Benjamin Van Ryseghem

Hazelnut 

dynamic creation of kernels
in a reflexive language

Tutors:
- Stéphane Ducasse
- Renaud Silvestri

June, the 18th 2012

Université Lille1
Sciences et Technologies

Thanks

I would like to especially thank Stéphane Ducasse and Jannik Laval who have always been there to show me the way, and answer my questions.

I also would like to thank Jean-Baptiste Arnaud and Igor Stasenko who have spent hours explaining to me how the VM works.

I would like to thank the whole team. It was a real pleasure to work with you.

I also would like to thank the reviewers: Damien Pollet, Marcus Denker, Camille Teruel, and Erwan Douaille.

I especially want to thank Guillermo Pollito who worked with me on the last steps of this project.

Résumé

Dans le cadre de mon stage de fin d'études, j'ai eu en charge le projet **Hazelnut** au sein de l'équipe RMoD au sein d'Inria à Lille. Le projet Hazelnut consiste en la création dynamique d'un nouveau noyau d'exécution à partir d'une implémentation de Smalltalk, Pharo.

Ce projet doit permettre de concevoir aisément des noyaux minimaux pouvant servir à des systèmes embarqués aussi bien qu'à redéfinir facilement la façon dont le noyau (et donc le système) fonctionne.

Abstract

During my internship at Inria in the RMoD team, I had in charge the **Hazelnut** project which consists in the dynamic creation of a new kernel from Pharo, an implementation of Smalltalk.

This project will be used to create minimal kernel for embedded systems or to easily redefine the way the kernel (and the whole system) works.

Contents

Chapter 1

Introduction

Currently student at the *Université Lille 1* at Villeneuve d'Ascq in the sixth semester of the licence Informatique[1] I had to do an internship for graduation in a research laboratory from 1st April 2012 to 31th June 2012. I have done this internship in the RMoD team working for the Inria, formerly named INRIA for "Institut National de Recherche en Informatique et en Automatique". I already did my DUT graduation internship [2] is this team on the same project.

Few weeks before the beginning of my previous intership, *John Maloney* released one of his projects named Micro Squeak consisting in a proof of concept of the creation of a new kernel (the core classes and methods of the system). We have decided to port this project to Pharo. It is named the project *Seed*.

Context: Smalltalk is an object-oriented language that has the property to save the entire state of the environment from one session to the other in a file called an image. An image contains a snapshot of the Smalltalk environment's memory. It basically contains all the classes and objects of the system at the moment it was saved. The Smalltalk environment is a "living thing", it is never created from scratch, but every new version is evolved from a previous image. For example, there are very probably in all current smalltalk images, objects that were created back in the first version of the first Smalltalk in the 70's (e.g. the "true" and "false" objects). After *John Maloney*'s project was released, we had a proof that the creation of a new kernel was possible, and we wanted to have this concept in Pharo. Some other projects[3] have provided the tools to create new kernels, but with a different approach. *Seed* is the first project inspired by Micro Squeak.

Problems: The problems are that Micro Squeak is based on the version 3.7 of Squeak, and even if the project has been released few months ago, it had

[1] a computer science formation in three years
[2] from 1st Novembre 2010 to 1st April 2011
[3] mainly Chacharas, Spoon

been developed in 2004. Due to that the project is not synchronized with the lastest evolutions of the system anymore. Moreover, the system I used during my internship Pharo, is a fork of Squeak. Pharo has already seen a large refactoring effort but it is still a monolithic system, with still a lot of useless or inefficient code. Because of that, it's quite difficult to define properly what the kernel is and to extract it without collecting irrelevant classes.

Goals The goal of the Hazelnut project is to automatically extract a kernel from a living Pharo image and to bootstrap this kernel into an image. The process has to be automatic to be able to follow the Pharo evolution. Moreover, in order to ease the kernel creation process, the Pharo structure has to be fixed. So in a nutshell the goals of the project are:

- Identify a kernel for Pharo;
- Automatize this identification;
- Fix the Pharo structure;
- Carry out the two last tasks in parallel because the system (and therefore its kernel) is currently evolving and under heavy modifications;
- Bootstrap a new image with this kernel.

Such a kernel could be used in embedded devices, due to the lightweight of the new image, or be used to modify the kernel of the system and to restart from this new kernel without old living objects. It's an argument for the agility of Pharo.

Contributions: My contributions to this project was to:

- Initiate the project;
- Write a kernel extraction script;
- Write kernel analysis tools;
- Work on Pharo kernel analysis to define the kernel;
- Work on the whole Pharo system to flag the system weaknesses such as wrong dependencies between packages;
- Work on different ways to generate a new image;
- Write unit tests;
- Reduce the generated image size;
- Set the initialization
- Ensure the reinitialization of the generated system;

- Fix the system, especially the Pharo kernel.

During my internship, I also worked on the integration of some tools of mine and on their maintenance.

Chapter 2

Context

Firstly I will introduce the work environment I used to live in during my internship in three parts, the institution I was working for, the team I was working in and the language I was working with.

2.1 Inria

Presentation: Inria is a french institution under the dual supervision of the ministries of research and industry which goal is to undertake research in basic and applied sciences and information technologies and communication (SITC). The institute also provides a strong technology transfer in a close attention to training through research, dissemination of scientific and technical development, expertise and participation in international programs.

Composition: Inria accommodates 3800 people in its eight research centers located in Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy, Bordeaux, Lille and Saclay, 2800 of them are scientists from Inria and partner organizations (CNRS, universities, colleges) working in over 160 project teams of joint research. Many Inria researchers are also professors and their students (about 1000) are preparing their thesis within the project teams of Inria research.

The research center of Lille: The Inria Lille – Nord Europe, led by David Simplot-Ryl, gathers from its creation 10 research teams located in a building of 4000m^2 acquired with the help of local government and European funds. It hosts more than 220 people, nearly half is paid by the Institute. This Inria center is an asset for the competitiveness of Nord - Pas de Calais in research and innovation.

2.2 The RMoD Team

Presentation: The goal of the RMoD team is to help re-modularization of object-oriented applications. This goal follows two complementary lines: re-engineering and definition of new constructs for programming languages. To help re-engineering, new analyses are proposed in order to understand and re-modularize big applications (specialized metrics, adapted visualizations, *etc*). In the context of programming languages, constructors for the modularity features and new systems modules validation are performed. The team is also working on a secured kernel for Pharo, an Integrated Development Environment (IDE) for Smalltalk used and maintained by the team.

Applications re-modularization: The evolution of an application is limited by strong dependencies between its inner parts. That's why it's crucial to answer the following questions: *"How can we substitute a part by another one with minimal impact ?"*, *"How to identify reusable elements ?"* or *"How to modularize an application when there is wrong links ?"*. To answer those questions is the goal of Moose, the team software analysis environment, provides a set of analyses. This work is divided in tree parts :

- Tools to understand big applications (packages/modules);
- Analysis for remodularization;
- Software quality.

Semantic elements for modularity. This second line focuses on the definition of new semantics elements for languages in order to construct flexible and reconfigurable software. The team continues its efforts on Traits and Classboxes but also works on new areas such as security in dynamic languages. It works on:

- The definition of a *Traits-only* language and;
- Reconciliation between reflective languages and security.

2.3 Smalltalk presentation in a nutshell

Smalltalk is the development environment used in the team, therefore this is the language I used during my internship. To understand the challenges I faced I will briefly present Smalltalk and its main characteristics.

2.3.1 What is Smalltalk ?

Smalltalk is an object-oriented, reflective, dynamically typed programming language. So let's explain each word:

- Object-oriented : Smalltalk programs are made of objects which communicate using messages (like in Java or C++);
- Reflective : any object can inspect or modify its own structure at runtime (like Java, but to a much greater extent);
- Dynamically typed : variables don't have a type at compilation, but only when a value is stored in them at runtime;
- *Everything is an object*: objects are the sole kind of runtime value.

2.3.2 Smalltalk basics

Smalltalk is based on two classes which constitute the conceptual core of this system, Object and Class (see Figure 2.1). Here you can see that each element cannot exist alone. The bootstrap[1] is the process which leads to this state. However, since Class and Object needs other objects such as string, characters, stream, numbers... the real bootstrap is more complex.

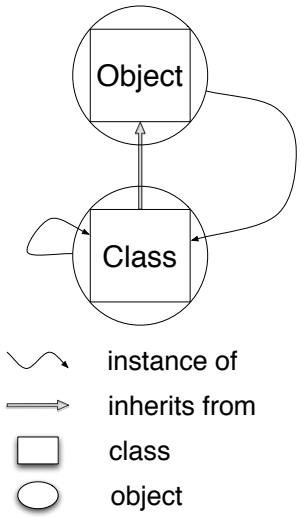

Figure 2.1: Class and Object bootstrap

The most important thing to know is that a bootstrap is a process where a system is initializing itself via its own execution. It's close to the *Chicken or egg dilemma* where each one deeply depends on the other one (more details will be given in the next chapter, page 19).

[1] the term *bootstrap* is often attributed to Rudolf Erich Raspe's story *The Surprising Adventures of Baron Munchausen*

2.3.3 Some basic code lines

Here few examples of Smalltalk code to know how to read further examples:

```
"Variables declaration"
| variable1 variable2 |

"Instance creation"
variable1 := Point new.
"Instance setting"
variable1 x: 1.
variable1 y: 2.

variable2 := Point new.
variable2 x: 1.
variable2 y: 2.

variable1 = variable2. true
variable1 == variable2. false
```

Here, we can see five things :

- | | : it allows you to declare variables.
- := : it is the assignment.
- new : it's a class method which creates a new instance of the receiver, e.g. "`Point new`" sends the message new to the class (which is also an object) Point.
- = : it tests if two objects represent the same object, it's a *logical* equality. It is a message, asking the receiver (before =) whether it is the same object as the parameter (after the =).
- == : it tests if two objects point to the same reference, it's a *physical* equality. It is a message also.

Let's see a basic method of the Integer class:

```
plus: integer1 andPlus: integer2

    ^ self + integer1 + integer2
```

Here we learn three new things:

- the colon : the way to specify parameters to most methods.
- self : the receiver of the method (similar to this in Java).
- the caret : it allows you to return a value[2]. By default a method returns self.

A method is often referred to by the notation Class≫#selector to have an unique notation. So the method we just saw is noted Integer≫#plus:andPlus:. One more example to see the last syntax elements, a method of class Class:

```
copyMethodDictionary
  "This method answer a copy of my method dictionary"

  | result |
  result := SortedCollection new sort: [:m1 :m2 | m1 selector < m2 selector].
  self methodDictionary do: [:method |
     result add: method.
     Transcript  show: method selector asString, ' added.';cr].
     ^ result
```

Here we have :

- "some text" : a comment.
- [:arg | code] : it's a block (a λ-expression). They act like anonymous methods where arg is an argument of the block which is used to execute the code. In addition it captures its creation environment - it is a lexical closure.
- rcvr m1; m2 : it's a cascade of messages. It means that the receiver of the second method (m2) is the same that the first method's (m1) receiver, in this case rcvr.

Now, you know the syntax of Smalltalk.

2.3.4 The SystemDictionary

The *System Dictionary* is a dictionary which contains all the global variables, including all the classes of the system. In Pharo, the variable *Smalltalk* is, normally, the sole SystemDictionary of the system. We can notice that *Smalltalk* is a global variable, so it contains itself.

2.3.5 Special object array

The Special Objects Array is basically an array shared between an image and the Virtual Machine (VM). It's an interface allowing the VM to know where are special objects it needs.

What is in the Special Objects Array ? Here I will give the first ten elements of the Special Objects Array:

- nil[3]

[2] You can sometime see ↑ instead.

[3] nil is te basic NullPattern object, like NULL in C or null in Java

- true
- false
- #Processor->Processor
- Bitmap
- SmallInteger
- ByteString
- Array
- Smalltalk
- Float
- ...

We can notice that the ninth element is Smalltalk, the current SystemDictionary.

2.3.6 The virtual machine

As in some other languages (especially Java), Smalltalk's methods are converted then interpreted by a VM. In fact, the Smalltalk compiler analyzes the code then createss a CompiledMethod which is a representation of the method but including more information ready to be executed by a *byteCode* interpreter or JustInTime translator:

- the *byteCode* : the source code converted into a language that the VM can interpret;
- the *literals* : they represent low level objects such as number true, false, strings that are referenced and read by the scanner at compilation time. *Literals* especially store pointers to class referred into the source code.

Method

Let's see an example, String≫#copy :

```
copy

    | string |
    string := String new: (self size).
    self doWithIndex: [:character :index |
            string at: index put: character].
    ^ string
```

First, let's explain what this method do :

- | string | : we declare a new variable named string.
- string := String new: (self size) : it creates a new instance of the class String which the size is set at the size of the receiver and then stores it in the variable named string.
- self doWithIndex: [:character :index | : we browse the receiver and for each element, we store the element in the variable character end the index of the element in the variable index.
- string at: index put: character : at the index index of string, we put character.
- ^ string : we finally return the variable string.

In a nutshell, this method basically parses the receiver (which is a String) and fills up a new String with the same value.

CompiledMethod

Now, let's take a look at the corresponding CompiledMethod

- the *byteCode* :

```
21 <40> pushLit: String
22 <70> self
23 <C2> send: size
24 <CD> send: new:
25 <68> popIntoTemp: 0
26 <70> self
27 <10> pushTemp: 0
28 <8F 12 00 05> closureNumCopied: 1 numArgs: 2 bytes 32 to 36
32  <12> pushTemp: 2
33  <11> pushTemp: 1
34  <10> pushTemp: 0
35  <C1> send: at:put:
36  <7D> blockReturn
37 <E1> send: doWithIndex:
38 <87> pop
39 <10> pushTemp: 0
40 <7C> returnTop
```

  Basically, the *byteCode* tells the VM how to manage the execution stack.

- the *literals* :
  - #String->String : this literal refers to the String called at the instantiation of String. This association is the one present in the Pharo SystemDictionary

- #doWithIndex: : this literal refers to a method invoked.
- #copy : this literal represent the selector of the method. It doesn't appear in the *byteCode* because it's not needed (moreover, historically, methods used to be anonymous).
- #String->String : this literal refers to the class of the method.

You can notice that new and at:put: are not in the literal. This is due to the fact that those methods are *special byteCodes*

It's important to keep in mind that all methods points to the current SystemDictionary through their literals, but you do not have to be able to read or understand the *byteCode* because it's a very low level tool. Moreover, *byteCode* is rarely read by developers.

Chapter 3

Bootstrapping Challenges

In this chapter we first define what is a system bootstrap and give some related definitions, then we present the current problems and the goals of our work.

3.1 Definitions and benefits

Reflective system. Smith defines reflexivity as: *« An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on and deals with its primary subject matter »*. [1]

In the context of programming languages, this definition can be stated as: *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation :* introspection *and* intercession *[...] Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called* reification. [1]

Maes has proposed in the first chapter of his thesis [3], precise definitions to clearly characterize reflective programming. We refer here to these definitions:

- A **computational system** is something that **reasons** about and **acts** upon some part of the world, called the **domain** of the system (p 13).
- A computational system may also be **causally connected** to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other (p 15).
- A **meta-system** is a computational system that has as its domain another computational system, called its **object-system**. [...] A meta-system has a representation of its object-system in its data. Its program specifies **meta-computation** about the object-system and is therefore called a **meta-program** (p 17).
- **Reflection** is the process of reasoning about and/or acting upon oneself (p 19) (see Figure 3.1).

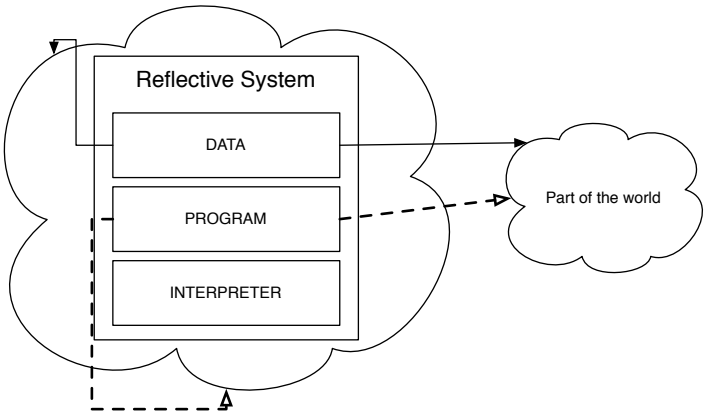

Figure 3.1: A Reflexive System

- A **reflective system** is a causally connected meta-system that has as object-system itself. The data of a reflective system contain, besides the representation of some part of the external world, also a causally connected representation of itself, called **self-representation** of the system. [...] When a system is reasoning or acting upon itself, we speak of **reflective computation** (p 19).

- A language with a **reflective architecture** is a language in which all systems have access to a causally connected representation of themselves.

- A programming environment has a **meta-level architecture** if it has an architecture which supports meta-computation, without supporting reflective computation (p 34).

- The **meta-object** of an object X represents the explicit information about X (e.g. about its behavior and its implementation). The object X itself groups the information about the entity of domain it represents (p 120).

Bootstrap. Bootstrapping a kernel is the process that builds the minimal structure of a language that is reusable to define this language. The idea is to use as early as possible the benefits of the resulting language by implementing a minimal core whose only goal is to be able to build the full system. As an example of a possible bootstrap: we write in C the minimal structures to represent and execute objects, and we then write with this core the full system. This avoids to have to write the full system (compiler for example in C). In ObjvLisp [2], the class Class is first defined using low level API, then Object is created, then Class is fully reimplemented using the first one.

3.2 Current problems in Pharo

The current structure of Pharo is a problem for creating easily a bootstrap at different levels:

- Pharo has a monolithic structure with a lot of old code (even though all our efforts are on that);
- There are hidden structural dependencies requiring deep analysis to be revealed. For example, Stream depends on Compiler while it should be the inverse;
- The definition of what is part of the Pharo kernel is fuzzy and dispatched through different packages which aren't autonomous and self contained.

3.3 Goals of the project

Bootstrapping. We need a process to create an autonomous kernel and to bootstrap it into a new image. This process has to be modular in order to be able to create a specific kernel. This kernel has to be autonomous which means it has to be isolated from the other classes and can only refer to itself.

Dealing with kernel changes. Pharo development is real active in all parts of the system and in particularly the kernel. Therefore it is not possible to take 6 or 8 months to just bootstrap it. We need a solution that can cope with the continuous changes and fixes in the kernel.

Therefore we will attack the problem from two angles: (1) cleaning the current kernel to ease the bootstrap, (2) building a bootstrap process that can be applied to the evolving kernel.

Chapter 4

Hazelnut

Hazelnut is one of the **Seed** projecst, where **Seed** was originally composed by different projects whose goal is to generate new kernels, all based on the project *Micro-Squeak*. For now we essentially distinguish two of them:

- PineKernel: it is a port of Micro Squeak in Pharo
- Hazelnut: it is the building of a kernel in Pharo starting from the Pharo kernel.

This project is composed by three different parts, the kernel collection, the kernel isolation and finally the image creation.

4.1 Kernel creation

Goal: The goal of this part is to create an alternative SystemDictionary (a SystemDictionary is a namespace holding all the classes of the system) starting from the existing one in Pharo, and to collect classes which are needed to build the kernel.

Problems:

- Which classes need to be collected ?
- How do we fill up the new SystemDictionary with those classes ?

Solutions:

- A first naive approach is to collect every class Object depends on in order to have an autonomous system. But due to bad dependencies in the system, the kernel collected this way contains half of the Pharo classes. This is clearly not satisfying. Therefore we have decided to have another approach. The second and final approach is to provide to the builder the

list of classes the user wants in the new kernel plus some classes absolutely needed by the system[1]. The problem is that we had to determine which classes are the absolutely needed ones. In order to answer this question, a tool to analyze classes dependencies has been written and recursively used starting from the Kernel package until we had a quite autonomous kernel composed of around 200 classes[2]. This tool also flags bad dependencies, but this part will be exposed in the next chapter (page 27).

- MicroSqueak's solution to fill up the new SystemDictionary is to recompile needed classes with a prefix and then to collect them. It's quite efficient when you have 20 classes to copy, but here we have the constraints that we do not know by advance what we will copy and then we want to be as fast as possible.

The solution we adopted is to create a new instance of SystemDictionary and to directly copy classes into it without recompiling them. The classes are still pointing to their original namespace as shown by Figure 4.1.

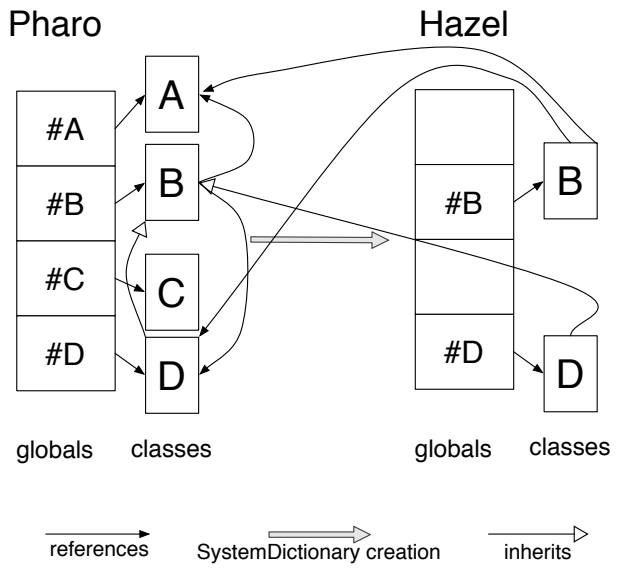

Figure 4.1: Step 1 - Copy the classes B and D into the new SystemDictionary

The second step is to make sure that the class and metaclass hierarchy is maintained in both the environments and that the methodDictionary[3] is also copied. To be sure to reconstruct the hierarchy, the copy method recursively rebuild class, metaclass and superclass hierarchy (see Figure 4.2).

[1] as Object, ProtoObject or MethodContext for example

[2] Pharo contains around 1800 classes

[3] a *methodDictionary* is a dictionary implemented in each class and containing all the methods of the class

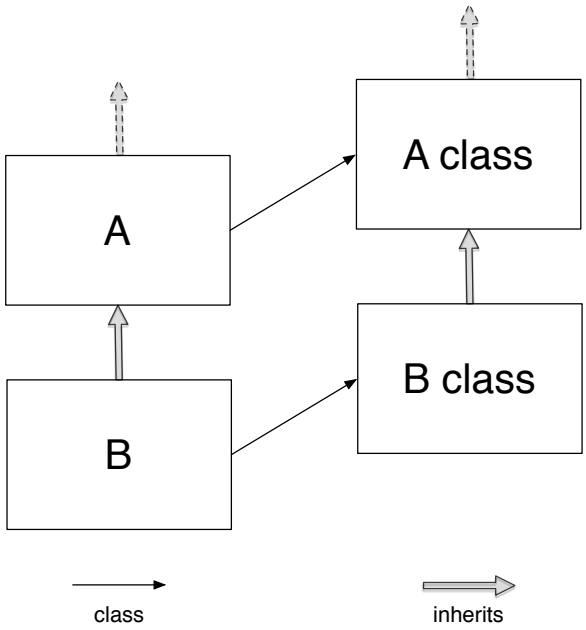

Figure 4.2: Class and MetaClass Hierarchy

Here is the pseudo code in Smalltalk that add a class in the Hazel SystemDictionary and check the hierarchy:

```
HazelKernelBuilder>>#addAClassInDictionary: class
"Add a copy of the class in the Hazel SystemDictionary then answer the copy"

    | hazel copy className |
     className := class name asSymbol.

    "Check if the class is already in the dictionary"
    (self list includesKey: class name)
          ifFalse: [^ nil].

     hazel := Smalltalk at: #HazelSmalltalk.
    (hazel globals includesKey: className)
          ifTrue: [^hazel at: className].

    "If not, add a copy in the dictionary"
     copy := self copyClass: class.
     self registerClass: copy.

    "then check the superclass"
```

```
copy superclass ifNotNilDo: [:superclass || superCopy |
    "add the superclass"
    superCopy := self addAClassInDictionary: superclass.
    "change the superclass"
    copy superclass: superCopy.
    "then change the metaclass's superclass"
    copy class superclass: (superCopy class)].

"Check all literals of all methods"
self checkMethods: copy.

"Check all class var"
self checkClassVar: copy.

^ copy
```

The last instructions will be commented in the next section.

The only wrong inheritance which remains is that ProtoObject in the Hazel world inherits from nil which is still in the Pharo world. But this will be fixed when we will change nil (see paragraph **??** page **??**).

In a nutshell: We are now able to copy wanted classes and needed classes into a new SystemDictionary, with a good hierarchy, but Hazel classes keep references to Pharo ones (see Figure 4.1).

4.2 Kernel isolation

Now that the kernel is created, we need to isolate it by removing dependencies to the Pharo world. There are two different types of dependencies which need to be fixed.

4.2.1 References to unwanted classes

Goal: Here we want to remove dependencies from Hazel classes to Pharo classes we haven't copied.

Problems:

- How to detect unwanted references ?
- How to remove those dependencies ?
- How to be sure it will not crash the system ?

Solutions:

- There are two places where unwanted references can be found:
  - In a method: in a method literal there are references to invoke classes (see page 14). Due to that, we can find references to unwanted classes;
  - In a class variable[4]: we can have an instance of an unwanted class or just an unwanted class itself.

  The solution adopted is to check a class methodDictionary and its class variables during class copy. If unwanted references are found, the following solution is applied.
- Removing dependencies is the key point of this cleaning step.

  For class variables, the solution was to set them to nil (in the minimal kernel creation process, only one class variable had to be set this way (HaloFont from StandardFonts).

  For methods we have considered several solutions. Our first thought was to remove the method and to recursively remove the sender of the method. But due to class structure,this would have removed almost all methods of the kernel. Then we thought about creating a *NullPattern* object implementing all the removed methods. The problem was to find which kind of answer is expected from each method, and how to dynamically replace the sender in the code source. Finally we chose to remove the method and to keep the senders.

[4] a class variable is a variable shared by all the instances of a class

- We haven't found a solution to ensure that you are not creating an unstable system. As long as your system can be changed, you can't certify that your kernel is totally functional. We are working on a better isolation of the Pharo kernel to reduce as much as possible the number of references (see Section 5.2 page 32).

In a nutshell: We have removed the references to unwanted class (see Figure 4.3), but because of that the integrity of the kernel may be corrupted.

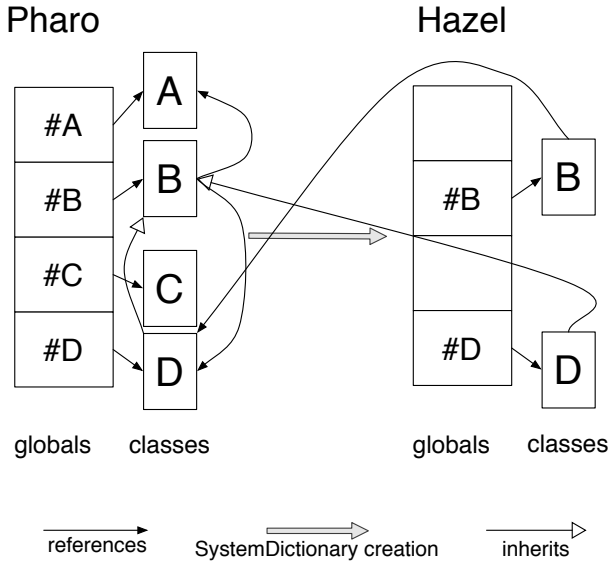

Figure 4.3: Step 2 - Remove references to unwanted classes

4.2.2 Dependencies from Hazel classes to original classes

Goal: Here we want to reroute dependencies from Hazel classes to Pharo classes to have only intern references in the Hazel kernel. Those references are stored into methods literals.

Problems:

- How to change those references ?
- How to check what the kernel contains ?

Solutions:

- This part is quite simple because the field was well prepared. Only methods refering copied classes are not fixed yet. So for those classes, the methodDictionary had just to be parsed in order to fix literals. And for fixing literals, we just change the Pharo associations to their corresponding Hazel one (and we are sure it exists).
- To check what the new image actually contains, and to be able to modify it f needed, we have modified a bit a couple of tools to be able to browse the living kernel before its serialization.

In a nutshell: We now have a kernel isolated with only internal references (see Figure 4.4).

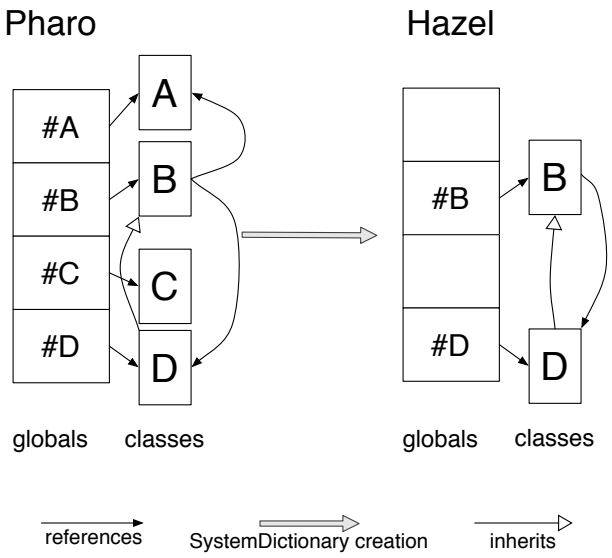

Figure 4.4: Step 3 - Reroute the remaining dependences

4.3 Image creation

Goal: The goal of this part is to successfully build a new image starting from an isolated kernel. An image is a snapshot of living objects binary saved in a file. They basically contains classes and some living instances.

Problems:

- Which technique should we use to create the image ?
- How to successfully replace the Special Objects Array ?

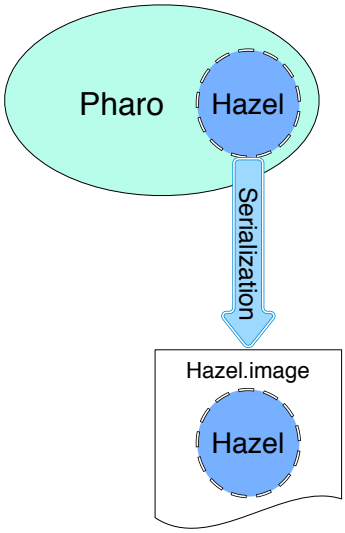

Figure 4.5: Micro Squeak - Serialization of needed objects

Solutions: Two solutions have been tested to create the image:

- The first solution was to take a lively image and to dynamically switch the Special Objects Array in order to make the unneeded object garbage collected (see *Garbage Collector* page 40). The difficulty of this method is that we are drastically modifying the image during its own execution. Some objects can easily be changed (i.e. nil or Character) when some others freeze the Virtual Machine (i.e. String or Semaphore). We think it's due to the fact that during the execution of the switching method, we are modifying the method context, and the Virtual Machine points to unaccessible pointers and we got an error[5] (see Figure 4.6).

[5] segmentation fault

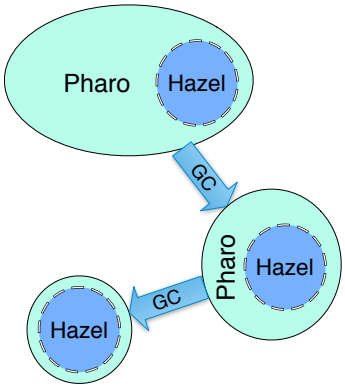

Figure 4.6: Hazel - Garbage Collection of unneeded objects

  - To replace the Special Objects Array, the idea was to take the current Special Objects Array, which is a Dictionary, and to replace its values. The problem is that some values called all the time by the image are buffered in the Virtual Machine (those values are nil,true and false) and updated at the opening of the image. So we have decided to use the primitives become: and becomeForward: which basically switch references between the receiver and the argument.
  - Because some objects can't easily be switched, we have considered another approach.

- The second solution, which is also the Micro Squeak solution, consists in collecting then serializing all the needed objects into a new image (see Figure 4.5). But since the tool used for Micro Squeak was not maintained anymore and based on a really old version of Pharo, first we had to make it works.

In a nutshell: We can produce a new image based on the previously created kernel.

4.4 Image initialization

Goal: The goal here is to properly initialize the newly created image to make it a real autonomous system.

Problems:

- What need to be initialized ?
- How to reinitialize class variables ?
- How to reinitialize classes in the correct order ?

Solutions:

- We know that each class variable has been set to nil during the process, so they need to be initialized to make the system works. But are there any references to a singleton that may have not been initialized ? I still do not know the answer, neither how to find such dependencies, except by checking each dependency by hand.
- Class variables should be initialized when the class they belong to is initialized. To ensure that, we checked by hand all classes which belong to the kernel, and ensured that class variables are initialized. If not, we have fixed it.
- But then, how to initialize classes ? You can have situations where two classes depend on each other for initialization (see the following example).

  ```
  A class>>#initialize

      myClassVarWhichShouldEqual1 := 1.
      myClassVarWhichShouldEqual2 := B myClassVarWhichShouldEqual2.

  B class>>#initialize

      myClassVarWhichShouldEqual1 := A myClassVarWhichShouldEqual1.
      myClassVarWhichShouldEqual2 := 2.
  ```

  In this case, which class to initialize first, A or B ? Since there is no obvious answer and that usually such circular dependencies are pointing to a bad design, the solution is to fix this kind of dependencies.

Chapter 5

System Preparation

To ease the kernel creation process, the system has had to be fixed or at least bugs have had to be flagged. This way we can at least point the structural anomalies and fix them to reduce the number of dependencies. In this optic, we have work on 3 topics:

- Text Constants;
- Kernel fixes;
- Pharo dependencies.

5.1 Text Constants

TextConstants used to be an old pool dictionary stored as a global variable which was used to store and share information between text related classes. Since the old way of managing pool dictionaries was not good (using global dictionaries), SharedPool s have been introduced. A SharedPool is a special class designed to store constants variables and to be shared between other classes. You easily use them by specifying the poolDictionaries field of a class.

Back in 2006 all dictionaries which only store constants were migrated except TextConstants. The reason of the non migration was probably that it was too deep in the system and that it involved low level fixes. In addition some applications started to use TextConstants not only to share constants but also to act as a bag to store temporary values, defeating the purpose of a Pool Dictionary.

Due to that, if one browses all classes, and for each browse poolDictionaries, you can have a SharedPool (i.e. a Class) or TextConstants (i.e. a Dictionary). Of course those two objects don't have the same interface. So instead of differentiating cases, we have decided to fix the situation and to finally convert TextConstants into a SharedPool.

Goal: Write a script which can automatically retrieve information from TextConstants (as a Dictionary), create a SharedPool class named TextConstants, then fill up this class with the retrieved information.

Problems:

- How to differentiate constant values from variable ones ?
- Where those methods can store information they used to store in TextConstants ?
- How avoid to rewrite all the methods ?

Solutions:

- To identify the problems, we read all methods invoking TextConstants and finally found a method in Text class which initialize TextConstants, even if it is in a strange way. So we have invoked this method but on a dummy dictionary instead of TextConstants to be able to retrieve all the variables name and value. With those information, it was easy to dynamically defined TextConstants and it initialize method.
- We have decided to add a class variable in TextConstants named *TextSharedInformation* which is a Dictionary used to store values.
- To avoid to rewrite all by hand, we built a script that automatically adds TextConstants in the poolDictionaries of classes which need it. When TextConstants was used has a Dictionary, the code is changed to invoke *TextSharedInformation* instead. But when TextConstants is used as a value holder, we had to change methods manually. By chance, only two methods needed to be rewritten this way.

In a nutshell: now TextConstants is a SharedPool and the whole system has been changed in order to use the new design of TextConstants. All variables founded in the field poolDictionaries are classes.

5.2 Kernel fix

In order to reduce the amount of bad dependencies during the kernel isolation (see section 4.2.2 page 27).

Goal: Minimise Pharo kernel dependencies by fixing classes to ease the Hazel kernel isolation.

Problems:

- How to identify bad dependencies ?
- How to fix them ?

Solutions:

- To identify bad dependencies, we have used Moose on Pharo 1.2 and have manually flag each dependencies.
- To fix them there is no magic formula, we have spent time on it, and we are far from having fix all bad dependencies. But a bug entry was opened on the developers platform for each bad dependence.

In a nutshell: 20 dependencies have already been fixed, but 40 bug entries are still open waiting for a fix.

5.3 Pharo dependencies

Here the topic is the same that the previous one. We want to flag dependencies in order to isolate modules, but here it's for the whole system.

Goal: Analyze (and fix) all Pharo bad dependencies in order to have isolated modules easily pluggable and un-pluggable and not circular references in class initialization.

Problems:

- Which tool to use to analyze a whole system ?
- How to automatize the creation of a new bug entry ?
- How to break circular references ?

Solutions:

- Moose have been used here too to build a tool that flag Pharo packages dependencies (about 1300 dependencies);
- We haven't found how to use the Google interface to automatize the creation of bug entries, so we have to add each bad dependence one by hand.
- To break circular references, there is no automatic answer. Each case has to be analyzed and deeply understood before being fixed.

In a nutshell: Now that all the bad dependencies are flagged, it's easier to focus on important things to fix.

Chapter 6

Conclusion

6.1 Technical results

I have technically learned a lot during my internship at many levels. Here is a non-exhaustive list of things I have learned or deepened.

Smalltalk. I already knew Smalltalk before the beginning of the internship. But I have improved my knowledge about the Smalltalk language especially because I used to borrow a lot of books from the lab, in particular the ones about dynamic languages. Thanks to Smalltalk, and the fact that you can browse living objects, I understand how an object-oriented language works more deeply.

Pharo. By analyzing the Pharo kernel then the whole system, I have read a lot of code and this way learned a lot about the Pharo internal structure. It allows me to think at the precise definition of a kernel, which classes are needed to run a system. Thanks to these analyses, I now have a better comprehension of modular packages.

The Virtual Machine. I have learned bases of how the VM works and how to create a new version of the current VM using VMMaker. I have also learned how to add new primitives in the system. I really would like to know more about the VM, because it used to looks like a black box even if the system can't work without the VM.

English. The team being multi-cultural with people from several countries, the english is used all the time for the internal communications. Moreover, all the mails shared in the Pharo mailing list are in english too. I started to learn how to write a research paper in english.

SVN. The team is using SVN for storing internal information, so I had to learn how it works.

6.2 Human results

6.2.1 Integrate a research team

Even while I learned a lot technically, humanly I discovered a new working environment inside the RMoD team, where communication and autonomy are really important.

- The communication is the backbone of the research work whether written or oral Another member of the team was working on a similar project (*PineKernel*) and each day we sent an email to the whole team with a sum up of our daily work. This way we were aware of each other work, and we shared a lot of knowledge. Moreover, I have often take a seat and ask a question to another team member, and spent hours sharing ideas and quickly test them. A large part of ideas used in Hazel were born this way, and it was really pleasant to work this way.

- The autonomy in work was import too because I had to make my own schedule and to learn how to manage my time. Moreover, I was alone working on the Hazel project, so I had to set a rhythm by myself. In the other hand, I forced myself not to work at home, to keep a regular rhythm which is big change compared to the Institut Universitaire de Technologie (IUT).

I also had multiple points of view on the work of a researcher, which is the job I would like to do. Moreover, the team being multi-cultural, I've learned some cultural parts from Argentinian culture (like Alfajoles), or Ukrainian one. It was really cool to practice my english with people from all over the world.

6.2.2 Pharo, a living community

Beside working for the Inria, I worked as a member of the active Pharo community. I had developed some projects before being a member of the team. Those projects have been improved and integrate into the current version of Pharo. These improvements have been done with the help of other members of the community especially during Sprints (coding session). This community is really reactive and any question, from the dumbest one to the more specific one, can be asked on the mailing list you will always have an answer.

In a nutshell: It was really a good experience that I hope I could reproduce. I really like to manage a project by myself, and to schedule my work alone. Moreover I really like to work on a research theme.

6.3 Conclusion

Context: After the John Maloney's MicroSqueak, project has been released, we had a proof that the creation of a new kernel was possible, and we wanted to have this concept in Pharo. Some other projects[1] have provided the tools to create new kernel, but with a different approach and not working anymore. Seed is the first project inpsired by Micro Squeak.

Goal: The goal of the project was to implement a process able to dynamically create a new kernel starting from a living image and a collection of classes the new kernel must provide. In parallel, I had to fix the Pharo structure in order to ease the previous process.

Problems: The most important problems encountered were:

- What is Pharo kernel ?
- How to collect needed classes ?
- How to create another kernel in a living image ?
- How to isolate the kernel ?
- Does the Pharo structure allow you to easily separate modules ?
- How to bootstrap the kernel ?
- How to create a new image with this kernel ?

Solution: After structural analysis, I have implemented a script which takes a list of classes as an argument, and build an autonomous kernel with all the classes needed and wanted. I have also provided another script to serialize this kernel as a new image which is a real system fully working.

Next Steps: The next steps will be to explore another way for filling the new name space. Instead of copying objects for the current system, trying to have a file based declarative boostrap. We could also fix the whole structure to ease the kernel isolation, and this way having a better isolated kernel in Pharo.

Conclusion: As a conclusion, Hazel provides tools to create a new isolated kernel, and also a new image with this kernel fully working. Working on the definition of a kernel, especially in Pharo allowed us to define which classes *are composing* the current kernel and which one *should compose* the kernel.

Working on the system structure had also revealed some problems in the packages architecture and dependencies.

[1] mainly Chacharas, Spoon

Glossary

Garbage Collector	The *Garbage Collector* is a mechanism which consist in automatically destroy unreferenced objects of a living system in order to recover memory.	28
IDE	Integrated Development Environment	10
IUT	Institut Universitaire de Technologie	36
Micro Squeak	*Micro Squeak* is a *John Maloney*'s project which produces a headless 57k image with a few basic classes as a proof of concept.	5, 21, 29, 37
Moose	*Moose* is a platform for software and data analysis maintained by a part of the RMoD team.	33
Pharo	*Pharo* is a fork of Squeak, an implementation of the object-oriented, dynamically typed, reflective programming language Smalltalk.	5–7, 13, 15, 19, 21, 22, 24–27, 33, 35–37
primitive	A *primitive* is a C implemented method directly interpreted by the VM.	29
Smalltalk	*Smalltalk* is an object-oriented, reflective, dynamically typed, *all is object* programming languages.	10, 11, 13, 35
Special Objects Array	The *Special Objects Array* is an array shared by the image and the VM which store primordial information.	13, 28, 29
SVN	*Subversion* is a tool for version control.	36
VM	Virtual Machine	13–15, 35

Bibliography

[1] D. G. Bobrow, R. P. Gabriel, and J. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[2] P. Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, Dec. 1987.

[3] P. Maes. *Computational Reflection.* PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, Jan. 1987.

Résumé

Dans le cadre de mon stage de fin d'études, j'ai eu en charge le projet **Hazelnut** au sein de l'équipe RMoD au sein d'Inria à Lille. Le projet Hazelnut consiste en la création dynamique d'un nouveau noyau d'exécution à partir d'une implémentation de Smalltalk, Pharo.

Ce projet doit permettre de concevoir aisément des noyaux minimaux pouvant servir à des systèmes embarqués aussi bien qu'à redéfinir facilement la façon dont le noyau (et donc le système) fonctionne.

Abstract

During my internship at Inria in the RMoD team, I had in charge the **Hazelnut** project which consists in the dynamic creation of a new kernel from Pharo, an implementation of Smalltalk.

This project will be used to create minimal kernel for embedded systems or to easily redefine the way the kernel (and the whole system) works.