snippet below. The function ***aMethod*** contains an infinite while cycle. If a thread is executing that method it will continue to execute the old implementation of that method regardless of dynamic updates of that method's body.

```
public void aMethod(){
     While(true){
     //Do something
     […]
     }
}
```

Code Snippet 6.1 - A method that contains an infinite loop

In order to solve the problem with updateability of these methods we should find a solution that allows the thread to exit from method's execution every time a cycle ends. We provided a solution for this issue that makes use of **TimerTask** class in order to emulate the behavior of a while loop. The solution is given in

```
public class myClass {

    private Timer loopTimer;
    public static final long LOOP_INTERVAL = 10;

    private void aMethodLoopBody() {
      //Do something
      […]
    }

    public void startLoop() {
     loopTimer = new Timer();
     loopTimer.scheduleAtFixedRate(new LoopRunner(), 0, LOOP_INTERVAL);
    }

    public void endLoop() {
     loopTimer.cancel();
    }

    private class LoopRunner extends TimerTask {
     @Override
     public void run() {
         aMethodLoopBody();
   }
}
```

Code Snippet 6.2 - Solution provided for the infinite loop problem

Classes **java.util.Timer** and **java.util.TimerTask** can be used for scheduling periodic tasks. We define a class that extends **TimerTask** and define the run method to run the former ***aMethod*** loop body statements. Whenever the loop has to start, the ***startLoop*** method is called scheduling the task for fixed rate execution. A **LoopRunner TimerTask** will be scheduled every LOOP_INTERVAL. According to the Java 2 Platform API specification "*Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes*". When the ***endLoop*** method is called the currently executed task terminates gracefully and, at that point, the loop is considered to be ended.

With fixed-delay execution scheduling, subsequent executions of the task take place at approximately regular intervals, separated by the specified period. Fixed-delay execution is appropriate for recurring activities that require "smoothness." In other words, it is appropriate for activities where it is more important to keep the frequency accurate in the short run than in the long run. This solution was appropriate in our case but should not be considered as a general solution for the infinite loop problem.

Twingleton problem

The Twingleton problem is an issue that affects serialization of TopComponent state and singletons on NetBeans platform 6.7. The TopComponent is the embeddable visual component on top of which the developer can build his application based using the windows API of NetBeans platform. When the developer starts to write an application using NetBeans' Rich Client Platform a wizard in the editor creates automatically a TopComponent subclass using default template. NetBeans platform takes care of serializing the unique instance of the TopComponent to disk at the end of the execution of the application and deserialize it at startup by redefining the methods of the Externalizable interface, providing custom serialization. The implementation provided with the template in platform 6.7 turned out to destroy the singleton pattern when windows that are attached to the TopComponent implement that pattern. Practically, when the application is started more than once, two instances of singletons are present.

In order to resolve this problem the developer has to provide an alternative implementation of method ***readProperties***.

```
Object readProperties(java.util.Properties p) {
      if ( instance == null ) {
          instance = this;
      }
```

```
        instance.readPropertiesImpl(p);
        return instance;
}
```

Code Snippet 6.3 - Twingleton issue's solution on TopComponent class

The implementation provided in version 6.8 RC2 is corrected and doesn't present the Twingleton issue.

Concrete classes cannot turn into abstract

Adding the abstract modifier to a concrete class definition always results in an exception casted at runtime when the dynamic update occurs. Consider the change represented in the listing below.

```
//Version 1
public class myClass{
        […]
}
------------------------------------
//Version 2
Public abstract class myClass{
        […]
}
```

Code Snippet 6.4- Adding abstract modifier to class

When the new module is reloaded Javeleon tries to create for each instance of myClass a new instance according to the new definition. This obviously turns out in an exception casted at runtime, as abstract classes cannot be instantiated.

This limitation poses constraints on naming. Consider the change we made in version 5.0 regarding the introduction of new kind of aliens. In order to introduce that feature we had to modify the class hierarchy regarding aliens by adding a new abstract level (Figure 5.37 – Modifications on entities hierarchy regarding the introduction of new classes of enemies (v4.0->v5.0)). In order to preserve state instances of **`Alien`**'s class we had to keep that class concrete without changing its complete name. Moreover we introduced the **`AbstractAlien`** class to represent the abstract behavior of the general alien.

Elimination of coupling between instances

In version 4.0 we tried to implement the Model-View-Controller pattern in our application design in order to separate domain logic from representation and control. We changed the superclass of class **`Game`** from **`JComponent`** to **`AbstractSubject`**, a new class introduced with the purpose of implementing the "Observer" design pattern. **`Game`** class in the former program code was a subclass of **`JComponent`**; the issue in this case was not related with the "change of superclass" refactoring itself, but rather with the coupling between the **`TopComponent`** and the **`Game`** class. At the application start the **`TopComponent`**

constructor is called and the `Game` instance is created and registered within the `TopComponent`. During the execution, the `TopComponent` instance calls method on its registered components, so, after the update,whenever the `TopComponent` tries to call a method of `JComponent` interface on the `Game` instance (which changed superclass), it tries to cast the `Game` instance to a `JComponent` causing a cast exception.

This problem is common in application design where there is strong coupling between entities derived by registration mechanism like in the “Observer” or “Composite” design patterns. These patterns should be avoided and loosely-coupled communications should be used instead. However this problem can arise when those patterns are applied in immutable classes, which are classes that are not affected by bytecode modifications of Javeleon.

Chapter 7 - Conclusions

Dynamic Software Updating systems represent a viable and cost-effective solution for updating applications that have to stay always on-line. Their fruition goes hand in hand with the advancement of agile processes for software development, in fact both their evolution is driven by the rising importance of requirements volatility. Javeleon is a software DSU's that allows easy integration in the software development process thanks to its transparency and flexibility of changes.

During the development of our case study application we proved several features of Javeleon:

- Easy integration in the development process thanks to the capability of modifying the code while the application is running, allowing fast dynamic update test and debugging. In fact Javeleon allows an immediate feedback on modifications that are made to the code;
- Greater flexibility regarding the set of possible modifications to the code. Compared to other DSU, Javeleon allows full redefinition of classes and modifications to the type hierarchy. Updates are performed at the granularity level of modules, so the developer can decide to update only parts of the application.
- The update mechanism implemented by Javeleon is transparent to the developer, meaning that the programmer doesn't have to write state transfer function or know the underlay mechanism that Javeleon uses to perform the on-line program change.
- The state mapping problem is transferred into a design problem, meaning that the developer has to use good design practices and do a deep planning of the application architecture before coding, following best practices known in the field of object oriented programming and rich client application development. While enforcing those rules for the sake of updateability, Javeleon teaches the developer to follow good programming practices.
- The process of learning good design practices in order to obtain dynamic updateability and the expected behavior is characterized by a fast learning curve. In our case, while the first releases took months to be thoroughly tested for dynamically updateability, the last iterations were completed in less than a week. Moreover, the programmer, while developing application with Javeleon, become with time more and more aware of the behavior that can turn out from choosing a particular design.

- The developer, by putting effort on choosing the right design for its application carefully designing the architecture before coding, reduces the probability of obtaining unexpected behavior. It was proved that a careful planning of the application state implies what gets instantly updated, defining the transition periods' duration and the dynamic behavior readiness of the software right after the update.
- While choosing the appropriate design for his application the developer has to cope with the trade-off between application performance and dynamic behavior readiness. Some of the good practices for dynamic updateability that we listed, like the ones that eliminate caching, are against application performance.

The development approach followed in order to test dynamic updates proved to be disadvantageous for the developer. The programmer, instead of implementing all the modifications to the code in a program release and then test for dynamic updateability, should go for a stepwise approach (Mannocci, 2010). The former approach burden test and debugging phases when dealing with design choices that affected negatively the dynamic behavior of the application. The latter approach, while easing those phases, enables the developer to be more aware of the of design rules that should be followed.

Up to now Javeleon works with applications built on top of NetBeans RCP, however it is possible to extend it in order to be capable of updating standalone applications. In the future we should continue to explore the modifications that are "dynamic update compliant" by extending the case study application, start a new project or plan specific scenarios. Scenarios could be based on the use of specific design patterns or refactorings. Moreover a document should collect a comprehensive set of good design practices in order to let the use of Javeleon be feasible for a software developer. We should also test the limits to the dynamism of application behavior attainable with Javeleon. In theory, a DSU system should be capable of changing the behavior of an application completely, therefore making it possible to pass from a program to a completely different other program by means of a state mapping function. We believe that following design practices commonly used when developing software frameworks can help on reaching this level on application's behavior flexibility.

As long as we deal with imperative languages the problem of state mapping will be always present. Writing the state mapping function means knowing the logic of the application, therefore this task is a burden for the developer. We believe that Javeleon, by presenting a novel transparent approach that allows high flexibility of modifications and eliminates the need of writing state transfer function, can be considered as a valid solution for developing dynamically updateable applications.

Bibliography

Bennet, P. L., & Swanson, E. B. (1980). *Software Maintenance Management.* Boston: Addison-Wesley Longman Publishing Co., Inc.

Boudreau, T., Tulach, J., & Wielenga, G. (2007). *Rich Client Programming: plugging into the NetBeans platform.* Prentice Hall - Pearson Education.

Brackeen, D., Barker, B., & Vanhelsuwé, L. (2003). *Developing Games in Java.* New Riders Publishing.

Cusumano, A. M., & Selby, W. R. (1997). How Microsoft Builds Software. *Communications of the ACM* , 53-61.

Dmitriev, M. (2001). *Safe Class and Data Evolution in Large and Long-Lived Java Applications.*

Fowler, M. (2000). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gregersen, A. R. (2010). *Extending Netbeans with Dynamic Update of Active Modules.*

Gregersen, A. R., & Jørgensen, B. N. (2009). Dynamic update of Java applications - balancing change flexibility vs programming transparency. *Journal of software maintenance and evolution: research and practice* , 81-112.

Gregersen, A. R., & Jørgensen, B. N. (2008). Module Reload through Dynamic Update - The Case of NetBeans. *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering* (pp. 23-32). IEEE Computer Society.

Gupta, D., Jalote, P., & Barua, G. (1996). A Formal Framework for On-line Software Version Change. *IEEE Transactions on software engineering* , 120-131.

Gustavsson, J. (2003). A Classification of Unanticipated Runtime Software Changes in Java. *ICSM '03: Proceedings of the International Conference on Software Maintenance* , 4-12.

Hicks, M., & Nettles, S. (2005). Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems* , 1049-1096.

Johnson, R. E., & Foote, B. (1991). *Designing Reusable Classes.*

Liskov, B. (1987). *Data Abstraction and Hierarchy.*

Liskov, B. (1987). Data Abstraction and Hierarchy. *Conference on Object Oriented Programming Systems Languages and Applications* (pp. 17-34). Orlando: ACM.

Mannocci, A. (2010). *Stepwise Evolution of Java Applications using Dynamic Updates.*

Martin, R. C. (1996, December). *Granularity.* Retrieved from www.objectmentor.com.

Martin, R. C. (1996, January). *The Open-Closed Principle.* Retrieved from www.objectmentor.com.

Martin, R. C., Beck, K., Grenning, J., Beedle, M., Highsmith, J., Mellor, S., et al. (2001, February 13). Manifesto for Agile Software Development.

Pescovitz, D. (2000). Monsters in a box. *Wired* .

Rajlich, V. (2006, August). Changing the paradigm of software engineering. *Communications of the ACM* , 67-70.

Rajlich, V. T., & Bennet, K. H. (2000). A Staged Model for Software Life Cycle. *Computer* , 66-71.

Sato, Y., & Chiba, S. (2005). Loosely-separated "Sister" Namespaces in Java. *Proceedings of ECOOP'05*, (pp. 49-70). Berlin.

Segal, M. E. (2002). Online Software Upgrading: New Research Directions and Practical Considerations. *International Computer Software and Applications Conference*, (pp. 977-981).

Snyder, A. (1986). Encapsulation and Inheritance in Object-Oriented Programming Languages. *Conference proceedings on Object-oriented programming systems, languages and applications*, (pp. 38-45). Portland.