Département de formation doctorale en informatique **École doctorale SPI Lille**
Numéro d'ordre : 41719

Virtualization Support for Application Runtime Specialization and Extension

Virtualisation pour Specialisation et Extension d'Environnements d'Execution

THÈSE

présentée et soutenue publiquement le 13. April 2015

pour l'obtention du

Doctorat Délivré Cojointement par
Mines Douai et l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Guillermo Polito

Composition du jury

Président :	Loïc LAGADEC
Rapporteur :	Oscar NIERSTRASZ, Christophe DONY
Examinateur :	Jan VITEK
Directeur de thèse :	Stéphane DUCASSE (Directeur de recherche – INRIA Lille Nord-Europe)
Co-Encadrant de thèse :	Noury BOURAQADI, Luc FABRESSE

Centre de Recherche en Informatique, Signal et Automatique de Lille
Département IA Mines Douai - INRIA Lille Nord Europe

RMoD
Inria Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley
59650 Villeneuve d'Ascq
France
http://rmod.inria.fr/

Acknowledgments

I would like to thank my thesis supervisors Stéphane Ducasse, Noury Bouraqadi and Luc Fabrese for allowing me to do in the first place a Ph.D at the RMoD and CAR teams, and for their advice during this thesis.

I thank the thesis reviewers and jury members Oscar Nierstrasz, Christophe Dony, Jan Vitek and Loïc Lagadec for accepting to be part of the jury of my defense.

I specially thank Santiago, Pablo, Alexis, Esteban, Camille, Charlotte, my family and all others that gave me their support me during the time I worked on this thesis.

I would also like to thank the Pharo community for their support and help during my learning. I expect part of this thesis is of value for them.

Abstract

An application runtime is the set of software elements that represent an application during its execution. Application runtimes should be adaptable to different contexts. Advances in computing technology both in hardware and software indeed demand it. For example, on one hand we can think about extending a programming language to enhance the developers' productivity. On the other hand we can also think about transparently reducing the memory footprint of applications to make them fit in constrained resource scenarios *e.g.*, low networks or limited memory availability.

We propose Espell, a virtualization infrastructure for object-oriented high-level language runtimes. Espell provides a general purpose infrastructure to control and manipulate object-oriented runtimes in different situations. A first-class representation of an object-oriented runtime provides a high-level API for the manipulation of such runtime. A hypervisor uses this first-class object and manipulates it either directly or by executing arbitrary expressions into it. We show with our prototype that this infrastructure supports language *bootstrapping* and application runtime *tailoring*. Using bootstrapping we describe an object-oriented high-level language initialization in terms of itself. A bootstrapped language takes advantage of its own abstractions and is easier to extend. With application runtime tailoring we generate specialized applications by extracting the elements of a program that are used during execution. A tailored application encompasses only the classes and methods it needs and avoids the code bloat that appears from the usage of third-party libraries and frameworks.

Keywords: application runtimes, object-oriented, high-level, virtualization, first-class runtimes, object spaces, bootstrapping, tailoring.

Résumé

Un environnement d'exécution est l'ensemble des éléments logiciels qui représentent une application pendant son exécution. Les environnements d'exécution doivent être adaptables à différents contextes. Les progrès des technologies de l'information, tant au niveau logiciel qu'au niveau matériel, rendent ces adaptations nécessaires. Par exemple, nous pouvons envisager d'étendre un language de programmation pour améliorer la productivité des developpeurs. Aussi, nous pouvons envisager de réduire la consommation memoire des applications de manière transparente afin de les adapter à certaines contraintes d'exécution *e.g.*, des réseaux lents ou de la mémoire limités.

Nous proposons Espell, une infrastructure pour la virtualisation d'environnement d'execution de langages orienté-objets haut-niveau. Espell fournit une infrastructure généraliste pour le contrôle et la manipulation d'environnements d'exécution pour différentes situations. Une représentation de 'premier-ordre' de l'environnement d'exécution orienté-objet fournit une interface haut-niveau qui permet la manipulation de ces environnements. Un hyperviseur est client de cette représentation de 'premier-ordre' et le manipule soit directement, soit en y exécutant des expressions arbitraires. Nous montrons au travers de notre prototype que cet infrastructure supporte le *bootstrapping* (*i.e.*, l'amorçage ou initialisation circulaire) des languages et le *tailoring* (*i.e.*, la construction sur-mesure ou 'taille') d'environnement d'exécution. En utilisant l'amorçage nous initialisons un language orienté-objet haut-niveau qui est auto-décrit. Un langage amorcé profite des ses propres abstractions se montrant donc plus simple à étendre. La taille d'environnements d'exécution est une technique qui génère une application spécialisé en extrayant seulement le code utilisé pendant l'exécution d'un programme. Une application taillée inclut seulement les classes et méthodes qu'elle nécessite, et évite que des librairies et des frameworks externes surchargent inutilement la base de code.

Contents

Introduction

Chapter 1

Contents

An application runtime is the set of software elements that represent an application during its execution. Focusing on high-level object-oriented applications, an application runtime includes *e.g.*, loaded libraries, classes and methods, created objects and threads. Within an application runtime, we find the *language runtime*. The language runtime is the subset of the application runtime that defines the concepts and behavior available in the language we use *i.e.*, the set of structures and constructs that describe the language internals. The language runtime implements the model that the language offers to the developers.

Manipulating and modifying these application and language runtimes, and therefore their language models, is becoming important. Multicore hardware brought new problems in concurrency and parallelism; the *cloud* increases the need of software adaptation for application migration and resource tailoring; new resource constrained devices such as cellphones are widely spread, making ubiquitous computing a real concern. These new technologies present new challenges to software and language developers. The software we use, and in particular the programming languages and tools we use should be easily tailorable to support many of the new challenges that come with new technology and needs.

We observe, however, that some software aspects are not usually designed to be easily changed. For example, changing a programming language model to provide better abstractions or to consume less memory would require modifications in the whole language infrastructure (*e.g.*, compiler, virtual machine) and the applications that use it. To support this kind of changes applications and languages should be either engineered from scratch or re-engineered with these change in mind. We need tools and methodologies that support such changes [NDG+08]. The languages and applications we develop should be adaptable to such unanticipated scenarios.

To address this goal we propose *Espell* : a runtime virtualization infrastructure. Espell virtualizes an application runtime for its control and manip-

ulation with the ultimate goal of generating specialized versions of it. Manipulations are transparent for the virtualized runtime. We show how Espell simplifies the generation of application and language runtimes in two different approaches: language runtime (re)creation by bootstrapping allows us to modify and redefine the concepts a language offers to its users; application runtime extraction reduces the memory consumption caused by the code units of an application.

1.1 Motivation

Advances in computing technology both in hardware and software demand that our applications and languages adapt themselves better to them. More resourceful machines can make use of richer programming languages with the purpose of enhancing the developer's productivity. We could think, for example, of extending an existing language with new concepts besides classes and inheritance such as first-class instance variables [VBLN11]. First-class instance variables provide a point of extension on instance variable access and allow the transparent implementation of alternative models for storing state (*e.g.*, putting an object's state inside a dictionary instead of an object's slot). First-class instance variables help in removing boilerplate code and to provide better programming abstractions.

At the same time, constrained resource devices pose completely different challenges. Limitations in memory, energy or CPU power may require us to *downgrade* some features from our applications and programs to reduce memory consumption. For example, we could consider removing the reflection support and the language meta-data from an application that does not use it. Moreover, we could consider removing every unused element from our application development artifact.

We believe applications and languages should provide support to tackle challenges as these ones. Moreover, they should not be limited to them but also be flexible to adapt themselves to other unknown situations.

1.2 Application Runtimes: Concepts

In the context of application runtime manipulation, we face the following question: *What are the elements of high-level programming languages we should focus on?* High-level programs are inherent complex pieces of software. For such a reason in this thesis we narrow our study to those programs that run on top of a VM because most of modern object-oriented languages (*e.g.*, Java, Python, Ruby, JavaScript, C#) are VM-based. This section proposes a dissection of a high-level language application runtime and introduces the termi-

nology used during the rest of this dissertation. We made this dissection with the objective of understanding the relationship between the software elements. We believe that understanding these relationships is important in the context of this thesis. Figure 1.1 shows a schema of this dissection.

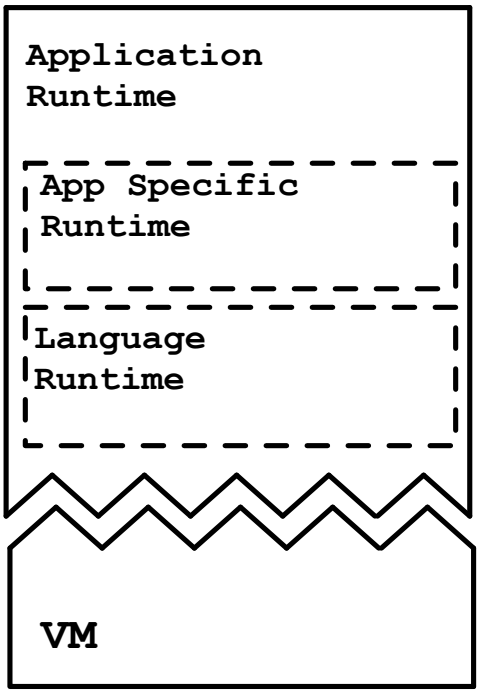

Figure 1.1: **Dissection of a running program.**

An application runtime is the set of software elements that constitute an application during its execution. An application runtime includes software elements that describe the application's structure during its execution, such as its libraries, classes and methods, and elements related with the application's execution, such as threads, the execution stack with its activation records and created objects. We identify inside an application runtime two main components: the *application specific* runtime contains the elements that are part of the application and the *language* runtime contains the elements that belong to the language, common to all applications.

A Virtual Machine (VM) provides execution support for the application runtime. The VM imposes an *execution model* that defines the semantics of a program *e.g.*, class-based, single inheritance. From the application runtime the distinction between VM and language is usually clear: the language elements cannot see nor access the VM elements. However, the opposite is not true. The VM has complete power over the elements that belong to the application runtime. Moreover, this leads inside the VM to a mixture between language and VM concerns that blurs their relationship. Note that some authors

may include the VM inside a broader runtime system. This is the reason why we use the terminology *application runtime* instead of runtime system and we consider this component separately from the VM.

1.2.1 Language Runtime

The subset of the application runtime that describes the programming language is the *language runtime*. The language runtime includes the set of structures and constructs that describe the language concepts and behavior. This language runtime is the representation at runtime of the model that the language proposes to the developers. For example, Smalltalk and Ruby propose that an application is structured in classes with implicit metaclasses *i.e.*, a class is instance of a (*meta*-)class that describes its behavior. Therefore, they also contain structures that represent the concepts of a Class and Metaclass and control the implicit creation of a metaclass for each new class. Additionally, a language runtime is not only composed of classes but it also includes objects. For example, it contains a table of unique strings or *symbols* used to univocally name objects. A language runtime is usually common to different applications in the same language: it is always present and contains always the same elements.

1.2.2 Application Specific Runtime

Within an application runtime we identify also the *application specific runtime*. The application specific runtime is the subset of the application runtime that includes the elements that belong to the particular application. It contains, in other words, the classes written by the application developer. The application specific runtime follows the model and concepts imposed by the language *i.e.*, it is expressed in terms of the language runtime, and thus coupled to it. For example, the application specific classes of a Smalltalk or Ruby application have an implicitly created metaclass.

1.3 Problem Statement

This thesis focuses on the generation of specialized high-level object-oriented application and language runtimes for languages that run on a VM. In this context, we aim at solving the following problems:

Extending languages. The model proposed by a language is not always simple to change. Changing the concepts provided by a language may involve changes in its runtime or its VM. Often, it requires us to have access to the VM code, and forces us to modify such representation without the proper level of abstraction. Also, the relationship between the

language and the VM, particularly its execution model, often remains unclear. This unclear interface comes from hardcoded assumptions in the multiple VM components. Its clients are exposed to the complexities of the VM internals when it comes to change the application runtime.

Reducing application runtime memory consumption. The application specific part of an application runtime often embeds several libraries and elements from different origins *e.g.*, the language base libraries and third party libraries and frameworks. Reducing the memory consumption by removing unused elements such as classes and methods should ensure that the application works as expected at the end of such process. However, the absence of type annotations in dynamically-typed languages and the broad usage of dynamic features such as reflection makes this a challenging problem.

Then, we pose the following research question:

Is there a general purpose infrastructure that supports the creation of specialized application runtimes, particularly to extend a language model and/or reduce its memory consumption?

We would like to have an infrastructure featuring safe application runtime manipulation, and allowing both the specialization and extension of application and language runtimes.

1.4 Contributions

To solve the stated problems, we make the following claim:

"Runtime virtualization and reification support the generation of specialized application and language runtimes, particularly to extend the language model and to discard unused runtime elements."

First-class runtimes should provide a clear and high-level VM-Language interface. This interface must hide the internal details of the VM complexities and allow us to easily manipulate the elements inside an application runtime. At the same time, it should ensure that the VM execution model is honored during such manipulations.

The contribution of this thesis is three-fold. The main contribution is Espell, an *infrastructure for application runtime virtualization*. This infrastructure allows us to manipulate and control a virtualized application runtime through a first-class runtime object, namely an object space. We validate this language virtualization infrastructure by exploring two approaches for application runtime generation:

Language Runtime Bootstrapping. We developed a bootstrapping process for an object-oriented high-level language. Bootstrapping is an explicit process that describes how a language runtime is created using the same language that it generates at the end. Bootstrapping provides us with full control on what are the elements installed in the application runtime at the end.

Application Runtime Tailoring. We developed a novel dynamic application runtime tailoring approach named Run-Fail-Grow (RFG). RFG starts with an empty application runtime and it installs elements inside it as they are needed during at runtime. With RFG we can create a specialized runtime that contains only the used elements of an application.

1.5 Thesis Outline

The rest of this dissertation is structured in four main parts that describe our contributions. After these four main parts, a final part presents the conclusive chapter of this thesis and several appendixes with complementary information for the reader of this thesis.

1.5.1 Part I: State of the Art

Chapter 2 presents the state of the art in the manipulation of application runtimes. This chapter analyses and classifies related work in three categories. Reflection models allow programs to ask about and act upon themselves to modify themselves and their own models. Metacircular runtimes and VMs bring the benefits of high-level languages to the development of VMs. Finally, application virtualization techniques allow certain grade of manipulation on a program from its outside.

Chapter 3 presents the state of the art in techniques to automatically tailor applications by extracting used code. We identify four different kinds of related work in this context. Dedicated platforms are general purpose programming platforms that are built with the purpose of minimizing memory consumption. Static tailoring techniques use static program information such as type annotations and class/method names to select used code. Dynamic techniques will benefit from the dynamic information that is available during a program's execution, such as the objects' concrete types and their references at runtime. Finally, hybrid techniques use a mixture of dynamic and static approaches.

1.5.2 Part II: Espell

Chapter 4 proposes Espell, a model for application runtime virtualization. In Espell a virtualized application runtime resides inside an object space. Externally, a hypervisor manipulates the internal state and controls the execution of the object space.

Chapter 5 presents the implementation details of a prototype implementation of Espell in Pharo. It describes how objects from the object space and the hypervisor can co-exist and communicate safely.

1.5.3 Part III: Bootstrapping

Chapter 6 presents a bootstrapping process based on Espell for object-oriented languages. This chapter starts by defining the concept of bootstrap, describes its challenges and presents how Espell solves them.

Chapter 7 presents the experiments we conducted to validate our bootstrap process. This chapter describes the bootstrap of four different languages on top of our infrastructure.

1.5.4 Part IV: Tailoring

Chapter 8 presents run-fail-grow (RFG), a tailoring technique based on runtime information. RFG extracts only the code that is effectively executed. Tornado, our implementation of RFG benefits from Espell to perform transparently for the application.

Chapter 9 presents the experiments we conducted to validate our tailoring process. This chapter describe the different case studies we used and how Tornado performs in each of them. We compare our results with Pharo's official distribution and other two dedicated platforms. We observe that Tornado can aggressively remove unused code.

1.5.5 Part V: Conclusion

Chapter 10 concludes this dissertation. It sums up the contributions of this thesis and presents several lines of future work.

Appendix A presents a list of the publications that came out of the work on this thesis.

Appendix B. briefly presents Pharo's syntax, important for the understanding of the code snippets appearing in the thesis.

Appendix C. lists the specific results of our bootstrapped languages. We list in here the classes and methods that are present in each of the languages we built.

Appendix D. lists the specific results of our tailored applications. We list in here the code units (methods and classes) that were automatically selected by Tornado to belong to the specialized version of several applications.

Part I

State of the Art

Application Runtime Manipulation

Chapter 2

Contents

Generating an application runtime requires the basic ability to manipulate such runtime *e.g.*, create and modify classes and methods; instantiate objects; begin and stop threads in it. This chapter explores the state of the art on application runtime manipulation. We identify three main concerns that arise when we attempt to modify an application runtime (Section 2.1): how easily a change can be applied to a language runtime, how coupled are the application runtime and VM concerns, and what is the existing tool support to modify such application runtime.

Based on those concerns, we defined a criterion to evaluate existing application runtime manipulation solutions (Section 2.2). We also classified the different techniques for application runtime manipulation in three main categories:

Reflection models. The literature presents several language runtime manipulation solutions based on metaprogramming and *reflection* (Section 2.3). These metaprogramming models allow either the modification of the language interpreter semantics or the scoping of modifications with one main objective: performing such modifications safely *i.e.*, without breaking the application runtime they are running on.

Metacircular runtimes. Metacircular runtimes aim at easing the modification of a runtime by means of simplifying the language-VM relation for development (Section 2.4). This simplification comes usually by expressing a VM in the same language it interprets at the end. Metacircular runtimes mostly focus on the simplification of complexity in VM

technology and not on the manipulation of the language runtime elements.

Virtualization techniques. Virtualization techniques applied to application runtimes are used for application control and manipulation (Section 2.5). Application co-existence allows the modification of an application without affecting others. Application manipulation provides control on an application's life cycle and its internal representation.

2.1 Concerns of Application Runtime Manipulation

We identify three main concerns when we aim at changing or replacing an application and language runtime. First, how flexible its language runtime is to be changed and adapted without running into inconsistent states. Second, how clear is the separation between the VM and language concerns to extract the language concerns. Finally, what are the abstraction level and the tool support needed to manipulate an application runtime.

Flexibility to Safely Change a Language Runtime

The VM is often the component in charge of initializing the application runtime and particularly the language runtime. This decision is indeed practical as the VM can safely initialize the language structures, solve the language bootstrapping issues avoiding recursions [KdRB91] (*e.g.*, create the first class or the first string without a class) and ensure the created language runtime complies to its execution model. This coupling is indeed necessary to run a program but does often remain hidden in hardcoded assumptions. A consequence of this is that the VM fixes the initial structures of the language runtime. Figure 6.1 illustrates this problem: Ruby's initial class hierarchy is imprinted inside the VM code, fixing BasicObject as the top superclass and followed by Object, Module and Class respectively[1].

A second problem arises as the VM includes code to manipulate objects that belong to the application runtime, introducing a duplication between the VM code and the language code. To illustrate this second problem, let us consider the code to the left of Figure 2.2 that creates a Dictionary object (a hash map object) in Smalltalk. If our language runtime is defined by a Dictionary object keeping *e.g.*, a map of global objects, we must execute that code to create the corresponding instance during the language initialization. However, since the language runtime and the VM are in the middle of their initialization, the VM cannot execute this code as it is, and thus it cannot enforce its

[1] Taken from the version 2.1 of the Ruby VM in `http://svn.ruby-lang.org/repos/ruby`

```
void Init_class_hierarchy(void) {
    rb_cBasicObject = boot_defclass("BasicObject", 0);
    rb_cObject = boot_defclass("Object", rb_cBasicObject);
    rb_cModule = boot_defclass("Module", rb_cObject);
    rb_cClass =  boot_defclass("Class",  rb_cModule);

    rb_const_set(rb_cObject, rb_intern("BasicObject"),rb_cBasicObject);
    RBASIC_SET_CLASS(rb_cClass, rb_cClass);
    RBASIC_SET_CLASS(rb_cModule, rb_cClass);
    RBASIC_SET_CLASS(rb_cObject, rb_cClass);
    RBASIC_SET_CLASS(rb_cBasicObject, rb_cClass);
}
```

Figure 2.1: **Code of the Ruby VM that initializes the class hierarchy (excerpt).** The VM code fixes the language class hierarchy.

own invariants. Production VMs will provide an alternative low-level representation of the same code respecting the same invariants, exemplified at the right of the same figure. This introduces a redundancy: the VM and the language have different code to honor the same invariants.

```
Dictionary class>>new: n
    ^ self new initialize: n

Dictionary>>initialize: n
    "Initialize array to an array size of n"
    array := Array new: n.
    tally := 0
```

```
void* createDictionaryWithSize(int n){
    void* dictionary, internalArray;
    dictionary = instantiate("Dictionary");
    internalArray := instantiate("Array", n);
    setInstanceVariable(dictionary, 1,
      internalArray);
    setInstanceVariableInt(dictionary, 2, 0);
    return dictionary;
}
```

Figure 2.2: **Code to create a** Dictionary **object.** To the left, the code is written in Smalltalk and used by the application runtime. To the right, a C function duplicates this behavior at the VM level.

Once the application runtime is initialized, reflective languages [Smi84, DM95] such as LISP or Smalltalk provide means to modify themselves at runtime. These languages are based on a reflective architecture presenting the notion of causal connection [Mae87]: a link between a programming element and its representation that keeps both in synchronization. However, as these languages contain meta-circular definitions in their kernel [CKL96], this can result in a metastability problem *i.e.*, a change in the language runtime may introduce a meta-call recursion and make the system unusable [KdRB91].

Finally, snapshot-based languages offer different means to evolve a language runtime. Instead of being reinitialized each time we need it, the state of the whole graph of objects that denotes the running program can be suspended and saved as a snapshot in a file. Later on, the program in this

snapshot can be restarted from the point it was suspended. As a consequence of this persistency property, some snapshot-based languages such as Squeak [IKM+97] or Pharo [BDN+09] do not have the infrastructure to be reinitialized from scratch. Indeed, their current deliverable snapshots are the result of a chain of side effects (updates and migrations) applied to the original Smalltalk-80 snapshot. Thus, even if we can change the application runtime during execution, they cannot ensure a reproducible initial state for their users.

Separation of VM and Language Concerns

The unclear interface between the language and the VM makes it difficult to recognize whether a piece of code in the VM belongs to a VM concern or a language one. This causes undesirable temporal couplings between these two elements and prevents us from extracting and replacing the application runtime by another one. To illustrate this problem, Figure 2.3 shows an excerpt of the JikesRVM[2] [AAB+00]. In this example, the memory manager is initialized in the middle of the initial class loading phase. These memory manager calls are indeed needed to avoid the collection of objects during the initialization. However, it is necessary to call it after some specific classes are initialized and not before.

```
1  private static void finishBooting() {
2      ...
3      MemoryManager.postBoot(); //Memory management
4      ...
5      runClassInitializer("java.lang.Runtime");
6      runClassInitializer("java.lang.System");
7      runClassInitializer("sun.misc.Unsafe");
8      ...
9      MemoryManager.fullyBootedVM(); //Memory management
10     ...
11     runClassInitializer("java.util.logging.Level");
12     runClassInitializer("gnu.java.nio.charset.EncodingHelper");
13 }
```

Figure 2.3: **Code of the JikesRVM that loads and initializes the initial classes of the runtime (excerpt).** The code performing the initial class loading is mixed with the code that initializes the memory manager of the VM.

Snapshot-based languages, on the other hand, show that language initialization and VM initialization can be orthogonal. Indeed, languages such as Pharo, LISP or the V8 flavor of Javascript can restart their system from a snapshot. The VM loads atomically the snapshot and binds it to the cur-

[2]Taken from the version 3.1.3 of the JikesRVM in `http://sourceforge.net/projects/jikesrvm`

rent executing VM. Loading a snapshot clarifies the VM startup steps as it replaces the runtime initialization phase. Moreover, as long as the snapshot satisfies the VM execution model, it can run any language model, as it is the case of Pharo and Newspeak that run on the same VM.

Abstraction Level and Tool Support

When defining the language runtime at the VM level, we rely on the tools and abstractions available to develop the VM to change the language. That means that the language developer uses often low-level abstractions to express higher-level concerns. This *mismatch* impacts on the VM/language developer efficiency, as the tools are not adequate for the task to accomplish. This problem is indeed aggravated with the complexity of such pieces of software, including elements such as the GC or the JIT that cross-cut every VM concern.

Indeed the developer would benefit from the productivity that a high-level language brings. In this sense, reflective languages support naturally the usage of the language abstractions and tools to modify the language runtime. Reflective languages benefit from the reification of the language concepts to be able to change them from the language itself. Languages such as Smalltalk show how debuggers and code browsers can be built using such features

2.2 Evaluation Criteria

This section presents the features we consider relevant to include in a runtime manipulation solution. These features serve as criteria to compare the state of the art we selected in the following sections.

Manipulate Objects. An ideal application runtime manipulation must provide access to objects. It should support class instantiation, and the manipulation of such objects, including access to their fields.

Manipulate Language Elements. An application runtime manipulation solution must support the creation and modification of elements that are part the language runtime *e.g.*, create and modify of classes and methods.

Manipulate Execution Elements. An application runtime manipulation solution must provide support to manipulate elements related to a program's execution *e.g.*, the creation, pausing and resumption of threads; the introspection of such threads to understand a program's execution.

Safety. An ideal application runtime manipulation solution must guarantee that both incorrect modifications cannot be applied, and that correct

modifications can be safely applied without leaving the system in an inconsistent state.

User abstraction level. An ideal application runtime manipulation solution must provide its user with the possibility to express its manipulations in a high-level language. In such a way, users can benefit from the abstractions and expressiveness of such a language.

API abstraction level. The API of an ideal application runtime manipulation solution must provide the abstractions to manipulate the application runtime in terms of the runtime's constructions.

Separation of concerns. The solution should clearly separate VM and language concerns to avoid transmitting to the language developer the complexities of VM technology such as the GC or the JIT compiler.

In the following sections we explore and compare three different categories of solutions that pursue different kind of application runtime manipulation. This comparison is based on the criteria defined in this section.

2.3 Reflection and Metaprogramming Models

Reflection is the capability of a program to reason about and act upon itself [Mae87]. Typically we distinguish two forms of reflective access: structural and behavioral [MJD96]. Structural reflection is concerned with the static structure of a program, while behavioral reflection focuses on the dynamic part of a running program. Orthogonally to the previous categorization we distinguish between introspection and intercession. Introspection refers to the access to a particular reified representation of a program's element, whereas for intercession we mean to alter the reified representation.

Structural Reflection. Structural reflection refers to the access to the static structure of a program. A typical example[3] is to access the class of an object at runtime.

```
1 'a string' class.
```

An example of structural intercession is to reflectively modify an instance variable of an object.

```
1 aCar instVarNamed: #driver put: Person new.
```

[3]Throughout this dissertation we use Pharo code examples. The syntax and the basic semantics are explained in Appendix B

Behavioral Reflection. Behavioral Reflection means to directly interact with the running program. For instance this includes reflectively activating a method.

```
1 #Dictionary asClass perform: #new
```

Another more complex example to dynamically switch the execution context and resend the current method with another receiver.

```
1 thisContext restartWithNewReceiver: Object new
```

Accessing the receiver of the current method through the execution context is an example of behavioral introspection.

```
1 thisContext receiver.
```

There is not always a clear separation between the two types of reflection. For instance adding new methods requires structural reflection. At the same time, adding a new method alters the future program execution, implying that it is also behavioral reflection. Typically we see that behavioral reflection stops at the granularity of a method. For instance in Pharo by default it is not possible to directly alter the execution on a sub-method level [DDT06].

Reflection provides by default most of the characteristics that we need for runtime manipulation, such as the ability to manipulate the execution of a program or the objects themselves. In the following sections we present several existing models implementing reflective behavior, and for each of them we put emphasis on their weaknesses and strengths.

Tower of Interpreters Model

The original model of reflection as defined by Smith [Smi82] is based on meta-level interpretation. A program is interpreted by an interpreter, which is interpreted by a meta-interpreter and so on, possibly *ad infinitum*. This leads to a tower of interpreters, where each level defines the semantics of the program (interpreter of application) it interprets (Figure 2.4).

The tower of interpreters presents a model where one can define and redefine the semantics of the program it is executing. We can modify the behavior of our program (in the floor zero) by jumping one level above it in the tower and modifying the interpreter running in that floor. In the same sense, we can jump one level above this interpreter to change also its behavior and so on. This allows the *indirect modification* of a program's behavior *i.e.*, a change in an interpreter in a level *n* changes the behavior of the interpreter in the level *n - 1*, which impacts on the interpreter below it and so on, up to the base level.

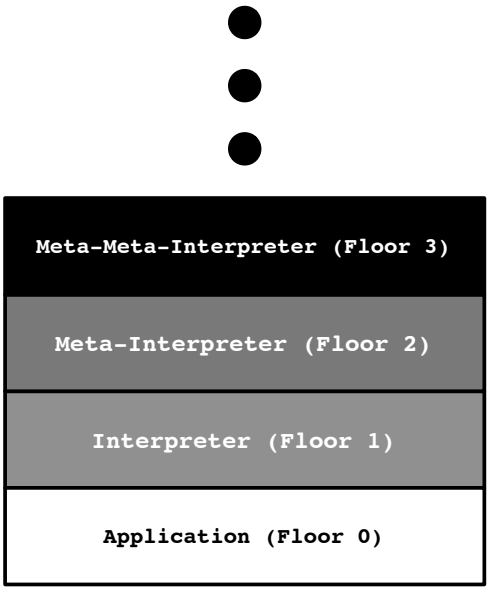

Figure 2.4: **The theoretical tower of interpreters model.** Each floor interprets the floor below itself.

Smith's tower of interpreters model is flexible and coherent regarding the manipulation of the reflective behavior of a program. The tower does not present a limit on the amount of interpreters we can stack, presenting a problematic infinite potential. This idea collides with the non-infinite resources in current hardware. On one hand, limited memory prevents us to have a tower of infinite interpreters running at the same time. On the other hand, above certain limit of interpreters, this approach becomes too slow and impractical, as discussed in [MJD96, MDC, MDC96]. We will show in the following subsections other reflective models that try to overcome these deficiencies while trying to keep some of the good properties of this model.

Black

Black is a reflective language based on Scheme that mimics the infinite tower of interpreters with the goal to make it practical. Its model is based on the same idea as the reflective tower: the base level is interpreted by an interpreter, which is interpreted by a meta-interpreter and so on. The main difference between the original tower of interpreters and the model presented by Black is that the latter avoids the infinite regression, making the model practical in finite resource machines.

Black avoids the infinite regression by limiting the real levels of interpretation: there is only one level of interpretation. For the rest of the levels, Black introduces a difference between directly-executed code in contrast with inter-

preted code. Directly-executed code is code that is executed by the machine, where no interpretation steps are involved. Then, the base-level application is the only interpreted code in the application. The rest of the tower, including the first interpreter, is implemented and executed directly in machine code (Figure 2.5).

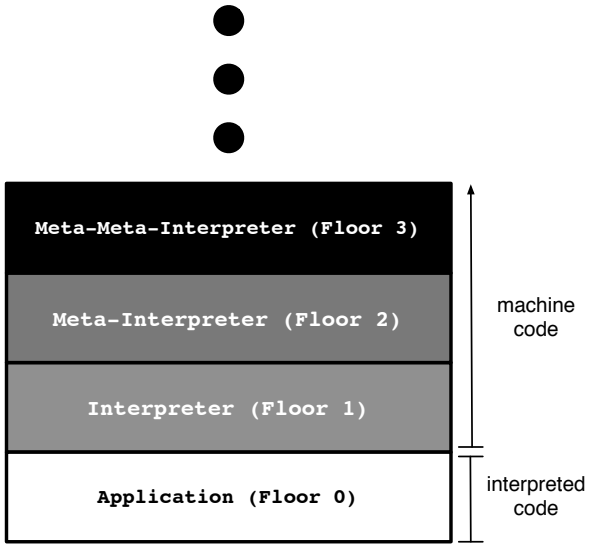

Figure 2.5: **Black model of interpreters.** Only the application level is interpreted. The levels above are directly executed on the machine.

By limiting the levels of interpretation, the model presented by Black forbids *indirect modification*. Changing the interpreter in a level *n* above the first level does not impact any more the interpreters below it, as they are directly-executed in the machine and not by the modified interpreter. Black supports, however, the modification of the first level of interpretation with the introduction of hooks inside the machine code. A hook detects whether a function in the interpreter (written in directly-executed code) is changed, and interprets it by a meta-level interpreter written also in directly executed code. Hooks degrade the performance in comparison with a non-hooked interpreter, but it enables to change and specialize the behavior of the directly-executed code.

Reflectivity: scoping reflection in Reflective Architectures

To enable reflection in mainstream languages such as Java, Ruby or JavaScript, the tower of interpreters is replaced by a reflective architecture [Mae87]. Instead of relying on a stack of interpreters interpreting each the level below it, a reflective architecture relies on the idea of *causal connection i.e.*, the programming language incorporates structures that represents

aspects of itself (*e.g.*, classes, objects), in such a way that if one structure changes the aspect it represents is updated accordingly, and vice-versa.

In languages presenting reflective architectures, reflection is controlled by meta-objects that share a same environment with the objects they reflect upon. One problematic corollary of this is that meta-objects rely on the same code and infrastructure as the objects they reflect upon *e.g.*, if a class stores its subclasses in a Set, changing the implementation of Set impacts the reflective behavior in addition of the base-application behavior. Therefore there is a risk of infinite meta-recursion when the meta-level instruments code that it relies upon (Figure 2.6).

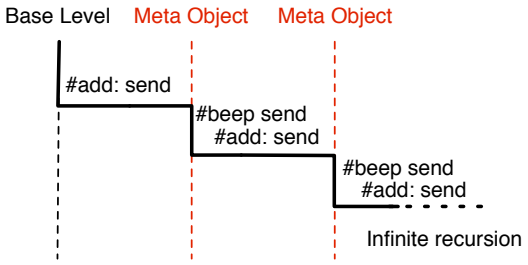

Figure 2.6: **Meta level recursion in reflective architectures.**

Denker et al. partially solve this problem in Reflectivity [DSD08]. Reflectivity is a reflective framework that avoids meta-recursions by tracking the degree of metaness of the execution context. In each reflective call, the MetaContext object is activated and it accounts for the meta-level jump. Likewise, when the reflective call returns, the MetaContext is deactivated. Using the accounted meta-level jumps of the MetaContext, meta-objects do only reflect on objects of a lower metaness (and not greater or equal metaness). Thus, it simulates the semantics of an infinite tower of distinct interpreters while there is only one of them that scopes the meta-operations using the accounted meta-level (Figure 2.7).

Reflectivity succeeds to modify and scope behavioral reflection for different meta-levels inside a single interpreter reflective architecture. However, it does not provide support to fully change the language semantics (residing in the VM) or to perform structural reflection.

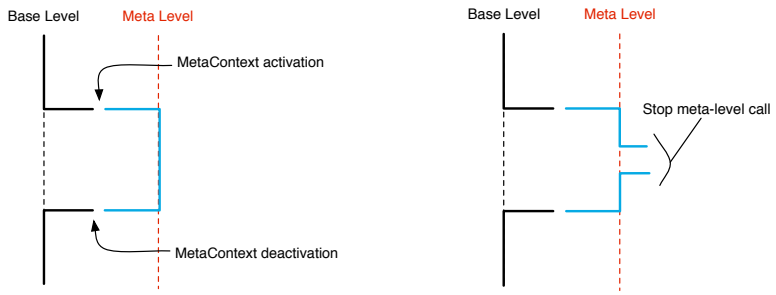

Figure 2.7: **Meta level jump using reflectivity.**

2.4 Metacircular Runtimes and VMs

The increased complexity of the VMs leads to more novel approaches on how to build VMs and therefore, their runtimes. Metacircular VMs are VMs programmed in the same language they support in the end *e.g.*, a Java VM written in Java or a Smalltalk VM written in Smalltalk. This approach is based on the principles of *high-level low-level programming i.e.*, expressing low-level concerns using high-level languages [FBC+09]. Metacircular VMs benefits from the abstraction power and tooling of a high-level language to manipulate their own VMs. This also means that during the build-time of a metacircular VM, we can express the manipulations of its application runtime in terms of the high-level language.

However, these projects are biased towards VM building techniques and not to the manipulation of the application runtimes that run on top of them. We can see this in the fact that most of the high-level manipulations inside a metacircular VMs do not survive the VM generation *i.e.*, once the VM is built we cannot access its high-level representations any more. Even if we do not focus on the modification of VMs, in this thesis we briefly study metacircular runtimes and VMs with the objective of understanding more concretely the benefits of their high-level low-level approach.

Squeak Smalltalk VM

The Squeak VM [IKM+97] is an early open source metacircular VM for the Smalltalk language. Its core building system is still in active use for the Cog VM[4] which introduces a JIT compiler. The Cog VM is used as default by the Pharo[5] programming language. The Squeak VM is developed using a Smalltalk subset called Slang that is exported to C to be compiled to the final VM binary.

[4] `http://www.mirandabanda.org/cogblog/`
[5] `http://pharo.org/`

Slang is limited to functionalities that can be expressed with standard C code. Slang in this case is mostly a high-level C preprocessor. Even though Slang basically has the same syntax as Smalltalk, it is semantically constrained to expressions that can be resolved statically at compilation or code generation time and are compatible with C. Hence Slang's semantics are closer to C's than to Smalltalk's. Unlike later metacircular frameworks, the Squeak VM uses little or no compile-time reflection to simplify VM designs. However, class composition helps to structure the source code. Next to the Slang source code which accounts for the biggest part of the interpreter code, some operating system related code and plugins are written in C. To facilitate the interaction with the pure C part, Slang supports inline C expressions and type annotations.

A great achievement of the Squeak VM is a simulator environment that enables programmers to interact dynamically with a simulated version of the running VM. The simulator is capable of running a complete Squeak Smalltalk image including graphical user interface. This means that programmers can change the sources of the running VM and see the immediate effects in the simulator. The VM developer has complete access and control to the VM internals and the application runtime it contains. It can, for example, change any object and class inside the simulated application runtime. However, to apply such a change it depends on a memory-oriented interface *i.e.*, object modification is achieved by a Smalltalk interface that provides C-like abstractions such as pointer arithmetics. Additionally, once the VM is generated, this low-level interface disappears and it is not accessible for the developer anymore.

Jikes: High-level low-level Programming with MMTK

Jikes (formerly Jalapeño) is an early metacircular research VM for Java written in Java [AAB+00]. The Jikes VM features several different garbage collectors and does not execute bytecodes but directly compiles to native code. With metacircularity in mind Jikes does not resort to a low-level programming language such as C for these typically low-level VM components. Instead they are written in Java as well using a high-level low-level programming framework. The Jikes VM had performance as a major goal, hence direct unobstructed interaction with the low-level world is necessary using a specialized framework.

Frampton et al. present a high-level low-level framework packaged as `org.vmmagic`, which is used as a system interface for Jikes. This framework introduces highly controlled low-level interaction in a statically type context. This framework provides a memory-oriented API to manipulate run-

time entities at VM generation time, which is used to implement VM concerns. Once the VM is compiled to native code, the interface exposed by the `org.vmmagic` framework is also compiled into native code and not accessible from Java programs executed on the top of the Jikes VM.

Maxine Java VM

Maxine is a metacircular Java VM [WHVDV+13] focused on an efficient developer experience. Typically VM frameworks focus on abstraction at the code-level which should yield simpler code and thus help reducing development efforts. However, in most situations the programmer is still forced to use existing general purpose tools for instance to debug the VM. In contrast, the Maxine VM provides dedicated tools to interact with the VM in development. Maxine uses abstract and high-level representations of VM-level concepts and consistently exposes them throughout the development process.

The Maxine project follows an approach where reflection is used at compile-time *i.e.*, once the VM is generated, the metacircular property of the VM is lost. However, during development Maxine tools provide a live interaction with the complete state of the running VM artifact while debugging it.

Klein VM

Klein[6] is a metacircular VM for the Self programming language that has no separation into VM and language [USA05]. A main difference between Klein and the already seen metacircular VM projects is that the reification of the VM-level elements survives the code generation or compilation time. Instead the VM structures are exposed to the language as Self objects, exposing them to the language and thus allowing their manipulation from the application runtime. Klein also supports advanced mirror-based [BU04] debugging tools to inspect and modify a remote VM.

Additionally to the advances in reflection and metacircularity, Klein focuses on fast compilation turnarounds and incremental modifications for a responsive development process. For comparison, the generation process of a Squeak VM takes minutes on modern hardware. The long compilation time prevents the Squeak VM developer to perform fast and incremental development cycles on the VM.

Development on the Klein VM stopped in 2009 and left the Klein VM in a fairly usable state. Yet, it proved that it is possible and build a language-runtime without the classical separation of the language-side and the VM.

[6] `http://kleinvm.sourceforge.net/`

From the literature presented about the Klein project we see a strong focus on the improvements of the development tools.

2.5 Language Virtualization Techniques

The most related family of work in virtualization are approaches like Xen [Chi07]. Xen is a Virtual Machine Monitor (VMM) that allows one to control and manage VMs in a high performance and resource-managed way. This approach targets the virtualization of full and unmodified operating systems (OSs), to facilitate their adoption in industrial/productive environments. They rely on support from the hardware platform, and in some cases from the guest OS, concentrating themselves on performance and production features.

Operating System virtualization technology is characterized by the existence of a *hypervisor* (named after the Operating System *supervisor* that controls the OS processes). The hypervisor is the VM component that allows one to observe or control the internals of one or many VMs. A VM hypervisor gives us, amongst others, the following services:

Co-location. Co-location is the ability to have co-existing applications on top of the same virtual machine. Co-located applications can use shared memory to communicate efficiently as they reside in the same operating system process.

Resource control. VMs should control how the different resources of their applications are used. However, state of the art VMs only control their consumed memory with the usage of a memory manager. They do not perform in general any control in other kind of resources such as CPU or energy consumption.

Security. VMs should control how applications access sensitive information such as files and network connections or execute potentially dangerous operations such as system calls.

Application mobility. As applications are portable, they should be easily migrated between different VMs also at runtime. Application mobility provides support for resource re-allocation.

High-level languages abstract the developer from the complexities of the underlying machine by running on top of a language VM. A language VM provides a language with an execution model closer to its semantics as well as several services such as automatic memory management or cross-cutting optimizations. Language VMs also provide portability *i.e.*, a program can run on different operating systems and hardware architectures because the VM

abstracts it from the underlying particular details. Although these language VMs are indeed *Virtual*, state of the art production-ready VMs do not provide by themselves the typical advantages of virtualized operating systems such as co-location, resource control, mature security or application mobility support.

With the objective of doing application runtime manipulation, we study in the following subsections techniques that to our understanding are virtualization-related techniques applied to application runtimes: application co-existence and application manipulation

Class Loaders

In Java, application co-existence can be achieved to some degree with class loaders. A class loader is a first-class entity responsible for dynamically loading classes: creating their runtime representation, loading their methods and linking their class references [LB98]. A class loader remembers all classes it loaded, and it is responsible for loading all classes they depend on. Class loaders define namespaces: different class loaders can load classes with the same name. These classes will be isolated in the sense that they will not be visible to the others statically.

Class loaders can be specialized and extended to provide custom behavior. For example, Fong et. al. [FO10] use the class loading mechanism to enforce scoping rules and determine the visibility of names in various region of the program. They allow the user to control untrusted namespaces and classes and they have defined a language to define security policy. Jensen et. al. [JLMT98] provide a formalization of the class loader with the means to enforce security. They also use a bytecode verifier on class loading to check if a class' bytecode doesn't try to perform overflow or underflow operations.

In the context of this thesis, we consider the class loader isolation mechanism as it can be used for application co-existence. Different versions of the same application can be loaded by different class loaders and be running at the same time. Class loaders present however a main limitation on the so called *bootstrap class loader*: the literature does not explore the means to load (and use) different versions of the main runtime classes of the language. Moreover, Java's bootstrap class loader is often implemented natively, and as such, we have no control on it from the language.

Notice additionally that class loaders provide only a load mechanism. Once we have several versions of the same application, it does not support the means to manage changes or updates of these applications. In this regard, the OSGI [OSG] architecture implementations often make use of class loaders to load classes into separately isolated components and manage them in

a higher level way.

Changeboxes

Changeboxes [DGL+07] is a change model designed to encapsulate and scope changes. Its main purpose is to allow several versions of a system to co-exist at runtime *i.e.*, the existence in the same environment of different versions of the same classes and methods. In changeboxes, a *changebox* is a first-class entity that encapsulates changes made on the language elements (*e.g.*, classes and methods) and an executable version of the system with its changes applied. The system can contain many changeboxes at the same time, and applications can be scoped to run within different changeboxes. This notion of dynamically scoping an application to a changebox allows one to have co-existing environments (*e.g.*, testing, development, production), increasing the developer's efficiency. Furthermore, it eases application update and migration to new versions, and reduces its update down-time as the application does not have to be stopped to be updated.

A Changeboxes prototype was developed in Smalltalk and its scoping mechanisms were implemented as follows:

Message send interception. Message sends can activate different methods, within different changeboxes. A MethodWrapper [BFJR98] is placed instead of the method that has multiple versions, and it delegates the execution to the method that corresponds to the currently valid changebox.

Class access interception. Smalltalk resolves class names at compile time, inserting a reference to the given class inside the method's literal array. However, accessing a class yields different class objects within different changeboxes. To resolve this, class accesses affected by a changebox are postponed until runtime, and the code is recompiled in such a way: instead of putting the class inside the literal array, the class is dynamically looked-up from the class table when it is accessed.

The Changeboxes model proves sound to update and migrate application and framework classes. However, it has the main drawback of not affecting critical classes in the system. The Changeboxes prototype does not work on language runtime classes such as Array or CompiledMethod as the underlying infrastructure (the VM) restricts the system to the existence of only one of them at the same time. The changes model does not provide a solution for this problem, as it focuses on application code update, leaving this as an open problem.

Caja\Cajita - Object Capability Languages

Caja [MSL+08] is an object-capability language [Lev84,MRC03,Spo00] subset of Javascript that pursues the safe co-existence of isolated Javascript scripts. Caja was conceived in the context of the web, where untrusted scripts can be loaded in any webpage and profit from any data available in the webpage. Caja defines a safe Javascript subset that removes elements from the language such as with or eval because their semantics are "strange" and in some cases unpredictable. Caja includes Cajita, a subset of Caja without the this keyword. Caja is meant for transforming and migrating already existing Javascript code, while Cajita is meant for newly written code.

Caja works with a combination of static and dynamic techniques. First, a static verifier checks and transforms Caja code into sanitized Javascript. This sanitized Javascript contains runtime checks that complement the static verifications. All these changes are meant to avoid exploits and vulnerabilities from untrusted sources. Caja also adds some new features to the Javascript libraries with support to freeze objects and turn them immutable.

To allow the safe co-existence of several scripts, Caja removes the Javascript global environment and replaces it by modules. Caja Modules co-exist transparently and can only access each other's data through an explicit and verified interface.

KaffeOS

KaffeOS [BHL00] is a multi-application Java Runtime System that supports application co-existence, isolation and resource accounting. The KaffeOS runtime allows the isolated co-existence of Java applications so they cannot access each other's data, nor interfere in their execution. For this, the KaffeOS VM performs CPU and memory accounting and prevents an application to starve others by consuming more resources than expected.

KaffeOS adds the process abstraction in the Java language, as in the sense of a process for an operating system. Each KaffeOS process owns a separate memory region (a process heap) where its objects are allocated. Shared objects reside in special shared memory regions. A *kernel heap* makes a distinction between code that runs in user mode or kernel mode. Regarding process communication, cross-process references become cross-heap references and are handled specially by the VM. Resources are accounted and controlled at the VM level for each process, so no process starves other processes.

KaffeOS lets several Java applications run side by side taking care of hardware resource consumption in addition of application's data. However, this solution is still limited with respect to base language classes. For example, the class java.lang.Object must be shared between different processes to al-

low their communication because of the type system checks.

JVMTI

The Java Virtual Machine Tool Interface (JVMTI) [JVM] is the interface offered by the Java VM for its manipulation and control. Originally called Java Platform Debugger Architecture (JPDA) and used in the context of debugging, it was extended to support other use cases such as monitoring and profiling. JVMTI exposes C functions to manipulate a Java VM (Figure 2.8). The JVMTI client, so called an *Agent*, queries and controls the given JVM through this interface. The manipulated JVM and an agent share the same operating system process. JVMTI agents are meant to be written in C, to be as compact as possible and allow maximal control with minimal intrusion.

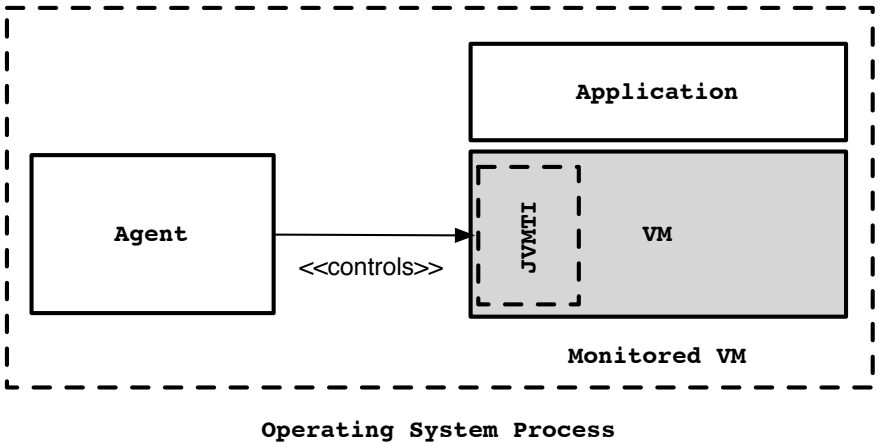

Figure 2.8: **JVMTI Architecture.** An agent controls a JVM though JVMTI. Both share the same operating system process.

JVMTI provides introspection and some limited intercession facilities at the VM and language levels. In particular it provides memory and heap management, thread control, execution stack manipulation, object and class manipulation and breakpoint support. JVMTI is used mainly for debugging, monitoring and analysing purposes, particularly profiling and thread analysis.

In the context of this thesis, we identify in JVMTI's architecture a similarity to a virtualization system. The Java VM takes the place of the virtualized operating system and the agent is its hypervisor.

MVM: a Multi User Virtual Machine

The Multi-user Virtual Machine [CDT03, CD01] is a general purpose virtual machine for the Java language that allows the co-existence of different appli-

cations, potentially from different users. Each application running on top of the MVM is an *isolate* based on the Java Application Isolation API specification [JCP]. This Java Application Isolation API defines a uniform mechanism to control the life-cycle of Java applications.

Many isolates co-exist not interfering with each other, as they believe they own their private JVM: the runtime is modified, so state is not shared between them by default. MVM allows several communication mechanisms to securely communicate between isolates: from standard mechanisms such as sockets, up to *links*, a low-level isolate-to-isolate mechanism introduced by the Isolate API.

MVM can run unmodified Java applications. The main difference between MVM and other co-existence solutions is that MVM-aware applications can use its high-level API to control the life-cycle (*e.g.*, creation, suspension, resuming and and termination) and the available resources of other isolates. Notice that differently from JVMTI, this control API is available to the applications running on the JVM and not only to native agents.

Safe-Tcl

Safe-Tcl [LOW97, Bor94] is a variation of Tcl whose main purpose is the execution of Tcl scripts in a safe environment, with restricted permissions and attributions. Safe-Tcl achieves this by using co-existing *twin interpreters i.e.*, a normal Tcl interpreter (the master interpreter) can invoke another interpreter and specialize its behavior. Both the master and child interpreters run isolated from each other (Figure 2.9).

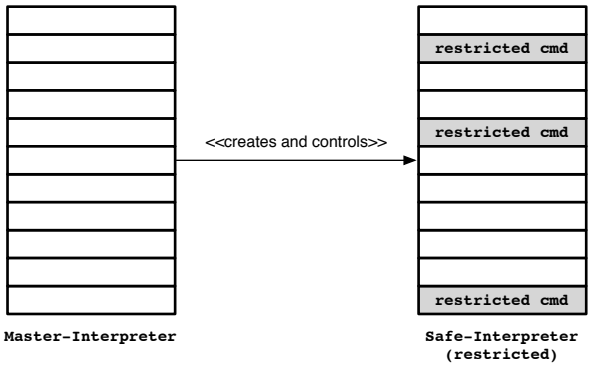

Figure 2.9: **SafeTcl Twin Interpretation.** The Master interpreter creates and configures a safe interpreter. The safe interpreter, with modified commands, will run the sandboxed Tcl application.

The master interpreter modifies the behavior of a safe interpreter by pro-

viding a security policy. A security policy grants or removes privileges to the scripts executed on an interpreter. Commands can be aliased so the untrusted interpreter call an aliased method and the command is fully implemented by a trusted interpreter. The twin interpreter is an interesting solution to have co-existing independent application runtimes where one has the power to manipulate the second one at the level of the interpreter.

2.6 Conclusion and Summary

In this chapter we studied the state of the art of application runtime manipulation techniques. We focus this study on three main concerns: the flexibility to change an application and language runtime, the mixture of low-level and high-level concerns and the tool support to do it. Based on these three concerns, we defined a criteria to present, evaluate and compare different work related to application runtime manipulation. We organized our state of the art in three categories: reflection models, metacircular runtimes and virtualization techniques. Table 2.1 shows how our evaluation criterion match each of these categories.

	Reflection	**Metacircular Runtimes**	**Virtualization Techniques**
Object Manipulation	+	+	~
Language Manipulation	+	+	~
Execution Manipulation	~	+	+
Safety	~	+	+
User Abstraction Level	+	+	~
API Abstraction Level	+	~	+
Separation of Concerns	+	-	+
(+) supported /achieved	**(~) partially supported**	**(−) not supported / not achieved**	

Table 2.1: Classification of application runtime manipulation into three different families of approaches and their evaluation based on the criteria defined in Section 2.2.

First, we studied different reflection and metaprogramming models such as the ones appearing in LISP, Smalltalk and Ruby. Metaprogramming and reflection support the modification of objects and language elements such as classes and methods. The reification of execution-related elements, though not always available, provides these models with the ability to also manipulate the execution: start, stop and alter the state of running processes. Reflection models require special support to safely apply arbitrary changes and not run into metastability issues *i.e.*, modifying an element that is used at the

same time to make such modification can break runtime assumptions. Additionally, as reflection is available from the language, runtime manipulations are expressed in a high-level language. This provides the ability to both express our manipulations with high-level abstractions and to provide a high-level API to perform reflective changes. Finally, reflection support provides runtime manipulation separated from VM concerns, as such support exists in the language.

Metacircular VMs explore the idea of expressing a VM in the same language it supports at the end, under the principles of *high-level low-level programming*. High-level low-level programming aims at enhancing a developer's productivity by using the abstractions of high-level languages to express lower-level concerns. Metacircular VMs have the same power as non metacircular VMs: they can freely manipulate the application runtime that runs on them. They have the support for manipulating runtime objects, language and execution-related elements. Application runtime manipulations can be safely applied as they are atomic for the application runtime being executed. However, the main advantage of metacircular VMs over regular low-level VMs is that they are expressed in high-level languages and thus, we can express our manipulations with the expression power of such language. Not all metacircular VMs provide an API written in terms of the runtime elements we want to manipulate, they do provide instead a memory-oriented API. Finally, metacircular VMs do often mix language and VM concerns, as they do not focus in the application runtime initialization but on the enhancements of VM technology (*e.g.*, the JIT compiler, the GC).

Virtualization techniques applied to application runtimes provide application co-existence and application runtime control. Application co-existence is used mainly with isolation purposes by allowing one to load several applications (potentially several versions of the same application) to run at the same time and be able to communicate. These application co-existence solutions are provided to the high-level language developer. Application control solutions provide mainly support to control an application's life cycle: start, pause or stop it. They often provide an API expressed in terms of such elements, hiding the low-level details of such manipulations. Particularly JVMTI provides fine grained manipulation and control covering also object and language elements manipulations. Its main drawback is however that JVMTI agents are expressed in a low-level language, limiting the application of our solutions.

From this, we can state that our solution should have the following properties:

High-Level. Expressing application runtime manipulations in a high-level language provides tooling support and a better level of abstraction for

the developer of such manipulations.

Metaprogramming support. Metaprogramming supports generally all kinds of manipulations that we desire for our solution. Making visible execution elements such as threads and the execution stack provides control over a program's execution.

To manipulate a co-existing application runtime. Manipulate a co-existing application runtime provides with the safety and atomicity of making such modifications from the VM as it avoids to modify itself. Additionally, co-existing runtimes provide support to such manipulations from a high-level language runtime.

Chapter 3 Application Runtime Tailoring

Contents

Application runtime tailoring is the specialization of an application runtime so it contains only the elements necessary for its execution and no more. In other words, tailoring is a technique that generates a specialized application runtime so it can better adapt to devices with constrained memory. This chapter starts by identifying the problem that motivates application tailoring: code bloat. Code bloat is mainly caused by unused code units appearing in general purpose libraries. We illustrate this problem through an example (Section 3.1).

Existing application extraction or *Tailoring* techniques build deployment artifacts containing a subset of the original code units inside an application. However, these existing techniques are not completely efficient because they have to overcome many different challenges such as reflection or the absence of type annotations in dynamically-typed languages (Section 3.2). We present an evaluation criterion that allows us to compare these different existing solutions (Section 3.3).

Finally, this chapter concludes by presenting and comparing existing techniques, grouped into four different categories:

Dedicated platforms. Pre-built tailored platforms with specialized VMs and language runtimes (Section 3.4).

Static techniques. Automatic tailoring techniques that depend only on static program information such as the source code and type annotations (Section 3.5).

Dynamic techniques. Automatic tailoring techniques that make use only of runtime information (Section 3.6).

Hybrid techniques. Automatic tailoring techniques that complement both dynamic and static techniques (Section 3.7).

3.1 Problems of Deployment on Constrained Devices

Deployed object-oriented applications often contain *code units* (e.g. packages, classes, methods) that the running application never uses. This problem is more evident and harder to control with third party software. Third party libraries and frameworks are designed in a generic fashion that allows multiple usages and functionalities, while applications use only few of them. Examples are logging libraries, web application frameworks or object-relational mappers.

Unused deployed code units have an undesired impact when targeting a constrained infrastructure. Some devices may constrain applications due to a restrictive hardware such as low primary or secondary memory [Mar12], or even software impositions such as the Android's Dalvik VM restriction to deploy only 65536 methods[1]. Big JavaScript mashup applications have an impact on loading time due to network speed and parsing time on the client. These limitations may forbid the deployment of applications that contain lots of code units, or limit the amount of applications and content a user can have in its device.

Existing solutions to this problem eliminate dead code by extracting used code units of an application, and thus reduce application size in secondary memory and primary memory footprint. The majority of the solutions in the field automatically detect and extract used code units, so called *tailoring*, with static call graph construction as the most dominant technique [GDDC97]. These static approaches present limitations in the presence of dynamic features such as reflection [LWL05], or in the absence of static type annotations. Additionally, they do not allow the user to customize the process of selection to cover different levels of an application's code *i.e.*, if third-party or base-language libraries are shared amongst several applications, a developer may want to extract only the used application specific code and leave the shared ones untouched; another developer may want to apply the process to the whole application.

To clearly show the problem, consider the application using a logging library in Figure 3.1. In this figure, we emphasize in gray the unused code units that can safely be removed. Figure 3.2 shows the code of this application, written in the Pharo language. This application contains a MainApp class with a

[1] According to dalvik's bytecode documentation (`http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`), the source register accepts values between 0 and 65535.

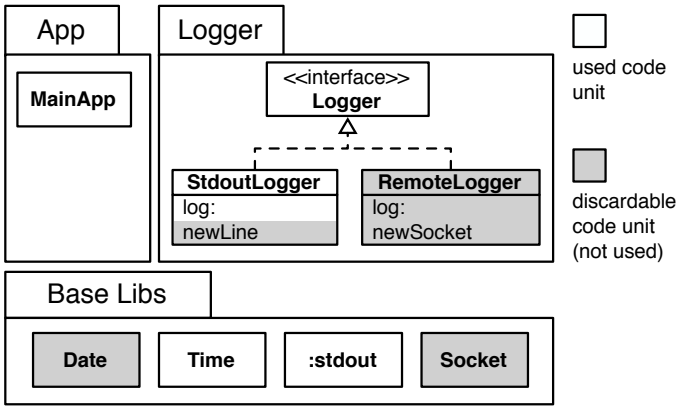

Figure 3.1: **Example of unused code units.** In gray, the unused code units that can safely be removed.

start method, which is the entry point of our application. The start method creates an instance of StdoutLogger and logs the application's start and end. In turn, the StdoutLogger uses the stdout global instance to log in the standard output the current time and the message. To print the time, the StdoutLogger makes use of the Time class from the base libraries of the language. Note that for the sake of clarity, we didn't include in the example all base libraries, though, in modern programming languages they represent a large codebase with several features going from networking to multithreading. For example, Java 8 SE contains 4240 classes[2], and the development edition of Pharo 3.0 [BDN+09] contains 4115 classes and traits.

In this example we can detect the following unused code units, shown in grey in Figure 3.1 and Figure 3.2:

1. The logger library includes two logging classes (StdoutLogger and RemoteLogger). Only the StdoutLogger is used and thus, the RemoteLogger class can be discarded.
2. Since the MainApp class does not use the Socket class nor the RemoteLogger class (the only user of the Socket class), the Socket class can be discarded.
3. No class in the application makes use of the Date class, and we assume for this example that it is not used in the base-libraries either. Then, this class can be safely removed.
4. The method newLine (lines 7-8 of Figure 3.2) of the StdoutLogger class is not used and can be also removed.
5. The StdoutLogger class uses the Time class to print the current time. Then, all code units that are not related to the Time now resolution or

[2] according to the javadoc API

```
1  MainApp>>start
2      logger := StdoutLogger new.
3      logger log: 'Application has started'.
4      "do something"
5      logger log: 'Application has finished'.
6
7  StdoutLogger»newLine
8      stdout newLine.
9
10 StdoutLogger>>log: aMessage
11     stdout nextPutAll: Time now printString.
12     stdout nextPutAll: aMessage.
13     stdout newLine.
14
15 RemoteLogger»log: aMessage
16     | socket |
17     socket := self newSocket.
18     socket nextPutAll: Time now printString.
19     socket nextPutAll: aMessage.
20     socket newLine.
21
22 RemoteLogger»newSocket
23     "...."
24     "creates an instance of socket given some configuration"
```

Figure 3.2: **Code of the example logging application.** In gray, methods not used by the application.

printing (*i.e.*, time arithmetic) could be considered as unused.

We would like to generate a new version of this application not containing these unused code units while keeping the application's behavior. We call this process *application tailoring*.

3.2 Challenges of Application Tailoring

A lot of work exists on the tailoring of statically-typed applications [CGV10, RK02, TSL03, PRT+04, TP01], where type annotations aid in the resolution of which piece of code will be used at runtime. However, static analysis is not an option in the context of dynamically-typed languages or in the presence of meta-programming and reflection [LWL05]. In this context of dynamically-typed and object-oriented programs that may use reflection, we identify the following main challenges in creating tailored applications:

Language Runtime Unused Code Units. As the core point of this thesis, we would like to extract not only application code but also code that belongs to the language base libraries, including its core meta-model. For

example, if metaclasses are not used in a program and the VM execution model does not require them to exist, we could safely remove them. The same could apply to classes and metaclasses that are part of the base-language libraries.

Dynamic typing. Dynamically-typed languages cannot benefit from the most powerful static analysis due to the absence of type annotations. Name-based static analyses (static analyses that build a simpler call graph based only on method names) can be used on them, but are not as efficient. Static techniques to detect code unit usage, such as call-graph analysis, need the support of more dynamic techniques *e.g.*, tracking runtime information, following the application's execution flow, or performing symbolic execution.

For example, an approach just following method names would not eliminate the asString method from the Object or String classes in the following code, as it cannot predict that only the asString method of the Integer class is used.

```
1  MainApp >> start
2      100 print
3
4  Integer >> asString
5      "the code to create a string from an integer"
6      ...
7
8  Object >> print
9      stdout nextPutAll: self asString
10
11 Object » asString
12     ^self class name
13
14 String » asString
15     ^self
```

Polymorphism and inheritance. Polymorphism in object-oriented languages allows a code unit to treat objects of different concrete types in the same way as soon as they share a common interface. Inheritance plays a similar role: any class can extend another class and provide different behavior while sharing a common API. As a consequence, both polymorphism and inheritance make the behavior of a program more difficult to predict by just statically analyzing its code units [TGP89].

Application runtime configuration. Modern applications often contain libraries and frameworks besides their proper code. To make these different code units fit together, applications rely on big configurations. These configurations are usually present in configuration files looked up dynamically by the application. Based on these configurations, the dependency injection pattern is usually used to dynamically set up the

application. This recurrent and standard process for configuring applications implies that static analysis will be inefficient to detect used code units without library-specific knowledge.

For example, an application configuration can be changed at any moment after deployment to use different code:

```
MainApp >> start
    userService := self configuration classById: #UserService.
    users := userClass new getUsers.
    ...

"XML with the application configuration"
<appContext>
    <classes id="UserService" class="LocalUserRepository" />
</appContext>

"Use this one if using a remote repository"
<appContext>
    <classes id="UserService" class="RemoteRestUserRepository" />
</appContext>
```

Reflection. Reflection makes static analysis inoperative by allowing an application to execute unanticipated pieces of code. Any String resulting from a program execution or program configuration can denote a message send, the name of a class to be instantiated, or even a script to be executed. We refer to method invocations as message sends because they represent better from our understanding the dynamic property of the invocation. Reflection is indeed important to cover, since it is a broadly used tool in industrial applications with object relational mappers such as Hibernate[3] or Glorp[4] and web frameworks such as Ruby On Rails[5], Struts[6] or Seaside[7].

Intrusive approaches. Intrusive tailoring approaches require us to change the application source code by using different APIs or particular interfaces. For example, J2ME [Jav] is a tailored version of the Java runtime libraries that presents different APIs than the default Java distribution. Using this kind of solution requires adapting the application's code to this particular infrastructure. This is not a suitable option when facing legacy code and already existing large code bases.

[3] http://hibernate.org/
[4] https://sites.google.com/site/glorpsite/
[5] http://rubyonrails.org/
[6] https://struts.apache.org/
[7] http://www.seaside.st/

3.3 Evaluation Criteria

This section presents properties that we consider the most relevant to evaluate techniques addressing the issue of unused deployment code units.

Base-Library Specialization. A programming language contains several base libraries covering very common and generic code. Not all the code units in these libraries are used in every application. An ideal tailoring solution must tailor base libraries of the language to reduce an application's deployment memory footprint.

Third-Party Libraries Specialization. Applications use several third-party libraries and frameworks covering several aspects of application development such as user interfaces, persistence or publication of services. Third party libraries contain large code bases and many dependencies. Thus, an ideal tailoring solution must tailor third-party libraries and frameworks too.

Legacy Code. An ideal tailoring solution must be applicable on already existing applications and not require modifications on them, independently of the size of their code-base.

Reflection Support. An ideal tailoring solution must handle correctly reflective code and resolve the unanticipated code executions in the same way as the application would do during runtime.

General Purpose Infrastructure. An ideal tailoring solution must produce a version of the application that is able to run on the official production infrastructure (such as the VM) without overhead.

Configurability. An ideal solution for tailoring an application must support many different levels of application. Some applications may not need to tailor base libraries because they are shared with other applications. However, tailoring base libraries may be useful on applications residing alone in constrained devices.

Dynamic typing. An ideal tailoring solution must be applicable and effective in dynamically-typed languages *i.e.*, with no type annotations.

Minimality. An ideal tailoring solution must succeed at selecting the minimal set of elements that a deployable application must contain. That is, it must not contain extra (non used) code units.

Completeness. An ideal tailoring solution must guarantee that the deployable application contains all the elements it needs to run without failure in all its possible execution paths.

In the following sections we explore and compare four different categories of solutions that pursue minimizing application memory footprint

through application tailoring. Such a comparison is based on the criteria presented in this section.

3.4 Dedicated platforms

Dedicated platforms are platforms containing frameworks and/or libraries customized to run under specific circumstances. For example, Java Micro Edition (J2ME) [Jav] is a dedicated version of the Java platform and Cocoa Touch is one for the Cocoa framework[8]. These specialized platforms are reduced platforms to run applications inside mobile and constrained devices. These platforms provide a reduced and fixed set of base libraries defined *a priori* and in a not customizable way. On one hand, this means that applications written for this platforms will never miss a code unit and fail at runtime. On the other hand, completeness comes at the cost of minimality: the final application includes more code units than the ones required at runtime.

Applications have to be written especially for and with these dedicated platforms. Thus, legacy code and third-party libraries not written with them are not compatible and cannot be deployed. Reflection is not a problem if the dedicated platform provides it. This is because the statically tailored base libraries are not built in an automatic fashion and the application code is not tailored.

3.5 Static Analysis-Based Techniques

Static analysis approaches for dead code elimination make use of the static information of a program to select the minimal subset of used elements. The bibliography describes four different algorithms to achieve this goal: unique name, class hierarchy analysis (CHA), rapid type analysis (RTA) and reachable members analysis (RMA) [BS96, Tit06]. These techniques share a common approach. They select an application's entry point method and deduce the execution flow from the available static information. For example, they use the available type annotations and class and method names. This information is often used to build a call-graph [GDDC97].

These techniques have been studied and applied in many environments and languages. Rayside et al. [RK02], Jax [TSL03] and the ExoVM System [Tit06] propose application extraction tools using these techniques for Java applications. Sallenave et al. [SD10] apply RTA to produce smaller .NET assemblies for embedded systems. Bournoutian et al. [BO14] use CHA to optimize on-device Objective-C applications. Agesen [Age96] presents in his thesis a static technique applied to Self, a dynamically-typed language. To

[8] `https://developer.apple.com/technologies/ios/cocoa-touch.html`

achieve this, Agesen uses type inference to obtain type information and use it to select which objects to extract.

In summary, these approaches are based on the static types found either in the source code or byte code. Thus, they are not applicable *efficiently* on dynamic languages with no static type declarations *i.e.*, some code units whose usage cannot be anticipated by these techniques will stay even if they are really not used during the application execution. However, these solutions are valuable as they allow one to tailor base and third-party libraries, and legacy code. Their tailoring approach generates new deployment units that can run on the standard runtime infrastructure. The main drawback appears in the presence of reflection and configuration files, which will only work with a subset of reflective invocations through complementary analyses on the strings found in the source code. Also, existing solutions in this family lack the possibility to configure the level of tailoring, making it an "all or nothing" approach.

3.6 Dynamic Analysis-Based Techniques

Dynamic analysis techniques use exclusively runtime information (*i.e.*, execution flow, alive objects, execution statistics) to perform dead code elimination. Amongst these, we identify two different approaches: *load on demand* and *code collection*. Load on demand approaches detect during runtime whenever a class or method needs to be installed and request it from a server. Code collection approaches deploy the full application and garbage collect unused code based on usage statistics. Most related work in this family share a common characteristic: these techniques are used inside ubiquitous computing systems *i.e.*, systems meant to be always connected. Ubiquitous computing systems, as they are always connected, have a possibility to fallback and recover in the case of incompleteness. However, to focus here on the dead code elimination techniques, we will discuss the incompleteness recovery techniques in Chapter 8.

JUCE [PRT+04, TP01]. It is a platform for ubiquitous computing devices supporting code load on demand and code collection. First, it initializes a minimal running application and code is loaded, with a method granularity, from a server located in a different machine. At runtime, the code that is not used for some time is collected following usage statistics, and loaded back again on demand if needed.

OLIE [GNM+03]. It is an engine that intelligently partitions and offloads objects during runtime to minimize memory consumption. It is part of the adaptive infrastructure for distributed loading (AIDE). In OLIE,

offloaded objects are indeed migrated to nearby remote devices. Migrated objects can be accessed later through proxies that perform remote invocations on them.

SlimVM [KWW+09, WGF11]. It is an ubiquitous computing system where all code resides on a remote server and is loaded only on demand on small devices. Some static analysis is performed only on the server to reduce the size of the transported code, by identifying most likely needed code. SlimVM changes the class format make it more compact and improve loading time. On the client side every code load is done dynamically.

Marea [Mar12]. It is an application-level virtual memory for object-oriented systems based on code collection. Marea allows not only the collection of code units but also the collection of arbitrary objects with a graph granularity. Collected graphs are swapped out to disk and swapped in back when the program tries to access one of the objects inside the object graph. Marea is supported by an object serializer with focus on fast deserialization to enable swap in's with small performance overhead, and a small memory footprint proxy library. The main drawback of this solution is that it lacks the automatic detection of objects to swap out and requires support from the developer for this.

All solutions inside this category except for Marea share one main property: they require to run the application inside a dedicated infrastructure to apply their techniques *e.g.*, dedicated VMs implementing remote lazy loading, code collection or special bytecode sets. The main challenge of these solutions resides on applying these techniques while minimizing their impact on performance during the runtime. Additionally, these solutions require their applications to run exclusively inside their infrastructure. Contrastingly, Marea solves this challenges with minimal overhead by using the existing reflective features of Pharo.

Regarding dynamic features such as reflection, these solutions are the ones that can, potentially, handle it in the best way since they have in runtime all the information needed to resolve it. JUCE and OLIE and Marea, handle naturally reflection as they do not change the runtime representation (which programs make assumptions of, when they use metaprogramming). SlimVM on the other hand, had to change the reflection support because they changed the object and class representation on their VM.

Regarding its applicability, SlimVM needs to recompile the whole application into its own format; Marea needs some development support for the collection of object graphs; OLIE and JUCE can tailor base and third party libraries without any modifications on them. Thus, the latter two can be ap-

plied to legacy code also for free. From all of these solutions, Marea is the only one that provides the ability to select the level of tailoring by customizing the serialization of the object graph to be swapped out.

The minimality of these approaches depends on the granularity of their loading mechanisms and the effectiveness of their code collection techniques. Big loading granularities may load more code units than the used, preventing minimality. The same problem appears in the case of Marea, where the code collection is managed by the application and depends on its implementation.

3.7 Hybrid Analysis-Based Techniques

Hybrid analysis techniques partially mix static and dynamic (*i.e.*, run-time) information. Their common approach is to start an application and pause it after some minimal run-time information is available *i.e.*, call stacks are created, some classes are loaded and initialized, and some objects are instantiated. Then, it uses the built stack of alive objects to perform a static analysis, as described in Section 3.5, with concrete type information. During the rest of the analysis, no more dynamic information is used.

Java in The Small (JITS) [CGV10] uses a hybrid approach to select the used parts of a program, and then loads them inside a binary image. A dedicated VM loads the binary image at startup. JITS's approach tailors base and third-party libraries as well as application specific code. It does not require modifications on the existent application to tailor it, so a legacy application could theoretically be tailored with this approach. JITS does not offer the possibility to configure the tailoring level, since it was designed to be used only in embedded devices where no more than one application would be running. Regarding reflection, JITS presents the same drawbacks as the other static call graph analysis approaches since not all the runtime information about the reflective invocations can be deduced.

3.8 Conclusion and Summary

Tailoring is a cross-cutting process that may affect any element of an application runtime, and thus, it requires particular support to detect and extract used code units. Several existing techniques were applied in the past with several results. Table 3.1 presents a comparison of these techniques, given the criteria we presented in Section 3.3.

Dedicated platforms are a first useful attempt to produce a specialized application runtime. They contain a pre-selection of base libraries that application developers must rely upon. Dedicated platforms are not tailored by an automatic process: that means that all the developed code will be part of

	Dedicated platforms	**Static Analysis**	**Hybrid Analysis**	**Dynamic Analysis**
Base Libraries	+	+	+	+
Third-Party Libraries	-	+	+	+
Legacy Code	-	+	+	~
Reflection Support	+	-	-	+
General Purpose Infrastructure	+	-	-	~
Configurability	-	-	-	~
Dynamic typing	+	-	-	+
Minimality	-	+	+	~
Completeness	+	-	-	~
(+) supported /achieved	**(~) partially supported**	**(−) not supported / not achieved**		

Table 3.1: Evaluation criteria applied to related work on deployment code unit tailoring techniques

the deployable distribution. This lack of automatic tailoring process means on one hand that the usage of reflection will not affect the result of the deployed artifact. On the other hand it means, however, that it does not offer the possibility to really tailor third-party libraries, legacy code nor even the developed application code. This is why we also consider dedicated platforms to ensure completeness while they are not as efficient to ensure minimality as other techniques.

We also see that automatic tailoring techniques, in contrast with dedicated platforms, base their tailoring on the existing application code: they can tailor base and third-party libraries as well as legacy code. Amongst them, dynamic-analysis based techniques, and hybrid-analysis ones in some grade, are able to handle reflective calls properly. We can see an almost general lack of support to configure whether a code unit should be subject to the tailoring or not.

Regarding minimality, automatic techniques are more efficient when they have more information at their disposal. Dynamic techniques use runtime information to know the concrete types of objects while static techniques use the types annotations in the source code. Between the two, dynamic techniques are more efficient because they benefit from more accurate informa-

tion.

Concluding, it is important to notice that none of the automatic tailoring techniques ensure completeness by themselves. We cannot anticipate the use of reflection and other dynamic techniques in complex scenarios. For this purpose, several techniques such as lazy loading or remote calls can be used to complement tailoring techniques. Chapter 8 explores a new dynamic tailoring technique based on Espell. We show how our technique succeeds to tailor even in the existence of reflection and adds configurability while others do not.

Part II

Espell: an Application Runtime Virtualization Infrastructure

Chapter 4

Espell: Virtualized Application Runtimes

Contents

Introduction

This chapter presents our Virtualization Application Runtime infrastructure, namely Espell. It focuses on the model of our solution and presents its core ideas. Espell supports application runtime virtualization through two key ideas:

Application co-location. The ability to have co-existing applications on top of the same virtual machine.

Object spaces. An object space is a first-class application runtime. It encapsulates the access to an application runtime and provides a high-level API to easily query and manipulate it.

In Espell two co-existing application runtimes have two well-defined roles. The *virtualized* runtime is the application runtime under manipulation. The *hypervisor* controls and manipulates the virtualized runtime (cf. Section 4.1) through the object space. Figure 4.1 shows a simplified schema of the solution.

A hypervisor can perform four basic kind of manipulations on the virtualized runtime, each with its own tradeoffs. First, it can directly manipulate its state through the object space interface (cf. Section 4.2). The other three manipulation mechanisms are based on this direct manipulation. Second, it can hook into the execution of the virtualized runtime to execute its own code (cf. Section4.3). Third, we can execute arbitrary expressions inside a virtualized runtime through process injection and virtual interpretation (Section 4.4).

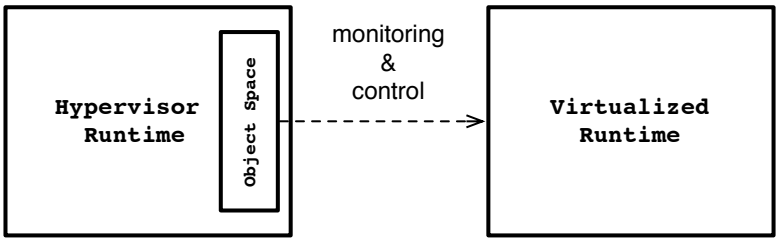

Figure 4.1: **Virtualization concepts in Espell.** Two co-existing application runtimes with two well-defined roles.

4.1 Controlling Virtualized Runtimes in Espell

Espell presents an architecture where multiple application runtimes co-exist independently from each other. An application runtime fulfills a role inside the virtualization infrastructure. An *hypervisor* monitors and controls *virtualized* application runtimes. Both the virtualized runtime and the hypervisor are full-fledged high-level application runtimes. On one hand, the virtualized application runtime is written in a high-level language as the focus of this thesis is the virtualization of such runtime. Although, we would like to emphasize that the hypervisor is as well written in a high-level language (in contrast with *e.g.*, JVMTI [JVM]), bringing three main benefits to our solution:

Expressiveness and abstraction. We can use the expressiveness and abstraction power supported by a high-level language to describe a hypervisor. For example, an hypervisor written in the Pharo language can benefit from the usage of inheritance, polymorphism and traits amongst others.

Infrastructure. The hypervisor can use all the infrastructure already available to our high-level language. It can access existing libraries such as collections, sockets and files. It also benefits from the infrastructure provided by the VM such as automatic memory management.

Tools. We can use the same tools that we use to describe our high-level language to manipulate our hypervisor. This means that we can benefit from code browsers, refactorings, unit tests and other existing tools.

A first-class object space resides inside the hypervisor and represents the virtualized runtime. A hypervisor object uses the object space and implements a particular manipulation on it *e.g.*, runtime update, failure detection, browsing or debugging. Figure 4.2 provides an overview of the Espell's architecture.

Notice that the hypervisor and the virtualized runtime do not share any VM or language state *i.e.*, each one has its own interpreter, stack, classes and

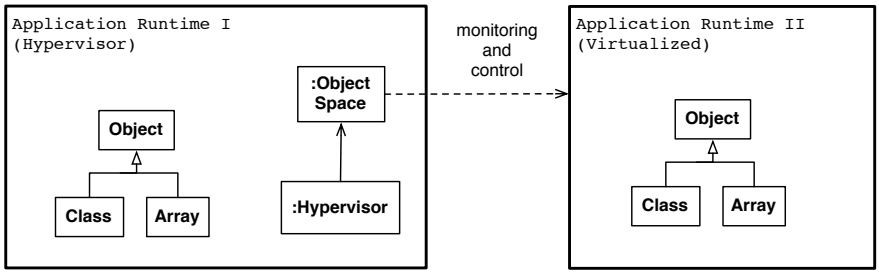

Figure 4.2: **Espell Coexistence.** The hypervisor controls the virtualized runtime through an object space.

objects. Their object graphs are isolated from each other *e.g.*, each one has its own Object and String class and their own set of unique interned strings (the symbols). This isolation strategy allows the hypervisor to change any part of the virtualized runtime without affecting itself, as it happens in reflective architectures. From the virtualized runtime point of view, all modifications come and are applied from the outside (the hypervisor), in contrast with reflective architectures where such changes are applied from within the application runtime under modification. This separation provides Espell with the following properties:

Transparency. Our solution does not require any changes to the applications residing inside the virtualized runtime. Thus, we can virtualize existing application runtimes without modifying them *i.e.*, Espell does not require virtualized applications to include particular libraries, use particular interfaces nor change their execution model.

Application independence. Our solution does not depend on the particular application inside the virtualized runtime. Different runtime hypervisors can be written for different use cases, including application specific and general ones. The limitation of this approach is, however, its dependance on the execution model imposed by the VM. We do not change the VM execution semantics such as the code interpreter or the method lookup.

4.2 Object Spaces: First-class Application Runtimes

An object space is a first-class representation of an object oriented application runtime, meant for its manipulation and control. An object space encapsulates an application runtime and provides a clear and explicit interface to manipulate it. This interface is split in two different kind of objects: an objectSpace object provides general operations on the virtualized runtime;

mirror objects [BU04] provide operations to manipulate individual objects that reside inside the virtualized runtime (Figure 4.3). Specific mirrors manipulate elements with a different object-format or runtime representation to enforce their internal representation, such as the ClassMirror.

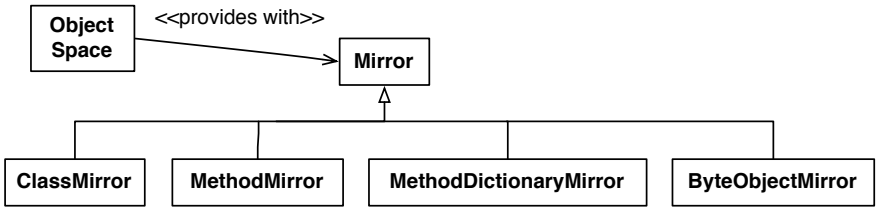

Figure 4.3: **Mirrors in Object Spaces.** An object space is the main entry point for runtime manipulation. An object space provides mirrors to modify particular objects.

These objects make explicit through the operations they expose two important parts of a VM-language interface, explained in detail in the following subsections. First, The VM setup interface exposes the elements of the language that should be initialized so it can run a program. Second, the runtime manipulation interface exposes operations that manipulate elements/structures that live during an application's execution, such as objects, classes, threads and the execution stack.

4.2.1 VM-Setup Interface

The VM-setup interface is the set of bindings between the VM and the language that the VM needs to run a program. For example, a VM may need the boolean objects true and false to push them as the result of a boolean operation at runtime. This configuration is usually done only during the language runtime initialization and before the execution of a program, as the VM cannot execute it before such elements exist. The VM setup interface allows us to configure the following kind of elements:

Well-known objects of the language. Objects such as nil, true or false are needed at runtime for different purposes. For example, the garbage collection needs nil at runtime to update weak references. Other particular objects may be held by the VM as prototypes to speed-up instantiation time of commonly used objects.

```
ObjectSpace {
    getNil −> Mirror
    getTrue −> Mirror
    getFalse −> Mirror

    setNil: Mirror −> Void
    setTrue: Mirror −> Void
    setFalse: Mirror −> Void
}
```

Special classes. Special classes are those needed by the VM at runtime. They are needed to perform safety checks, create instances or handle special cases such as immediate objects. For example, when mutating a String object the Pharo VM checks that the introduced object is a Character object, to avoid putting the String into an invalid state. Also, the VM requires references to those classes that it instantiates directly (instead of being created through the new message in program). For example, the Pharo VM instantiates a BlockClosure object when it founds the bytecodes that correspond a block expression at runtime. Finally, immediate objects are objects that are encoded inside an object reference instead of occupying extra memory inside the heap. Immediate objects include a tag in the reference that the VM knows how to map to its corresponding class to perform the method-lookup.

```
ObjectSpace {
    setArrayClass: ClassMirror −> Void
    setBlockClass: ClassMirror −> Void
}
```

Special messages. Special messages are callbacks that the VM will invoke in the application to notify particular events. The probably most known of such messages is Smalltalk's doesNotUnderstand (and its equivalent Ruby's methodMissing). When the VM does not find a method matching a message-send, it will send a doesNotUnderstand message to the receiver object and let the user program decide what to do in such a case.

```
ObjectSpace {
    setDoesNotUnderstandSelector: Mirror −> Void
}
```

This VM setup interface avoids to hardcode particular knowledge of a language inside the VM. This gives us the possibility to easily change particular language internals such as renaming special messages or changing the class hierarchy of special classes without leaving the virtualized runtime in an inconsistent state. Being radical, we could think about porting another language to this VM, as long as they share similar execution semantics *i.e.*, the

VM execution model is compatible with the given language and we configure this interface accordingly.

4.2.2 Runtime Manipulation Interface

The runtime manipulation interface includes operations to monitor and modify the virtualized runtime during execution. This interface provides access to the virtualized runtime's internal elements and encapsulates implementation details behind them (*i.e.*, the object-format imposed by the VM). This interface provides the following kind of operations:

Global Access. An object space offers operations to query and modify the global state of the virtualized runtime. For example, it provides access to the loaded classes or the running threads/processes. It exposes as well operations to install new and remove existing ones.

```
1  ObjectSpace {
2      "Classes"
3      createClassNamed: String withFormat: Integer -> ClassMirror
4      getClasses -> List<Mirror>
5      getClassNamed: String -> Mirror
6      removeClassNamed: String -> Void
7      ...
8
9      "Threads"
10     getThreads -> List<Mirror>
11     installThread: ThreadMirror -> Void
12     ...
13 }
```

Runtime Object Access. Mirrors [BU04] expose operations to query or alter a particular object or element in the virtualized runtime. Different kinds of mirrors enforce the object-format of the different types of objects we can manipulate. For example, we expose specific mirrors for normal objects, classes, methods, activation records or processes. Notice that these mirrors are low-level mirrors as they expose operations to mutate and access objects in their VM representation. Additionally, through mirrors we can execute VM primitives on objects, providing the correct primitive ID (an integer or native function pointer, depending on the implementation) and the corresponding arguments. These primitives are operations that the VM exposes normally to the language.

```
Mirror {
    "Class access"
    getClass −> ClassMirror
    setClass: ClassMirror −> Void

    "Field Access"
    getInstanceVariableNamed: String −> Mirror
    setInstanceVariableNamed: String varName withValue: Mirror −> Void
    ...

    "Primitive execution"
    executePrimitive: PrimitiveID withArguments: Array −> Mirror
}
```

```
ClassMirror {
    "Instantiation"
    instantiate −> Mirror
    instantiateWithSize: Integer −> Mirror
    ...

    "Method manipulation"
    compileMethod: String −> Void
    removeMethodNamed: String −> Void
    ...
}
```

4.3 First-Class Hypervisors

The client of an object space is a *hypervisor*. A hypervisor is first-class object that implements a particular monitoring/modification strategy for a virtualized runtime. For this we split a virtualized application's execution in cycles. A cycle is an execution that lasts at least a time window and finishes when it finds the next safe suspension point (Figure 4.4). A safe suspension point is a point during execution where suspending a thread will not leave it in an inconsistent state. To illustrate with a concrete example, in our Espell prototype written for the Pharo language described in Chapter 5 safe suspension points are the activation of message sends and bytecode backjumps. During each execution cycle, the virtualized runtime runs unmanaged using the full VM's capacity. When the cycle finishes because the execution completed a time window and found a suspension point, the control is given to the hypervisor. The hypervisor can monitor and modify the virtualized application at this point and then resume the execution of the virtualized runtime from the last suspension point.

To enable cycle execution, the objectSpace interface allows us to execute an application runtime during a cycle of time. An Espell implementation should additionally include support for suspension points. This operation will wake up the virtualized runtime processes and execute them for at least a given time. Once the cycle is finished, it will be suspended in the next suspension

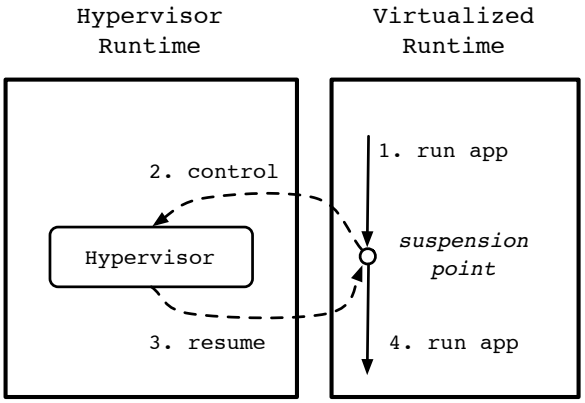

Figure 4.4: **Execution Cycle.**

point it finds and the control will return to the hypervisor.

```
ObjectSpace {
    runCycle -> Void
}
```

Then, our hypervisor class hierarchy presents four basic methods. First, the run template method implements the basics of the hypervision cycle: it resumes the execution of the virtualized runtime inside a loop. Two methods (before and after) provide hooks for the specific hypervisor implementations. Figure 4.5 shows a class hierarchy example and code with two sketched hypervisors. A NullHypervisor allows the virtualized runtime to run without any intervention. The UpdateHypervisor instead checks the existence of a file with updates on every cycle.

```
Hypervisor >> run
    [ true ] whileTrue: [
        self before.
        self basicRun.
        self after.
    ]

Hypervisor >> basicRun
    objectSpace runCycle.

NullHypervisor >> before
    "Nothing"

NullHypervisor >> after
    "Nothing"

UpdateHypervisor >> before
    self checkForUpdate.

UpdateHypervisor >> checkForUpdate
    "We check if a given file exists"
    'update.txt' asFile exists ifTrue: [ ...
    ...
```

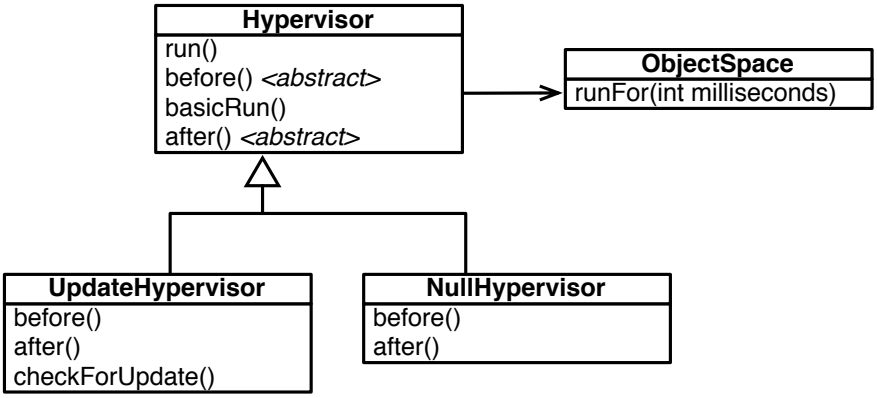

Figure 4.5: **Example Hypervisors Class Hierarchy.** A NullHypervisor does nothing while the UpdateHypervisor checks for updates before every cycle execution.

4.4 Cross-Runtime Communication

A hypervisor may require to apply some operation in a virtualized runtime that would be cumbersome through direct object manipulation with mirrors. Let's take as an example a virtualized application that uses a particular logging library where disabling logging means executing the following statement:

```
1 Logger disable.
```

And the code invoked is:

```
1 Logger class>>disable
2   self uniqueInstance disable
3
4 Logger>>disable
5   enabled := false
```

Doing the same through our abstract layer of mirrors presents the following drawbacks (a) to replicate the behavior of the disable method inside the hypervisor and (b) to couple it to the internal representation of such a logging library.

```
1 OurHypervisor>>disableLogging
2   logger := objectSpace getClass: #Logger.
3
4   "the logger is a singleton"
5   "in a class variable named uniqueInstance"
6   loggerInstance := logger classVariableAt: #UniqueInstance.
7
8   "The 'enabled' instance variable is the first"
9   loggerInstange instanceVariableAt: 1 put: false.
```

A hypervisor may benefit from the execution of an arbitrary expression or statement within the scope of the virtualized runtime. It will reduce in this example the coupling to only the public API of the logging library. For ex-

ample, enabling or disabling a logger from the virtualized application can be easily achieved through executing the following statement in the hypervisor instead of manually modifying the state of the logger objects:

```
1 objectSpace execute: [ Logger disable ].
```

To achieve this kind of communication Espell provides two mechanisms: process injection and virtual interpretation.

4.4.1 Process Injection

The hypervisor and the virtualized runtime do not share core-libraries nor special objects, preventing us to easily perform a message-send between our co-existing application runtimes. Using the existing message-send mechanism provided by the VM has the following challenges:

Cross-Runtime Method-Lookup. A *cross-runtime message-send* (from the hypervisor to the virtualized runtime) cannot be simply achieved by a usual message-send mechanism. The usual message-send mechanism looks up in the receiver's class hierarchy a method with an *object identical* method signature. However, our not-sharing strategy prevents the normal method-lookup work on a *cross-runtime message-send* as symbols and classes are not shared between the different application runtimes. In such a case, the method-lookup mechanism fails because the elements of a message signature and the method signature we target are indeed *equals but not identical* (Figure 4.6).

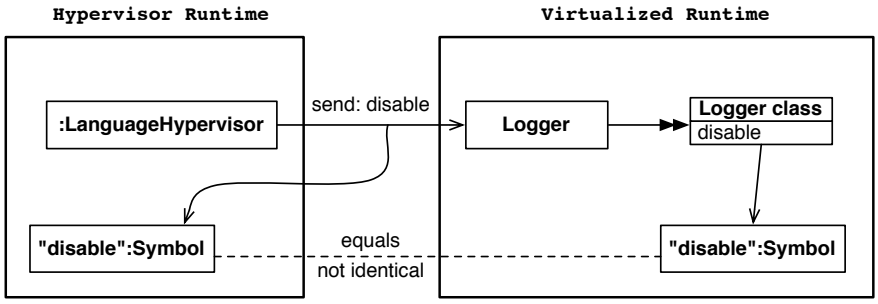

Figure 4.6: **Failing Cross-Runtime Method Lookup.** The hypervisor sends the size message to an array in the virtualized runtime. This message-send signature is the hypervisor's size symbol. The size method exists but its signature has a different size symbol. Both symbols are equals but not identical, and thus the lookup fails.

Exceptions and Stack. A cross-runtime method-lookup succeeds if we modify the hypervisor message-send to contain the right symbol from the virtualized runtime. In such case, the execution stack contains a mixture of activations that belong to the hypervisor and the virtualized

runtime. This poses a problem under the presence of techniques that traverse the execution stack indiscriminately, such as the exceptions or stack manipulation operations such as Smalltalk's thisContext special variable. These operations may leak object references from an application runtime to another, leaving them in inconsistent state (Figure 4.7) [MWC10].

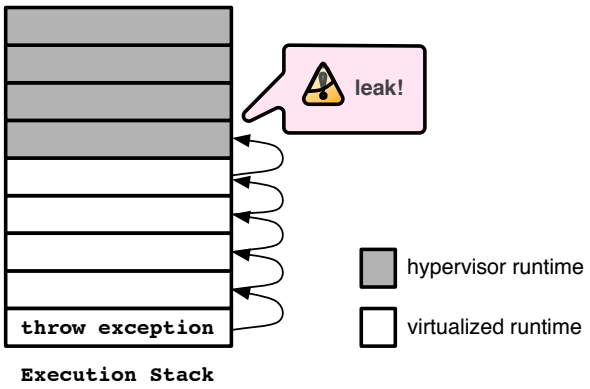

Figure 4.7: **Reference Leaks in a Mixed Stack on Exception.** When mixing the execution stack between the virtualized and hypervisor runtimes, an exception may traverse the stack and have access to a reference from a different application runtime.

To overcome these problems we base the cross-runtime communication on *process injection i.e.*, we create a new thread inside the virtualized runtime containing the expression to execute. This new thread executes in cycles until it is finished. Once finished, from the hypervisor we can access the result of the execution through a mirror on the process (cf. Figure 4.8).

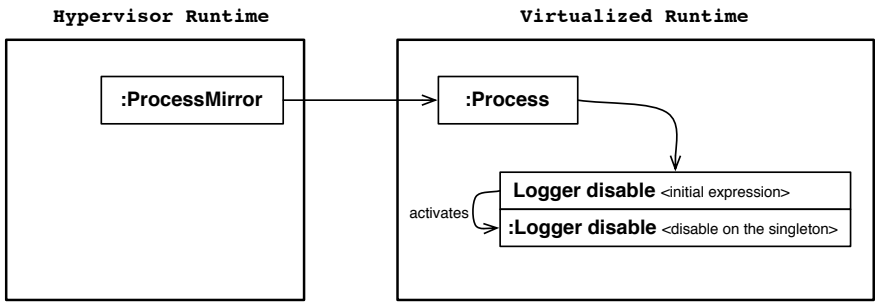

Figure 4.8: **Process injection.** The hypervisor can install and manipulate a process through a mirror. The result can accessed from the mirror.

To perform process injection, we need to marshall the arguments, literals and the returned object between their representation in the hypervisor from/to their equivalent representation in the virtualized runtime. For example, marshaling a String object means to create a new String object in the

other application runtime with its corresponding class and copying all its bytes. Global object references such as classes are mapped to their corresponding mirrors in the virtualized runtime. Non-global non-literal objects can be specified explicitly as an argument with their corresponding mirrors.

```
objectSpace
    execute: [ :aLogger | aLogger disable ]
    with: aLoggerInstanceMirror.
```

This solution for cross-runtime communication solves both problems observed before. In a first place, it prevents the method-lookup failure by translating the objects part of the method signature. Second, the new process executes in its own stack ensuring none of the objects from the hypervisor are leaked to the virtualized runtime.

4.4.2 Virtual Interpretation

It may happen that a virtualized runtime is in an state where it cannot execute code by itself. An virtualized runtime may be in a buggy state due to a change, or it may be under construction (as in the subject of this thesis). In those cases, we cannot delegate the execution to the virtualized runtime itself and thus, we cannot use the process injection mechanism. Let's for example consider that we accidentally break or remove the at:put: method from the MethodDictionary class. Since that method is the one used to install new methods, from that moment on we cannot install methods in classes using process injection any more. Moreover, solving this problem would require to install a method, preventing us to do it through process injection.

```
methodDictionaryClass := objectSpace classNamed: #MethodDictionary.
methodDictionaryClass remoteMethod: #at:put:.

"now we cannot install methods using process injection any more"
```

For such cases Espell allows us to execute an expression inside a virtualized runtime within the context of the hypervisor through the *virtual interpretation* of *e.g.*, abstract syntax trees (ASTs) or bytecodes. A virtual interpreter is a first-class entity that resides inside the hypervisor and interprets code inside a virtualized runtime. As it is a first-class entity, we can easily modify its implementation details such as the method lookup or the execution of primitives. These kinds of modifications allow us to handle the special cases that cannot be handled by process injection. Then, we can use a virtual interpreter to execute an AST equivalent to the at:put: method to install back our corresponding method. This puts the virtualized runtime in our example back into a stable state before it can continue its own execution.

```
"newMethod is a method that fixes the bug"
newMethod := objectSpace compileMethod: 'at: aKey put: aMethod ....'.

interpreter := VirtualInterpreter newOn: anObjectSpace.
interpreter
   execute: [ Object methodDictionary at: #brokenSelector put: newMethod ]
   with: newMethod.

"now we can continue working with process injection"
```

4.5 Conclusion and Summary

Espell presents an architecture where several application runtimes can share the same process. A virtualized application runtime is subject to monitoring and manipulation from another application runtime, namely the hypervisor runtime. The hypervisor runtime contains a first class representation of the virtualized runtime, namely an object space. The object space exposes a clear interface to safely access and modify an application runtime. This interface includes operations to configure the virtualized application runtime and operations to modify particular objects from it. The latter is encapsulated in mirror objects.

The execution of a virtualized runtime is organized in cycles. In between cycles, the hypervisor can perform an operation and then resume the virtualized runtime execution. We ensure that the state of the virtualized runtime keeps its coherence by only suspending it in safe suspension points.

Besides the direct manipulation through mirrors, a hypervisor can perform cross-runtime message-sends for a richer communication using process injection. Process injection prevents failures in the method lookup and avoids stack traversing techniques such as exception to leak cross-runtime references. Additionally, a virtual interpreter allows us to execute code in the virtualized runtime with full intercession. A virtual interpreter is a first class entity, allowing us to easily specialize and change it.

In following chapters we will show our prototype implementation of this infrastructure and how this infrastructure supports the evolution of a programming application runtime in two different scenarios. First, the recreation of a application runtime using an explicit bootstrap process. Second, an application tailoring technique that specializes an application runtime to contain only elements that are used during runtime.

The Espell Prototype

Chapter 5

Contents

Introduction

To validate our model, we implemented our Espell prototype on the Pharo platform (Section 5.1). Our solution virtualizes Pharo application runtimes and provides, as already described, the ability to manipulate their object graph and control their execution. Our implementation includes a language side library containing the object space and hypervisor classes. Our main modification to Pharo is an extension to its stack-based VM. This VM is a version of the Pharo VM without its Just In Time (JIT) compiler. We made the choice of the JIT-less VM to simplify the development efforts of our prototype. Our extension enables the co-existence of several application runtimes on the same VM and the modification of the VM setup interface.

We implemented object spaces by exposing Pharo's *special objects array*. This array provides an object space with support to modify the VM setup interface (Section 5.2) and to replace the VM interpreter state to execute several co-existent application runtimes on top of the same VM (Section 5.3). Espell mirror's do not require particular changes in the Pharo VM (Section 5.4). We base our mirror implementation on two existing VM primitive operations that allow us to execute a primitive on an arbitrary object by changing the primitive's receiver. Additionally, the already existing reifications of objects such as CompiledMethod, Process (threads) and Context (stack frames or activation records), allow us to manage such runtime elements using the same mirror mechanism and avoid the development of specific ad-hoc solutions.

Espell presents a memory layout where the objects of both the virtualized runtime and the hypervisor co-exist in the same heap (Section 5.6). This means on one hand that there is no need for implementing special support of *cross-runtime references* for mirrors. On the other hand, we can reuse the existing memory management of the virtual machine almost transparently.

This chapter finishes by presenting the non-implemented aspects of our solution: JIT compilation, shared state enforced by VM implementation and correct management of external resources such as files or sockets (Section 5.8).

5.1 Pharo Execution Model in a Nutshell

To understand the constraints and limitations of our solution, we start this chapter by presenting the execution model imposed by the Pharo VM. The Pharo VM features a bytecode-based stack interpreter with a generational garbage collector and a JIT compiler. For the interested reader, several publications describe its details and how it evolved over time [GR83, IKM+97, Mir11]. On the execution model side, this VM imposes the following contract:

Object Format. All objects in the 32 bit version of the Pharo VM have a header and a list of fields. The object header is one, two or three words long and describes amongst others how large is the field list of the object, if those fields contain weak or strong pointers, and which is the class of the object. The VM also assumes the format of other special objects such as classes, method dictionaries and methods. Each class must have three mandatory fields: the superclass, its method dictionary and an integer describing the class format. Method dictionaries are hash tables that base their hash function on the identity of key objects. Methods are objects that contain a list of bytes with the bytecodes of the method and a list of the literal objects of that method encoded in the bytes.

Object Model. The Pharo VM enforces, in its lowest level, a class-based object oriented model with single inheritance. Each object has a reference to its class (that appears inside its header). Each class has only one superclass (or nil denoting the end of the hierarchy) and one method dictionary. The VM during its execution does not enforce the existence of metaclasses nor a particular class hierarchy. Instead, meta-classes are built on top of this much simpler model. Indeed, this simple model allows one to implement other language extensions such as Traits [SDNB03] or mirrors as in Metatalk [NBD+11].

Bytecode set. The Pharo VM constrains methods to a limited number of predefined bytecode sets. These bytecode sets are based on the VM's stack based interpreter. This means that every language that is meant to run on top of this VM must be compiled to any of these bytecode sets, independently of its original syntax and semantics.

5.2 The Special Objects Array as VM-Setup Interface

The VM-setup interface is the set of bindings between the VM and the language needed by the VM to run *e.g.*, which are the boolean objects and special classes. Pharo VM holds the state of the VM-setup interface inside a *special objects array* object. Pharo's special objects array contains 56 entries whose indexes are well known by the VM for their access at runtime. To implement our VM-setup interface, we provide access and enable the modification of this special object array. Following, we detail the special objects array entries that our API offers.

Special Instances. Special instances such as nil, true and false are directly pushed by the VM interpreter at runtime instead of residing in a method literal list. A flyweight Character table references the first 256 character objects to ensure their identity and save memory.

System Dictionary. The system dictionary references all installed classes in the system. An object space uses this entry to query the installed classes and to install new ones. The VM does not make any special use of this object as it is managed directly from the language.

Process Scheduler. The process scheduler references all existing processes/threads in the runtime. We can install and remove processes from the process scheduler. The VM uses this same process scheduler to manage the runtime's execution.

Symbol Table. The symbol table gathers the set of *unique* symbols in the system. Symbols are mainly used to denote method signatures and ensure reference equality during the method lookup. An object space uses the symbol table to map symbols between runtimes and so it ensures no duplicate symbols are created. The VM does not make any special use of this object as it is managed directly at the language level.

Literal Classes. Literal classes are the classes of literal objects. Literal objects are those objects whose values are known at parse time. The best known examples of literal objects are strings, numbers or literal arrays. These literal objects are then encoded inside the method where they are embedded. For example, numbers are usually included inside the method's literal list. Other literal objects such as block closures cannot

however be created at compile time because they are required to know runtime information, such as the stack context where they were created. Those objects are instead encoded in the method's bytecode. Then, the VM creates at runtime a BlockClosure instance when it finds the corresponding bytecodes. Generally speaking, on one hand, the VM uses the classes available in this list to directly instantiate objects of these types or perform safety checks at runtime (which cannot be performed at compile time because of reflection or the dynamically-typed nature of the language). On the other hand, an object space uses these well-known classes to perform transformations between objects in one runtime to another *e.g.*, translate a string from the hypervisor runtime to a string in the virtualized runtime.

Special Selectors. The special selectors denote those messages that the VM will send to the image under special situations. This is the case of the doesNotUnderstand selector that is sent to the receiver object when the method lookup fails finding a method under a message-send. Other selectors in this category are for example (a) cannotInterpret which is sent when a class is involved in the method lookup with no method dictionary, (d) run:with:in: when the method lookup finds in the method dictionary an object that is not a method, (c) mustBeBoolean which is sent when a branch operation founds a non-boolean object.

5.3 Cycle Execution and Context Switch

The Pharo VM has single threaded execution *i.e.*, only one operating system thread is used to execute Pharo code. Process scheduling is handled internally by the virtual machine. Processes scheduled using this approach are also called *green threads*. Green threads provide process scheduling without native operative system support while limiting the proper usage of modern multicore CPUs. By using green threads only one process, the *active process*, is executed at each instant in time. If there are any waiting processes with a greater priority, the active process gets preempted *i.e.*, it is suspended and the process with the highest priority becomes the new active process.

In Espell, we modified part of the green thread mechanism to allow the scheduling of different application runtimes. We call *context switch* the mechanism that changes from one application runtime to another. A virtualized application runtime runs for a window of time after which it gets preempted if another runtime with higher priority (the language hypervisor runtime) is waiting. Internally, the VM has a single bytecode interpreter shared amongst the different running runtimes. To perform the context switch, we exchange the special objects array of the VM interpreter and we resume the VM exe-

cution. This will resume the execution of the application runtime whose special objects array we installed (cf. Figure 5.1). This solution keeps the single threaded nature of the VM: the application runtime under execution is the *active* application runtime while the others are considered as *suspended* ones. This implementation has the benefit of avoiding concurrency problems between the different application runtimes, allowing us to focus on the runtime modification features.

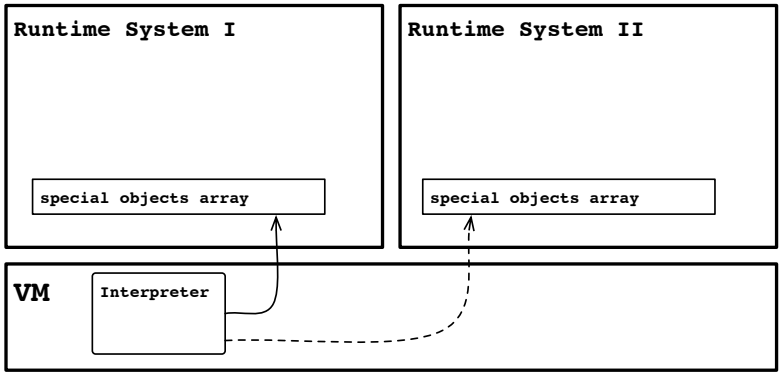

Figure 5.1: **Context Switch Internals.**To perform a context switch, we change the special objects array of the VM's interpreter.

Finally, the window time control was implemented by hooking into the existing process preemption mechanism in the VM. A separate VM thread independent of the application runtimes, namely the *heartbeat*, is awakened with a frequency of 5Hz (*i.e.*, every 200 milliseconds) and activates a flag that indicates a preemption may occur. The chosen frequency is indeed important as it determines the precision of the clock measurements and how often the VM checks for events such as delays or semaphores being signalled. If the frequency is unstable or low the clock resolution will be poor, delays will fire erratically and input from outside the VM won't be responded to promptly.

When the VM interpreter arrives to one of the safe suspension points (*i.e.*, a point where suspending the current execution will not leave the current process in an incoherent state) it preempts the active process if the heartbeat signalled so [DS84]. Our safe suspension points correspond to the Cog VM suspension points: back jump and message-send bytecodes [Mir11]. These suspension points are designed on one hand to minimize the execution overhead and on the other hand to not suspend a process in between of a bytecode combination that could be unknown by the debugger. The suspension point before a message-send corresponds to the VM's stack-overflow check and the one in backjumps allows preemption to happen in the case of long run (possible infinite) cycles. In Espell we modified the preemption code to activate another application runtime if available.

5.4 Espell Mirror Implementation

We implement mirrors in Espell through the direct manipulation of the objects inside a virtualized runtime. An Espell mirror includes a direct reference to its *target*, an object inside a virtualized runtime. The isolation between the hypervisor and the virtualized runtime is handled by mirrors. The only object with grants to reference an object from other runtime is a mirror. To accomplish this, the operations on mirrors may yield either another mirror or objects that belong to the hypervisor runtime (marshaled from their representation inside the virtualized runtime).

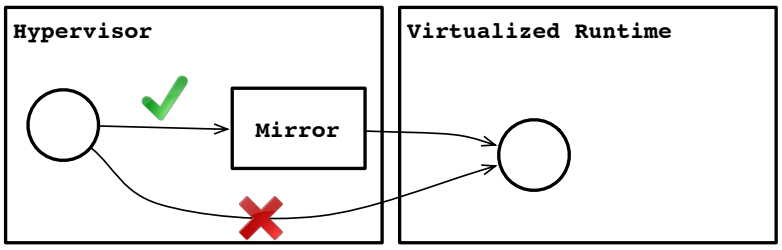

Figure 5.2: **Mirrors.** To perform a context switch, we change the special objects array of the VM's interpreter.

Mirrors manipulate their targets through VM primitives. A primitive in Pharo is a function executed at the VM level. The VM assigns to each primitive an id, generally an integer index. To call a primitive at the language level we must add a method describing that it calls a primitive with a given id. Then, we have to send the corresponding message to the target of such primitive. The following code snippet shows an example where the size of an array is calculated by invoking a primitive. Primitive **62** is the primitive that returns the number of variable fields of an object *e.g.*, the number of fields of an array. We make a method size available in the Array class implemented as a primitive invocation with a *pragma* or an annotation *i.e.*, the message-like expression between angle brackets (<>). When sending the message size to an array object, this method is looked up and found in this object class hierarchy, and the VM activates the primitive on the receiver object.

```
1 Array >> size
2     <primitive: 62>
3
4 "Then we can execute"
5 {1 . 2 . 3} size.
```

We want, however, to avoid cross-runtime message-sends. We then bypass the message-send with the combination of two existing primitives:

receiver:withArguments:executeMethod: This primitive executes a *given method* on an object. Given a method, it is possible to execute it on an

object, avoiding method lookup in the object. In current Pharo VM, this primitive is available in the CompiledMethod class with the number 188. This method receives as arguments the object on which the primitive will be executed, an array of arguments, and the methods to execute.

tryPrimitive:withArgs: This primitive executes a *primitive* on an object. It is possible to send a message to an object, so a primitive is executed on the receiver. This primitive is implemented in Pharo's ProtoObject class as tryPrimitive:withArgs: with number 118 and receives as argument the number of the primitive and an array or arguments.

Our mirrors use then a combination of these primitives to execute an arbitrary primitive while avoiding the method lookup. We use receiver:withArguments:executeMethod: to execute the primitive method tryPrimitive:withArgs: on the object from the virtualized runtime. The first avoids performing a message-send to the object inside the virtualized runtime. The second provides the ability to execute any arbitrary primitive. The following code snippet illustrates how we implemented the size primitive using mirrors.

```
1  Mirror >> size
2      ^ self executePrimitive: 62
3
4  Mirror >> executePrimitive: aPrimitiveNumber
5      ^ self executePrimitive: aPrimitiveNumber withArguments: #()
6
7  Mirror >> executePrimitive: aPrimitiveNumber withArguments: anArrayOfArguments
8      "target is a reference to an object inside an object space"
9      ^ CompiledMethod
10         receiver: target
11         withArguments: { aPrimitiveNumber . anArrayOfArguments }
12         executeMethod: (ProtoObject >> #tryPrimitive:withArgs:)
```

Using this mechanism, Espell implements one mirror per type of object supported by the Pharo VM. Notice that as some runtime elements such as methods, activation records or even processes are reified as objects in Pharo, we can manipulate them through mirrors without developing new support for them in the VM:

Object mirrors. Mirrors for objects containing just object references such as an Array or a Person.

Word mirrors. Mirrors for objects containing only non-reference word size fields such as a Float or a WordArray object.

Byte mirrors. Mirrors for objects containing non-reference byte size fields such as a ByteArray or a ByteString.

Class mirror. Mirrors for class like objects. They control the manipulation of classes and enforce the format of a class in the Pharo VM which should

have as its first three instance variables the superclass of the class, its format and method dictionary.

Method dictionary mirror. Mirrors for method dictionaries enforcing the VM invariants for method installation *i.e.*, using the identity as a hash for the key to lookup in the method dictionary.

Method mirror. Mirrors for method objects. Method objects in the Pharo VM are objects with a mixed format. They contain object references to their literals, and byte fields with their bytecode.

Context mirror. Mirror for context objects. A context object is reified lazily by the Pharo VM and contains a variable number of fields denoting the local variables of a scope.

Process mirror. Mirror for process objects. Process objects are used to manage the execution of a Pharo runtime. They can be resumed, suspended or finalized.

5.5 Espell Virtual Interpreter

We implemented a virtual interpreter for Espell by extending the existing Pharo AST interpreter. This interpreter parses Pharo code and executes the resulting AST in our application using reflection. We extended this AST interpreter with the purpose of executing the given AST inside a virtualized runtime instead of inside the current application runtime. Our extensions include the following main points:

Name Resolution. Names inside the source code representing *e.g.*, classes or globals must be resolved to the corresponding elements inside a virtualized runtime. For this, we provide an alternative source of bindings to the AST semantic analysis process.

Instance Variable Access. Accessing or writing an object's instance variable is mapped by the virtual interpreter to a field access or write in the mirror representing the receiver object.

```
1 VirtualInterpreter >> readInstanceVariableNumber: index
2     ^ self receiver instanceVariableAtIndex: index
3
4 VirtualInterpreter >> writeInstanceVariableNumber: index with: aMirror
5     ^ self receiver instanceVariableAtIndex: index put: aMirror
```

Method Lookup. On a message-send, the virtual interpreter performs the method lookup by inspecting the class hierarchy of the receiver object. The receiver object state is available to the interpreter by a mirror. We illustrate this with a simplified method lookup:

```
VirtualInterpreter >> lookupSelector: selector
    | classToLookup |
    "classToLookup is a class mirror"
    classToLookup := self receiver getClass.
    [ classToLookup isNotNil ] whileTrue: [
        (self class: classToLookup hasSelector: selector)
            ifTrue: [ ^ classToLookup ].
    classToLookup := classToLookup superclass.
    ]
```

Primitive Execution. In the leaves of the execution, message-sends invoke the language primitives. These language primitives are executed through the object space interface.

```
VirtualInterpreter >> invokePrimitiveMethod: aMethod
    ^ self receiver
        executePrimitive: aMethod primitive
        withArguments: self arguments
```

Being a first-class entity, we can specialize the virtual interpreter to instrument code execution in order to override normal behavior or trace the execution. We can for example trace all instance variable writes by overwriting one of the methods above.

```
TracingVirtualInterpreter >> writeInstanceVariableNumber: index with: aMirror
    | result |
    result := super writeInstanceVariableNumber: index with: aMirror.
    self logWriteIn: self receiver atIndex: index with: aMirror.
    ^ result
```

5.6 Espell Memory Layout

In Espell, all objects belonging to the different application runtimes share a unique object heap. Objects are mixed in this heap and objects from the same application runtime are not necessarily contiguous, as shown in Figure 5.3. However, they are logically separated as they belong to different object graphs. Pharo being a safe-language [HCC+98, HvE02] prevents us to forge references (*i.e.*, creating references from numbers using pointer arithmetic) and isolates the object graphs of each runtime system.

This decision is intended to minimize the changes made to the virtual machine and reduce the complexity in our prototype implementation. Our decision, while easing the development of our solution, has the following consequences:

Reuse memory handling mechanisms. We use the same existing memory infrastructure as when Espell is not used. Existing mechanisms for allocating objects or growing the object memory when a limit is reached can be reused transparently by our implementation.

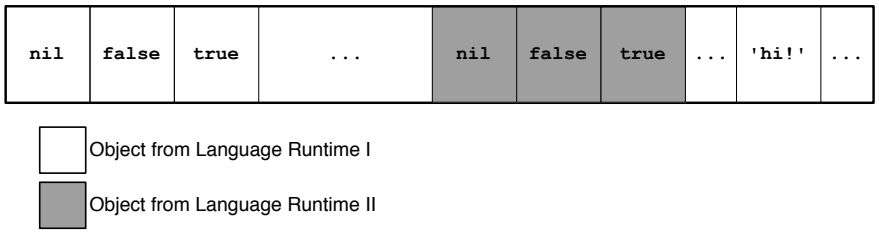

Figure 5.3: **A unique heap containing objects from different application runtimes.** Objects from the application runtime I and application runtime II are mixed in the heap. In this figure, after the nil, true and false instances that belong to application runtime I, follow the corresponding ones of the application runtime II, which can in order be followed by objects of the former, like the string **'hi!'**.

Simplify the object reference mechanism. Cross-runtime references are normal object references. No extra support from the virtual machine was developed in this regard. Isolation is ensured at the language level through mirrors.

Shared garbage collection. Since objects from the different runtimes are mixed in the object memory, and their boundaries are not clear from the memory point of view, the garbage collector (GC) is shared between them. Every GC run must iterate over all their objects, increasing its time to run.

Limited to two co-existent application runtimes. The current memory layout limits our prototype to two co-existent application runtimes due to the interference of the GC and weak references. When the GC collects weak references, it replaces the reference to a reference to nil. In the context of Espell, the GC should replace weak references to the nil object from the correct application runtime. We can conclude that indeed, we need some kind of ownership mechanism *i.e.*, the GC may know to which application runtime an object belongs to correctly replace object references. In our prototype implementation we *mark* ownership with only one available bit in the object header of objects. We look forward to replace this mechanism in the future by an alternative and more flexible implementation *e.g.*, separate memory regions per application runtime or extra object headers to represent ownership.

5.7 Benchmarks

We present in this section a series of benchmarks we applied to our implementation[1]. The intent of these benchmarks is to provide an idea of the overhead of our implementation of mirrors, process injection and the execution cycle. However, notice that the focus of this thesis is not on the performance of our technique for its extensive usage. Instead we focused on its flexibility for application runtime generation. We acknowledge that there is room for optimization in the techniques we used.

5.7.1 Mirrors Micro-Benchmarks

To evaluate the mirror usage overhead we measure six different aspects of our implementation: instance variable read and write, variable field read and write, object instantiation and primitive execution. We measured these aspects first in a normal program execution without reflection or mirrors, second by using the standard reflection library provided by the language and finally by using mirrors. To have visible results, we used a problem size of 50.000 between each measurement (*i.e.*, we actually measured the time it took to execute 50.000 times each benchmark).

Benchmark	**Normal**	**Reflection**	**Mirrors**
Instance Variable Read	2.00ms +/-0.47	2.70ms +/-0.61	104.5ms +/-1.1
Instance Variable Write	2.90ms +/-0.63	2.80ms +/-0.53	62.80ms +/-0.70
Variable Field Read	1.80ms +/-0.59		105.20ms +/-0.45
Variable Field Write	1.70ms +/-0.54		63.00ms +/-0.97
Object Instantiation	3.20ms +/-0.93		383.9ms +/-2.1
Primitive Execution	1.60ms +/-0.67	3.50ms +/-0.78	89.40ms +/-0.82

Table 5.1: **Mirrors Micro Benchmarks.** Comparing the time to execute basic operations using the language as usual, using reflection and Espell mirrors.

As we can notice, this mirror implementation introduces a considerable overhead in execution when used. Table 5.2 shows the actual overhead by comparing the benchmark results. The overhead varies between 22 and 56 times the normal execution with the exception of object instantiation. We attribute this difference to the fact that object instantiation can trigger garbage collection every certain number of new objects created.

[1] `http://www.smalltalkhub.com/#!/~Guille/ObjectSpace`

Benchmark	Overhead over Normal Execution	Overhead over Reflection
Instance Variable Read	52x	39x
Instance Variable Write	22x	22x
Variable Field Read	58x	
Variable Field Write	37x	
Object Instantiation	116x	
Primitive Execution	56x	26x

Table 5.2: **Mirrors Overhead.** The actual overhead of using this mirror implementation over the normal language constructs.

5.7.2 Process Injection Overhead

To measure the overhead introduced by process injection over a normal message send, we present three different scenarios. The two first measure the injection of message sends that take a short time by only sending the yourself message to an object. In the first case, the message is sent to a global object. In the second case the message is sent to a literal string, implying its marshalling to inside the virtualized runtime during the injection of the process. The third scenario is the execution of a process that takes longer time (the calculation of factorial of 5000). We compare each of these scenarios by executing them in out of the Espell infrastructure and by injecting them inside a virtualized runtime. Benchmarks are run with a problem size of 5000 to have visible measurements *i.e.*, we run each benchmark 5000 times between measurements. Table 5.3 shows our results.

Benchmark	Normal Execution	Process Injection
Single message-send (1)	0.40ms +/-0.48	38s +/-4
(1) with literal marshaling	0.40ms +/-0.48	37s +/-0.1
Long message-send	438s +/-18	521s +/-31

Table 5.3: **Process Injection Benchmarks.** Measuring the overhead included in the execution by process injection.

The results we obtained are measured in terms of a problem size of 5000, meaning that they are several orders of magnitude bigger than a single pro-

cess injection. We then *estimate* the overhead of process injection performing the following simple calculation (cf. Table 5.4). Notice that we don't include in this calculation the confidence of the result. We can also see that the overhead is bigger in the case of smaller computations. This is explained by the cost of process injection, which depends on mirrors to create and install processes. However, the cost of process injection is paid only once per process injected when the process is created. We can see that in longer computations this cost is not as evident.

$$estimatedOverhead = \frac{processInjection}{normalExecution}$$

Benchmark	Estimated Overhead
Single message-send (1)	95070x
(1) with literal marshaling	91835x
Long message-send	1.18x

Table 5.4: **Process Injection Overhead.** Comparing the overhead included in the execution by process injection with the normal execution.

5.7.3 Execution Cycle Overhead

With the objective of testing the execution cycle overhead, we executed a series of more complex benchmarks. We chose a subset of the computer language benchmarks game[2]. We executed each of these benchmarks ten times in three different setups. We first benchmarked the Vanilla JIT-less Pharo VM without any of our changes, to make it a fair base of comparison with our JIT-less solution. Next, we benchmarked a non-virtualized solution running on our modified VM. Finally, we executed the same series of benchmarks using a Null hypervisor *i.e.*, an hypervisor that takes the execution control on each cycle and returns it to the virtualized runtime without performing any particular action. Table 5.5 presents the results of our benchmarks, indicating the corresponding problem size of each benchmark.

These benchmarks specify a *problem size* that defines how many times a benchmark should be executed between measurements to obtain visible results (this is because some benchmarks are unnoticeable if performed only once). This means that we cannot read our measurements as an estimation of the cost of execution but as a base for comparison. We *estimate* the overhead

[2]Original list of benchmarks: `http://benchmarksgame.alioth.debian.org/`
Pharo Implementation: `http://www.smalltalkhub.com/#!/~StefanMarr/SMark`

Benchmark (problem size)	Vanilla Pharo VM	No Hypervisor	Null Hypervisor
RegexDNA (10000)	5711ms +/-22	5905ms +/-11	5878ms +/-18
KNucleotide (1000)	94.00ms +/-0.85	95.60ms +/-0.77	100.1ms +/-3.4
SpectralNorm (100)	79.8ms +/-1.4	84.9ms +/-1.1	86.6ms +/-1.4
BinaryTrees (9)	36.80ms +/-0.65	34.60ms +/-0.40	37.10ms +/-0.83
Mandelbrot (200)	422.3ms +/-1.4	468.8ms +/-2.9	488.0ms +/-6.5
ReverseComplement (1000)	5.40ms +/-0.48	6.00ms +/-0.71	6.80ms +/-0.24
ThreadRing (1000)	2.50ms +/-0.41	2.20ms +/-0.59	2.50ms +/-0.56
PiDigits (27)	0.90ms +/-0.69	0.70ms +/-0.66	0.70ms +/-0.66
Meteor (2098)	2260.2ms +/-2.6	2373ms +/-12	2398.2ms +/-8.5
NBody (1000)	21.10ms +/-0.57	20.80ms +/-0.80	21.70ms +/-0.66
Fasta (1000)	4.30ms +/-0.54	4.80ms +/-0.70	5.20ms +/-0.65

Table 5.5: **Execution Cycle Benchmarks.** Comparing the cycle overhead from a non virtualized Pharo VM to a virtualized one.

of the execution cycle during each benchmark by comparing the measured times (Table 5.6). Note that the more time the benchmark takes, the less relevant the cost of process injection is. Notice also that we don't include in this estimation the confidence of the result, which in cases such as the PhDigits benchmark show that the results were not stable. We do our estimations with the following calculation. Our estimation shows an overhead below the 10% for eight of the eleven benchmarks in both measured cases.

$$estimatedOverheadOf(X) = \frac{measurementOn(X)}{normalExecution}$$

5.8 Non Implemented Aspects

For the sake of completion, we document in this subsection the aspects that have not been yet implemented in our prototype solution.

5.8.1 JIT Compilation

Just In Time (JIT) compilation is available in the production distribution of the Pharo VM. The existing JIT compiler doubles the performance of Pharo Stack VM while adding yet another complex component in the VM machinery. To reach such performance, it makes assumptions on the memory layout of the

Benchmark	Espell VM Overhead	Hypervisor Overhead
RegexDNA	1.03x	1.03x
KNucleotide	1.02x	1.06x
SpectralNorm	1.06x	1.09x
BinaryTrees	0.94x	1.01x
Mandelbrot	1.11x	1.16x
ReverseComplement	1.11x	1.26x
ThreadRing	0.88x	1x
PiDigits	0.77x	0.77x
Meteor	1.05x	1.06x
NBody	0.99x	1.03x
Fasta	1.12x	1.21x

Table 5.6: **Execution Cycle Overhead.** Comparing the cycle overhead from a non virtualized Pharo VM to a virtualized one.

running application runtime. For example, it requires commonly accessed objects such as nil, true or false to remain in the same memory position to optimize their access.

However, Espell introduces more than one application runtime in the same heap *e.g.*, more than one version of special objects such as nil are present in the same heap. Moreover, these objects can be moved in memory on GC compaction. This breaks the JIT assumptions. Thus, supporting JIT compilation in Espell would require a complete reengineering of the existing JIT compiler and possibly the VM.

5.8.2 Plugin and Native Libraries State

The Pharo VM allows one to access resources from outside the application runtime through native libraries and VM plugins. A VM plugin is a native library that satisfies a particular interface to communicate with the VM. Native libraries may keep their own state and may not be designed to be loaded several times in the same process. In particular, VM plugins are often developed assuming the existence of only one application runtime so their state is global for the whole VM.

The main VM modifications we made in our prototype to support the basic execution of several application runtimes do not include modification

on VM plugins, nor does it handle the case of loading multiple versions of the same native library from different runtimes. The usage of some of these elements in Espell may not be fully working.

5.8.3 Finalization of External Resources

Pharo VM supports the concept of weak references. If an object is only referenced by a weak reference, this object will be garbage collected and this weak reference replaced by a reference to nil. However, before garbage collecting the object, the *finalization mechanism* will send the finalize message to a finalizer object in charge of releasing any resources it may be holding. Finalization is useful to release resources external to the application runtime such as files or sockets. In Pharo, this finalization process is activated from the VM but executed by the application runtime.

In Espell and because of the shared garbage collection, it may happen that a GC activated by one application runtime (the active application runtime) collects an object from another suspended application runtime. In such a case, the VM will activate only the finalization of the active application runtime. Then, the collected object from the suspended application runtime does not have the possibility to finalize its resources. This may cause external resource leaks, since they can be garbage collected but not properly finalized and released. A possible solution is to force garbage collection before each context switch, which would have an impact in performance.

5.9 Conclusion and Summary

This chapter explores the implementation aspect of our Espell prototype on the Pharo programming language. Our prototype includes a language-side library written for the Pharo and modifications in the Pharo VM. We based our prototype in the *Stack* Pharo VM flavor. This VM is the same as the production Pharo VM except for the absence of a JIT compiler.

The Espell prototype allows the co-existence of two object-oriented application runtimes by the means of manipulating Pharo's special object array, an array object referencing objects that the VM access directly at runtime. One of these application runtimes takes the role of the hypervisor and manages the other one (the virtualized one) through an object space. An object space has a high-level interface to manipulate the interactions between the Pharo VM and the special objects array, and so, control the context switch. The hypervisor is outside the virtualized runtime and controls it though the API it offers.

Objects inside the virtualized runtime are manipulated through mirrors

objects inside the hypervisor. These mirrors are implemented by the combination of existing VM primitives and do not require new VM support for their implementation. Our mirror hierarchy covers all kind of objects that the VM supports. Particularly, the already existing reification of execution-related elements such as processes and contexts allows us to implement their manipulation through mirrors and use the same mechanisms as for regular objects.

Internally, all running application runtimes in Espell share the same object heap. This decision eases our prototype implementation as no special support for cross-runtime references is needed. However, this has an impact on the time consumed by the GC over a bigger heap.

The benchmarks we run let us make several conclusions. First, that there are improvements to make in our mirror implementation. Second, that our process injection implementations introduces an important overhead for the injection of smaller computations while it shows an acceptable overhead in the case of longer computations. Finally, the game benchmark suite shows that the execution cycle control has an acceptable overhead that is for the majority of our benchmarks below 10%. We would also like to note that our implementation was biased towards its flexibility and no real effort was spent on optimizations, where there is still room for improvement. A fully engineered solution should improve specifically the JIT compiler, the state of native libraries and plugins, the shared GC and the problems in object finalization.

Part III

Bootstrapping: Explicit Runtime Generation

Bootstrapping Object-Oriented Languages

Chapter 6

Contents

Introduction

The language initialization is the step during the execution of a program where the application and language runtime of such program is generated *i.e.*, the initial structures and constructs of a language runtime are set up in primary memory. As described in The Art of the Metaobject Protocol (AMOP) [KdRB91], a language initialization solves the *bootstrapping issues* of a language runtime. To achieve this the language initialization is often located in the virtual machine (VM), where it can solve these issues without depending on running code on the language under construction.

For example, Figure 6.1 shows an excerpt of the code that initializes the class hierarchy in the Ruby VM written in C[1]. From this code, Ruby's basic class hierarchy is composed by BasicObject as its root, followed by Object, Module and Class. These classes are first created manually without instantiating any class, and once the class Class is available, their class references are updated. As we stated in Chapter 2, this has many negative consequences. First, the VM (the Ruby VM in this case) fixes this basic class hierarchy and prevents us to change it without changing the VM. In addition, an unclear separation between the VM and language concerns in the VM code makes

[1]Taken from the version 2.1 of the Ruby VM in `http://svn.ruby-lang.org/repos/ruby`

```
1  void Init_class_hierarchy(void) {
2      rb_cBasicObject = boot_defclass("BasicObject", 0);
3      rb_cObject = boot_defclass("Object", rb_cBasicObject);
4      rb_cModule = boot_defclass("Module", rb_cObject);
5      rb_cClass =  boot_defclass("Class",  rb_cModule);
6
7      rb_const_set(rb_cObject, rb_intern("BasicObject"),rb_cBasicObject);
8      RBASIC_SET_CLASS(rb_cClass, rb_cClass);
9      RBASIC_SET_CLASS(rb_cModule, rb_cClass);
10     RBASIC_SET_CLASS(rb_cObject, rb_cClass);
11     RBASIC_SET_CLASS(rb_cBasicObject, rb_cClass);
12 }
```

Figure 6.1: **Code of the Ruby VM that initializes the class hierarchy (excerpt).** The VM code fixes the language class hierarchy.

it harder to change and adapt the language to other circumstances. Finally, when this VM is written in a low-level language, we rely on the tools and abstractions this low-level language provides instead of the more powerful ones from the high-level language.

Notice that this language initialization step is often baptized as the language bootstrapping step. However, we use for it the name *language initialization* and present the concept of **bootstrapping** as it is known in the context of compilers (where a compiler can compile itself). Then, bootstrapping is the process of expressing the language initialization in the language it defines at the end. Though this concept is know in the context of compilers, we explore and expand this concept in the context of object-oriented languages. We can then see bootstrapping as a high-level low-level programming approach for language runtime generation [FBC+09] (cf. Section 6.1).

We illustrate a bootstrap process for an object-oriented language runtime through an example: the bootstrap of the Pharo language (cf. Section 6.2). However, executing such a bootstrap is not straight forward: we cannot execute code in the runtime under construction using the same runtime. For example, we cannot send a message when no methods are yet installed. This example raises the question: *how do we execute such a bootstrap?*. We show how Espell solves this problem and represents a robust infrastructure for language runtime bootstrapping (cf. Section 6.3). Using Espell, the bootstrapped application runtime is hosted inside a virtualized runtime and the bootstrap process takes the role of an hypervisor. The virtualized runtime is initially empty and the bootstrap hypervisor fills it with objects gradually.

By bootstrapping, the definition of the language runtime becomes circular *i.e.*, the language definition expresses its own initialization in itself. This presents the benefit of using the abstraction power and tools of the language it defines (cf. Section 6.4). The bootstrapping hypervisor can execute the lan-

guage's circular definition through a virtual interpreter specialized for the case of bootstrapping, the *bootstrapping interpreter*. The bootstrapping interpreter manipulates objects inside a virtualized runtime while they cannot execute code by themselves, solving the circularity problems (cf. Section 6.5). Additionally, our infrastructure allows us to trace the execution of the bootstrap code to detect the impact that a change in the language may have in the bootstrap process (cf. Section 6.6).

6.1 Bootstrapping

The idea of a *bootstrap* is well known in the context of compilers, where a compiler is considered bootstrapped when it can compile its own source code to produce itself. For example, a bootstrapped C compiler is a compiler that, by using its own source code written in C, can produce another compiler with its same behavior. Notice that the input source is not a direct description of the C language, but a description of the compiler itself *i.e.*, the description of a program that builds a program. Notice as well that the output of this bootstrap is an executable representation of that description *i.e.*, the machine code that will be loaded and run by the operating system.

Bootstrapping an object-oriented language runtime should not be mistaken for just writing a compiler of the language in the same language. The compiler of a high-level object-oriented language produces usually the bytecode of a class or method, which is an incomplete view of it: it does not describe other objects that are indeed needed to run this program nor the relation of this compiled code with other objects during runtime. However, following the idea of the C compiler we can then define the bootstrapping process of an object-oriented language runtime as follows:

Definition 1 (Language runtime Bootstrap) *It is a process whose input is the definition of a language written in the same language, and whose output is a language runtime.*

This definition applied to an object-oriented high-level language implies the following:

The language elements should be self-described. The input language definition must include a description of the classes, methods and/or objects that are part of the language runtime to build. This description should be expressed in the same language we are building. From now on we will call these entities the *base-level* entities.

The procedure to create the language elements is also self-described. The input definition must also include the knowledge to recreate itself: the

basic operations to create its classes and methods, or initialize the basic structures of the language runtime *e.g.*, the runtime table of symbols or its threads. From now on we will call the entities in charge of this task the *meta-level* entities.

The output is a graph of objects. The output of a bootstrap process is the graph of objects that represents the language definition *i.e.*, the classes, methods, threads and other objects that allow programs to run. These elements are created by the bootstrap inside an application runtime for its execution.

In the following sections we will explore how object-oriented language can be bootstrapped starting by an example. Then, we present how the virtualization features of Espell provide support for such a task.

6.2 Bootstrapping Through an Example

The main component of a bootstrap process is the *language definition i.e.*, the definition of the elements of the language and the procedure to create them. The language definition of a bootstrap is circular *i.e.*, it is written in the same language it defines. In this section we illustrate a bootstrapping process through an example: the bootstrap process for the Pharo language. For this, the example follows the excerpt of Pharo's language definition shown in Figure 6.2. Notice that while such a language definition is not difficult to express *per se*, it raises the question of *how it can be executed*. This is specially challenging due to the circularity of the language definition. For example, we need classes to be defined to execute this exact bootstrap description. This question leads us to ask ourselves about the infrastructure required to be able to manage this and other different bootstraps. The sections that follow describe how Espell provides a suitable infrastructure to tackle these problems.

Step 0: Setup of the environment. The precondition of a bootstrap process is to start an empty application runtime. The bootstrap process will fill this application runtime as it is executed. This application runtime will contain at the end of the process all the elements of the language runtime we are building.

Step 1: Create the first well-known objects. The first step of the process is to create the nil, true and false objects that are needed for execution (cf. Figure 6.3). It is important that nil is the first object we create in Pharo, as the rest of the objects we create will have their fields initialized to it. We also

```
1  nilObject := UndefinedObject basicNew.
2  trueObject := True basicNew.
3  falseObject := False basicNew.
4
5  globalTable := GlobalTable basicNew.
6  globalTable
7      at: #GlobalTable
8      put: globalTable.
9
10 SymbolTable initialize.
11 ...
12 ...
13
14 nil subclass: #ProtoObject
15     instanceVariableNames: ''.
16
17 ProtoObject subclass: #Object
18     instanceVariableNames: ''.
19
20 Object subclass: #UndefinedObject
21     instanceVariableNames: ''.
22 ...
23 ...
24
25 ProtoObject >> isNil
26     ^ false
27
28 UndefinedObject >> isNil
29     ^ true
30 ...
31 ...
32
33 Float initialize.
34 Processor initialize.
35 ...
36 ...
```

Figure 6.2: **Excerpt of the Pharo bootstrap language definition.**

create true and false, as they are required for code execution. Notice that to execute this step we use the UndefinedObject class. However, this class is not in the bootstrapped runtime yet and creating it would introduce another *paradox* (*e.g.*, creating the UndefinedObject class requires the nil object). To execute this and other similar pieces of code the bootstrap process breaks the circularity by automatically introducing a *stub class*. Stub classes contain the minimal requirements to execute simple operations such as the instantiation primitive (basicNew in this example). We will dive into stub classes in Section 6.5.

Step 2: Create basic language structures. The basic language structures are the minimal structure we need to create all the rest of the language elements (cf. Figure 6.4). For example, creating a class requires the existence

```
1 nilObject := UndefinedObject basicNew.
2 trueObject := True basicNew.
3 falseObject := False basicNew.
```

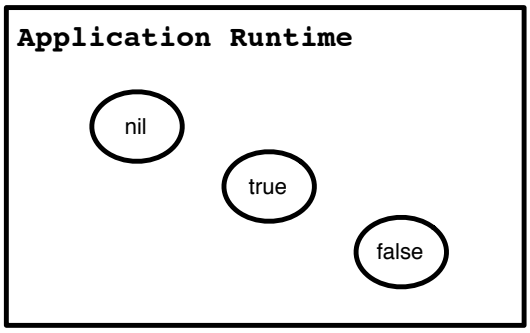

Figure 6.3: **Step 1: create the first well-known objects.** The nil, true and false instances are created inside the application runtime. Their classes are however not yet installed. They are installed as stubs that will be later on replaced.

of a table of global objects to install the class, and a table of unique strings or symbols to have a unique identifier for it. We must create these basic structures from the very beginning, as the rest of the process rely on them.

Note that this example presents new paradoxes. First, the apparition of GlobalTable and SymbolTable present the same problems as UndefinedObject did before, and thus it is solved in the same way. Additionally, we need to send the at:put: message to the GlobalTable while the at:put: method does not yet exist. Also, this method belongs to a superclass of GlobalTable that is not yet installed: Dictionary. To solve this, our bootstrap process will look up the code to execute inside the language definition and execute it using the bootstrapping interpreter. By doing that, our bootstrap process can benefit from all the code that exists in the language to define itself. We will dive into how methods are executed before they are installed in Section 6.5.

Step 3: Create classes. We create the classes that the language definition requires in the language runtime (cf. Figure 6.5). For this, we use the *class builder* present in Pharo's language definition through the bootstrapping interpreter. The class builder validates that a class can be created, creates it and installs it inside the global table. It encapsulates the complexity of class creation *e.g.*, first-class layouts, traits, first class instance variables.

We replace existing stubs (*e.g.*,UndefinedObject, True or GlobalTable) by their new versions. This step does not yet install methods, since they can reference classes that are about to be created. We will create all methods at once in the next step once all classes are created.

Step 4: Installing methods. We compile (if needed) and install each of the methods present in the language definition into their respective classes. Method literals are bound to their corresponding literals or global objects (*e.g.*, classes).

```
globalTable := GlobalTable basicNew.
globalTable
    at: #GlobalTable
    put: globalTable.

SymbolTable initialize.
```

```
Dictionary >> at: key put: anObject
   | index assoc |
   index := self findElementOrNil: key.
   assoc := array at: index.
   ...
```

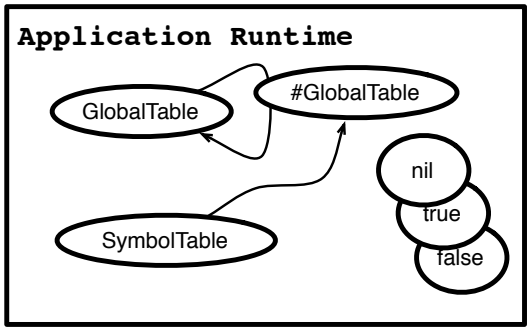

Figure 6.4: **Step 2: Create basic language structures.** We create those structures that are required to create the rest of the language elements, such as the symbol and global tables. In the figure we can see that the global table references itself as a global and the symbol table references the GlobalTable symbol. Methods such as at:put: are looked up in the language definition and executed using the bootstrapping interpreter.

```
nil subclass: #ProtoObject
    instanceVariableNames: ''.

ProtoObject
    subclass: #Object
    instanceVariableNames: ''.
```

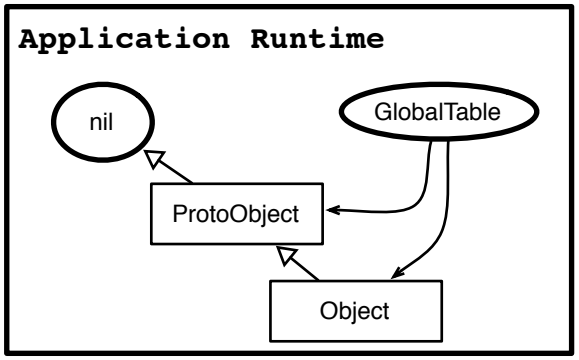

Figure 6.5: **Step 3: Create classes.** Classes are created by the class builder and installed inside the global table.

Step 5: Initialization. This last step consists in the execution of the class initializers. In Pharo, it consists in sending the message initialize to those classes that require some kind of extra initialization. For example, at this point we create well-known float values (*e.g.*, NaN or Infinity) and the thread machinery. At the end of this step, the language runtime is able to execute code by itself.

As we observed, a bootstrap looks easy to express as we are using high-level abstractions to do so. However, executing it presents several challenges in the form of paradoxes (or informally, the chicken-egg problem). For example, creating an object requires a class and creating a class requires other objects. Installing a method requires sending a message and sending a message

```
1 ProtoObject >> isNil
2     ^ false
3
4 UndefinedObject >> isNil
5     ^ true
```

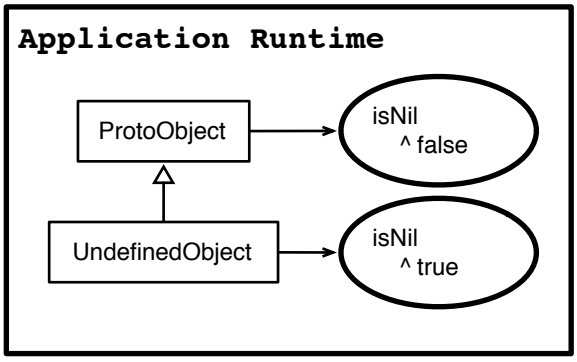

Figure 6.6: **Step 4: Installing methods** The methods that appear in language definition are installed inside the application runtime. In this example the ProtoObject and UndefinedObject classes contain each an isNil method.

```
1 Float initialize.
2 Processor initialize.
```

Figure 6.7: **Step 5: Initialization.** Class initializers are executed to setup the state of the class. In this example we initialize the Float and ProcessScheduler classes sending them the initialize message.

requires methods to be already installed. In the following sections we will describe what is an infrastructure that allows us to execute such bootstrap process.

6.3 Bootstrapping with Espell

To execute such a language definition, we developed a bootstrap infrastructure for an object-oriented language based on Espell (Figure 6.8). This infrastructure presents three explicit components that decouple the initialization of the language runtime from the VM initialization so we can define different languages on top of the same VM. Additionally we can modify the language runtime using the abstractions and tools of the high-level language.

Language definition. The code that defines the initialization of the language runtime is extracted and expressed in the same language it defines. This code is expressed as normal code of the language it defines. Thus, during the language initialization we can benefit from the abstractions and tools of the language we are defining. For our prototype implementation this self-description is a set of files.

Virtualized Bootstrapped Language Runtime. The bootstrapped language is initialized inside a virtualized runtime. This virtualized runtime is

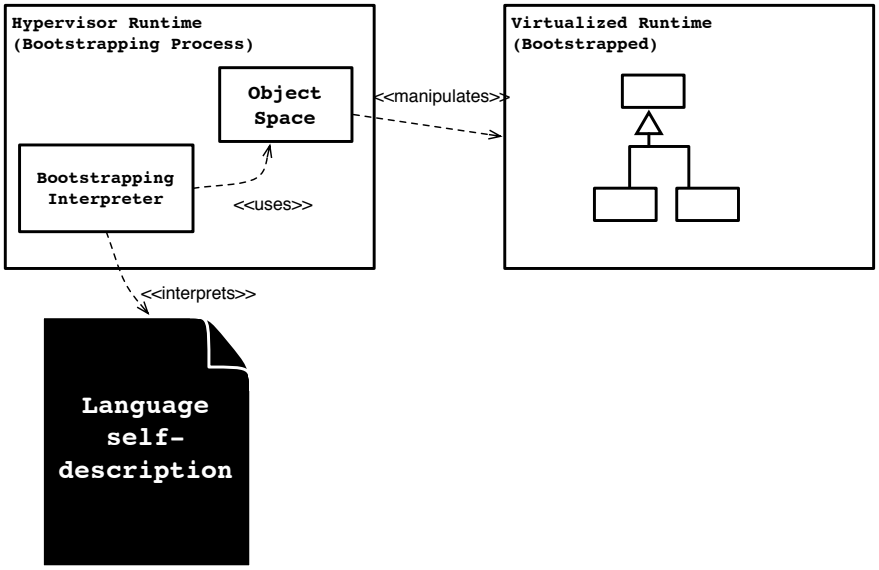

Figure 6.8: **Solution overview.** A bootstrapping interpreter uses the self-description in the language definition to build the language through a clear VM-language interface.

initially empty and the bootstrap process fills it with objects before it reaches the execution point. As such, we can use the object space VM-language interface for its manipulation. This also serves the purpose of identifying what are the VM and language concerns during language initialization to easily decouple them.

Bootstrapping Hypervisor and Interpreter. A *bootstrapping hypervisor* contains a virtual interpreter specialized for bootstrapping, the *bootstrapping interpreter*. The bootstrapping interpreter is a specialized AST interpreter that can execute the code available in the language definition under the initial absence of classes and objects.

6.4 The Circular Language Definition

The language definition describes the language we are bootstrapping. During the bootstrap process, the language runtime under construction traverses several stages until it is finished. When the bootstrap process starts, the language runtime is not yet able to execute code by itself *e.g.*, it cannot resolve the method lookup because the class hierarchy is not created. As we install packages or classes the language runtime reaches its *execution point*, where it contains already the minimal set of elements it needs for execution. Later on, we can install reflective features into the language runtime to get it inside the *reflective spectrum*. Figure 6.9 illustrates the stages of a language runtime construction during the bootstrap process.

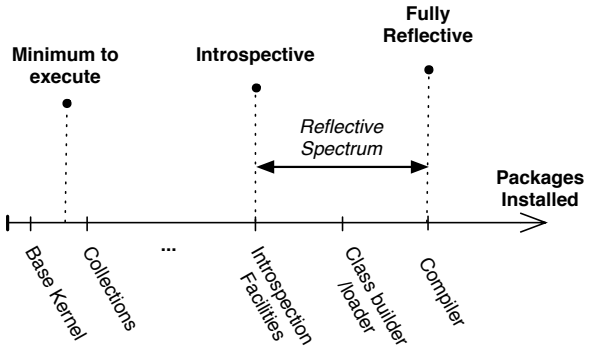

Figure 6.9: **Bootstrap Phases.** Initially, a language runtime does not contain the minimal elements to execute code. As the bootstrap process installs elements, it reaches the point of execution where it can run autonomously. Later, when (if) reflective features are installed, it reaches the reflective spectrum.

The Espell-based bootstrap process takes as input this language definition, parses it and builds the abstract syntax trees (ASTs) of the language elements (*e.g.*, classes and methods). ASTs ease the manipulation of the language definition for its compilation, interpretation and extraction of information *e.g.*, we can easily extract variable and class names, superclasses, and know how to bind names during compilation. Figure 6.10 illustrates our basic AST model. For example, a Class definition contains amongst others its name, its superclass, a list of the instance variables that it defines and the source code of its definition. A method definition contains its selector, a list of its statements and its full source code. For clarity of the figure, we don't include sub-method nodes such as statements, assignments or message sends in it.

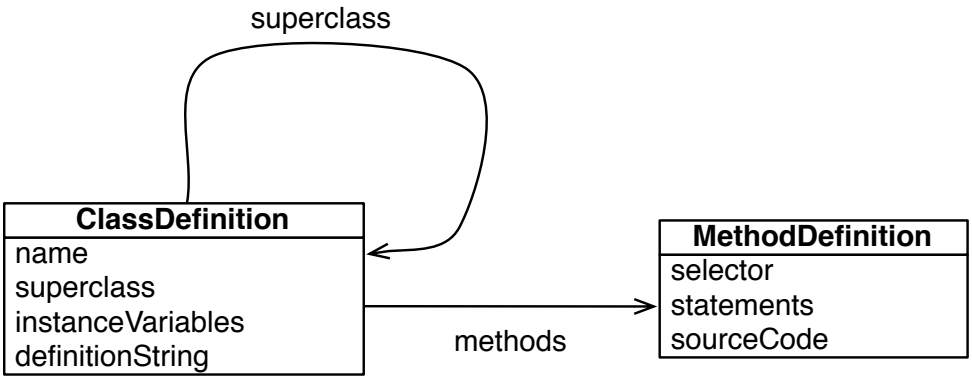

Figure 6.10: **Basic Bootstrap AST Model.**

Inside a language definition we can identify two main sub-components, the language base-level entities and the language meta-level ones.

6.4.1 Language base-level entities

The language definition contains all those classes and methods that must be built and available to a program to run. We call these the base-level entities of a language definition. The base-level entities describe the basic class hierarchy of the language and basic objects such as strings or integers. The following code snippet illustrates how the base-elements of Pharo are described inside the language definition. Notice that this code snippet has the same responsibility as the Ruby VM code we showed in the introduction of this chapter.

```
1  nil subclass: #ProtoObject
2      instanceVariableNames: ''.
3
4  ProtoObject >> isNil
5      ^ false
6
7  ProtoObject subclass: #Object
8      instanceVariableNames: ''.
9
10 Object subclass: #UndefinedObject
11     instanceVariableNames: ''.
12
13 UndefinedObject >> isNil
14     ^ true
15
16 Object variableByteSubclass: #String
17     instanceVariableNames: ''.
18
19 ...
```

6.4.2 Language meta-level entities

The language definition must also include elements that know how to create the base-level entities *e.g.*, a compiler or compiler interface (to an external compiler). We need the language meta-level elements to create methods and classes during the bootstrap. Notice however that a language runtime must not necessarily include these elements at the end of the bootstrap. A bootstrap process that builds and introduces the meta-level elements inside the bootstrapped runtime is a *reflective language runtime*. A language that contains both introspection and full intercession facilities is a fully reflective language. A fully reflective language does not only have the minimal set of elements to run, but also the minimal to be autonomous: it can create/load classes and methods without any external component (compiler, class builder, interpreter).

```
Object subclass: #Compiler
    instanceVariableNames: ''.

Compiler >> compile: sourceCode in: aClass
    ...

Object subclass: #ClassBuilder
    instanceVariableNames: ''.

ClassBuilder >> buildClassNamed: aClassName withSuperclass: aSuperclass
    ...
```

The meta-level entities include in addition the code that defines the bootstrapping process *i.e.*, the steps that must be followed to create a coherent and well formed language runtime.

```
nilObject := UndefinedObject basicNew.
trueObject := True basicNew.
falseObject := False basicNew.

globalTable := GlobalTable basicNew.
globalTable
    at: #GlobalTable
    put: globalTable.

SymbolTable initialize.

...
...
Float initialize.
ProcessScheduler initialize.
```

6.5 The Bootstrapping Interpreter

The bootstrapping interpreter is a virtual interpreter that interprets code expressed in the bootstrapped language. The bootstrap process uses the bootstrap interpreter to execute the language definition inside the virtualized runtime before it reaches the execution point. Its design presents the following important properties that allow it to achieve this:

Alternative method lookup. Before reaching the execution point, the class hierarchy of the language runtime is incomplete, or part of its methods are not yet installed. The bootstrapping interpreter implements an alternative method lookup mechanism to allow message sending before we reach the execution point: methods are looked up in the definition of the language instead of the hierarchy in the language runtime; a mapping is kept between classes created in the language runtime and their definitions in the language definition to know where the lookup should start.

Automatic class stubs. The bootstrapping interpreter does also solve most of the well known bootstrapping issues (*e.g.*, how to create an instance

before a class exists) in a generic way by using class stubs. When an inexistent class is needed during the bootstrap process, the bootstrapping interpreter creates an empty class to take its place respecting the VM format for it. The bootstrapping interpreter will be able to create instances of this class and map it to its corresponding definition to perform the method lookup. This class cannot, however, initially perform reflective operations as it does not contain any reflective information. When the real class is created later on in the process, it replaces the stub. For this purpose, we extended the object space interface with one operation that allows the creation of an object whose class pointer is not initialized.

```
1 objectSpace {
2     mirror allocateObjectOfSize(int size);
3 }
```

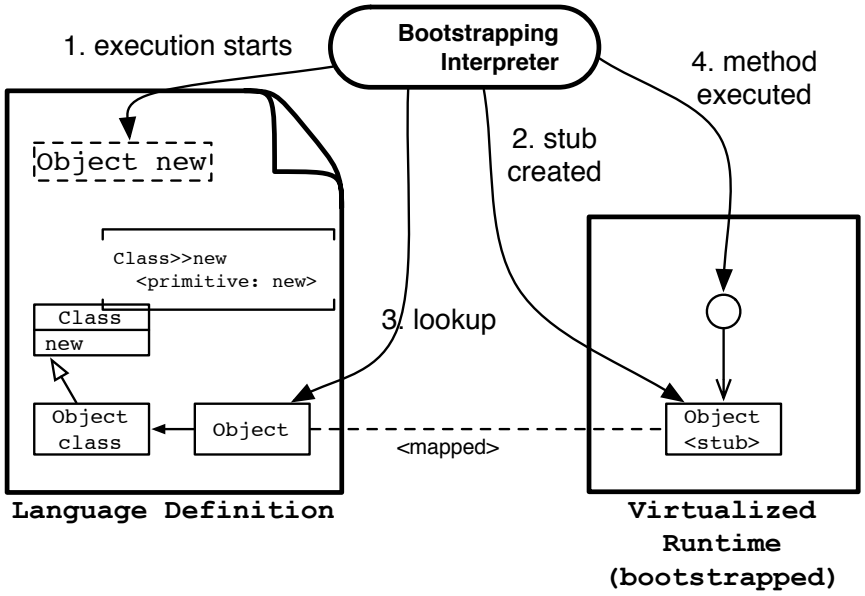

Figure 6.11: **The Bootstrapping interpreter in action.** A stub class is created for a non existent class. Each class is mapped to its description in the language definition. The lookup is then performed inside the language definition. Once the method is found, it is executed inside the language runtime.

Figure 6.11 illustrates with an example the behavior of the interpreter, particularly in the execution of the "Object new" expression. First, if the class Object does not exist, it creates a stub Object class and maps it to its corresponding definition in the language definition. To interpret the new message, the bootstrapping interpreter performs the method lookup from the class of the object in the language definition. As the class from the language runtime and the language definition are mapped, the bootstrapping interpreter

knows where to start the method lookup. The method is found in the Class class and executed in the language runtime. As a result, an instance of the Object class is created.

By using the bootstrapping interpreter, the language definition serves both as a source for compiling the code that will be part of the application runtime, and for the execution of the bootstrapping process itself. A change to the language definition will affect both and avoids major code and logic duplications between VM, language, and language initialization. For example, Figure 6.12 illustrates how we can use the bootstrapping interpreter to use the Dictionary code from the language definition instead of the C function that is part of the VM, as we showed in Chapter 2.

```
1 Bootstrap>>createDictionaryWith: n
2     "Create a dictionary in the new language runtime"
3     ^ interpreter
4           execute: 'Dictionary new: size'
5           binding: { 'size' -> n }.
```

Figure 6.12: **Avoiding logic duplications with the bootstrapping interpreter.** This example shows how the bootstrapping interpreter does not duplicate the logic of the Dictionary»new: method, but executes it inside the virtualized runtime.

6.6 Continuous Bootstrapping

Building continuously a language runtime provides the language engineers with the same benefits of continuously building another application: automated integration and testing, quick and continuous feedback on the applied changes. This continuous feedback should give the language developer with the information and tools to resolve conflicts and problems: it should clearly show which was the *impact* of such a change in the process. A change introduced in the language impacts directly on the definition of the language (Figure 6.13). The changed definition is used in turn by the bootstrap process to bootstrap the new version of the language runtime, thus the change has also an indirect impact on the bootstrapped language.

However, not every change in the language definition may impact the bootstrap process: some code is only meant to be in the result of the bootstrap process but it is not used by the bootstrapping interpreter *e.g.*, changing the set of final classes introduced by the bootstrap does not alter the bootstrap process. To identify the impact of a change on the language in the bootstrap process we introduced as a second output of our bootstrapping interpreter an execution trace containing all the language elements that were used to

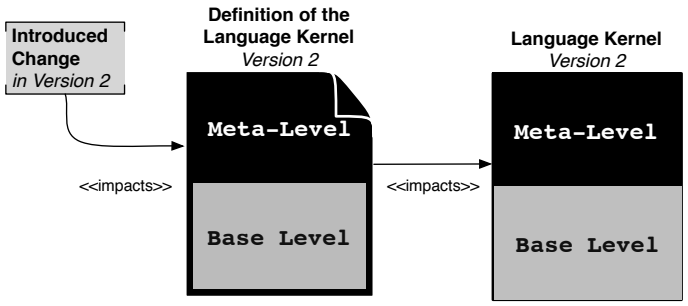

Figure 6.13: **How a change impacts the bootstrap process.** A change in the language may impact directly the definition of the language, which in turn impacts the bootstrap process.

bootstrap: any change on these elements may have an impact on the process. Then, to produce useful feedback for the changes made by a language developer, an *impact resolver* measures the impact of a change on the bootstrap process by comparing the introduced change to the trace of previous bootstrap execution (Figure 6.14).

Our bootstrapping infrastructure measures the impact by making a diff between the traced and changed language elements. In case a change breaks the bootstrap process, the language engineer has hints that help him spot the problem and act on it. We are working to obtain in the future more information using impact analysis techniques.

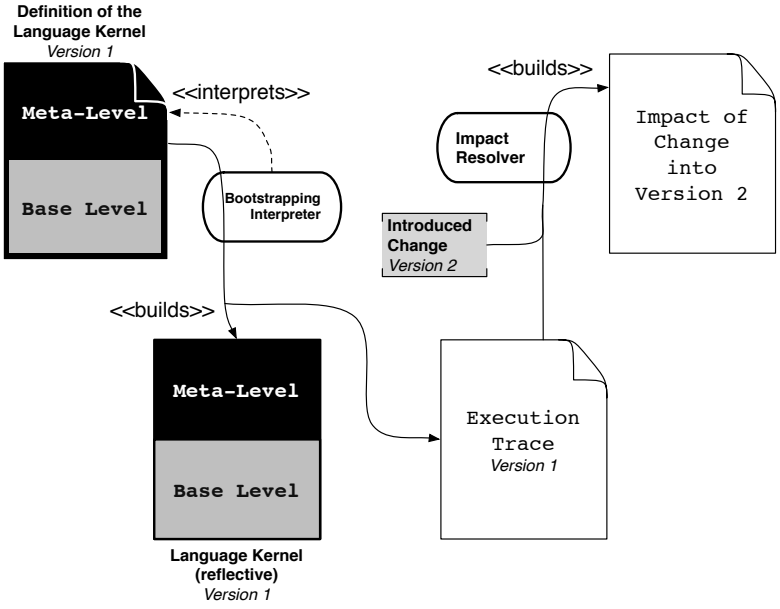

Figure 6.14: **How a change impacts the bootstrap process.** The bootstrap process execution is traced. An impact resolver decides if the introduced change will impact the bootstrap process or not.

6.7 Conclusion and Summary

Bootstrapping is commonly known by its usage on compiler building, where a compiler can compile itself. It can be generalized to the introduction of any software system to its own building process. A bootstrap process allows us to easily change this system as it is expressed in terms of itself, taking advantage of its abstractions and tools.

In this chapter we explored the bootstrap of an object-oriented language using Espell. By using Espell we can easily modify the bootstrapped language runtime even when it is empty. A virtual interpreter specialized for bootstrapping allows the execution of the language definition in an initially empty virtual runtime. On one hand, it overrides the method lookup mechanism to look for method ASTs inside the language definition instead of inside the half-built bootstrapped language, avoiding duplications. On the other hand it solves the bootstrapping issues by automatically creating class stubs and replacing them once we create and install the real classes.

Chapter 7 Bootstrapping Validation

Contents

Introduction

In this chapter we present our results while bootstrapping four different case study languages. To reuse the parsing infrastructure and the bootstrapping interpreter, the four bootstrapped languages share also the same syntax: a Smalltalk syntax. Despite these similarities, all these four language runtimes possess different models and semantics (cf. Section 7.1):

Pharo. Pharo is a fully-reflective language composed of classes and traits with first class slots and object layouts [VBLN11].

Metatalk with and without meta-level. Metatalk [NBD+11] is a language that fully decomposes the meta-level from the base-level using mirrors, allowing us to bootstrap a reflective and a non-reflective version of Metatalk.

Candle. Candle is a partially reflective Smalltalk-80 based mini-kernel that includes introspection and some self-modification features.

Finally this chapter presents some measurements (cf. Section 7.2). To keep bootstrapping practical, we optimized the critical parts of the process for both the language user and the language engineer. On one hand language users do not usually need to modify the language runtime but use it, independently of the language initialization process it provides. To suit this scenario we do not build the language runtime each time: we generate a snapshot with a cached version of it. On the other hand we find language designers/engineers whose job is to change the language runtime. For them the bootstrap process must provide with an acceptable development cycle for activities like debugging. With this case in mind, we optimized the *bootstrapping interpreter* with a dynamic compilation technique. Each of the measurements we present below

were made on a 2.2 Ghz Intel Core i7 machine with 8 GB memory 1333 Mhz DDR3.

7.1 Languages Used for Experimentation

Using Espell we bootstrapped four different object-oriented languages. These languages share two common properties: the execution model of its VM and the Smalltalk syntax. They present different programming models for the developer *e.g.*, the absence of reflective features, their inclusion through mirrors, first-class variables and traits. Figure 7.1 shows how these four languages are placed in the language spectrum. We explain more in detail these languages in the following subsections. Also, Appendix C details the list of classes and methods that are included in each of them.

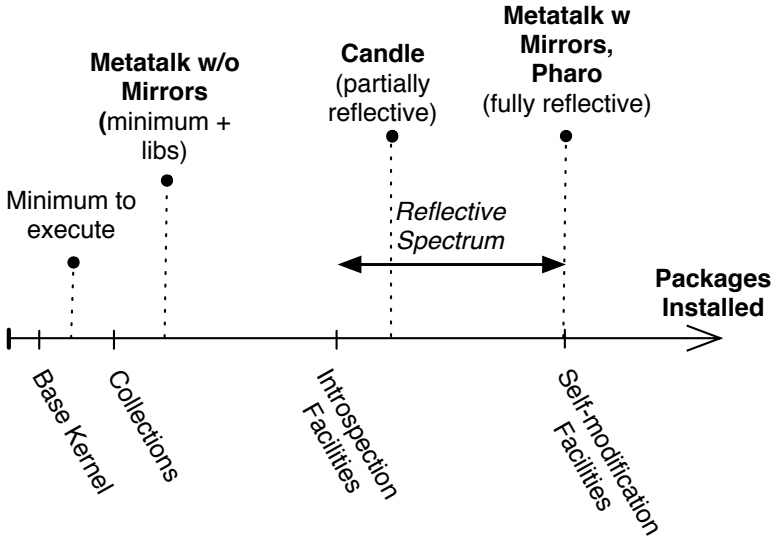

Figure 7.1: **Bootstrapped Languages Spectrum.** How the languages we bootstrapped are placed in the reflective spectrum. In particular, Metatalk with and without its mirrors is in different extremes of the spectrum.

7.1.1 Language I: Pharo

Pharo [BDN+09][1] is an object-oriented reflective Smalltalk-inspired programming language. As it is a Smalltalk-80 inspired language, its class model includes implicit metaclasses: each class has its own metaclass, an instance of Metaclass. Pharo also extends the execution model of its VM with traits [SDNB03] and class extensions (*i.e.*, the ability to add methods to a class that belongs to another package). Finally Pharo has first class instance

[1]The bootstrapped version of Pharo resides in `https://github.com/guillep/PharoKernel`

variables (slots) structured in object layouts [VBLN11]. Figure 7.2 shows how the elements of the language are related to each other; the diagram is not meant to reflect the actual class graph (which is more complex) but the language concepts.

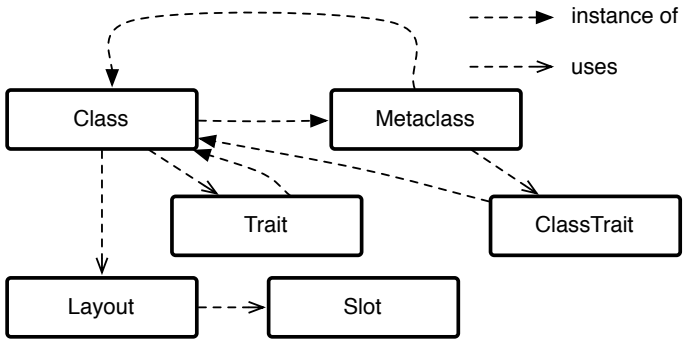

Figure 7.2: **Simplified Pharo object model schema.** In Pharo each class has a metaclass. Metaclasses are defined circularly. Both classes and metaclasses makes use of trait objects to define part of their behavior. Classes also have layout that includes first class instance variables (slots). This schema does not represent the actual object graph, but a simplified picture.

Pharo is a fully-reflective language, placed at the end of the reflective spectrum. The Pharo language includes introspection in the kernel itself, and also self-modification stratified in three levels: object mutation facilities, a class builder and a compiler. The main challenge in Pharo is that the kernel itself of Pharo is defined by Traits: *e.g.*, the Trait class uses a Trait. First class slots are also part of the self-description of the language. This introduces new bootstrapping issues that must be resolved at bootstrapping time.

7.1.2 Languages II and III: Metatalk with and without Mirrors

Metatalk [NBD+11][2] is a reflective language where reflection is fully decomposed in explicit meta-objects, namely mirrors [BU04]. Metatalk makes the usage of reflection explicit: a program's execution takes place in the base-level of the language runtime, and it jumps to a meta-level when a mirror is used. Notice that Metatalk mirrors are not related to Espell mirrors. Metatalk mirrors inhabit inside the virtualized runtime while Espell mirrors are outside it and Metatalk does not know about them. Metatalk's class model is simpler than Smalltalk's class model. It does not impose metaclasses. Instead, all classes are instances of the single Class class. If there is a need for metaclasses (to share behavior between classes), the developer can write his or here own explicit metaclasses (Figure 7.3).

[2]The bootstrapped version of MetaTalk resides in `https://github.com/guillep/MetaTalk`

Metatalk mirrors decompose reflective behavior as well as the language meta-information *i.e.*, class' names, field order and names amongst others are part of its mirrors, and thus, they belong to the meta-level. When there is not a need for reflection, a Metatalk program can discard its meta-level with all the meta-information in it. The only staying meta-information is the one required by the VMs execution model, such as the superclass relations, or the method selectors. This decomposition allows us to bootstrap Metatalk with or without its meta-level. This results in two different language runtimes: Metatalk base-level has no reflection at all, while Metatalk with both the base and the meta level is a fully-reflective language.

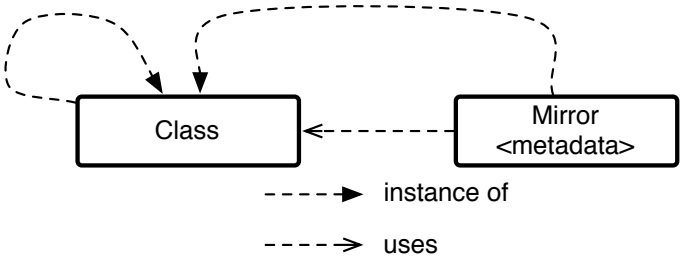

Figure 7.3: **Simplified Metatalk object model schema.** In Metatalk classes have no implicit metaclass. All classes share the same class. Mirrors are simple objects, thus instances of classes, that reflect on a class and contain their metadata. This schema does not represent the actual object graph, but a simplified picture.

Metatalk can be bootstrapped in two different ways. A non-reflective bootstrap initializes only the main classes of the language but does not create its meta-level. The non-reflective bootstrap does not contain mirrors. A second bootstrap creates a reflective Metatalk, which based on the latter one introduces the mirror instances with their corresponding metadata. We could bootstrap easily Metatalk in such a way due to the clear decomposition of its reflective elements.

7.1.3 Language IV: Candle

Candle[3] is a Smalltalk-based language with a micro language runtime. Its class model includes implicit metaclasses like Smalltalk's and Pharo's. However, Candle has no support for traits or slots (Figure 7.4). We built Candle's language runtime by adapting MicroSqueak [Mal] to run on top of the Pharo VM primitives. This micro language runtime was designed with the explicit goal of being the minimal distribution for the Squeak Smalltalk language.

[3]The bootstrapped version of Candle resides in `https://github.com/guillep/PharoCandle`

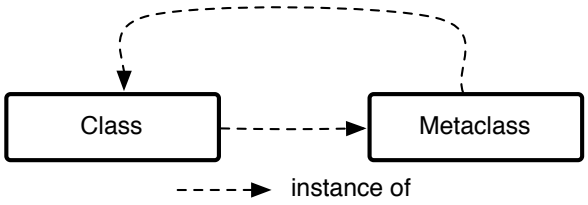

Figure 7.4: **Simplified Candle object model schema.** Candle follows a more traditional Smalltalk-80 model. In Candle each class has a metaclass. Metaclasses are defined circularly. There are no traits. This schema does not represent the actual object graph, but a simplified picture.

Candle is a partially reflective language defined by a total of 49 classes and a reduced set of methods. Candle includes a minimal core of the language, a basic collection library and basic file IO support. It also provides object introspection and mutation facilities. It does not include, however, a class builder or compiler to extend itself.

7.2 Measurements

In this section we present the benchmarks we did to measure the bootstrap time of each of our four languages using our standard infrastructure. Table 7.1 shows the time to bootstrap each of the four languages using an unoptimized AST interpreter. This time includes just the bootstrap process: from parsing the code in the language definition to its complete setup. It does not include the execution time once it is bootstrapped. Indeed, once bootstrapped the language runs at full VM speed. We executed each of these benchmarks 10 times. The results table puts also the results in context: it presents how many code entities (classes, traits, mirrors) and methods are built for each language. Notice that the bootstrapping time depends on the amount of elements it builds and also on their complexity. For example, creating a class in Pharo involves a biggest graph of objects than in the other two languages (because of the introduction of traits and class layouts). Section 7.3 introduces two optimizations we did based on these measurements, that focus on the startup time and the development cycle of the bootstrap.

We can observe from our measurements that bootstrapping Metatalk takes on average 1 second if no mirrors are created and 13 seconds if we include reflection in it. Candle bootstrap is slower, on the order of 1 minute and a half, mainly because it contains eight times more methods than the Metatalk. We can see that a plain AST-based bootstrapping interpreter has a bigger impact on the bootstrap time if the language contains complex structures to initialize. Indeed, creating a Pharo class using the AST inter-

Language	Code entities / Methods	Bootstrap time
Pharo	626* / 6812	2h30m +/-10m
Candle	100* / 875	87s +/-8s
Metatalk w/o mirrors	25 / 114	0.957s +/-0.112s
Metatalk reflective	58* / 166	13.697s +/-0.061s

Table 7.1: **Building Benchmarks.** Comparing the execution time of the bootstrapped languages using AST interpretation. (*) Pharo and Candle have implicit metaclasses, meaning that for each created class, an associated metaclass is created even if not necessary. Metatalk introduces a mirror object for each of the classes in the language.

preter is an operation that takes on average 17 seconds, because each class contains a reification of its memory layout and slots [VBLN11]. This problem is aggravated by the number of classes and methods in this language definition.

Particularly about bootstrapping Pharo, a lack of modularity of the language impacts on the number of code elements we have to build. Pharo's language runtime is historically a monolithic system which precludes us to build a minimal system. In fact, the Pharo language runtime we are bootstrapping represents a subset of the full Pharo language as it is distributed. This subset is planned to be the starting point for a more modular Pharo distribution.

7.3 Optimizations

To be useful in practice, we understand that the bootstrap process should have the following two properties: (a) be fast enough to provide a good feedback loop and allow debugging to the language engineer and (b) provide a short startup time for the language users. Optimizing a bootstrap process is indeed a challenge since we cannot optimize it statically by fixing the meta-level semantics, as changing them is the main purpose of the bootstrap. In the following sections we show how snapshotting and dynamic compilation aid in these two optimization scenarios.

Enhancing Bootstrap Time: Dynamic Compilation

Since the main purpose of the bootstrap process is to easily change the meta-level semantics and structure of the language entities we cannot fix them statically to optimize them. In exchange, we chose to optimize the interpretation cycle using a dynamic compiler. The dynamic compiler compiles the interpreted code on demand. This compiled code is cached and executed directly

on the VM bypassing the interpretation step in following executions. We implemented dynamic compilation to optimize Pharo as it presents the worst of our results (cf. Table 7.2). We reduced the total bootstrap time by a factor of 2.85. Additionally, we observed a major improvement on class creation, where the time improves from 17 seconds to less than half a second. The long class creation time impacts Pharo total bootstrap time because it is executed 313 times. Contrastingly, the initial setup of the language structures (*e.g.*, the symbol and character table, the initial threads) is executed only once where the cost of our dynamic compilation implementation increases the execution time. Notice that the current implementation does not optimize method compilation nor parsing, meaning there is still a room for improvement.

Case	**AST Interpretation**	**Dynamic Compilation**	**Gain Factor**
Total Bootstrap	2h30m +/-10m	53m +/-3.65m	2.85x
Initial Setup	4m8s +/-9s	5m20s +/-40s	0.77x
Creation of one class	17s +/-1s	0.432s +/-0.189s	39.85x

Table 7.2: **Comparison of bootstrap time in absence and presence of dynamic compilation.**

Optimizing Startup Time: Snapshotting.

The user of a programming language is concerned about writing applications that run on this programming language instead of changing the programming language. From a user perspective the initialization of the language is hidden within the startup of an application. It should be however fast and ensure always the same state. The language initialization present in production VMs provides both properties. Bootstrapping, in the sense of this paper, makes this process slower due to the interpretation step.

For language users, we overcome this slow-down by *caching* the result of our bootstrap process in a snapshot. Thus, we bootstrap a language runtime only when we change it, and otherwise we load the cached version. Caching keeps both properties of application startup: it guarantees the same state and it is faster. Table 7.3 shows a comparison in the startup time of four different programs: (a) the Pharo VM loading no-runtime and quitting to avoid the usage of the snapshot and eliminate its startup time; the Pharo VM loading (b) Pharo, (c) Metatalk and (d) Candle using snapshots; (e) starting a Ruby program that quits Ruby. We benchmarked the startup times by running each of them 10 times and making an average.

From the results, we observe that our startup time is bigger than ruby's

Artifact	Startup time
Pharo VM - no runtime	166ms +/-13
Pharo	280.8ms +/-3.4
Candle	186ms +/-7.6
Metatalk w/o mirrors	202ms +/-13
Metatalk reflective	205ms +/-11
Ruby	64ms +/-7.1

Table 7.3: **Startup time benchmarks.** Benchmarking the startup time of a Ruby application with the same in Pharo or Candle using a snapshot.

but still reasonable, under half of a second. Additionally, we can see that the startup time of the Pharo VM itself represents a big part of the startup time of each language (Figure 7.4). This particular issue is indeed orthogonal to the fact of the bootstrap and requires the VM to be reengineered to minimize its startup time.

Artifact	Benchmarked time	Pharo VM Overhead
Pharo	280.8ms +/-3.4	59.12%
Candle	186ms +/-7.6	89.25%
Metatalk w/o mirrors	202ms +/-13	82.18%
Metatalk reflective	205ms +/-11	80.98%

Table 7.4: **Startup time in perspective.** Calculating the overhead of the Pharo VM on the startup time of Pharo, Candle and Metatalk using snapshots.

Implementation-wise, the snapshot we used is a memory dump of the VM heap. This heap will contain all the objects, classes and methods we created during the bootstrap. At load time, the memory dump is restored into memory and the VM internals are re-configured to use this heap using the VM setup interface. This idea is the same used by languages such as Smalltalk, Lisp, Javascript in V8 or the JikesRVM [AAB+00].

7.4 Conclusion and Summary

We bootstrapped four different languages that have key differences in their meta-models: the core of the Pharo language is defined by traits, class layouts and first-class slots, Candle is a minimal Smalltalk with implicit metaclasses,

finally Metatalk decomposes reflection from the base level and stores meta information in the meta level of the language. By using Espell these four languages run on top of the same Pharo Virtual Machine.

Then, we showed also that bootstrapping can be applied in a real environment. A fast startup can be achieved by caching the language runtime and a fast development cycle can be obtained by optimizing the bootstrapping interpreter through dynamic compilation.

Part IV

Tailoring: Automatic Extraction of Application Runtimes

Chapter 8 Run-Fail-Grow: Dynamic Tailoring

Contents

Introduction

This chapter describes the *run-fail-grow* (RFG) technique: an alternative solution to dead code elimination that identifies the code units that are actually used in an application during runtime (Section 8.1). Using RFG, we launch a *reference* application runtime containing all code units (base libraries, third party libraries and application code) and a *nurtured* application runtime containing a *seed i.e.*, a minimal set of code units we want to ensure in our application. RFG generates an application runtime by "growing" the nurtured runtime into a deployable specialized version of the reference application. RFG runs the nurtured version of the application runtime and feeds it with the code units that were detected missing through failures (Section 8.2).

The resulting deployable application runtime only embeds the seed and used code. By carefully choosing the seed, the user configures the scope of the tailoring process making possible different levels of tailoring. For example, a seed that includes all base libraries makes the tailoring process only select used code in the application-specific part; whereas an empty seed makes the tailoring process select used code in all parts: base libraries, application libraries and application-specific part (Section 8.4).

The dynamic nature of our solution allows its usage in dynamically-typed languages, and applications using reflection. Our solution does not need to modify the original application because our run-fail-grow works outside of the nurtured runtime, transparently. It also successfully deals with applications that make use of programming language features such as reflection or open classes.

We developed Tornado, an RFG implementation based on Espell (Section 8.5). In Tornado, the nurtured runtime resides inside a virtualized runtime. With Espell, we can monitor the execution of the nurtured runtime to detect whether a code unit is missing or not. Using Espell's mirrors we can also install the required code units.

8.1 Run Fail Grow: Dynamic dead code elimination

We propose a tailoring technique we named run-fail-grow (RFG). Briefly, RFG works by launching a *reference* runtime encompassing the full application with all its code units (base libraries, third party libraries and application code) and a *nurtured* runtime that has only part of its required code units installed. The nurtured runtime is run, and when a failure is detected because it misses a code unit, we install into it the corresponding code unit from the reference runtime. Thus, the nurtured runtime grows progressively as missing code units are found and solved. Once finished, the nurtured runtime is ready to be deployed on the target devices. Figure 8.1 depicts the basics of our run-fail-grow approach.

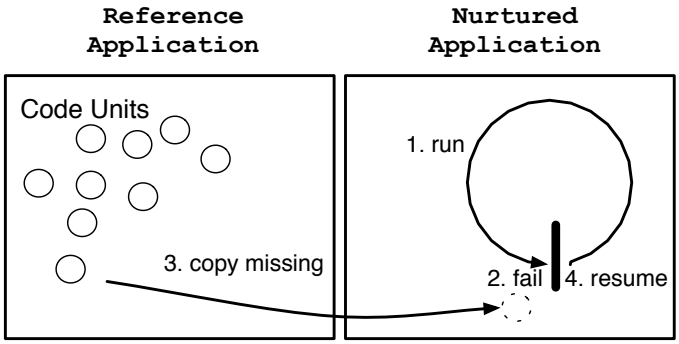

Figure 8.1: **Application tailoring with a run-fail-grow approach.** We (1) run the nurtured runtime and (2) detect the missing units on failure. For each failure, (3) we copy missing code units from the reference runtime and then (4) the execution is resumed (just before the failure point) until the process finishes.

On one hand, we start the reference runtime and pause it at the point where either it contains all its collection of code units, or we can load them dynamically (under a lazy loading strategy). The reference runtime remains paused to avoid to mutate its state during the tailoring process. Pausing consists in suspending all processes and threads from the application runtime.

On the other hand, we fill initially the nurtured runtime with the *seed i.e.*, the set of code units that we want to ensure in the final application runtime.

This seed allows us to specify the level of specialization of our deployable application runtime. By using a seed that contains all base libraries, RFG will only affect the application specific code units and third-party libraries. However, by using an empty seed, it will also tailor base libraries (cf. Section 8.4).

Running the nurtured runtime consists in two main steps. We first install in the nurtured runtime one or more *application entry points* in the form of threads, and then we resume them. The execution of these entry points results in sending messages to objects in the seed. These messages will signal *missing code failures* when the respective classes and methods to resolve the message are not available. We detect the missing code failures and solve them by fetching the needed code units from the reference runtime and install them into the nurtured runtime. RFG installs only code units on demand *i.e.*, the content of installed classes and objects is not installed until it is actually needed; methods are not installed until they are invoked. The process repeats until we end it explicitly. As part of the process we can interact with the application runtime under construction through *e.g.*, its user interface. These interactions may signal more missing code failures, complementing the tailoring process. Ideally, the nurtured runtime reaches a stable point where it needs no more code units. The nurtured runtime is then ready for deployment.

The dynamic nature of RFG addresses all our challenges (cf. Chapter 3). Missing code units are detected and resolved at runtime, where two main elements are available: the exact messages that are sent with their corresponding receiver and arguments, and their concrete types. The methods and classes to install can be easily deduced from the available concrete types, *depending neither on type declarations nor their inference*. RFG *takes into account the configuration of the application* such as the ones present in files, since the code that reads and interprets them is actually executed, without the need of custom code for them. *Reflection is supported for free* since reflective invocations are treated as simple message sends and executed as any other code, and strings composed dynamically by the application are available at run time.

8.2 Run-Fail-Grow through an example

We illustrate in this section the ideas behind RFG with the example introduced originally in Chapter 3 (cf. Figure 8.2). For the sake of clarity, in this example we will tailor the application's code units only (*i.e.*, not the base libraries). This means that in this example the seed includes the base libraries.

Setup of the Environment. First, we launch the reference runtime (cf. Figure 8.3) and an empty nurtured runtime (cf. Figure 8.4 Step a). We fill the

```
1  MainApp>>start
2      logger := StdoutLogger new.
3      logger log: 'Application has started'.
4      "do something"
5      logger log: 'Application has finished'.
6
7  StdoutLogger»newLine
8      stdout newLine.
9
10 StdoutLogger>>log: aMessage
11     stdout nextPutAll: Time now printString.
12     stdout nextPutAll: aMessage.
13     stdout newLine.
14
15 RemoteLogger»log: aMessage
16     | socket |
17     socket := self newSocket.
18     socket nextPutAll: Time now printString.
19     socket nextPutAll: aMessage.
20     socket newLine.
21
22 RemoteLogger»newSocket
23     "...."
24     "creates an instance of socket given some configuration"
```

Figure 8.2: **Code of the example logging application.** Methods in gray are not used by the application.

nurtured runtime with the seed containing the language base libraries. Thus, each application has its own copy of the base libraries, as shown in this case with the Date and Time classes and the stdout object.

Install the application's entry point. We install into the nurtured runtime our application's entry point *i.e.*, a MainApp instance (aMainApp) and a process that will execute the statement "aMainApp start" (cf. Figure 8.4 Step b). Note that although we are referencing an instance of the class MainApp, the MainApp class is not installed yet.

When the execution starts, the mainApp instance receives the start message, and we detect the MainApp class and its start method as a missing code unit failure. We install these two missing code units (cf. Figure 8.4 Step c) and finally the MainApp»start method is activated and starts running.

Activating the start method. The method start defined in Figure 8.2 is executed, as we can see in Figure 8.4 Step c. During the execution of its first statement (Figure 8.2 line 2) we detect a missing code unit failure: The StdoutLogger class does not exist. Thus, before continuing, we install a StdoutLogger

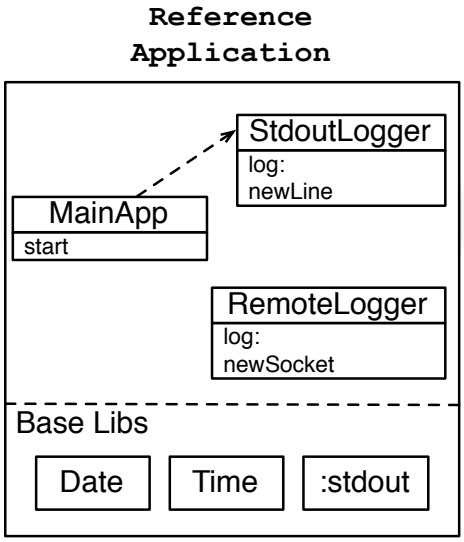

Figure 8.3: **Reference runtime with all code units.**

class with the same shape as its reference counterpart (cf. Figure 8.4 Step d). This class does contains however neither all the methods nor the meta-data (*e.g.*, superclass, package, subclasses) from the reference class since they may not be necessary.

Once we install the StdoutLogger class, we resume the execution. The first statement results in a new StdoutLogger instance. Note that the new method is already installed because it is part of the language base library, already available in the seed. During the second statement's execution (Figure 8.2 line 3), we detect a missing code unit failure on the log: message (cf. Figure 8.4 Step e): the corresponding method is not installed in the StdoutLogger class. We install it and resume the execution. This time the method is found, and the log: method is activated.

Once the log: method finishes, the execution returns to the start method. There, the third statement (Figure 8.2 line 5) is executed with no intervention of our technique, since the log: method is already available. Figure 8.4 Step f shows the final state of the nurtured runtime: it contains only the methods and classes that are actually used by the application. Leaf objects used during the process have been garbage collected.

8.3 Detecting Missing Code Units

RFG depends on getting notified when a missing code unit failure appears. RFG's algorithm is based on traps to achieve this task, as shown in Algorithm 1. Traps are placeholders that are installed in the nurtured runtime in the place of real elements. They are triggered whenever the application tries

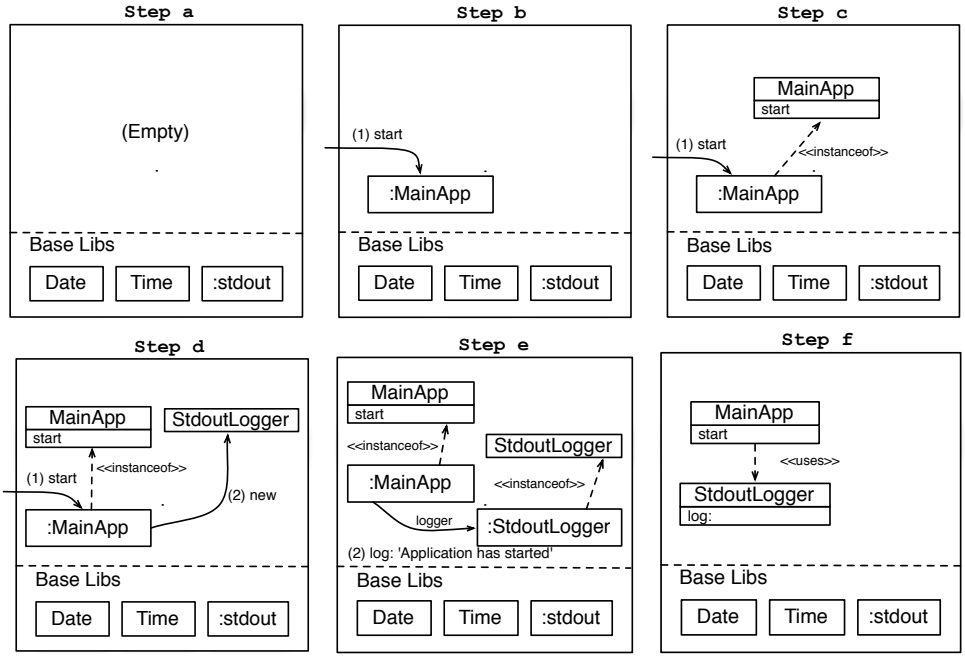

Figure 8.4: **The nurtured runtime at different steps of tailoring.**

to access them. In case a trap is triggered, we suspend the nurtured runtime execution, we install the missing code units replacing their corresponding traps, and finally resume the execution from the moment immediately before the trap was triggered. Traps are installed dynamically in the nurtured runtime following the information flow of the application *e.g.*, when a method A is installed some traps are installed inside it to capture possible missing code unit failures it may cause.

```
Initialize reference runtime;
Initialize nurturing application with the seed;
Install entry point(s);
while not finished do
    run the nurtured runtime;
    if trap was activated then
        install missing code units;
        restart message send;
    end
end
```

Algorithm 1: An abstract view of the run-fail-grow process.

RFG follows the following rules to install code units inside the nurtured runtime:

1. A class is installed when we install a method that belongs to it or one of its superclasses.
2. A method is installed when it needs to be executed, following the rules of the method-lookup in the underlying VM.
3. An object is installed when *any* message is sent to it.
4. An object is installed when it is needed by the VM to execute some operation (*e.g.*, primitives and method literals).

We identify the following as the basic traps that are necessary to tailor an application.

Message Received Trap. A *message received* trap takes the place of a normal object and captures *all* messages sent to it. These traps are activated when a message is sent to the trap. When it is activated, it installs the object that corresponded to the trap (Rule 3). RFG installs the object as a *partial* clone of the original object *i.e.*, we set all its state to traps to capture the access to its class and fields.

Method Nonexistent Trap. A *method nonexistent* trap captures message-sends to already existing object that cannot be resolved because the corresponding method does not yet exist in the nurtured runtime yet. When the application execution triggers one of these traps, RFG installs the corresponding method in the class hierarchy of the object (Rule 2). If the class owner of the method does yet not exist, RFG installs it (Rule 1). When RFG installs a method, its method literals (*e.g.*, strings and numbers that are part of the method definition) are installed along with it because the VM can manipulate them directly during the bytecode interpretation (Rule 4).

Method Invoked Trap. A particular case of method nonexistent traps are *method invoked traps*. A method invoked trap captures the execution of a particular method. Method invoked traps are useful to capture overridden methods. Overridden methods must be captured to avoid the method lookup to answer a superclass implementation instead of the right one (Rule 2), resulting in an unexpected behavior. Figure 8.5 illustrates this problem: the class B from the reference runtime contains an override, while it is not present in the nurtured runtime. If no trap is placed to capture the override, the method doSomething from class A would be executed, thus changing the semantics of our application.

Note that there is no need for a *missing class trap* or similar, as classes are installed when a *method nonexistent trap* is triggered. In the case of languages that expose classes as objects, classes could also be installed by message received traps.

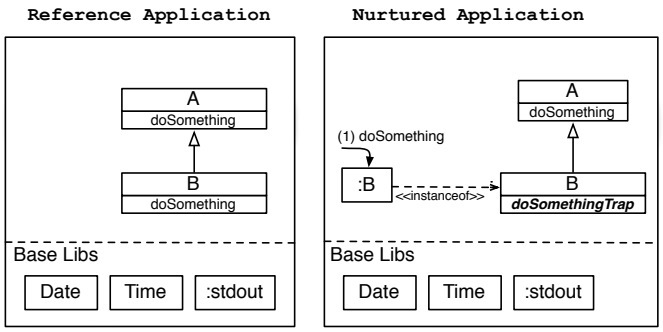

Figure 8.5: **The need for overriding traps.** Method traps should capture the overridden doSomething message-send to avoid the superclass method to be executed wrongly.

8.4 Customizing Dead Code Elimination with Seeds

The level of tailoring of RFG can be specified using seeds. A seed is a collection of code units whose installation is forced into the nurtured runtime. These code units are available for the nurtured runtime and thus, accessing them does not trigger missing code unit failures. A seed can contain any arbitrary code unit, including package, classes, methods and even already initialized objects. Seeds are useful to cover different tailoring scenarios.

Let's take as a first example a smartphone where the base libraries of the language are already available, so they are shared amongst the many applications installed in it. When targeting such a smartphone, base libraries are already present and we do not need to produce a specialized version of them. We need to specialize only third-party libraries and application code. In this case, we use a seed providing the language base libraries.

Let's take as a second example a constrained device robot-like which will contain only our application. When targeting this robot as deployment scenario, we want to specialize all the code to deploy including base libraries. In such a case, the seed is empty to allow the RFG algorithm to work on every code unit.

Figure 8.6 presents two tailoring maps showing example usages of seeds. Each application contains code units corresponding to the base libraries, third-party libraries and application code. To the left the seed covers base and third party libraries, thus RFG applies and selects a subset of the application code units only. To the right, the seed covers only the base libraries, thus RFG applies and selects a subset of the code units from the third party libraries and application code.

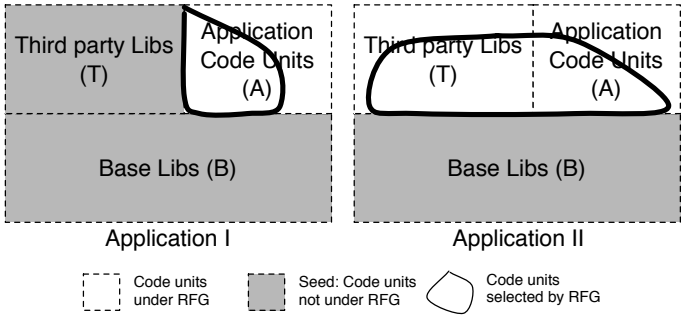

Figure 8.6: **Tailoring Map.** A tailoring map describes which code units of an application are included in the seed (in gray), which ones are subject to the RFG technique (in white) and the number of them that are finally selected (within the thick area).

8.5 Tornado: RFG using Espell

We implemented our RFG technique as a tool called *Tornado*. Tornado is implemented using Espell to tailor applications written in the Pharo programming language. Tornado's architecture combines Espell (cf. Chapter 4) and Ghost proxies [PBF+14] (cf. Section 8.5.1) illustrated in Figure 8.7. The nurtured runtime is hosted inside a virtualized runtime. Tornado's hypervisor runs and monitors the nurtured runtime. It installs traps in the nurtured runtime as Ghost proxies, and uses the object space interface to query and install code units in it. Notice that due to the limitations of our prototype implementation, the reference runtime cohabits with the hypervisor, which may cause problems when trying to tailor the tailoring process itself. We now detail how we fulfilled each of RFG's requirements in our solution:

Execution cycles. We use Espell execution cycles to monitor and control the execution of a nurtured runtime. When a cycle is finished the Tornado hypervisor checks if the virtualized runtime is suspended on a trap. In such case, it installs the corresponding code unit and resumes the execution with another cycle.

Advanced proxies. Pharo's libraries includes Ghost, an advanced proxy implementation. Ghost allows one to capture all kind of message sends, intercept particular method executions, and even to proxy classes and special objects. We use the Ghost model to implement execution traps.

Object space runtime manipulation. An object space provides already with operations to query and install the classes and methods in the virtualized runtime.

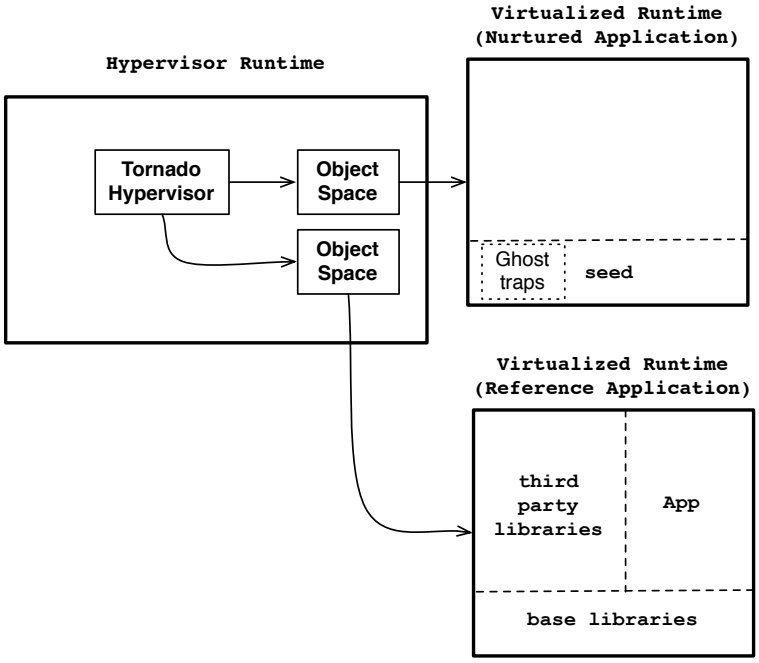

Figure 8.7: **Tornado's architecture overview.** Tornado controls both the reference and nurtured runtimes through Espell. Traps are installed into the nurtured runtime with the Ghost library.

8.5.1 Execution Traps with Ghost Proxies

Implementing execution traps such as the ones described in Section 8.3 requires powerful intercession support to capture *all* message-sends.To achieve this, we implemented a set of proxies following the Ghost model [PBF+14] (similar to JavaScript proxies [VCM10]). Ghost proposes a low-memory footprint, general purpose proxy implementation for the Smalltalk language supporting the creation of proxies for normal objects as well as classes and methods. Using Ghost proxies we are able to detect all situations corresponding to our traps. Ghost proxies represent missing objects, classes and methods. When a proxy is accessed, Tornado replaces it by its corresponding real object. Tornado correctly respects identity by using a table relating each proxy to the code unit or object it represents in the reference runtime. Additionally, each proxy is attached to a *handler* that may perform some action when the proxy receives a message. We use proxy handlers to perform the right action for each trap. We discuss below the different kinds of proxies and handlers we use and how they support RFG.

Message Received Trap. We implemented the message received trap as a normal object proxy. An object proxy takes the place of an object and receives all messages that were originally meant for it. This trap is trig-

gered when the proxy receives a message. Its handler replaces the proxy by a *partial* copy of the original object from the reference runtime. The copy is created and all references to the proxy are replaced by references to this new object (achieved through the become: facility of the Pharo language that dynamically swaps object references). Each field of this new installed object is set to message received traps.

Class Message Received Trap. In Pharo classes are reified as objects, meaning that they can receive messages in addition to participating in the VM's method lookup. To handle this case Tornado uses special message received traps to capture messages sent to classes. These *class message received* traps are implemented with proxies in the same way as normal message received traps. A main difference relies, however, in that the proxies that replace classes have the same internal structure as the normal classes from the Pharo VM (we guarantee they have the three instance variables for their superclass, method dictionary and format). By doing that, the VM can take the proxy/trap as a class and continue its execution without crashing.

Method Nonexistent Trap. We implemented the method nonexistent trap in Tornado as a class proxy located at the top of the class hierarchy. Whenever a message is sent to an object, the VM looks up the method in the object's class hierarchy. This trap is triggered when a message arrives to it, meaning that there was no method for it in the hierarchy of the receiver. When triggered, the handler installs the classes part of the hierarchy of this method and the missing method in its corresponding class. The literal objects that belong to the method are installed with it because they are objects that could be used directly by the VM during bytecode interpretation.

Particularly, we can find the case where no method is found in the reference runtime due to either a bug in the application or a design choice to use the dynamic features of Pharo. In those cases Tornado sends the doesNotUnderstand: message to the receiver object (an equivalent to *e.g.*, Ruby's method_missing and Python's __getattr__) which in turn could be handled by the application or throw a corresponding exception.

Method Invoked Trap. We implemented method invoked traps in Tornado using method proxies. Method proxies are placed in the method dictionaries of classes containing overridden methods, taking the place of the original method. When Tornado installs a class into the nurtured runtime that contains overridden methods in the reference runtime, it installs into this class a method proxy for each of its overridden meth-

ods. This trap is triggered whenever the method proxy it is about to be executed. The handler of this trap compiles a new method with the same source as the corresponding method from the reference runtime and installs it inside the nurtured runtime.

Primitive invoked trap. Primitive invoked traps are implementation specific related to the Pharo language. Pharo's primitive operations such as number arithmetic are implemented through primitive methods. Primitive methods are implemented in the Virtual Machine and do often access directly the fields of its receiver and arguments by forging references and manipulating directly the memory, bypassing our traps. Thus, we face an issue when a *message received trap* proxy is the argument of such a method: the VM can modify this proxy without activating the trap. To honor Tornado's Rule 4 (an object is installed if needed by the VM), we decorate primitive methods with *primitive invoked traps*. We implemented primitive invoked traps with method proxies, as a special case of method invoked traps. When a primitive invoked trap is triggered, its handler triggers each of the missing object traps received as arguments, if any. In this way, Tornado forces the installation of the arguments and the primitive is executed with actual objects instead of proxies, as expected.

8.5.2 Object Installation and Propagation Rules

As we explained before, Tornado installs all objects inside the nurtured runtime on demand, as *partial copies*, *i.e.*, the objects referenced by the original object will be replaced by default by traps. When Tornado installs an object inside the nurtured runtime, this new object has the same format and size as its original counterpart. *Propagation rules* determine how each of the object's fields are propagated on installation. Tornado provides the following propagation rules to customize installation:

Message Received Trap. This is the *default propagation rule* and end user applications can usually be tailored with just them. This propagation rule installs a message received trap in each field of the object that is being installed.

Materialization. This propagation rule forces the installation of the object referenced by the field. This is used for those cases where some structure should be guaranteed to the Virtual Machine *e.g.*, the objects referenced by the first three fields of a class (superclass, format and method dictionary) cannot be proxified because they are used by the VM for the method lookup. The same happens with other objects reifying low level concepts such as activation records or semaphores.

Swapping. This propagation rule forces the reference of the object installed to be swapped to another object's reference. The usual use case of this rule is replacing some object reference by nil, and so, to force lazy initializations.

8.5.3 Object Identity and Proxies

Tornado takes care of the identity of objects with an identity table. The identity table is important because Tornado works at the object granularity. Due to the inherent graph nature of object-oriented programs, an object being installed may reference another object that is already installed inside the nurtured application. Identity is an important concern in the presence of proxies. Tornado guarantees that identity checks inside a program (*e.g.*, comparison through ==) always preserve object identity by following the following invariant:

An object and its proxy do not exist concurrently in the nurtured application.

That is, the nurtured application contains either an object or the proxy trap that represents its absence, but not both at the same point in time. When the proxy is replaced by the actual object's copy, all references to the proxy are swapped to references to the new object. The proxy is no longer referenced and thus, garbage collected. This invariant guarantees that identity checks in a program that should be true are indeed true. Either the compared references point both to the same proxy, or both to the same copy.

8.5.4 Implementing Seeds in Tornado

Tornado's seeds specify the level of tailoring. The seeds are in charge of initializing the nurtured application's virtualized runtime with the elements we want to ensure on it. Our current prototype supports two ways of describing and building seeds:

Loading an already existing memory snapshot. The nurtured application's virtualized runtime is initialized by loading an already existing snapshot or image (*i.e.*, this is an image in the same sense as Smalltalk or Lisp). This technique consists in using a memory dump from an object heap containing all the classes and objects desired in the seed. This memory snapshot should follow Pharo's object format.

Creating all seed code units from scratch. The nurtured application's virtualized runtime is initialized with objects built from scratch. This technique uses a bootstrapping process as described in Chapter 6.

8.5.5 Preparing the Application for Deployment

Once Tornado is stopped, the nurtured runtime contains all the code units needed to run. Tornado proceeds to prepare the application for deployment *i.e.*, it removes all trap leftovers and extracts the nurtured runtime. Tornado identifies the traps by the presence of proxies and replaces the references to those proxies by references to another object, defaulting to the nil object. Proxy objects do not then represent a drawback in space consumption because they are garbage collected.

Once the traps are removed, the nurtured runtime keeps no dependencies to Tornado nor its infrastructure. Thus, the application can run outside the Espell infrastructure with no performance penalties. Finally, Tornado extracts the application code units using one of two different techniques: (a) the creation of a snapshot file containing all code units and already initialized objects; or (b) build a static description of the application containing the code for all classes and methods that should be part of it.

8.6 Conclusion and Summary

In this chapter we presented a run-fail-grow (RFG) approach for application tailoring. RFG tailors an application by starting it and initializing it with a seed that contains the minimal set of code units we want to ensure. Then, we install and execute the application's entry points. As the application executes, missing code units are found and installed on demand, ensuring that only the needed code units are introduced. By following the runtime execution, it supports dynamic features such as reflection and meta-programming.

We implemented RFG in a tool called Tornado based on Espell. With Espell, we are able to monitor the nurtured runtime's execution using the execution cycles. Ghost-like proxies represent traps that detect missing code units during execution. We use Mirrors to install required code and restart the execution as it is needed.

Chapter 9 Run-Fail-Grow Validation

Contents

Introduction

We evaluate Tornado by conducting five experiments that tailor different Pharo applications, with increasing requirements (Section 9.1). We chose our experiments in a way to target the objectives to (a) understand how minimal are the applications resulting from the tailoring process, (b) explore how successfully we address the challenges we stated in Chapter 3 and (c) exercise those cases that push to the limit the interaction between the language and the VM. Our experiment methodology consists in the following steps:

1. **Setting up a seed for the application.** Most of our experiments use what we called an *empty seed*. This seed is, however, not completely empty. The empty seed we used contains some minimal infrastructural objects that are needed for language-VM interaction, and is therefore 10KB large. Our last experiment, the largest one, uses both this empty seed and an additional seed containing the base libraries.
2. **Preparing the application entry points.** This step consists in the installation of the one or more processes that will run our application.
3. **Run the application.** The application is run by executing its threads. In particular in our last experiment (an interactive web application), we interact with our application through a web browser.
4. **Stop and extract the application.** Once the tailoring process finishes, we stop Tornado and extract the resulting application by making a snapshot of it in a Pharo image file. We test the generated snapshots

to verify they work properly, either by using the application or debugging them when they involve no I/O. We evaluate the behavior of the tailored application under the assumption that only the features we used during the tailoring should work.

5. **Perform measurements.** Finally, we measure the size of the generated snapshots files and compare them against the Pharo distribution prepared for production (cf. Section 9.2).

Additionally, we present an evaluation of Tornado according to the evaluation criteria stated in Chapter 3. Our evaluation includes a comparison with the already presented related work on application tailoring (Section 9.4). Finally, we discuss some aspects and trade-offs of the run-fail-grow approach and our implementation (Section 9.5).

9.1 Experiments

This section enumerates the different experiments we conducted to evaluate Tornado RFG. Each of these experiments details the purpose of the experiment and the particular features that are evaluated.

Experiment I: Adding Two Numbers

The smallest (in terms of size) interesting program to tailor is adding two numbers, without the involvement of any I/O *i.e.*, an application that just executes the "2 + 3" statement as entry point. Tailoring this program is challenging because it stresses the infrastructure by installing only the minimal elements an application needs to run. It makes evident how small a tailored application can be. Additionally, it is interesting since it makes use of the following features of the Pharo language and infrastructure:

Immediate objects. Immediate objects are objects encoded in the object reference instead of being allocated in the heap. Immediate objects do not contain a reference to their class in the object header, as there is no object header. Instead, the object reference where the object is encoded contains a bit tag that the VM uses to identify the immediate object. This means that the Pharo VM must have references to the immediate object classes (or their proxies) in order to send messages to these immediate objects. In this experiment we use immediate small integers, instances of SmallInteger.

Special selectors. The method selector + is a special selector for the Pharo VM. Special selectors are optimized as they are broadly used messages,

for example for arithmetics. First, they are implemented as special bytecodes to avoid method lookup. If the special bytecode cannot be executed because some VM assertions are not valid (*e.g.*, class and object format assumptions), the VM performs the default method lookup. In this experiment the VM should take care of small integer arithmetic *i.e.*, it should fulfill all VM assumptions and not perform a method lookup; Tornado should install no extra methods nor classes.

Experiment II: Factorial of a small number

The following a bit more complex experiment is the factorial of a small number. Again, it is free from any I/O. The application simply executes the "10 factorial" statement as its entry point. Factorial uses arithmetics as the latter experiment (sums and multiplications), while it also adds the following interesting cases:

Method lookup. The factorial message is sent to a small integer but not optimized as it is not a special selector. Thus, the VM looks up the corresponding method up in its class hierarchy. The method factorial is defined in a superclass (Integer).

Recursion. The factorial implementation in Pharo base libraries is recursive. Additionally, this recursion activates the factorial method many times, creating many activation records in the VM. Activation records are reified lazily whenever it is accessed reflectively, or when the stack depth is deeper than the maximum supported.

Experiment III: Factorial of a large number

Following, we experimented with an application whose entry point was the "100 factorial" statement. This application does not either make use of any I/O. The factorial of a large integer creates eventually integers that exceed 32 bits, and thus, do not fit as immediate small integers. This experiment adds the following interesting cases:

Large integers. Large integers in Pharo are represented, in contrast to immediate small integers, as standard objects allocated in the heap with their own object header and arbitrary length. Large integers are created automatically by the VM when the result of some integer calculation produces a number that overflows 31 bits. That is, the LargeInteger class (or its proxy) should be available to the VM in order to instantiate the correct object. Additionally, large integers implement their arithmetic methods by calling primitives from external plugins (the large integers plugin).

Polymorphism. The introduction of large integers introduces also *polymorphism* between them and the immediate small integers. They share the same class hierarchy (Integer is the superclass of SmallInteger and LargePositiveInteger), since the method factorial is implemented in the superclass and each of the subclasses has its own implementation of the arithmetic methods for adding and multiplying.

Experiment IV: Reflective invocations

The fourth experiment introduces reflective invocations. Figure 9.1 introduces the code we used for this experiment. The User class in our example has two fields (name and age), and four methods. Two of these methods (age and name) return directly the field with the same name, the method hasWritePermissions is annotated with the property annotation (a pragma in Pharo's terminology) and the method isMinor is a normal method. We introduce also PropertyExtractor class with the responsibility of returning the name of those methods that are properties of an object *i.e.*, all methods that only return a field, and all those methods annotated with the property annotation. The statement we introduced as the entry point for this experiment is "PropertyExtractor new extractPropertiesFrom: User new".

```
1  Object subclass: #User
2    instanceVariableNames: 'name age'.
3
4  User>>age
5    ^ age
6
7  User>>hasWritePermissions
8    <property>
9    ^ true
10
11 User>>name
12   ^ name
13
14 PropertyExtractor>>extractPropertiesFrom: anObject
15   ^ anObject class methods
16     select: [ :each | each isReturnField
17       or: [ each pragmas anySatisfy: [ :pragma | pragma keyword = #property ] ] ]
18     thenCollect: [ :each | each selector ]
```

Figure 9.1: **Code of the reflective invocations experiment.** The PropertyExtractor class does the reflective invocations, the User is the class we will be reflecting on.

This experiment evaluates how we handle a reflective application with Tornado's RFG. The PropertyExtractor queries the methods from the User class, that are included as part of the tailored application (since they receive the messages isReturnField and pragmas). These reflective invocations include:

(a) access to an object's class, (b) access class methods and (c) query those methods to know if they correspond to the criteria of the PropertyExtractor.

Experiment V: Adding I/O

A fifth experiment introduces I/O to each of the previous experiments, adding a statement printing to the standard output the obtained results. Figure 9.2 shows the code from our entry point in the case of summing up two numbers. The entry points for the other experiments have the same structure, differing only on the expression that is printed (the "1+2" expression in this case). Notice that the FileStream class needs to be initialized before the proper printing into the stdout stream because the code needed for class initialization is not installed by default in the empty seed.

```
1 FileStream startUp: true.
2 FileStream stdout
3   nextPutAll: (1 + 2) asString;
4   crlf.
```

Figure 9.2: **Entry point of the experiment that sums two numbers and prints the result in the standard output stream.**

In this experiment, besides testing the proper usage of I/O streams such as the standard output stream, we evaluate the ability of Tornado to handle **platform identification**. The stdout stream initialization for Pharo is done by the File package written in Pharo, and it depends on which is the current operating system. This experiment shows that Tornado prepares tailored versions of applications to run on a single operating system or platform.

Experiment VI: Seaside Web Application

Our last experiment consists in tailoring a web application using the Seaside application framework [DLR07]. Seaside is a web application framework featuring continuations thanks to stack reification. We configured it with its default values, without making any customizations. The web application under tailoring has a single webpage that allows one to send requests to the web server to increment or decrement a counter. This experience shows that Tornado works in presence of Pharo's **multi-threading**. The Seaside application framework makes use of Pharo processes. One process listens for incoming connections and opens new processes to handle requests. Seaside uses semaphores to synchronize processes and wait for incoming data from sockets.

For this case, we proceeded to do two different experiments, with two different seeds. We first used the empty seed (*Seaside Web Application A*), as

in the previous experiments, and then used a seed containing all Pharo base libraries (*Seaside Web Application B*). Appendix D details the entry points and list of extracted code units for each of these cases.

9.2 Results

We gathered our experiments' results into Table 9.1. Additionally, Appendix D presents in detail the entry points and list of selected code units in the cases of usage of I/O and the Seaside Web App. This table shows:

Experiment. The name of the experiment under evaluation, followed by our measurements.

Reference Runtime. The size of the reference runtime containing all of its code units, in KB. We present here the size of the official Pharo distribution prepared for production for each application. This distribution voids some caches and removes some well known objects that are only required at development time.

Seed. The size in KB of the chosen seed for the experiment.

Nurtured Runtime. The final size of the nurtured runtime once Tornado finishes its process and the application is extracted.

Saved. The percentage of space saved in comparison with the reference application. We calculated this percentage using the following equation:

$$saved = \frac{100 * (reference - nurtured)}{reference - seed}$$

Note that we subtract the size of the seed from both the nurtured and reference runtimes sizes, since the seed is shared between both. That way, we compare only those parts of the application that were subject of the RFG algorithm.

Table 9.2 shows the size in KB of the code units we used in our experiments. This table details the size of the Pharo base libraries, third party libraries such as Seaside and our particular experiments, which aid in the understanding of the results. We obtained these sizes by measuring the size of the code units once loaded in memory.

Our experiments show that Tornado aggressively reduces the size of code units required for an application. Our examples save from 95% to 99% of space, compared with their reference runtimes (which contains all base libraries and third party libraries in case of Seaside). Our first three experiments (the sum of two numbers, and the factorial of 10 and 100) show that Tornado succeeds to create minimal deployment versions of our applications,

Experiment	**Reference App**	**Seed Size**	**Nurtured App**	**Installed Code**	**Saved(%)**
Sum Two Numbers (I)	12873	10	11	1	**99.99%**
Fact 10 (II)	12873	10	15	5	**99.96%**
Fact 100 (III)	12873	10	18	8	**99.94%**
Reflective App (IV)	12873	10	32	22	**99.83%**
(I) + I/O	12873	10	81	71	**99.45%**
(II) + I/O	12873	10	82	72	**99.44%**
(III) + I/O	12873	10	89	79	**99.39%**
(IV) + I/O	12873	10	95	85	**99.34%**
Seaside Web App A	17250	10	573	563	**96.73%**
Seaside Web App B	17250	12872	13090	218	**95.02%**

Table 9.1: **Results of the tailored experiments.** Sizes are displayed in KB. The percentage of saved space does not take into account the seed, as it is not subject of Tornado and it is shared by both the reference and nurtured runtimes.

taking into account that our seed forces a minimum of 10KB in each of them. The reflective application is indeed also minimal, but bigger than the other three, as Tornado installs inside the nurtured runtime (a) all the code that is accessed by reflection and (b) code from the collections package to iterate over the methods of a class.

We detect a notable growth in size when adding I/O to our experiments, which varies from 63KB to 71KB extra. According to the list of installed code units, we identity a problem in the design of the I/O streams library from Pharo: a set of character tables meant for character encoding and conversion are initialized, even if not all of them are later on used by the application. This problem shows that this part of Pharo base libraries should be redesigned.

The Seaside experiments show that Tornado can be used in a complex setting such as a web application that runs a web server, while still achieving good results. It is interesting to note, from the comparison of both experiments, that more of half of the size of the final nurtured runtime in *Seaside*

Component	Size (KB)
Pharo Base Libraries	12872
Seaside Application Framework Libraries	4378
Seaside Web App	47
Reflective Invocations App	104

Table 9.2: **Component sizes in our experiments.** Size presented in KB.

Web Application A comes from the base libraries. When we introduce the base libraries in the seed, the amount of installed code is reduced to less than the half.

9.3 Comparison with a Dedicated Platform

To have a broader view of our results, we compare them also with two dedicated platforms that exist in Pharo's infrastructure.

Candle. Candle (Chapter 7) is a dedicated platform that runs on the Pharo platform based on MicroSqueak [Mal]. It is a specialized platform containing an alternative implementation of base libraries, as Java Micro Edition (J2ME) [Jav] is for Java. MicroSqueak and thus, Candle, were designed with the explicit goal to be the smallest practical Squeak kernel. It contains a total of 49 classes with a reduced set of methods. It offers a minimal core of the language, a basic collection library and basic file IO support. Candle presents a minimal memory footprint of 80KB, when we build an application that performs no computation. Candle presents crucial differences with Pharo base libraries: it does not provide the same libraries (*e.g.*, it does not contain socket support) and it does not ensure the same API of those libraries that it contains. Thus, applications such as the one in our Seaside experiment cannot run on top of MicroSqueak without a dedicated version of the Seaside framework.

PharoKernel. The PharoKernel is a shrunk version of Pharo official distribution. PharoKernel shrinking is an ad-hoc process *i.e.*, there is a script that hardcodes the knowledge of which objects should be removed from the distribution for its deployment. On one hand, PharoKernel conserves compatibility with Pharo: we can still load Pharo code on it and most of the APIs are not broken. On the other hand, PharoKernel does not achieve minimality as PharoCandle does. We can, for example, install the Seaside application framework on PharoKernel.

Table 9.3 shows a comparison between our results and dedicated platforms. On one hand, we can see that Tornado ensures smaller memory footprints than the dedicated runtimes when working on small applications. We can see a degradation on the used memory when starting using I/O. This problem comes from the fact that a simple I/O operation in regular Pharo requires a lot of objects and methods. Candle on the other hand uses a dedicated implementation of I/O.

Regarding the seaside web application, we can observe that while Seaside can be installed inside PharoKernel, its memory footprint grows even bigger than the one in Pharo's official distribution prepared for production. On the other hand, we do not compare the our solution applied to Seaside with Candle, as we cannot install Seaside under Candle. This last problem appears in most of the dedicated platforms. For example, you cannot easily execute a J2SE program on top of J2ME.

Experiment	**PharoKernel**	**Candle**	**Tornado Vs PharoKernel**	**Tornado Vs Candle**
Sum Two Numbers (I)	3799	80	**99.97%**	**86.25%**
Fact 10 (II)	3799	80	**99.87%**	**77.5%**
Fact 100 (III)	3799	80	**99.79%**	**60%**
Reflective App (IV)	3799	80	**99.42%**	**72.5%**
(I) + I/O	3799	80	**98.13%**	**-1.25%**
(II) + I/O	3799	80	**98.10%**	**-2.5%**
(III) + I/O	3799	80	**97.92%**	**-11.25%**
(IV) + I/O	3799	80	**97.76%**	**-18.75%**
Seaside Web App A	20254	n/a	**97.17%**	**n/a**
Seaside Web App B	20254	n/a	**35.37%**	**n/a**

Table 9.3: **Comparison with two dedicated platforms.** Sizes are displayed in KB. The percentage of saved space compares the full deployable runtime.

9.4 Evaluation of Tornado

In this section we evaluate Tornado according to the criteria we presented in Chapter 3. Table 9.4 shows an overview of the criteria presented in Chapter 3 and their possible values to evaluate tailoring solutions. We include Tornado in the last column of this table.

	Dedicated platforms	**Static Analysis**	**Hybrid Analysis**	**Dynamic Analysis**	**Tornado**
Base Libraries	+	+	+	+	**+**
Third-Party Libraries	-	+	+	+	**+**
Legacy Code	-	+	+	~	**+**
Reflection Support	+	-	-	+	**+**
General Purpose Infrastructure	+	-	-	~	**+**
Configurability	-	-	-	~	**+**
Dynamic typing	+	-	-	+	**+**
Minimality	-	+	+	~	**+**
Completeness	+	-	-	~	**~**
(+) supported /achieved		**(~) partially supported**		**(−) not supported / not achieved**	

Table 9.4: Evaluation criteria applied to related work on deployment code unit tailoring techniques

Tornado's model and implementation present themselves as a complete solution in the area of application tailoring. Regarding the area of application, we can see that Tornado tailors code units written by the application's developer as well as those from the base language and third-party libraries. This approach, based on runtime execution, offers also three main advantages: (a) it does not require modifications in the nurtured runtime's code allowing its usage on legacy code and libraries in a transparent way, (b) it is applicable in dynamically typed languages as it does not rely on the type annotations in the source code and (c) it supports reflection naturally since the code exercised during the tailoring is the same that will be executed once deployed.

Tornado requires a dedicated infrastructure only during the tailoring: tools to monitor and manipulate the tailoring application. However, once the tailoring is finished and the application reaches a stable point, Tornado extracts and prepares the application to run in the deployment-ready unmod-

ified infrastructure. Tornado offers also a flexible solution in the sense that it allows one to configure the level of tailoring by means of a seed. The seed contains a pre-selection of code units available in the tailoring application before the tailoring starts. In such a way, we can use the seed to specify whether, for example, the base or third-party libraries should be tailored or not.

Tornado also handles modern programming language features such as reflection, open classes and class extensions [BDW03] (*i.e.*, a package can define methods to classes from other packages) and traits [SDNB03], out of the box. Tornado installs methods from other packages or behavior units such as traits seamlessly because during runtime it knows the exact concrete type of each object involved in the execution. Thus, no extra static or string analysis is needed. This is possible thanks to Ghost proxies [PBF+14], which can capture all message sends and specific method invocations.

Finally, we can see that Tornado achieves minimality by selecting only those code units that were used during the tailoring process. No extra code units are installed, besides the objects needed by the VM to run. Regarding completeness, we see Tornado as a partially complete technique, such as the dynamic ones. Every piece of code that was *exercised* during the tailoring process will be available in the deployable version of the application runtime.

9.5 Discussions on the run-fail-grow approach

9.5.1 Ensuring Completeness

Dead code elimination techniques never ensure completeness by themselves. Static approaches cannot efficiently predict the need of those elements used by reflection, or configured in external files/resources. Dynamic approaches depend on the code coverage of the application during runtime, *i.e.*, if the parts of the application that are not used will be not available afterwards. Hybrid approaches share both weaknesses. Orthogonal to the dead code elimination techniques, these two complementary mechanisms are used by existing solutions to guarantee *completeness* and avoid runtime errors due to missing code.

Lazy Loading. JUCE [PRT+04, TP01] and SlimVM [KWW+09, WGF11], as well as Tornado, load missing code from remote servers on demand, Marea [Mar12] implements application-level virtual memory with lazy loading of unloaded unused objects. These solutions differ in their lazy loading approaches by the granularity they use. JUCE loads code with a method granularity to control memory consumption. SlimVM uses as its main loading granularity, a *basic block* granularity, but they can work at the class and method level also. Marea uses an object-cluster granu-

larity. It loads object graphs containing not only classes but also individual objects, which were unloaded to reduce the application's memory footprint.

Remote Invocations. OLIE [GNM+03] uses remote invocations to invoke methods from those objects that were offloaded and migrated to other devices. This approach may introduce several latency problems due to network communications. OLIE tries to minimize it by offloading those elements that degrade less the performance of the system. For that, it takes at runtime object and bandwidth usage statistics.

For dynamic and hybrid techniques, application coverage must ensure that every code unit that is interesting to be deployed is covered. This should include testing special and boundary cases as well as the straightforward cases. We can enforce the coverage and installation of code with several testing techniques.

Manual Testing. Manual testing provides a simple but inefficient way to cover an application's code. Its main benefit is that the code units selection is based on user interactions. Its main drawback is the possibility of human omission during the testing, which impacts directly the detection of used code.

Automated Testing. Automated testing counters the human omissions by adding repeatability in the generation of the deployment unit. Different levels of testing have different impacts on the coverage and will produce different results. For example, using unit tests to cover the application and libraries' code may exercise more code than the one that is actually needed, since they use to test smaller units and tend to cover the whole code. Acceptance tests may not exercise enough parts of the application. UI tests should be considered as part of the solution for maximizing coverage.

9.5.2 Application Designs that get along with Tornado

As shown in Section 5.2, the design of the tailored application directly impacts on the results obtained by Tornado. A series of issues appear regarding global state (*e.g.*, class variables and global variables). A first issue is related to the initialization of such a global state [Ung95]. Since Tornado follows the application's execution flow, eager initializations force Tornado to install objects and methods that may not be used later by the application. In contrast, lazy initializations will only be triggered on usage. Thus, better results could be obtained if a lazy initialization strategy is adopted for the global state.

A second issue appears with residual side-effects. Our tailoring technique builds the deployment application by running it. Thus, those executed global side-effects may reside in the tailored application. For example, a web application framework may hold a cache of HTTP sessions in a class variable. When the tailoring process finishes, the application will keep this cache if we do not handle the case. Solving this problem in Tornado may require either minimizing global state in an application, or either installing a new entry point to reinitialize such global state when the tailoring is finished *e.g.*, clean caches and session dependent state such as file and socket descriptors.

9.6 Conclusion and Summary

In this chapter we presented a validation on our run-fail-grow (RFG) approach for application tailoring. We conducted our validation by first showing in several experiments how our solution succeeds in tailoring applications of different sizes and styles. Tailoring an application just adding two numbers demonstrates that the smallest application runtime we can produce with Tornado occupies only 10KB. Other experiments included I/O, reflection or even the tailoring of a multi-threaded web application using the Seaside framework. They show that Tornado can aggressively shrink applications.

We also evaluate Tornado under the evaluation criteria we stated in the state of the art of this dissertation. The flexibility of our solution through seeds, its ability to handle reflection and its non-dependance on a particular infrastructure for deployment are characteristics that highlight our solution in comparison with the others.

Finally, we present a discussion on several important aspects of RFG and tailoring solutions in general: how to ensure completeness with a tailoring solution and what are the particular application design decisions that help in reducing even more the size of the application runtimes produced by RFG.

Part V

Conclusion

Conclusion

Contents

Introduction

This thesis focuses on the ease of manipulation of application runtimes for Object-Oriented languages, particularly reflective ones. Having access and the power to change such application runtimes is indeed an issue for both language and application developers.

Application runtimes are at the center of the activity of language developers. Implementing a language requires designing its execution model and its runtime representation besides its syntax and exposed concepts. Moreover, extending an existing language to add new features requires its runtime to be easily extensible. Also, reflective languages add the challenge of circularities. To perform these tasks, language developers need tools that allow them to modify a language runtime, extend it and address its circularities.

Application developers are also exposed to the modification and specialization of application runtimes. Particularly, the spreading of new constrained devices such as embedded systems or sensor networks, require application developers to reduce the memory occupied by their applications. To address these concerns, application developers should have access and control on the application runtimes they develop. They should be able to shrink them.

To address these issues we introduced Espell: an infrastructure for application runtime virtualization. We show that runtime virtualization is a general purpose tool that can be used for different purposes. In particular we use it to explore the two challenges presented above. Our first scenario is language bootstrapping. The second is the tailoring of application runtimes.

10.1 Contributions

This section lists the main contributions made during the period of this thesis: Espell application runtime infrastructure and its usages, bootstrapping and tailoring. Additionally, we published the results of several facets of our work. We list these publications in Appendix A.

10.1.1 Espell

The main contribution of this thesis is Espell, a *language virtualization infrastructure* [PDFB13]. In Espell, a first-class application runtime, namely an object space, allows the manipulation, control and monitoring of a virtualized runtime through a clear API. A first-class language hypervisor implements such runtime manipulations with the expression power and abstractions of the high-level language we are manipulating.

Espell allows application runtime manipulation and control through several techniques. Mirrors allow the direct manipulation of objects inside a virtualized runtime while enforcing the invariants of the VM execution model. The hypervisor can execute a virtualized runtime in cycles and perform custom operations between each of those cycles. This allows the hypervisor to control the execution inside a virtualized runtime. Finally, to execute arbitrary code inside a virtualized runtime Espell provides with process injection and virtual execution. The first one injects code inside a virtualized runtime so it can run normally. The latter is based on code interpretation and while slower provides finer control on what code is executed.

10.1.2 Bootstrapping

Bootstrapping is commonly known by its usage on compiler building, where a compiler can compile itself. It can be generalized to the introduction of any software system to its own building process. A bootstrap process has the property of describing the system under construction in terms of the system itself. This allows us to easily change and extend this system, taking advantage of its abstractions and tools.

We applied the idea of a bootstrap in object-oriented languages by providing a circular language definition [PDF+14] *i.e.*, a definition of the language runtime defined in itself. Our virtualization infrastructure eases the execution of such a language definition. First, an object space provides a clear VM-language interface to help with the manipulation of the language runtime under creation. Second, a bootstrapping virtual interpreter allows the execution of the language definition when the language cannot yet be executed.

We used this infrastructure to bootstrap three different object-oriented languages with different programming models:

Candle. A minimal Smalltalk with implicit metaclasses.

Pharo. The core of the Pharo language which is defined by traits, first-class layouts and first-class variables.

MetaTalk. A Smalltalk based language that decomposes reflection into mirrors. The language meta information is hosted inside the meta level of

the language, which can be dynamically removed.

10.1.3 RFG Tailoring

Application tailoring is a technique that reduces the memory footprint of an application by removing code bloat. Code bloat is an issue in constrained scenarios, when the size of an application limits its deployment. For example, devices with limited memory or web-applications in slow networks. Application tailoring reduces unused code units (*e.g.*, classes, methods) to produce a specialized version of an application for its deployment in such scenarios.

Using Espell, we developed run-fail-grow (RFG), an approach for dynamic application tailoring. RFG tailors an application by starting it inside an initially empty virtualized runtime. We can additionally ensure a set of code units inside our application by introducing them inside the seed. Then, a set of application entry points describing where the application starts are installed inside the virtualized runtime and executed. As the application executes, the application will *fail* due to missing code units. RFG reacts to missing code failures by installing the required code units. Then, code units are only installed on demand inside the virtualized runtime. Using this technique, we ensure that only the needed code units are introduced.

By performing during execution, RFG tailors programs written in dynamically typed languages and using features such as reflection and polymorphism. It works transparently, being able to tailor legacy and third-party code without modifying it. Tornado, our RFG implementation, succeeds to produce applications with minimal footprint for deployment. Our results show that we can aggressively tailor challenging cases. In our experiments we observe memory reductions from 95.02% to 99.99% when comparing with the production ready Pharo distribution [PDBF11].

10.2 Future Work

On the engineering side, Espell presents some future work to do such as improving its performance, the integration of a JIT compiler, the memory separation for a more efficient GC or the correct management of external libraries state. On the research side, Espell presents a language runtime virtualization model that opens several directions for future work that we consider for exploration.

10.2.1 Security

Application runtime virtualization opens the door to flexibly forbid or constrain operations to a virtualized runtime. An application runtime could

be transparently sandboxed inside a virtualized application believing that it *owns* the entire machine for itself. Additionally, a first-class hypervisor could allow us to dynamically change the security policy from the high-level hypervisor instead of being a fixed policy in the VM. For example, we could transparently limit a virtualized runtime by the number of open sockets it can open, or the directories inside a machine that it can access.

10.2.2 Resource Control

By enhancing the control over execution of a virtualized runtime, we could use virtualization to account and restrict the consumption of critical resources such as CPU or memory. A hypervisor could dynamically modify the resource control policy to execute applications in specific scenarios. This feature could be used, for example, to develop simulators for constrained devices, or testing applications in extreme situations.

10.2.3 Application distribution and migration.

Application runtime virtualization opens the door to managing application migration in a novel fashion. Virtualized application runtimes could be transparently migrated between different machines or processes without being stopped. Moreover, an application may not need to be developed in a specific way as the migration process would reside in a hypervisor, external to it.

10.2.4 Dynamic Adaptation.

Language virtualization can be also useful in the context of dynamic adaptation of applications with almost zero downtime. Indeed, the same mechanisms we used for bootstrapping and tailoring can be used to update a running application. Mirrors provide a mechanism to directly modify an application runtime objects and classes. The execution in cycles allows safe points of suspension inside an application to make atomic changes.

10.2.5 VM-Language Co-Evolution

One of the main limitations of our approach is that it does not address the modification of the VM. Indeed, object spaces expose the border-line between the VM and the language to change the latter. We do not change the VM's execution model. For example, we would like to flexibly change the VM's object format and see this change automatically reflected in Espell's mirrors. A next step of this virtualization infrastructure is its evolution and co-evolution with the VM.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. 14, 22, 106

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph.D. thesis, Stanford University, December 1996. 40

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 14, 35, 100

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best Paper Award. 135

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. 26

[BHL00] G. Back, W. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management and sharing in java. In *4th USENIX International Symposium on Operating System Design and Implementation (OSDI)*, 2000. 27

[BO14] Garo Bournoutian and Alex Orailoglu. On-device objective-c application optimization framework for high-performance mobile processors. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 85:1–85:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. 40

[Bor94] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *Proceedings of the IFIP TC6/WG6.5 International Conference on Upper Layer Protocols, Architectures and Applications*, pages 389–402, New York, NY, USA, 1994. Elsevier Science Inc. 29

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM. 40

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press. 23, 52, 54, 101

[CD01] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 36(11):125–138, 2001. 28

[CDT03] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *USENIX Annual Technical Conference, General Track*, pages 85–98, 2003. 28

[CGV10] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Transaction on Embedded Computer Systems*, 9:21:1–21:53, mar 2010. 36, 43

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Open Source Software Development Series. Prentice Hall, 2007. 24

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996. 13

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006. 17

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007. 26

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007. 129

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995. 13

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, January 1984. 67

[DSD08] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008. 20

[FBC+09] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, Eliot, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM. 21, 84

[FO10] Philip W. L. Fong and Simon Orr. Isolating untrusted software extensions by custom scoping rules. *Comput. Lang. Syst. Struct.*, 36(3):268–287, 2010. 25

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. 34, 40

[GNM+03] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, PERCOM '03, pages 107–, Washington, DC, USA, 2003. IEEE Computer Society. 41, 136

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. 64

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association. 71

[HvE02] C. Hawblitzel and T. von Eicken. Luna: a flexible java protection system. *ACM SIGOPS Operating Systems Review*, 36(SI):391–403, 2002. 71

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997. 14, 21, 64

[Jav] Java micro edition. http://java.sun.com/javame/index.jsp. 38, 40, 132

[JCP] Java-Community-Process. Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121. 29

[JLMT98] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in java: A formalization. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society. 25

[JVM] Sun microsystems, inc. jvm profiler interface (jvmpi). 28, 50

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 12, 13, 83

[KWW+09] Christoph Kerschbaumer, Gregor Wagner, Christian Wimmer, Andreas Gal, Christian Steger, and Michael Franz. Slimvm: A small footprint java virtual machine for connected embedded systems. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 133–142, New York, NY, USA, 2009. ACM. 42, 135

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98*, pages 36–44, 1998. 25

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984. 27

[LOW97] Jacob Y. Levy, John K. Ousterhout, and Brent B. Welch. The safe-tcl security model. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1997. 29

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of Asian Symposium on Programming Languages and Systems*, 2005. 34, 36

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987. 13, 16, 19

[Mal] John Maloney. Microsqueak. http://web.media.mit.edu/ jmaloney/microsqueak/. 102, 132

[Mar12] Martinez Peck Mariano. *Application-Level Virtual Memory for Object-Oriented Systems*. PhD thesis, Université de Lille, 2012. 34, 42, 135

[MDC] J. Malenfant, C. Dony, and P. Cointe. Behavioral reflection in a prototype-based language. In *Proceedings of International Workshop on Reflection and Meta-Level Architectures*, pages 143–153. 18

[MDC96] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantics of introspection in a reflective prototype-based language. In *LISP AND SYMBOLIC COMPUTATION*, pages 153–180, 1996. 18

[Mir11] Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011. 64, 67

[MJD96] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of Reflection*, pages 1–20, 1996. 16, 18

[MRC03] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003. 27

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja safe active content in sanitized javascript. Technical report, Google Inc., 2008. 27

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Proceedings of Annual Network and Distributed System Security Symposium (ISOC NSSS)*, pages 375–388, 2010. 59

[NBD+11] Papoulias Nikolaos, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Towards structural de-

composition of reflection with mirrors. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'11)*, Edingburgh, United Kingdom, 2011. 64, 99, 101

[NDG+08] Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, and Matthias Hölzl, editors, *Challenges for Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 64–79. Springer-Verlag, 2008. 1

[OSG] Osgi alliance. 25

[PBF+14] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. Ghost: A uniform and general-purpose proxy implementation. *Journal of Object Technology*, 2014. 119, 120, 135

[PDBF11] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, and Luc. Fabrese. Extended results of Tornado: A Run-Fail-Grow approach for Dynamic Application Tayloring. Technical report, RMod – INRIA Lille-Nord Europe, 2011. 143

[PDF+14] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 2014. 142

[PDFB13] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual smalltalk images: Model and applications. In *IWST - International Workshop on Smalltalk Technology, Co-located within the 21th International Smalltalk Conference - 2013*, 2013. 142

[PRT+04] Lucian Popa, Costin Raiciu, Radu Teodorescu, Irina Athanasiu, and Raju Pandey. Using code collection to support large applications on mobile devices. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, MobiCom '04, pages 16–29, New York, NY, USA, 2004. ACM. 36, 41, 135

[RK02] Derek Rayside and Kostas Kontogiannis. Extracting java library subsets for deployment on embedded systems. *Sci. Comput. Program.*, 45(2-3):245–270, November 2002. 36, 40

[SD10] Olivier Sallenave and Roland Ducournau. Efficient compilation of .net programs for embedded systems. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of*

Object-Oriented Languages, Programs and Systems, ICOOOLPS '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM. 40

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. 64, 100, 135

[Smi82] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982. 17

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984. 13

[Spo00] Lex Spoon. Objects as capabilities in squeak, 2000. 27

[TGP89] David Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press. 37

[Tit06] Ben L. Titzer. Virgil: objects on the head of a pin. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 191–208, New York, NY, USA, 2006. ACM. 40

[TP01] Radu Teodorescu and Raju Pandey. Using jit compilation and configurable runtime systems for efficient deployment of java programs on ubiquitous devices. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, pages 76–95, London, UK, UK, 2001. Springer-Verlag. 36, 41, 135

[TSL03] Frank Tip, Peter F. Sweeney, and Chris Laffra. Extracting library-based java applications. *Commun. ACM*, 46(8):35–40, August 2003. 36, 40

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 73–87, New York, NY, USA, 1995. ACM. 136

[USA05] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM. 23

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 959–972, New York, NY, USA, 2011. ACM. 2, 99, 101, 104

[VCM10] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Dynamic Language Symposium*, volume 45, pages 59–72. ACM, oct 2010. 120

[WGF11] Gregor Wagner, Andreas Gal, and Michael Franz. “slimming” a java virtual machine by way of cold code removal and optimistic partial program loading. *Sci. Comput. Program.*, 76(11):1037–1053, November 2011. 42, 135

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. 23

Published Papers

A.1 Journals

Bootstrapping Reflective Systems: The Case of Pharo.
Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin Ryseghem.
In Science of Computer Programming, 2013. Impact Factor: 0,548

Run-Fail-Grow: a Dynamic Dead Code Elimination Technique.
Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi.
Under submission in Software Practice and Experience. Impact Factor: 1.148

A.2 Workshops

Clara Allende, Guillermo Polito. Virtual Smalltalk Images: Model and Applications. In WISIT - Workshop de Ingeniería en Sistemas y Tecnologías de la Información, 2014.

Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Understanding Pharo's global state to move programs through time and space. In IWST - International Workshop on Smalltalk Technology, Co-located within the 22th International Smalltalk Conference - 2014, 2014.

Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual Smalltalk Images: Model and Applications. In IWST - International Workshop on Smalltalk Technology, Co-located within the 21th International Smalltalk Conference - 2013, 2013.

Appendix A: Pharo Programming Language

Chapter B

Pharo is a Smalltalk inspired object-oriented and dynamically-typed general-purpose language with its own programming environment. The language has a simple and expressive syntax which can be learned in a few minutes. Concepts in Pharo are very consistent, everything is an object: classes, methods, numbers, strings, even the execution context.

Pharo runs on top of a bytecode-based *virtual machine*. Development takes place in an *image* in which all objects reside. All these objects can be modified by the programmer, this includes classes and methods. Hence, we eliminate the typical edit/compile/run cycle and instead incrementally add, remove or modify classes and methods. It is worth noting that *all* classes can be extended with new methods in Pharo. For instance, one can add new operations on integers or strings, classes that are treated as unchangeable internal objects by many other high-level languages. For deployment and debugging, the state of a running image can be saved at any point and subsequently restored.

B.1 Minimal Syntax

Reserved Words	
`nil`	the undefined object
`true`, `false`	boolean objects
`self`	the receiver of the current message
`super`	the receiver, in the superclass context
`thisContext`	the current invocation on the call stack

Literal Object Syntax	
`'a string'`	
`#symbol`	unique string
`$a`	the character `a`
`12 2r1100 16rC`	integers twelve in decimal, binary and hexadecimal encoding
`3.14 1.2e3`	floating-point numbers
`#(abc 123)`	literal array containing the symbol `#abc` and the number `123`
`#[12 16rFF]`	literal byte array containing the bytes/integers 12 and 255
`{foo . 3+2}`	dynamic array built from 2 expressions

Reserved Characters in Expressions	
`"a comment"`	
`.`	expression separator (period)
`;`	message cascade (semicolon)
`:=`	assignment
`^`	return a result from a method (caret)
`[ :p | expr ]`	code block with a parameter
`| foo bar |`	declaration of two temporary variables
`<pragma>`, `<primitive: 3>`	pragma or annotations used in methods, for instances to declare a primitive method.

B.2 Message Sending

A method is called by sending a message to an object called the *receiver*. Each message returns an object. Messages are modeled from natural languages with a subject a verb and complements. There are three types of messages with descending precedence: unary, binary, and keyword.

Unary messages have no arguments.

```
Array new.
```

The first example creates and returns a new instance of the `Array` class, by sending the message `new` to the class `Array` that is an object.

```
#(1 2 3) size.
```

The second message returns the size of the literal array which is `3`.

Binary messages take only one argument and are named by one or more symbol characters.

```
3 + 4.
```

The + message is sent to the integer object `3` with `4` as the argument.

```
'Hello', ' World'.
```

In the second case, the string `'Hello'` receives the message , (comma) with the string `' World'` as the argument.

Keyword messages can take one or more arguments that are inserted in the message name.

```
'Smalltalk' allButFirst: 5.
```

The first example sends the message `allButFirst:` to a string, with the argument `5`. This returns the string `'talk'`.

```
3 to: 10 by: 2.
```

The second example sends `to:by:` to `3`, with arguments `10` and `2`; this returns a collection containing `3`, `5`, `7`, and `9`.

B.3 Precedence

There is a fixed global precedence when evaluating expressions in Pharo: Parentheses > unary > binary > keyword, and finally from left to right.

```
(10 between: 1 and: 2 + 4 * 3) not
```

Here, the messages + and * are sent first, then `between:and:` is sent, and finally `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*, so `2 + 4 * 3` reads left-to-right and thus yields 18 and not the expected 14!

B.4 Cascading Messages

Multiple messages can be sent to the same receiver with `;`.

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi.
```

The message `new` is sent to `OrderedCollection` which results in a new collection to which three `add:` messages are sent with different arguments. The value of the whole message cascade is the value of the last message sent (here, the symbol `#ghi`). This example is the equivalent of first assigning the new collection to a temporary variable and sending three separate `add:` messages:

```
| newCollection |
newCollection := OrderedCollection new.
newCollection add: #abc.
newCollection add: #def.
newCollection add: #ghi.
```

To return the original receiver of the message cascade (*i.e.*, the collection) instead of the last result (*i.e.*, `#ghi`), the `yourself` message is used:

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi;
  yourself.
```

B.5 Blocks

Blocks are objects containing code that is executed on demand, (anonymous functions or closures). They are the basis for control structures like conditionals and loops.

```
2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

The first example sends the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled.

```
#('Hello World' $!)
  do: [ :e | Transcript show: e ]
```

The next example sends the message `do:` to an array. This evaluates the block once for each element, passing it via the `e` parameter. As a result, `Hello World!` is printed.

B.6 Methods

Methods are first-class objects in Pharo and can be inspected and modified on the fly. Methods are created by saving expressions in the Pharo development environment. Typically methods are printed with a special first line indicating the class the method is installed on and the name or selector it is given.

```
Array >> helpMethod
    2 = 2
        ifTrue: [ Error signal: 'Help' ].
```

This example would denote a simple method with a unary selector on the `Array` class. This method could be invoked by evaluating `Array new helpMethod`.

Certain methods are marked with a pragma to use predefined primitives from the VM. These are used for expressions that cannot be expressed in Pharo. For instance the `basicNew` which allocates new objects uses the primitive number 70:

```
Behavior >> basicNew
    "Answer a new instance of this class"
    <primitive: 70>
    OutOfMemory signal.
```

Bootstrap Extracts

Chapter C

This appendix shows the extracts of classes and/or methods for the languages we bootstrapped.

C.1 Pharo Bootstrap Extract

This section lists the classes that were extracted from Pharo (version 3) for its bootstrapping. For brevity we do not list the methods in these classes. Additionally, our extraction for bootstrap includes all methods in such classes.

ASTCache
Abort
AbstractClassInstaller
AbstractClassModification
AbstractCompiler
AbstractFieldModification
AbstractLayout
AbstractMethodUpdateStrategy
AbstractModification
AbstractTimeZone
AddedField
AdditionalMethodState
AllProtocol
Announcement
AnnouncementLogger
AnnouncementSet
AnnouncementSubscription
Announcer
AnonymousClassInstaller
ArithmeticError
Array
ArrayedCollection
AssertionFailure
Association
Author
AuthorNameRequest
Bag
Beeper
Behavior
BitsLayout
BlockCannotReturn
BlockClosure
BlockLocalTempCounter
Boolean
ByteArray
ByteLayout
ByteString
ByteSymbol
CCompilationContext
Categorizer
CategoryAdded
CategoryRemoved
CategoryRenamed
ChangesLog
Character
CharacterSet
CharacterSetComplement
ChronologyConstants
CircularHierarchyError
Class
ClassAdded
ClassAnnouncement
ClassCategoryReader
ClassCommentReader
ClassCommented
ClassDescription

ClassModification
ClassModificationPropagation
ClassModifiedClassDefinition
ClassOrganization
ClassRecategorized
ClassRemoved
ClassRenamed
ClassReorganized
ClassTrait
Collection
CollectionIsEmpty
CombinedChar
CompilationContext
CompiledMethod
CompiledMethodLayout
CompiledMethodTrailer
ContextPart
Continuation
DangerousClassNotifier
Date
DateAndTime
DateParser
DebuggerMethodMapOpal
DeepCopier
DefaultExternalDropHandler
Delay
DelayWaitTimeout
DependentsArray
Deprecation
Dictionary
DomainError
DosTimestamp
DuplicatedSlotName
DuplicatedVariableError
Duration
DynamicVariable
EmptyLayout
Error
EventManager
EventSensorConstants
ExactFloatPrintPolicy
Exception
ExceptionSet
ExceptionSetWithExclusions
Exit
ExpressionEvaluated
ExternalDropHandler
ExternalSemaphoreTable
False
FixedLayout
Float
FloatArray
FloatPrintPolicy
FloatingPointException
Fraction
Generator
Halt
HashTableSizes
HashedCollection
Heap
IRAccess
IRBlockReturnTop
IRBuilder
IRBytecodeDecompiler
IRBytecodeGenerator
IRBytecodeScope
IRInstVarAccess
IRInstruction
IRInterpreter
IRJump
IRJumpIf
IRLiteralVariableAccess
IRMethod
IRPop
IRPrimitive
IRPrinter
IRPushArray
IRPushClosureCopy
IRPushDup
IRPushLiteral
IRReceiverAccess
IRReconstructor
IRRemoteArray
IRRemoteTempAccess
IRReturn
IRSend
IRSequence
IRStackCount
IRTempAccess
IRTempVector
IRThisContextAccess
IRTranslator

IRVisitor
IdentityBag
IdentityDictionary
IdentitySet
IllegalResumeAttempt
InMidstOfFileinNotification
IncompatibleLayoutConflict
InexactFloatPrintPolicy
InputEventFetcher
InputEventHandler
InputEventSensor
InstVarRefLocator
InstanceModification
InstructionClient
InstructionPrinter
InstructionStream
Integer
IntegerArray
Interval
InvalidGlobalName
InvalidSlotName
InvalidSuperclass
Job
JobChange
JobDetector
JobEnd
JobNotification
JobProgress
JobStart
JobStartNotification
KeyNotFound
KeyedTree
LargeInteger
LargeNegativeInteger
LargePositiveInteger
LayoutAbstractScope
LayoutClassScope
LayoutEmptyScope
LegacyWeakSubscription
LimitedWriteStream
LimitingLineStreamWrapper
Link
LinkedList
LocalTimeZone
LookupKey
Magnitude
ManifestASTCore
ManifestOpalCompilerCore
Margin
Matrix
Message
MessageCatcher
MessageNotUnderstood
MessageSend
Metaclass
MethodAdded
MethodClassifier
MethodContext
MethodDictionary
MethodModification
MethodModified
MethodRecategorized
MethodRecompileStrategy
MethodRemoved
Model
ModifiedField
Monitor
MonitorDelay
Month
MultiByteBinaryOrTextStream
Mutex
MutexSet
NaNException
NonBooleanReceiver
NonInteractiveTranscript
NotFound
NotYetImplemented
Notification
NullStream
Number
NumberParser
OCASTClosureAnalyzer
OCASTSemanticAnalyzer
OCASTTranslator
OCASTTranslatorForEffect
OCASTTranslatorForValue
OCAbstractLocalVariable
OCAbstractMethodScope
OCAbstractScope
OCAbstractVariable
OCBlockScope
OCClassScope

OCCopyingTempVariable
OCInstanceScope
OCInstanceVariable
OCKeyedSet
OCLiteralList
OCLiteralSet
OCLiteralVariable
OCMethodScope
OCOptimizedBlockScope
OCRequestorScope
OCSemanticError
OCSemanticWarning
OCShadowVariableWarning
OCSourceCodeChanged
OCSpecialVariable
OCTempVariable
OCUndeclaredVariable
OCUndeclaredVariableWarning
OCUninitializedVariableWarning
OCUnknownSelectorWarning
OCUnusedVariableWarning
OCVectorTempVariable
Object
ObjectFinalizer
ObjectFinalizerCollection
ObjectLayout
OldClassBuilderAdapter
OpalCompiler
OrderedCollection
OrderedIdentityDictionary
OutOfMemory
PackageInfo
PackageOrganizer
PharoClassInstaller
PluggableDictionary
PluggableSet
Point
PointerLayout
PositionableStream
Pragma
PragmaAdded
PragmaAnnouncement
PragmaCollector
PragmaCollectorReset
PragmaRemoved
PragmaUpdated
PrimitiveFailed
Process
ProcessLocalVariable
ProcessSpecificVariable
ProcessorScheduler
ProtoObject
Protocol
ProtocolAdded
ProtocolAnnouncement
ProtocolOrganizer
ProtocolRemovalException
ProtocolRemoved
PseudoClassOrganization
RBArgumentNode
RBArrayNode
RBAssignmentNode
RBAssignmentToken
RBBinarySelectorToken
RBBlockNode
RBBlockReplaceRule
RBCascadeNode
RBClassReference
RBConfigurableFormatter
RBErrorToken
RBExplicitVariableParser
RBIdentifierToken
RBKeywordToken
RBLiteralArrayNode
RBLiteralArrayToken
RBLiteralNode
RBLiteralToken
RBLiteralValueNode
RBMessageNode
RBMethodNode
RBMultiKeywordLiteralToken
RBNumberLiteralToken
RBParseErrorNode
RBParseTreeRewriter
RBParseTreeRule
RBParseTreeSearcher
RBParser
RBPatternBlockNode
RBPatternBlockToken
RBPatternMessageNode
RBPatternMethodNode
RBPatternParser

RBPatternPragmaNode
RBPatternScanner
RBPatternVariableNode
RBPatternWrapperBlockNode
RBPragmaNode
RBProgramNode
RBProgramNodeVisitor
RBReadBeforeWrittenTester
RBReplaceRule
RBReturnNode
RBScanner
RBSearchRule
RBSelfNode
RBSequenceNode
RBShortAssignmentToken
RBSpecialCharacterToken
RBStringReplaceRule
RBStringReplacement
RBSuperNode
RBTemporaryNode
RBThisContextNode
RBToken
RBValueNode
RBValueToken
RBVariableNode
RPackage
RPackageAnnouncement
RPackageConflictError
RPackageCreated
RPackageOrganizer
RPackageRenamed
RPackageSet
RPackageTag
RPackageUnregistered
RWBinaryOrTextStream
Random
ReadStream
ReadWriteStream
Rectangle
RelativeInstructionPrinter
RemovedField
ScaledDecimal
Schedule
SelectorException
Semaphore
SequenceableCollection
Session
Set
SetElement
SharedPool
SharedQueue
ShiftedField
ShouldBeImplemented
ShouldNotImplement
SimulationExceptionWrapper
SizeMismatch
Slot
SlotClassBuilder
SlotClassBuilderError
SlotClassBuilderWarning
SlotNotFound
SmallDictionary
SmallIdentityDictionary
SmallInteger
SmallIntegerLayout
SmalltalkImage
SnapshotDone
SortedCollection
SparseLargeArray
SparseLargeTable
Stopwatch
Stream
String
SubclassResponsibility
SubscriptOutOfBounds
SubscriptionRegistry
Symbol
SyntaxErrorNotification
SystemAnnouncement
SystemAnnouncer
SystemDictionary
SystemNavigation
SystemOrganizer
SystemVersion
TApplyingOnClassSide
TBehavior
TBehaviorCategorization
TClass
TClassDescription
TComparable
TComposingDescription
TIRVisitor

TRBProgramNodeVisitor
TSortable
TTransformationCompatibility
TextStream
ThreadSafeTranscript
Time
TimeStamp
TimeZone
TimedOut
Timespan
Trait
TraitAlias
TraitBehavior
TraitComposition
TraitCompositionException
TraitDescription
TraitException
TraitExclusion
TraitMethodDescription
TraitTransformation
True
UndefinedObject
UnhandledError
UnmodifiedField
UserInterruptHandler
ValueLink
ValueNotFound
VariableLayout
VirtualMachine
Warning
WeakActionSequence
WeakAnnouncementSubscription
WeakArray
WeakFinalizationList
WeakFinalizerItem
WeakIdentityKeyDictionary
WeakKeyAssociation
WeakKeyDictionary
WeakKeyToCollectionDictionary
WeakLayout
WeakMessageSend
WeakOrderedCollection
WeakRegistry
WeakSet
WeakSubscriptionBuilder
WeakValueAssociation
WeakValueDictionary
Week
WideCharacterSet
WideString
WideSymbol
WordArray
WordLayout
WriteStream
Year
ZeroDivide

C.2 Candle Bootstrap Extract

This section lists the methods from Candle for its bootstrapping.

PCAssociation»hash
PCAssociation»value:
PCAssociation»key
PCAssociation»key:
PCAssociation»=
PCAssociation»value
PCAssociation»key:value:
PCAssociation»printOn:
PCAssociation»<
PCOrderedCollection»remove:ifAbsent:
PCOrderedCollection»last
PCOrderedCollection»size
PCOrderedCollection»errorNoSuchElement
PCOrderedCollection»add:
PCOrderedCollection»at:
PCOrderedCollection»at:put:
PCOrderedCollection»makeRoomAtFirst
PCOrderedCollection»copyFrom:to:
PCOrderedCollection»grow
PCOrderedCollection»setCollection:
PCOrderedCollection»collect:
PCOrderedCollection»makeRoomAtLast
PCOrderedCollection»
copyReplaceFrom:to:with:

PCOrderedCollection»addFirst:
PCOrderedCollection»do:
PCOrderedCollection»first
PCOrderedCollection»insert:before:
PCOrderedCollection»removeFirst
PCOrderedCollection»removeIndex:
PCOrderedCollection»removeLast
PCOrderedCollection»select:
PCIdentityDictionary»scanFor:
PCIdentityDictionary»keys
PCFile»primWrite:from:startingAt:count:
PCFile»close
PCFile»cr
PCFile»position
PCFile»position:
PCFile»localFolderPath
PCFile»name
PCFile»nextPutAll:
PCFile»primClose:
PCFile»openReadWrite:
PCFile»primGetPosition:
PCFile»size
PCFile»readInto:startingAt:count:
PCFile»primOpen:writable:
PCFile»next:
PCFile»openReadOnly:
PCFile»primRead:into:startingAt:count:
PCFile»primSize:
PCFile»primImageName
PCFile»primSetPosition:to:
PCMessage»lookupClass
PCMessage»sentTo:
PCMessage»arguments
PCMessage»printOn:
PCMessage»selector
PCIdentitySet»scanFor:
PCFloat»ln
PCFloat»printOn:base:
PCFloat»reciprocalLogBase2
PCFloat»sqrt
PCFloat»tan
PCFloat»truncated
PCFloat»reciprocal
PCFloat»raisedTo:
PCFloat»hash
PCFloat»/
PCFloat»rounded
PCFloat»radiansToDegrees
PCFloat»-
PCFloat»abs
PCFloat»=
PCFloat»adaptToInteger:andSend:
PCFloat»arcCos
PCFloat»>
PCFloat»arcTan
PCFloat»cos
PCFloat»floorLog:
PCFloat»significand
PCFloat»timesTwoPower:
PCFloat»fractionPart
PCFloat»exponent
PCFloat»isNaN
PCFloat»asFloat
PCFloat»arcSin
PCFloat»sin
PCFloat» =
PCFloat»degreesToRadians
PCFloat»isInfinite
PCFloat»log
PCFloat»sign
PCFloat»*
PCFloat»<=
PCFloat»>=
PCFloat»exp
PCFloat»negated
PCFloat»absPrintOn:base:
PCFloat»+
PCFloat»<
PCClass»name:
PCClass»classSide
PCClass»
    weakSubclass:instanceVariableNames:
    classVariableNames:
PCClass»initFrom:methodDict:
PCClass»
    variableWordSubclass:
    instanceVariableNames:
    classVariableNames:
PCClass»
    newClassBuilderForSubclass:
    instanceVariableNames:
    classVariableNames:

PCClass»instVarNames
PCClass»
variableByteSubclass:
instanceVariableNames:
classVariableNames:
PCClass»name
PCClass»isMeta
PCClass»instVarNames:
PCClass»theNonMetaClass
PCClass»subclass:
instanceVariableNames:
classVariableNames:
PCClass»variableSubclass:
instanceVariableNames:
classVariableNames:
PCClass»classVariables
PCClass»classVariables:
PCForm»bits
PCForm»fillRectX:y:w:h:
PCForm»setColorR:g:b:
PCForm»primScreenSize
PCForm»beDisplayDepth:
PCForm»depth
PCForm»setWidth:height:depth:
PCForm»height
PCForm»copyX:y:width:height:
PCForm»drawForm:x:y:rule:
PCForm»width
PCCompiledMethod»numLiterals
PCCompiledMethod»objectAt:put:
PCCompiledMethod»header
PCCompiledMethod»objectAt:
PCCompiledMethod»numTemps
PCCompiledMethod»frameSize
PCCompiledMethod»initialPC
PCCompiledMethod»flushCache
PCCompiledMethod»isCompiledMethod
PCArray»hash
PCArray»replaceFrom:to:with:startingAt:
PCArray»asArray
PCArray»asDictionary
PCArray»elementsExchangeIdentityWith:
PCArray»printOn:
PCLinkedList»linkAt:
PCLinkedList»at:putLink:
PCLinkedList»firstLink
PCLinkedList»removeAllSuchThat:
PCLinkedList»copyWith:
PCLinkedList»at:put:
PCLinkedList»first
PCLinkedList»linkOf:ifAbsent:
PCLinkedList»add:after:
PCLinkedList»isEmpty
PCLinkedList»do:
PCLinkedList»indexOf:startingAt:ifAbsent:
PCLinkedList»last
PCLinkedList»removeFirst
PCLinkedList»linksDo:
PCLinkedList»postCopy
PCLinkedList»add:afterLink:
PCLinkedList»removeAll
PCLinkedList»removeLink:
PCLinkedList»linkOf:
PCLinkedList»at:
PCLinkedList»removeLast
PCLinkedList»copyWithout:
PCLinkedList»addLast:
PCLinkedList»add:
PCLinkedList»add:before:
PCLinkedList»add:beforeLink:
PCLinkedList»lastLink
PCLinkedList»linkAt:ifAbsent:
PCLinkedList»collect:
PCLinkedList»addFirst:
PCLinkedList»removeLink:ifAbsent:
PCLinkedList»validIndex:
PCLinkedList»species
PCLinkedList»swap:with:
PCLinkedList»remove:ifAbsent:
PCBehavior»basicNew:
PCBehavior»superclass
PCBehavior»sharedPools
PCBehavior»superclass:
PCBehavior»initialize
PCBehavior»printOn:
PCBehavior»inheritsFrom:
PCBehavior»isCompact
PCBehavior»allInstancesDo:
PCBehavior»name
PCBehavior»someInstance
PCBehavior»format
PCBehavior»allInstances

PCBehavior»isBits
PCBehavior»isBytes
PCBehavior»isPointers
PCBehavior»methodDict
PCBehavior»isVariable
PCBehavior»new
PCBehavior»basicNew
PCBehavior»allInstVarNames
PCBehavior»classPool
PCBehavior»»
PCBehavior»instSize
PCBehavior»new:
PCBehavior»indexIfCompact
PCBehavior»selectorAtMethod:setClass:
PCBehavior»isBehavior
PCBehavior»lookupSelector:
PCBehavior»canUnderstand:
PCBehavior»setFormat:
PCBehavior»instSpec
PCBehavior»setCompactClassIndex:
PCUndefinedObject»printOn:
PCUndefinedObject»subclass:
instanceVariableNames:
classVariableNames:
PCUndefinedObject»isNil
PCUndefinedObject»ifNotNil:
PCUndefinedObject»basicCopy
PCUndefinedObject»ifNil:
PCUndefinedObject»ifNil:ifNotNil:
PCLargePositiveInteger»digitLength
PCLargePositiveInteger»negative
PCLargePositiveInteger»\
PCLargePositiveInteger»highBit
PCLargePositiveInteger»negated
PCLargePositiveInteger»/
PCLargePositiveInteger»//
PCLargePositiveInteger»+
PCLargePositiveInteger»<=
PCLargePositiveInteger»>
PCLargePositiveInteger»>=
PCLargePositiveInteger»normalize
PCLargePositiveInteger»
replaceFrom:to:with:startingAt:
PCLargePositiveInteger»digitAt:
PCLargePositiveInteger»abs
PCLargePositiveInteger»bitAnd:
PCLargePositiveInteger»sign
PCLargePositiveInteger»-
PCLargePositiveInteger»*
PCLargePositiveInteger»bitXor:
PCLargePositiveInteger»=
PCLargePositiveInteger»digitAt:put:
PCLargePositiveInteger»bitOr:
PCLargePositiveInteger»=
PCLargePositiveInteger»bitShift:
PCLargePositiveInteger»quo:
PCLargePositiveInteger»<
PCSmallInteger»>
PCSmallInteger»<=
PCSmallInteger»\\
PCSmallInteger»digitAt:
PCSmallInteger»>=
PCSmallInteger»identityHash
PCSmallInteger»printOn:base:
PCSmallInteger»hash
PCSmallInteger»=
PCSmallInteger»//
PCSmallInteger»=
PCSmallInteger»asFloat
PCSmallInteger»hashMultiply
PCSmallInteger»-
PCSmallInteger»basicIdentityHash
PCSmallInteger»highBit
PCSmallInteger»*
PCSmallInteger»+
PCSmallInteger»basicCopy
PCSmallInteger»bitAnd:
PCSmallInteger»bitOr:
PCSmallInteger»digitAt:put:
PCSmallInteger»isSmallInteger
PCSmallInteger»<
PCSmallInteger»bitShift:
PCSmallInteger»quo:
PCSmallInteger»digitLength
PCSmallInteger»/
PCSmallInteger»bitXor:
PCInterval»size
PCInterval»last
PCInterval»at:
PCInterval»at:put:
PCInterval»species
PCInterval»add:

PCInterval»=
PCInterval»first
PCInterval»hash
PCInterval»includes:
PCInterval»setFrom:to:by:
PCInterval»remove:
PCInterval»do:
PCInterval»increment
PCInterval»collect:
PCInterval»printOn:
PCMagnitude»>
PCMagnitude»=
PCMagnitude»<=
PCMagnitude»>=
PCMagnitude»between:and:
PCMagnitude»hash
PCMagnitude»min:
PCMagnitude»max:
PCMagnitude»<
PCSequenceableCollection»
indexOf:startingAt:ifAbsent:
PCSequenceableCollection»
replaceFrom:to:with:
PCSequenceableCollection»
replaceFrom:to:with:startingAt:
PCSequenceableCollection»last
PCSequenceableCollection»at:ifAbsent:
PCSequenceableCollection»copyWith:
PCSequenceableCollection»
copyReplaceFrom:to:with:
PCSequenceableCollection»,
PCSequenceableCollection»do:
PCSequenceableCollection»first
PCSequenceableCollection»select:
PCSequenceableCollection»size
PCSequenceableCollection»=
PCSequenceableCollection»copyFrom:to:
PCSequenceableCollection»asArray
PCSequenceableCollection»indexOf:ifAbsent:
PCSequenceableCollection»remove:ifAbsent:
PCSequenceableCollection»swap:with:
PCSequenceableCollection»collect:
PCBitBlt»fillWords:
PCBitBlt»rule:
PCBitBlt»copyBits
PCBitBlt»sourceForm:
PCBitBlt»copyBitsTranslucent:
PCBitBlt»destForm:
PCBitBlt»sourceX:y:
PCBitBlt»destX:y:width:height:
PCBitBlt»fillWords
PCBitBlt»width:height:
PCBitBlt»initialize
PCBitBlt»fillR:g:b:
PCBitBlt»clipX:y:width:height:
PCNumber»adaptToInteger:andSend:
PCNumber»to:by:
PCNumber»to:by:do:
PCNumber»floorLog:
PCNumber»abs
PCNumber»cos
PCNumber»isNumber
PCNumber»arcTan
PCNumber»log
PCNumber»reciprocal
PCNumber»exp
PCNumber»//
PCNumber»/
PCNumber»asInteger
PCNumber»log:
PCNumber»raisedToInteger:
PCNumber»printStringBase:
PCNumber»rem:
PCNumber»roundUpTo:
PCNumber»adaptToFloat:andSend:
PCNumber»rounded
PCNumber»to:do:
PCNumber»arcSin
PCNumber»printOn:
PCNumber»+
PCNumber»
PCNumber»ln
PCNumber»degreesToRadians
PCNumber»negated
PCNumber»quo:
PCNumber»radiansToDegrees
PCNumber»tan
PCNumber»-
PCNumber»arcCos
PCNumber»ceiling
PCNumber»negative

PCNumber»to:
PCNumber»roundTo:
PCNumber»truncated
PCNumber»sin
PCNumber»sign
PCNumber»truncateTo:
PCNumber»*
PCNumber»floor
PCNumber»raisedTo:
PCNumber»sqrt
PCLargeNegativeInteger»abs
PCLargeNegativeInteger»negative
PCLargeNegativeInteger»normalize
PCLargeNegativeInteger»sign
PCLargeNegativeInteger»printOn:base:
PCLargeNegativeInteger»negated
PCProcess»nextLink
PCProcess»printOn:
PCProcess»priority
PCProcess»errorHandler:
PCProcess»resume
PCProcess»suspendedContext
PCProcess»suspend
PCProcess»terminate
PCProcess»priority:
PCProcess»nextLink:
PCProcess»initSuspendedContext:
PCProcess»errorHandler
PCCharacter»=
PCCharacter»isLetter
PCCharacter»printOn:
PCCharacter»isSpecial
PCCharacter»isVowel
PCCharacter»to:
PCCharacter»asCharacter
PCCharacter»setValue:
PCCharacter»>
PCCharacter»asLowercase
PCCharacter»<
PCCharacter»asInteger
PCCharacter»asString
PCCharacter»asUppercase
PCCharacter»basicCopy
PCCharacter»tokenish
PCCharacter»hash
PCCharacter»digitValue
PCCharacter»asciiValue
PCCharacter»isDigit
PCCharacter»isUppercase
PCSet»fullCheck
PCSet»fixCollisionsFrom:
PCSet»keyAt:
PCSet»copy
PCSet»init:
PCSet»=
PCSet»grow
PCSet»withArray:
PCSet»asArray
PCSet»findElementOrNil:
PCSet»includes:
PCSet»noCheckAdd:
PCSet»asSet
PCSet»remove:ifAbsent:
PCSet»atNewIndex:put:
PCSet»scanFor:
PCSet»add:
PCSet»do:
PCSet»size
PCSet»collect:
PCSet»swap:with:
PCReadStream»peek
PCReadStream»atEnd
PCReadStream»position:
PCReadStream»next
PCReadStream»skip:
PCReadStream»position
PCReadStream»contents
PCReadStream»peekFor:
PCReadStream»size
PCReadStream»on:
PCReadStream»next:
PCMetaclass»theNonMetaClass
PCMetaclass»new
PCMetaclass»initMethodDict:
PCMetaclass»isMeta
PCMetaclass»name
PCMetaclass»soleInstance:
PCObject»ifNil:
PCObject»pointsTo:
PCObject»class
PCObject»ifNotNil:ifNil:
PCObject»isCompiledMethod

PCObject»respondsTo:
PCObject»
PCObject»asLink
PCObject»perform:
PCObject»perform:withArguments:inSuperclass:
PCObject»shouldNotImplement
PCObject»instVarAt:
PCObject»someObject
PCObject»printOn:
PCObject»yourself
PCObject»=
PCObject»initialize
PCObject»isBehavior
PCObject»nextInstance
PCObject»become:
PCObject»putAscii:
PCObject»species
PCObject»tryPrimitive:withArgs:
PCObject»basicAt:
PCObject»isNil
PCObject»putcr
PCObject»isNumber
PCObject»errorSubscriptBounds:
PCObject»isContextPart
PCObject»isSelfEvaluating
PCObject»beep
PCObject»nextObject
PCObject»isInteger
PCObject»basicAt:put:
PCObject»handleExceptionName:context:
PCObject»identityHash
PCObject»asString
PCObject»errorNonIntegerIndex
PCObject»perform:withArguments:
PCObject»hash
PCObject»primitiveFailed
PCObject»printString
PCObject»instVarAt:put:
PCObject» =
PCObject»at:put:
PCObject»shouldBePrintedAsLiteral
PCObject»isSmallInteger
PCObject»error:
PCObject»basicSize
PCObject»subclassResponsibility
PCObject»->
PCObject»at:
PCObject»ifNotNil:
PCObject»basicCopy
PCObject»basicIdentityHash
PCObject»copy
PCObject»doesNotUnderstand:
PCObject»ifNil:ifNotNil:
PCObject»perform:with:
PCObject»==
PCObject»mustBeBoolean
PCObject»putString:
PCObject»isKindOf:
PCObject»errorImproperStore
PCByteArray»asByteArray
PCByteArray»
replaceFrom:to:with:startingAt:
PCByteArray»asString
PCDictionary»includes:
PCDictionary»remove:ifAbsent:
PCDictionary»associationAt:
PCDictionary»remove:
PCDictionary»removeKey:
PCDictionary»keysDo:
PCDictionary»collect:
PCDictionary»associationAt:ifAbsent:
PCDictionary»keyAtValue:ifAbsent:
PCDictionary»noCheckAdd:
PCDictionary»printOn:
PCDictionary»scanFor:
PCDictionary»errorKeyNotFound
PCDictionary»at:
PCDictionary»at:ifAbsent:
PCDictionary»errorValueNotFound
PCDictionary»keyAt:
PCDictionary»keyAtValue:
PCDictionary»keys
PCDictionary»associationsDo:
PCDictionary»copy
PCDictionary»removeKey:ifAbsent:
PCDictionary»add:
PCDictionary»at:put:
PCDictionary»do:
PCDictionary»includesKey:
PCDictionary»select:
PCSemaphore»=
PCSemaphore»signal

PCSemaphore»initialize
PCSemaphore»critical:
PCSemaphore»hash
PCSemaphore»wait
PCValueLink»value:
PCValueLink»nextLink
PCValueLink»nextLink:
PCValueLink»value
PCValueLink»=
PCValueLink»asLink
PCValueLink»hash
PCValueLink»printOn:
PCMethodDictionary»associationsDo:
PCMethodDictionary»at:put:
PCMethodDictionary»keysDo:
PCMethodDictionary»keyAt:
PCMethodDictionary»do:
PCMethodDictionary»grow
PCMethodDictionary»at:ifAbsent:
PCMethodDictionary»includesKey:
PCMethodDictionary»removeKey:ifAbsent:
PCMethodDictionary»
keyAtIdentityValue:ifAbsent:
PCMethodDictionary»add:
PCMethodDictionary»copy
PCMethodDictionary»scanFor:
PCMethodDictionary»swap:with:
PCBlock»value:
PCBlock»value:value:
PCBlock»asContext
PCBlock»outerContext
PCBlock»asContextWithSender:
PCBlock»valueWithArguments:
PCBlock»home
PCBlock»ifError:
PCBlock»method
PCBlock»msecs
PCBlock»numArgs
PCBlock»numCopiedValues
PCBlock»value
PCContext»blockCopy:
PCContext»sender
PCContext»isContextPart
PCProcessorScheduler»initProcessLists
PCProcessorScheduler»installStartProcess
PCProcessorScheduler»remove:ifAbsent:
PCProcessorScheduler»installIdleProcess
PCProcessorScheduler»activeProcess
PCProcessorScheduler»highestPriority
PCProcessorScheduler»idleProcess
PCCollection»add:
PCCollection»asByteArray
PCCollection»do:
PCCollection»errorNotFound
PCCollection»collect:
PCCollection»includes:
PCCollection»emptyCheck
PCCollection»asArray
PCCollection»printOn:
PCCollection»select:
PCCollection»detect:ifNone:
PCCollection»sum
PCCollection»isEmpty
PCCollection»asSet
PCCollection»remove:
PCCollection»remove:ifAbsent:
PCCollection»size
PCCollection»errorEmptyCollection
PCArrayedCollection»sort:
PCArrayedCollection»size
PCArrayedCollection»sort
PCArrayedCollection»mergeSortFrom:to:by:
PCArrayedCollection»add:
PCArrayedCollection»
mergeFirst:middle:last:into:by:
PCArrayedCollection»
mergeSortFrom:to:src:dst:by:
PCWriteStream»on:
PCWriteStream»pastEndPut:
PCWriteStream»contents
PCWriteStream»nextPut:
PCWriteStream»position:
PCWriteStream»size
PCWriteStream»space
PCWriteStream»nextPutAll:
PCString»>
PCString»
findString:startingAt:caseSensitive:
PCString»substrings
PCString»size
PCString»<
PCString»asSymbol

PCString»
findSubstring:in:startingAt:matchTable:
PCString»asByteArray
PCString»compare:with:collated:
PCString»findDelimiters:startingAt:
PCString»hash
PCString»indexOfAscii:inString:startingAt:
PCString»findTokens:
PCString»numArgs
PCString»<=
PCString»replaceFrom:to:with:startingAt:
PCString»>=
PCString»at:
PCString»indexOf:startingAt:
PCString»printOn:
PCString»at:put:
PCString»translate:from:to:table:
PCString»asLowercase
PCString»asString
PCString»=
PCString»skipDelimiters:startingAt:
PCString»compare:
PCString»indexOf:startingAt:ifAbsent:
PCClassBuilder»isVariable
PCClassBuilder»initialize
PCClassBuilder»instSize
PCClassBuilder»beBytes
PCClassBuilder»name:
PCClassBuilder»classVariableNames:
PCClassBuilder»compactClassIndex
PCClassBuilder»beCompiledMethod
PCClassBuilder»isPointers
PCClassBuilder»isWords
PCClassBuilder»
isCompiledMethodClassIndex
PCClassBuilder»beVariable
PCClassBuilder»instVarNames:
PCClassBuilder»isCompiledMethod
PCClassBuilder»beWords
PCClassBuilder»isWeak
PCClassBuilder»newClassFormat
PCClassBuilder»bePointers
PCClassBuilder»build
PCClassBuilder»superclass:
PCClassBuilder»beWeak
PCClassBuilder»compactClassIndexFor:
PCClassBuilder»classVariableNames
PCSymbol»flushCache
PCSymbol»printOn:
PCSymbol»initFrom:
PCSymbol»species
PCSymbol»=
PCSymbol»asString
PCSymbol»asSymbol
PCSymbol»hash
PCSymbol»errorNoModification
PCSymbol»replaceFrom:to:with:startingAt:
PCSymbol»at:put:
PCSymbol»basicCopy
PCInteger»bitOr:
PCInteger»=
PCInteger»floor
PCInteger»asCharacter
PCInteger»//
PCInteger»lastDigit
PCInteger»printOn:base:
PCInteger»<
PCInteger»truncated
PCInteger»digitDiv:neg:
PCInteger»bitInvert
PCInteger»copyto:
PCInteger»benchFib
PCInteger»rounded
PCInteger»*
PCInteger»benchmark
PCInteger»asInteger
PCInteger»bitClear:
PCInteger»bitShift:
PCInteger»ceiling
PCInteger»digitAdd:
PCInteger»bitXor:
PCInteger»isInteger
PCInteger»growby:
PCInteger»bitAnd:
PCInteger»/
PCInteger»quo:
PCInteger»+
PCInteger»-
PCInteger»digitRshift:bytes:lookfirst:
PCInteger»hash
PCInteger»digitCompare:
PCInteger»digitLogic:op:length:

PCInteger»digitMultiply:neg:
PCInteger»replaceFrom:to:with:startingAt:
PCInteger»>
PCInteger»asFloat
PCInteger»digitSubtract:
PCInteger»timesRepeat:
PCInteger»normalize
PCInteger»growto:
PCInteger»digitLshift:
PCProcessList»first
PCProcessList»remove:ifAbsent:
PCProcessList»removeFirst
PCProcessList»add:
PCProcessList»do:
PCProcessList»isEmpty
PCProcessList»addLast:
PCProcessList»size
PCFalse»&
PCFalse»not
PCFalse»or:
PCFalse»printOn:
PCFalse»and:
PCFalse»ifTrue:
PCFalse»|
PCFalse»ifFalse:
PCFalse»ifTrue:ifFalse:
PCTrue»or:
PCTrue»ifTrue:
PCTrue»&
PCTrue»basicCopy
PCTrue»ifTrue:ifFalse:
PCTrue»not
PCTrue»ifFalse:
PCTrue»printOn:
PCTrue»and:
PCTrue»|
PCMethodContext»privRefresh
PCMethodContext»stackp:
PCMethodContext»removeSelf
PCMethodContext»
setSender:receiver:method:closure:startpc:
PCMethodContext»tempAt:put:
PCMethodContext»asContext
PCMethodContext»method
PCMethodContext»home
PCPoint»abs
PCPoint»degrees
PCPoint»hash
PCPoint»min:
PCPoint»*
PCPoint»-
PCPoint»r
PCPoint»rounded
PCPoint»printOn:
PCPoint»dist:
PCPoint»adaptToFloat:andSend:
PCPoint»theta
PCPoint»asPoint
PCPoint»crossProduct:
PCPoint»max:
PCPoint»y
PCPoint»dotProduct:
PCPoint»/
PCPoint»setX:setY:
PCPoint»+
PCPoint»adaptToInteger:andSend:
PCPoint»truncated
PCPoint»=
PCPoint»setR:degrees:
PCPoint»//
PCPoint»x
PCPoint»negated

C.3 MetaTalk Bootstrap Extract

This section lists the methods from MetaTalk for its bootstrapping. This list includes mirrors and base-level classes.

ObjectMirror»initialize
ObjectMirror»classMirror
ObjectMirror»setClassMirror:
ObjectMirror»baseObject
ObjectMirror»setBaseObject:
ClassMirror»atMethod:put:

ClassMirror»baseObject
ClassMirror»super
ClassMirror»allocate
ClassMirror»subClass:instanceVariableNames:
ClassMirror»setClassMirror:
ClassMirror»initialize
ClassMirror»name:
ClassMirror»methods
ClassMirror»classMirror
ClassMirror»newWithBaseObject:
ClassMirror»super:
ClassMirror»instanceVariables:
ClassMirror»methodSourceOf:
ClassMirror»isClass
ClassMirror»indexOf:
ClassMirror»methodAt:
ClassMirror»atMethod:compile:
ClassMirror»instanceVariables
ClassMirror»name
ClassMirror»setBaseObject:
File»nextPutAll:
File»primOpen:writable:
File»position:
File»primWrite:from:startingAt:count:
File»openReadWrite:
File»cr
File»primClose:
File»primSetPosition:to:
File»close
File»size
File»primSize:
Char»asString
Char»asCharacter
Point3D»initialize
Point3D»z:
Point3D»z
Point3D»asString
True»asString
True»not
True»ifTrue:ifFalse:
True»|
True»ifTrue:
True»ifFalse:
True»&
True»initialize
Number»negated
Number»
Number»+
Number»
Number»asNumber
Number»<
Number»
Number»decimalDigitLength
Number»
Number»-
Number»to:
Number»asString
Number»>
Base»reflect:
Class»new
Class»new:
Class»allocate
Class»allocate:
Block»value
Block»valueWithArgs:
Block»whileTrue:
UndefinedObject»asString
Point»asString
Point»initialize
Point»y:
Point»x:
Point»x
Point»y
False»ifTrue:
False»ifTrue:ifFalse:
False»asString
False»ifFalse:
False»|
False»&
False»not
False»initialize
Array»asString
Array»size
Array»indexAndValuesDo:
Array»at:put:
Array»do:
Array»indexOf:
Array»at:
AbstractMirror»baseObject
AbstractMirror»variables
AbstractMirror»at:put:
AbstractMirror»perform:withArguments:

AbstractMirror»setBaseObject:
AbstractMirror»obj:perform:withArguments:
AbstractMirror»obj:atIndex:put:
AbstractMirror»asString
AbstractMirror»isClass
AbstractMirror»classMirror
AbstractMirror»at:
AbstractMirror»baseObject:
AbstractMirror»obj:atIndex:
AbstractMirror»
obj:perform:withArguments:lookupClass:
AbstractMirror»setDirectClass:on:
AbstractMirror»baseClass:
AbstractMirror»perform:
AbstractMirror»directClassOn:
AbstractMirror»setClassMirror:
MirrorFactory»setDirectMirror:on:
MirrorFactory»newOn:
MirrorFactory»directClassOn:
MirrorFactory»on:
MirrorFactory»initialize
MirrorFactory»directMirrorOn:
MirrorFactory»setDirectClass:on:
Dict»values
Dict»at:
Dict»keysAndValuesDo:
Dict»remove:
Dict»add:
Dict»keyOf:
Dict»atKey:
Dict»at:put:
Dict»keys
Object»==
Object»print
Object»=
Object»yourself
Object»asList
Object»quit
Object»copy
Object»size
Object»initialize
Object»asString
List»at:
List»tail
List»=
List»at:put:
List»size
List»,
List»isEmpty
List»remove:
List»do:
List»indexAndValuesDo:
List»asList
List»indexOf:
List»head
List»asString
List»basicAddAll:
List»initialize
List»add:
String»,
String»byteAt:put:
String»copyReplaceFrom:to:with:
String»replaceFrom:to:with:startingAt:
String»print
String»size
String»at:
String»asString
String»at:put:
String»==

Tornado Result Extracts

This appendix lits the entry points and resulting code units we obtained from three different tailoring cases.

D.1 I/O App Extract

This section lists the methods extracted from a nurtured Hello World application using I/O. This case was tailored using an empty seed. The used entry point is the following:

```
1 FileStream startUp: true.
2 1 to: 10 do: [ :i | FileStream stdout nextPutAll: 'hello'; crlf ].
```

This list includes all methods installed from the Pharo base libraries and the simple Hello World application.

Array class»new:
ArrayedCollection»size
Association class»key:value:
Association»value:
Association»value
BlockClosure»on:do:
BlockClosure»repeat
BlockClosure»valueNoContextSwitch
ByteString class»compare:with:collated:
ByteString class»
findFirstInString:inSet:startingAt:
ByteString class»stringHash:initialHash:
ByteString»at:put:
ByteString»at:
ByteString»isByteString
ByteString»replaceFrom:to:with:startingAt:
ByteTextConverter class»
unicodeToByteTable
ByteTextConverter»nextPut:toStream:
ByteTextConverter»unicodeToByte:
Character class»cr
Character class»lf
Character class»value:
Character»=
Character»asInteger
Character»asciiValue
Character»charCode
Collection»detect:ifNone:
Dictionary»at:ifAbsent:
Dictionary»at:ifPresent:
Dictionary»at:put:
Dictionary»noCheckAdd:
Dictionary»scanFor:
FileStream class»newForStdio
FileStream class»new
FileStream class»standardIOStreamNamed:forWrite:
FileStream class»startUp:
FileStream class»stdioHandles
FileStream class»stdoutCharacter»isCharacter
FileStream class»voidStdioFiles
FileStream»collectionSpeciesStandard
FileStream»enableReadBufferingSmalltalkImage»vm
FileStream»isBinaryStandard
FileStream»next:putAll:startingAt:Standard
FileStream»nextPut:Standard
FileStream»openOnHandle:name:forWrite:Standard

FileStream»primWrite:from:startingAt:count:
GreekEnvironment class»supportedLanguages
HashTableSizes class»atLeast:
HashTableSizes class»sizes
HashedCollection class»newProto
HashedCollection»atNewIndex:put:
HashedCollection»findElementOrNil:
HashedCollection»fullCheck
HashedCollection»grow
HashedCollection»initialize:
Integer»asCharacter
JapaneseEnvironment class»
    supportedLanguages
KoreanEnvironment class»
    supportedLanguages
LanguageEnvironment class»
    currentPlatform
LanguageEnvironment class»
    defaultSystemConverter
LanguageEnvironment class»
    initKnownEnvironments
LanguageEnvironment class»
knownEnvironments
LanguageEnvironment class»
    localeID:
LanguageEnvironment»localeID:
Latin1Environment class»
    supportedLanguages
Latin2Environment class»
    supportedLanguages
Latin9Environment class»
    supportedLanguages
Latin9Environment class»
    systemConverterClass
Locale class»currentPlatform
Locale class»determineCurrentLocale
Locale»determineLocaleID
Locale»determineLocale
Locale»fetchISO2Language
Locale»languageEnvironment
Locale»localeID:
Locale»localeID
Locale»primCountry
Locale»primLanguage
LocaleID class»isoLanguage:isoCountry:
LocaleID class»isoLanguage:
LocaleID class»isoString:
LocaleID»=
LocaleID»hash
LocaleID»isoCountry
LocaleID»isoLanguage:isoCountry:
LocaleID»isoLanguage
LookupKey class»key:
LookupKey»key:
LookupKey»key
Magnitude»max:
MultiByteFileStream»
    basicNext:putAll:startingAt:
MultiByteFileStream»basicNextPut:
MultiByteFileStream»converter:
MultiByteFileStream»converter
MultiByteFileStream»
    installLineEndConventionInConverter
MultiByteFileStream»nextPut:
MultiByteFileStream»nextPutAll:
Number»negative
OSPlatform class»isWin32
OSPlatform class»platformName
Object»=
Object»at:put:Object»isCharacter
Object»at:
Object»class
Object»hash
Object»isInteger
Object»species
Object»~~
OrderedCollection class»arrayType
OrderedCollection class»new:
OrderedCollection class»new
OrderedCollection»add:
OrderedCollection»addLast:
OrderedCollection»at:
OrderedCollection»ensureBoundsFrom:to:
OrderedCollection»resetTo:
OrderedCollection»reset
OrderedCollection»setCollection:
OrderedCollection»size
PositionableStream class»on:
PositionableStream»isBinary
PositionableStream»on:
ProtoObject»basicIdentityHash
ProtoObject»flag:

ProtoObject»identityHash
ProtoObject»initialize
RussianEnvironment class»supportedLanguages
Semaphore»critical:
SequenceableCollection»copyFrom:to:
SequenceableCollection»copyUpTo:
SequenceableCollection»do:
SequenceableCollection»first:
SequenceableCollection»first
SequenceableCollection»identityIndexOf:ifAbsent:
SequenceableCollection»identityIndexOf:
SequenceableCollection»indexOf:ifAbsent:
SequenceableCollection»second
SequenceableCollection»writeStream
SimplifiedChineseEnvironment class»supportedLanguages
SmallInteger»bitXor:
SmallInteger»hash
Stream»basicNextPut:
String class»new:
String class»with:
String»=
String»compare:with:collated:
String»findDelimiters:startingAt:
String»findTokens:
String»hash
String»indexOf:startingAt:ifAbsent:
String»isString
String»skipDelimiters:startingAt:
TextConverter class»defaultSystemConverter
TextConverter class»initializeLatin1MapAndEncodings
TextConverter class»latin1Encodings
TextConverter class»latin1Map
TextConverter»initialize
TextConverter»installLineEndConvention:
TextConverter»nextPutAll:toStream:
TextConverter»nextPutByteString:toStream:
VirtualMachine class»getSystemAttribute:
WriteStream»contentsStandard
WriteStream»crlf
WriteStream»nextPut:
WriteStream»on:
WriteStream»reset

D.2 Seaside Web Application Entry Points

This section lists the entry points as used to tailor the Seaside web application with a full Pharo seed and an empty seed. The following code snippet corresponds with a full Pharo seed. It only consists in starting the web server as the base libraries are initialized and available in the seed.

```
1 ZnZincServerAdaptor startOn: 8888.
```

The following code snippet corresponds with an empty seed. This entry point includes also the initialization of the minimal runtime needed to do networking.

```
"We initialize some classes of the system"
SmalltalkImage initializeForTornado.
Symbol initializeForTornado.
Object initialize.
ExternalSemaphoreTable initialize.
Socket initialize.
Delay initialize.
Delay startUp: true.
Delay shutDown: true.
OSPlatform initialize.
DiskStore initialize.
FileStream initialize.
NetNameResolver initialize.
DateAndTime initialize.
ProcessorScheduler initialize.
WeakFinalizationList initialize.
UUIDGenerator initialize.
WeakArray initialize.
GRPharoRandomProvider initialize.
WASlime initialize.
UIManager basicDefault: DummyUIManager new.
ZnServer initialize.
WAServerManager initialize.
Smalltalk instVarNamed: 'session' put: Smalltalk newSessionObject.
Smalltalk startupImage: true snapshotWorked: true.

"Finally we start the web server"
ZnZincServerAdaptor startOn: 8888.
```

D.3 Seaside App A Extract

This section lists the methods extracted from the nurtured Web application when using an empty seed. This list includes all methods installed from Seaside framework, the Counter application and the base library of Pharo.

Array class»new:
Array»isSelfEvaluating
Array»printOn:
Array»replaceFrom:to:with:startingAt:
Array»shouldBePrintedAsLiteral
ArrayedCollection class»new:withAll:
ArrayedCollection class»new
ArrayedCollection class»with:with:with:
ArrayedCollection class»with:with:
ArrayedCollection class»with:
ArrayedCollection»mergeSortFrom:to:by:
ArrayedCollection»size
ArrayedCollection»sort:
Association class»key:value:
Association class»key:value:
Association»expireWeakKey
Association»expiredWeakKey
Association»key:WeakKey
Association»key:value:WeakKey
Association»keyWeakKey
Association»value:WeakKey
Association»value:
Association»valueWeakKey
Association»value
BlockClosure»argumentCount
BlockClosure»asContextWithSender:
BlockClosure»asContext
BlockClosure»assert
BlockClosure»cull:
BlockClosure»ensure:
BlockClosure»fixCallbackTemps
BlockClosure»forkAt:named:
BlockClosure»forkAt:
BlockClosure»ifCurtailed:

BlockClosure»ifError:
BlockClosure»isClosure
BlockClosure»newProcess
BlockClosure»numArgs
BlockClosure»numCopiedValues
BlockClosure»on:do:
BlockClosure»on:fork:
BlockClosure»outerContext
BlockClosure»renderOn:
BlockClosure»repeatWithGCIf:
BlockClosure»repeat
BlockClosure»startpc
BlockClosure»value:value:value:
BlockClosure»value:value:
BlockClosure»valueNoContextSwitch
BlockClosure»valueWithArguments:
BlockClosure»valueWithPossibleArguments:
ByteArray»asByteArray
ByteArray»replaceFrom:to:with:startingAt:
ByteString class»compare:with:collated:
ByteString class»findFirstInString:inSet:startingAt:
ByteString class»indexOfAscii:inString:startingAt:
ByteString class»stringHash:initialHash:
ByteString class»translate:from:to:table:
ByteString»asByteArray
ByteString»at:put:
ByteString»at:
ByteString»beginsWith:
ByteString»byteAt:put:
ByteString»byteSize
ByteString»
findSubstring:in:startingAt:matchTable:
ByteString»
findSubstringViaPrimitive:in:startingAt:matchTable:
ByteString»isByteString
ByteString»isOctetString
ByteString»replaceFrom:to:with:startingAt:
ByteSymbol class»stringHash:initialHash:
ByteSymbol»at:
ByteSymbol»
findSubstring:in:startingAt:matchTable:
ByteSymbol»isByteString
ByteSymbol»privateAt:put:
ByteSymbol»species
ByteSymbol»string:
CNGBTextConverter class»encodingNames
CP1250TextConverter class»encodingNames
CP1253TextConverter class»encodingNames
ChangesLog class»default
ChangesLog»recordStartupStamp
Character class»codePoint:
Character class»cr
Character class»lf
Character class»space
Character class»value:
Character»=
Character»asCharacter
Character»asInteger
Character»asSymbol
Character»asUppercase
Character»asciiValue
Character»charCode
Character»characterSet
Character»codePoint
Character»digitValue
Character»greaseInteger
Character»isAlphaNumeric
Character»isCharacter
Character»isDigit
Character»isLetter
Character»isOctetCharacter
Character»isSeparator
Character»isVowel
Character»leadingChar
Character»to:
Collection class»withAll:
Collection»addAll:
Collection»allSatisfy:
Collection»anySatisfy:
Collection»asArray
Collection»detect:ifNone:
Collection»emptyCheck
Collection»inject:into:
Collection»isCollection
Collection»isEmptyOrNil
Collection»isEmpty
Collection»noneSatisfy:
Collection»notEmpty
Collection»printElementsOn:
Collection»printNameOn:
Collection»printOn:
Collection»removeAll:

Collection»removeAllFoundIn:
Collection»sorted:
Collection»sorted
CommandLineUIManager class»replacing:
CommandLineUIManager»initialize
CommandLineUIManager»replacing:
CompiledMethod»frameSize
CompiledMethod»header
CompiledMethod»initialPC
CompiledMethod»isPrimitive
CompiledMethod»numLiterals
CompiledMethod»numTemps
CompiledMethod»objectAt:
CompiledMethod»primitive
CompoundTextConverter class»encodingNames
ContextPart class»contextEnsure:
ContextPart class»contextOn:do:
ContextPart class»newForMethod:
ContextPart class»theReturnMethod
ContextPart»activateMethod:withArgs:receiver:class:
ContextPart»activateReturn:value:
ContextPart»at:put:
ContextPart»at:
ContextPart»bottomContext
ContextPart»cut:
ContextPart»doPop
ContextPart»exceptionClass
ContextPart»exceptionHandlerBlock
ContextPart»exceptionHandlerIsActive:
ContextPart»exceptionHandlerIsActive
ContextPart»findContextSuchThat:
ContextPart»findNextHandlerContextStarting
ContextPart»findNextUnwindContextUpTo:
ContextPart»handleSignal:
ContextPart»insertSender:
ContextPart»isDead
ContextPart»jump
ContextPart»methodReturnTop
ContextPart»nextHandlerContext
ContextPart»pop
ContextPart»privSender:
ContextPart»push:
ContextPart»pushTemporaryVariable:
ContextPart»releaseTo:
ContextPart»resume:through:
ContextPart»resume:
ContextPart»return:from:
ContextPart»return:through:
ContextPart»return:
ContextPart»runUntilErrorOrReturnFrom:
ContextPart»sender
ContextPart»singleRelease
ContextPart»stackp:
ContextPart»stepToCallee
ContextPart»step
ContextPart»terminateTo:
Date class»fromSeconds:
Date class»fromString:
Date class»readFrom:
Date class»starting:
Date class»year:month:day:
Date»dayMonthYearDo:
Date»monthIndex
Date»printOn:format:
Date»printOn:
DateAndTime class»clock
DateAndTime class»fromSeconds:offset:
DateAndTime class»fromSeconds:
DateAndTime class»fromString:
DateAndTime class»initializeOffsets
DateAndTime class»initialize
DateAndTime class»localOffset
DateAndTime class»localTimeZone
DateAndTime class»
milliSecondsSinceMidnight
DateAndTime class»millisecondClockValue
DateAndTime class»now
DateAndTime class»readFrom:
DateAndTime class»
readOptionalSeparatorFrom:
DateAndTime class»
readTimezoneOffsetFrom:
DateAndTime class»
readTwoDigitIntegerFrom:
DateAndTime class»startUp:
DateAndTime class»todayAtMilliSeconds:
DateAndTime class»waitForOffsets
DateAndTime class»
year:month:day:hour:minute:offset:
DateAndTime class»
year:month:day:hour:minute:
second:nanoSecond:offset:

DateAndTime class»
year:month:day:hour:minute:
second:offset:
DateAndTime class»
year:month:day:hour:minute:second:
DateAndTime class»
year:month:day:hour:minute:
DateAndTime class»
year:month:day:offset:
DateAndTime class»
year:month:day:
DateAndTime»<
DateAndTime»asDateAndTime
DateAndTime»asTimeStamp
DateAndTime»asUTC
DateAndTime»dayMonthYearDo:
DateAndTime»dayOfMonth
DateAndTime»dayOfWeekAbbreviation
DateAndTime»dayOfWeekName
DateAndTime»hour24
DateAndTime»hour
DateAndTime»julianDayNumber
DateAndTime»julianDayOffset
DateAndTime»localSeconds
DateAndTime»minute
DateAndTime»monthAbbreviation
DateAndTime»monthName
DateAndTime»month
DateAndTime»normalize:ticks:base:
DateAndTime»offset:DateAndTime»midnight
DateAndTime»offset
DateAndTime»printHMSOn:
DateAndTime»second
DateAndTime»setJdn:seconds:nano:offset:
DateAndTime»ticks:offset:DateAndTime»dayOfWeek
DateAndTime»ticks
DateAndTime»year
Delay class»forMilliseconds:
Delay class»handleTimerEvent
Delay class»initialize
Delay class»primSignal:atMilliseconds:
Delay class»restoreResumptionTimes
Delay class»runTimerEventLoop
Delay class»saveResumptionTimes
Delay class»scheduleDelay:
Delay class»shutDown
Delay class»startTimerEventLoop
Delay class»startUp
Delay class»stopTimerEventLoop
Delay class»unscheduleDelay:
Delay»beingWaitedOn:
Delay»beingWaitedOn
Delay»delayDuration
Delay»resumptionTime:
Delay»resumptionTime
Delay»schedule
Delay»setDelay:forSemaphore:
Delay»signalWaitingProcess
Delay»unschedule
Delay»wait
DelayWaitTimeout»isExpired
DelayWaitTimeout»setDelay:forSemaphore:
DelayWaitTimeout»signalWaitingProcess
DelayWaitTimeout»wait
Dictionary»addAll:GRSmall
Dictionary»associationsDo:
Dictionary»at:ifAbsent:GRSmall
Dictionary»at:ifAbsent:Small
Dictionary»at:ifAbsent:
Dictionary»at:ifAbsentPut:GRSmall
Dictionary»at:ifAbsentPut:
Dictionary»at:ifPresent:GRSmall
Dictionary»at:ifPresent:
Dictionary»at:put:GRSmall
Dictionary»at:put:Small
Dictionary»at:put:
Dictionary»at:
Dictionary»do:
Dictionary»findIndexFor:GRSmall
Dictionary»findIndexForKey:Small
Dictionary»fixCollisionsFrom:
Dictionary»includesKey:GRSmall
Dictionary»includesKey:
Dictionary»initialize:GRSmall
Dictionary»initializeSmall
Dictionary»isEmptyGRSmall
Dictionary»keyAtValue:ifAbsent:
Dictionary»keysAndValuesDo:
Dictionary»keysAndValuesDo:
Dictionary»keysAndValuesDo:
Dictionary»keysDo:GRSmall
Dictionary»noCheckAdd:

Dictionary»postCopyGRSmall
Dictionary»privateAt:put:GRSmall
Dictionary»privateAt:put:Small
Dictionary»rehash
Dictionary»removeKey:ifAbsent:
Dictionary»scanFor:
Dictionary»seasideRequestFieldsGRSmall
Dictionary»sizeGRSmall
Dictionary»valuesDo:
DiskStore class»checkVMVersion
DiskStore class»initialize
DiskStore class»reset
DiskStore class»shutDown:
DiskStore class»startUp:
DiskStore class»useFilePlugin
Duration class»days:hours:minutes:
seconds:nanoSeconds:
Duration class»days:hours:minutes:
seconds:
Duration class»days:
Duration class»nanoSeconds:
Duration class»seconds:nanoSeconds:
Duration class»seconds:
Duration»+
Duration»-
Duration»asDuration
Duration»asMilliSeconds
Duration»asNanoSeconds
Duration»asSeconds
Duration»days
Duration»isZero
Duration»negated
Duration»seconds:nanoSeconds:
Duration»ticks
DynamicVariable class»value:during:
DynamicVariable»value:during:
EUCJPTextConverter class»encodingNames
EUCKRTextConverter class»encodingNames
EncodedCharSet class»charsetAt:
EventManager class»actionMaps
EventManagerclass»flushEvents
Exception class»,
Exception class»handles:
Exception class»signal:
Exception class»signal
Exception»description
Exception»isResumable
Exception»messageText:
Exception»messageText
Exception»printOn:
Exception»privHandlerContext:
Exception»receiver
Exception»resume:
Exception»resumeUnchecked:
Exception»signal:
Exception»signalerContext
Exception»signal
ExceptionSet»,
ExceptionSet»add:
ExceptionSet»handles:
ExceptionSet»initialize
ExtendedNumberParser»allowPlusSign
ExtendedNumberParser»nextNumber
ExternalSemaphoreTable class»clearExternalObjects
ExternalSemaphoreTable class»collectionBasedOn:withRoomFor:
ExternalSemaphoreTable class»freedSlotsIn:ratherThanIncreaseSizeTo:
ExternalSemaphoreTable class»initialize
ExternalSemaphoreTable class»registerExternalObject:
ExternalSemaphoreTable class»safelyRegisterExternalObject:
ExternalSemaphoreTable class»safelyUnregisterExternalObject:
ExternalSemaphoreTable class»slotFor:in:
ExternalSemaphoreTable class»unprotectedExternalObjects:
ExternalSemaphoreTable class»unprotectedExternalObjects
ExternalSemaphoreTable class»unregisterExternalObject:
False»not
False»|
FilePath class»pathName:isEncoded:
FilePath»asSqueakPathName
FilePath»pathName:isEncoded:
FilePath»pathName
FileStream class»flushAndVoidStdioFiles
FileStream class»initialize
FileStream class»shutDown:
FileStream class»startUp:
FileStream class»stdioHandles
FileStream class»voidStdioFiles
Float class»precision
Float»adaptToInteger:andSend:
Float»asFloat
Float»isInfinite
Float»timesTwoPower:
Float»truncated

Fraction class»numerator:denominator:
Fraction»>=
Fraction»reduced
Fraction»setNumerator:denominator:
Fraction»truncated
GRCodecStream class»on:
GRCodecStream»atEnd
GRCodecStream»initializeOn:
GRNullCodec class»codecName
GRNullCodec class»supportsEncoding:
GRNullCodec»encoderFor:
GRNullCodec»url
GRNullCodecStream»nextPutAll:
GRObject class»new
GRObject»initialize
GROrderedMultiMap»allAt:
GROrderedMultiMap»at:add:
GRPharoConverterCodecStream class»on:converter:
GRPharoConverterCodecStream»contents
GRPharoConverterCodecStream»initializeOn:converter:
GRPharoConverterCodecStream»size
GRPharoGenericCodec class»supportedEncodingNames
GRPharoGenericCodec class»supportsEncoding:
GRPharoLatin1Codec class»supportedEncodingNames
GRPharoLatin1Codec class»supportsEncoding:
GRPharoPlatform»addToShutDownList:
GRPharoPlatform»addToStartUpList:
GRPharoPlatform»includesUnsafeUrlCharacter:
GRPharoPlatform»includesUnsafeXmlCharacter:
GRPharoPlatform»semaphoreClass
GRPharoRandomProvider class»initialize
GRPharoRandomProvider class»nextInt:
GRPharoRandomProvider class»randomClass
GRPharoRandomProvider class»startUp
GRPharoUtf8Codec class»basicForEncoding:
GRPharoUtf8Codec class»supportsEncoding:
GRPharoUtf8Codec»decode:
GRPharoUtf8Codec»decoderFor:
GRPharoUtf8Codec»encoderFor:
GRPharoUtf8Codec»name
GRPharoUtf8Codec»url
GRPharoUtf8CodecStream»encodeFast:
GRPharoUtf8CodecStream»next:
GRPharoUtf8CodecStream»nextPut:
GRPharoUtf8CodecStream»nextPutAll:
GRPlatform class»current
GRPlatform»reducedConflictDictionary
GRSmallDictionary class»new
GRSmallDictionary class»new:
GRSmallDictionary class»new
GRSmallDictionary class»withAll:
HashTableSizes class»atLeast:
HashTableSizes class»sizes
HashedCollection class»new:
HashedCollection class»new
HashedCollection class»sizeFor:
HashedCollection»array
HashedCollection»atNewIndex:put:
HashedCollection»findElementOrNil:
HashedCollection»fullCheck
HashedCollection»grow
HashedCollection»initialize:
HashedCollection»size
Heap class»withAll:sortBlock:
Heap»add:
Heap»do:
Heap»downHeapSingle:
Heap»growHeap»reSort
Heap»growSize
Heap»growTo:
Heap»isEmpty
Heap»privateRemoveAt:
Heap»remove:ifAbsent:
Heap»removeFirst
Heap»setCollection:tally:
Heap»size
Heap»sortBlock:
Heap»sorts:before:
Heap»upHeap:
Heap»updateObjectIndex:
ISO885915TextConverter class»encodingNames
ISO88592TextConverter class»encodingNames

IdentitySet»scanFor:
InstructionStream»interpretExtension:in:for:
InstructionStream»interpretNextInstructionFor:
Integer class»readFrom:base:
Integer class»readFrom:
Integer»*
Integer»+
Integer»<
Integer»=
Integer»»
Integer»asCharacter
Integer»asInteger
Integer»copyto:
Integer»denominator
Integer»digitCompare:
Integer»digitDiv:neg:
Integer»digitMultiply:neg:
Integer»digitSubtract:
Integer»floor
Integer»isFraction
Integer»isInteger
Integer»noMask:
Integer»normalize
Integer»numerator
Integer»printOn:
Integer»printStringBase:length:padded:
Integer»printStringLength:padded:
Integer»quo:
Integer»rounded
Integer»timesRepeat:
Integer»truncated
Interval class»from:to:by:
Interval class»new
Interval»collect:
Interval»setFrom:to:by:
Interval»size
Interval»species
KOI8RTextConverter class»encodingNames
LanguageEnvironment class»
    defaultFileNameConverter
LargeNegativeInteger»negative
LargeNegativeInteger»normalize
LargePositiveInteger»*
LargePositiveInteger»+
LargePositiveInteger»-
LargePositiveInteger»//
LargePositiveInteger»<
LargePositiveInteger»asFloat
LargePositiveInteger»digitAt:
LargePositiveInteger»digitLength
LargePositiveInteger»highBitOfMagnitude
LargePositiveInteger»negated
LargePositiveInteger»negative
LargePositiveInteger»normalize
LargePositiveInteger»quo:
LimitedWriteStream»nextPut:
LimitedWriteStream»nextPutAll:
LimitedWriteStream»setLimit:limitBlock:
LookupKey class»key:
LookupKey»key:
LookupKey»key
MacOSXPlatform class»isActivePlatform
MacRomanTextConverter class»
    encodingNames
Magnitude»>
Magnitude»between:and:
Magnitude»max:
Magnitude»min:
MethodContext class»
    sender:receiver:method:arguments:
MethodContext»aboutToReturn:through:
MethodContext»blockReturnTop
MethodContext»closure
MethodContext»contextTag
MethodContext»methodReturnContext
MethodContext»method
MethodContext»privRefresh
MethodContext»receiver
MethodContext»setSender:receiver:method:arguments:
MethodContext»setSender:receiver:method:closure:startpc:
MethodContext»tempAt:put:
MethodContext»tempAt:
MethodDictionary»at:ifAbsent:
MethodDictionary»includesKey:
MethodDictionary»scanFor:
Month class»daysInMonth:forYear:
Month class»indexOfMonth:
Month class»nameOfMonth:
Mutex»critical:
Mutex»initialize
NetNameResolver class»initializeNetwork
NetNameResolver class»initialize

NetNameResolver class»primNameResolverStatus
NetNameResolver class»resolverStatus
Number class»one
Number class»readFrom:
Number»//
Number»%
Number»\\
Number»abs
Number»asDuration
Number»floor
Number»fractionPart
Number»integerPart
Number»isNumber
Number»isZero
Number»negated
Number»negative
Number»quo:
Number»raisedToInteger:
Number»rem:
Number»strictlyPositive
Number»to:
NumberParser class»on:
NumberParser»nextElementaryLargeIntegerBase:
NumberParser»nextIntegerBase:
NumberParser»nextUnsignedIntegerBase:
NumberParser»nextUnsignedIntegerOrNilBase:
NumberParser»on:
NumberParser»peekSignIsMinus
NumberParser»readExponent
OSPlatform class»determineActivePlatformStartingWith:
OSPlatform class»initialize
OSPlatform class»isMacOS
OSPlatform class»platformName
OSPlatform class»shutDown:
OSPlatform class»startUp:ISO88597
OSPlatform class»version
OSPlatform»shutDown:
OSPlatform»startUp:
Object class»flushDependents
Object class»flushEvents
Object class»initializeDependentsFields
Object class»initialize
Object class»newFrom:
Object»=
Object»actAsExecutor
Object»as:
Object»asSetElement
Object»asString
Object»assert:
Object»at:put:
Object»at:
Object»basicAt:put:
Object»basicAt:
Object»basicSize
Object»breakDependents
Object»class
Object»copyFrom:
Object»copySameFrom:
Object»copy
Object»enclosedSetElement
Object»encodeOn:
Object»executor
Object»greaseString
Object»hash
Object»instVarAt:put:
Object»instVarAt:
Object»instVarNamed:put:
Object»isArray
Object»isCharacter
Object»isInteger
Object»isKindOf:
Object»isLiteral
Object»isMemberOf:
Object»isSelfEvaluating
Object»isString
Object»isSymbol
Object»myDependents:
Object»notNil
Object»perform:with:
Object»perform:withArguments:inSuperclass:
Object»perform:withArguments:
Object»perform:
Object»postCopy
Object»printOn:
Object»printStringLimitedTo:
Object»printString
Object»readFromString:
Object»renderOn:
Object»respondsTo:
Object»restoreFromSnapshot:
Object»shallowCopy
Object»shouldBePrintedAsLiteral

Object»snapshotCopy
Object»species
Object»split:do:
Object»split:
Object»value
Object»yourself
Object»~=
OrderedCollection class»arrayType
OrderedCollection class»new:
OrderedCollection class»new
OrderedCollection»add:
OrderedCollection»addAll:
OrderedCollection»addAllLast:
OrderedCollection»addFirst:
OrderedCollection»addLast:
OrderedCollection»asArray
OrderedCollection»at:
OrderedCollection»collect:
OrderedCollection»copyEmpty
OrderedCollection»do:
OrderedCollection»ensureBoundsFrom:to:
OrderedCollection»growAtLast
OrderedCollection»makeRoomAtFirst
OrderedCollection»makeRoomAtLast
OrderedCollection»postCopy
OrderedCollection»remove:ifAbsent:
OrderedCollection»removeFirst
OrderedCollection»removeIndex:
OrderedCollection»resetTo:
OrderedCollection»reset
OrderedCollection»reverseDo:
OrderedCollection»select:
OrderedCollection»setCollection:
OrderedCollection»size
PositionableStream class»on:
PositionableStream»atEnd
PositionableStream»isBinary
PositionableStream»isEmpty
PositionableStream»on:
PositionableStream»originalContents
PositionableStream»peekFor:
PositionableStream»peek
PositionableStream»position:
PositionableStream»position
PositionableStream»reset
PositionableStream»skip:
PositionableStream»skipSeparators
PositionableStream»skipTo:
Process class»forContext:priority:
Process»activateReturn:value:
Process»calleeOf:
Process»complete:
Process»isActiveProcess
Process»name:
Process»popTo:
Process»primitiveResume
Process»priority:
Process»priority
Process»psValueAt:put:
Process»psValueAt:
Process»resume
Process»return:value:
Process»suspendedContext:
Process»suspendingList
Process»suspend
Process»terminate
ProcessLocalVariable class»value:
ProcessLocalVariable»value:
ProcessSpecificVariable class»soleInstance
ProcessSpecificVariable class»value
ProcessSpecificVariable»default
ProcessSpecificVariable»value
ProcessorScheduler class»idleProcess
ProcessorScheduler class»initialize
ProcessorScheduler class»
relinquishProcessorForMicroseconds:
ProcessorScheduler class»startUp
ProcessorScheduler»activePriority
ProcessorScheduler»activeProcess
ProcessorScheduler»highIOPriority
ProcessorScheduler»highestPriority
ProcessorScheduler»lowIOPriority
ProcessorScheduler»lowestPriority
ProcessorScheduler»terminateActive
ProcessorScheduler»timingPriority
ProcessorScheduler»userInterruptPriority
ProtoObject»basicIdentityHash
ProtoObject»flag:
ProtoObject»identityHash
ProtoObject»initialize
ProtoObject»instVarsInclude:
ProtoObject»isNil

ProtoObject»pointsTo:
ProtoObject»~~
Random»initialize
Random»nextInt:
Random»nextValue
Random»next
ReadStream class»on:from:to:
ReadStream»next
ReadStream»on:from:to:
ReadStream»upTo:
ReadStream»upToEnd
Semaphore class»forMutualExclusion
Semaphore class»new
Semaphore»critical:ifError:
Semaphore»critical:
Semaphore»initSignals
Semaphore»signal
Semaphore»waitTimeoutMSecs:
Semaphore»wait
SequenceableCollection class»new:streamContents:
SequenceableCollection class»ofSize:
SequenceableCollection class»streamContents:limitedTo:
SequenceableCollection class»streamContents:
SequenceableCollection class»streamSpecies
SequenceableCollection»,
SequenceableCollection»allButFirst:
SequenceableCollection»at:ifAbsent:
SequenceableCollection»atAllPut:
SequenceableCollection»copyAfter:
SequenceableCollection»copyFrom:to:
SequenceableCollection»copyReplaceFrom:to:with:
SequenceableCollection»copyUpTo:
SequenceableCollection»do:separatedBy:
SequenceableCollection»do:
SequenceableCollection»doWithIndex:
SequenceableCollection»first:
SequenceableCollection»first
SequenceableCollection»from:to:put:
SequenceableCollection»grownBy:
SequenceableCollection»includes:
SequenceableCollection»indexOf:ifAbsent:
SequenceableCollection»indexOf:startingAt:ifAbsent:
SequenceableCollection»indexOf:
SequenceableCollection»indexOfSubCollection:startingAt:
SequenceableCollection»keysAndValuesDo:
SequenceableCollection»last
SequenceableCollection»readStream
SequenceableCollection»replaceFrom:to:with:
SequenceableCollection»reverseDo:
SequenceableCollection»second
SequenceableCollection»select:
SequenceableCollection»split:indicesDo:
SequenceableCollection»splitOn:
SequenceableCollection»swap:with:
SequenceableCollection»withIndexDo:
SequenceableCollection»writeStream
Set class»new
Set»add:
Set»do:
Set»fixCollisionsFrom:
Set»grow
Set»includes:
Set»noCheckAdd:
Set»remove:ifAbsent:
Set»scanFor:
SmallInteger class»maxValCP1252
SmallInteger»*
SmallInteger»/
SmallInteger»asFloat
SmallInteger»decimalDigitLength
SmallInteger»gcd:
SmallInteger»hashMultiply
SmallInteger»hash
SmallInteger»highBitOfPositiveReceiver
SmallInteger»highBit
SmallInteger»identityHash
SmallInteger»isLarge
SmallInteger»numberOfDigitsInBase:
SmallInteger»printOn:base:length:padded:
SmallInteger»printOn:base:
SmallInteger»printString
SmallInteger»quo:
SmalltalkImage class»initializeForTornado
SmalltalkImage»addDeferredStartupAction:
SmalltalkImage»addToShutDownList:
SmalltalkImage»addToStartUpList:
SmalltalkImage»at:ifAbsent:
SmalltalkImage»at:
SmalltalkImage»clearExternalObjects
SmalltalkImage»executeDeferredStartupActions:
SmalltalkImage»garbageCollectMost

SmalltalkImage»imagePath
SmalltalkImage»includesKey:
SmalltalkImage»installLowSpaceWatcher
SmalltalkImage»isHeadless
SmalltalkImage»isInteractive
SmalltalkImage»
logStartUpErrorDuring:into:tryDebugger:
SmalltalkImage»lowSpaceThreshold
SmalltalkImage»lowSpaceWatcher
SmalltalkImage»newSessionObject
SmalltalkImage»primImagePath
SmalltalkImage»primSignalAtBytesLeft:
SmalltalkImage»
primitiveGetSpecialObjectsArray
SmalltalkImage»processShutDownList:
SmalltalkImage»processStartUpList:
SmalltalkImage»recordStartupStamp
SmalltalkImage»registerExternalObject:
SmalltalkImage»send:toClassesNamedIn:with
SmalltalkImage»shutDownImage:
SmalltalkImage»specialObjectsArray
SmalltalkImage»
startupImage:snapshotWorked:
SmalltalkImage»unregisterExternalObject:
SmalltalkImage»vm
SmalltalkImage»wordSize
Socket class»acceptFrom:
Socket class»initializeNetwork
Socket class»initialize
Socket class»newTCP
Socket class»register:
Socket class»registry
Socket class»standardTimeout
Socket class»unregister:
Socket»acceptFrom:
Socket»accept
Socket»closeAndDestroy:
Socket»closeAndDestroy
Socket»close
Socket»dataAvailable
Socket»destroy
Socket»initialize:
Socket»isConnected
Socket»isOtherEndClosed
Socket»isValid
Socket»listenOn:backlogSize:
Socket»
primAcceptFrom:
receiveBufferSize:
sendBufSize:
semaIndex:
readSemaIndex:
writeSemaIndex:
Socket»primSocket:receiveDataInto:startingAt:count:
Socket»primSocket:sendData:startIndex:count:
Socket»primSocket:setOption:value:
Socket»primSocketConnectionStatus:
Socket»primSocketDestroy:
Socket»primSocketReceiveDataAvailable:
Socket»primSocketRemoteAddress:
Socket»primSocketSendDone:
Socket»readSemaphore
Socket»register
Socket»remoteAddress
Socket»sendSomeData:startIndex:count:for:
Socket»sendSomeData:startIndex:count:
Socket»setOption:value:
Socket»socketHandle
Socket»unregister
Socket»waitForAcceptFor:
Socket»waitForConnectionFor:ifTimedOut:
Socket»waitForDataFor:ifClosed:ifTimedOut:
Socket»waitForDataFor:
Socket»waitForDisconnectionFor:
Socket»waitForSendDoneFor:
SparseLargeTable»at:
SparseLargeTable»noCheckAt:
SparseLargeTable»pvtCheckIndex:
SparseLargeTable»size
SqNumberParser»allowPlusSign
SqNumberParser»makeIntegerOrScaledInteger
SqNumberParser»readScale
Stream»basicNext
Stream»nextPutAll:
Stream»print:
String class»crlf
String class»empty
String class»new:
String:startingAt:
String»=
String»asDateAndTime
String»asDate

String»asLowercase
String»asNumber
String»asString
String»asSymbol
String»asUppercase
String»asZnMimeType
String»asZnUrl
String»compare:caseSensitive:
String»compare:with:collated:
String»convertFromWithConverter:
String»encodeOn:
String»findString:startingAt:caseSensitive:
String»findString:
String»find
String»greaseInteger
String»hash
String»includesSubstring:
String»indexOf:startingAt:ifAbsent:
String»indexOf:startingAt:
String»indexOf:
String»indexOfSubCollection:startingAt:ifAbsent:
String»indexOfSubCollection:
String»isString
String»isWideString
String»match:
String»putOn:
String»renderOn:
String»sameAs:
String»seasideMimeType
String»startingAt:match:startingAt:
String»subStrings:
String»translateFrom:to:table:
String»translateToLowercase
String»translateToUppercase
String»translateWith:
String»trimBoth:
String»trimBoth
String»trimLeft:right:
Symbol class»initializeForTornado
Symbol class»intern:
Symbol class»internCharacter:
Symbol class»lookup:
Symbol class»shutDown:
Symbol»=
Symbol»asString
TextConverter class»allEncodingNames
TextConverter class»encodingNames
TextConverter class»encodingNames
TextConverter class»encodingNames
TextConverter class»encodingNames
TextConverter class»encodingNames
TextConverter class»encodingNames
TextConverter class»latin1Encodings
TextConverter class»latin1
GRCodec class»forEncoding:
TextConverter»initialize
TextConverter»nextFromStream:UTF8
Time class»dateAndTimeFromSeconds:
Time class»dateAndTimeNow
Time class»fromSeconds:
Time class»hour:minute:second:nanoSecond:
Time class»millisecondClockValue
Time class»milliseconds:since:
Time class»millisecondsSince:
Time class»primSecondsClock
Time class»readFrom:
Time class»seconds:nanoSeconds:
Time class»secondsWhenClockTicks
Time class»totalSeconds
Time»hour24
Time»hour
Time»minute
Time»nanoSecond
Time»print24:showSeconds:on:
Time»printOn:
Time»seconds
Time»second
Time»ticks:
TimeZone»offset
Timespan class»starting:duration:
Timespan»<
Timespan»dayOfMonth
Timespan»duration:
Timespan»month
Timespan»start:
Timespan»start
Timespan»year
True»ifFalse:
True»not
True»|
UIManager class»basicDefault:

UIManager class»default:
WAUnescapedDocument»initializeWithStream:encoding:
UIManager class»default
UIManager»activate
UIManager»beDefault
UIManager»boot:during:
UIManager»deactivate
UIManager»onSnapshot:
UTF16TextConverter class»encodingNames
UTF8DecomposedTextConverter class»encodingNames
UUIDGenerator class»initialize
UUIDGenerator class»startUp
UndefinedObject»encodeOn:
UndefinedObject»isNil
UndefinedObject»notNil
UndefinedObject»seasideUrl
UndefinedObject»shallowCopy
Unicode class»isDigit:
Unicode class»isLetter:
Unicode class»toUppercase:
VirtualMachine class»allocationsBetweenGC:
VirtualMachine class»getSystemAttribute:
VirtualMachine class»interpreterClass
VirtualMachine class»
interpreterSourceDate
VirtualMachine class»
interpreterSourceVersion
VirtualMachine class»isPharoVM
VirtualMachine class»isRunningCogit
VirtualMachine class»
maxExternalSemaphores
VirtualMachine class»parameterAt:put:
VirtualMachine class»parameterAt:
VirtualMachine class»setGCParameters
VirtualMachine class»tenuringThreshold:
VirtualMachine class»version
VirtualMachine class»wordSize
WAAccessIntervalReapingStrategy»
defaultConfiguration
WAAccessIntervalReapingStrategy»
initialize
WAAccessIntervalReapingStrategy»
interval
WAAccessIntervalReapingStrategy»
reap
WAAccessIntervalReapingStrategy»
stored:key:
WAActionCallback»block:
WAActionCallback»evaluateWithArgument:
WAActionCallback»isEnabledFor:
WAActionCallback»signalRenderNotification
WAActionPhaseContinuation»continue
WAActionPhaseContinuation»handleRequest
WAActionPhaseContinuation»renderContext:
WAActionPhaseContinuation»renderContext
WAActionPhaseContinuation»runCallbacks
WAActionPhaseContinuation»shouldRedirect
WAAdmin class»defaultServerManager
WAAdmin class»serverAdaptors
WAAnchorTag»callback:
WAAnchorTag»tag
WAAnchorTag»url
WAAnchorTag»with:
WAApplication»contentType
WAApplication»doesHandlerSupportCookies:
WAApplication»handleDefault:
WAApplication»handleFiltered:
WAApplication»isApplication
WAApplication»isImplemented:
WAApplication»keyField
WAApplication»libraries
WAApplication»mainClass
WAApplication»mimeType
WAApplication»newSession
WAApplication»resourceBaseUrl
WAApplication»sessionClass
WAApplicationConfiguration»parents
WAAttributeSearchContext class»key:target:
WAAttributeSearchContext»at:ifPresent:
WAAttributeSearchContext»at:put:
WAAttributeSearchContext»attribute
WAAttributeSearchContext»cachedValues
WAAttributeSearchContext»findAttributeAndSelectAncestorsOf:
WAAttributeSearchContext»initializeWithKey:
WAAttributeSearchContext»isAttributeInheritedOn:
WAAttributeSearchContext»isAttributeLocalOn:
WAAttributeSearchContext»key
WABrush»initialize
WABrush»parent
WABrush»setParent:canvas:
WABrush»with:
WABufferedResponse class»on:

WABufferedResponse»contents
WABufferedResponse»destroy
WABufferedResponse»initializeOn:
WABufferedResponse»stream
WACache»at:ifAbsent:
WACache»expiryPolicy
WACache»initializeCollections
WACache»initializeMutex
WACache»initialize
WACache»keyAtValue:ifAbsent:
WACache»keyAtValue:
WACache»keySize
WACache»missStrategy
WACache»notifyRemoved:key:
WACache»notifyRetrieved:key:
WACache»notifyStored:key:
WACache»pluginsDo:
WACache»reapingStrategy
WACache»reap
WACache»removalAction
WACache»setExpiryPolicy:
WACache»setMissStrategy:
WACache»setReapingStrategy:
WACache»setRemovalAction:
WACache»store:
WACacheCapacityConfiguration»describeOn:
WACachePlugin»configuration
WACachePlugin»defaultConfiguration
WACachePlugin»initialize
WACachePlugin»removed:key:
WACachePlugin»retrieved:key:
WACachePlugin»setCache:
WACachePlugin»stored:key:
WACacheReapingStrategy»reap
WACallback class»on:
WACallback»convertKey:
WACallback»evaluateWithFieldValues:
WACallback»key
WACallback»setKey:callbacks:
WACallback»valueForField:
WACallbackRegistry»advanceKey
WACallbackRegistry»handle:
WACallbackRegistry»increaseKey
WACallbackRegistry»initialize
WACallbackRegistry»nextKey
WACallbackRegistry»store:
WACanvas»brush:
WACanvas»flush
WACanvas»nest:
WACanvas»render:
WACanvas»text:
WAComponent»accept:
WAComponent»acceptDecorated:
WAComponent»decoration
WAComponent»initialize
WAComponent»updateStates:
WAConfigurationDescription»add:to:
WAConfigurationDescription»addAttribute:
WAConfigurationDescription»attributes
WAConfigurationDescription»expressions
WAConfigurationDescription»initialize
WAConfigurationDescription»integer:
WAConfiguredRequestFilter»configuration
WACounter»count:
WACounter»decrease
WACounter»increase
WACounter»initialize
WACounter»renderContentOn:
WACounter»states
WADefaultScriptGenerator»close:on:
WADefaultScriptGenerator»open:on:
WADevelopmentConfiguration»parents
WADispatcher class»default
WADispatcher»handleFiltered:named:
WADispatcher»handleFiltered:
WADispatcher»handlerAt:ifAbsent:
WADispatcher»handlerAt:with:
WADispatcher»handlers
WADispatcher»nameOfHandler:
WADispatcher»urlFor:
WADocument class»on:codec:
WADocument class»on:
WADocument»closeWADocument»destroy
WADocument»initializeWithStream:codec:
WADocument»nextPut:
WADocument»nextPutAll:
WADocument»open:
WADynamicVariable class»use:during:
WADynamicVariable class»value
WAEncoder class»on:table:
WAEncoder class»on:
WAEncoder»initializeOn:table:

WAEncoder»nextPut:
WAErrorHandler class»exceptionSelector
WAExampleComponent»rendererClass
WAExceptionFilter»exceptionHandler
WAExceptionFilter»handleFiltered:
WAExceptionHandler class»context:
WAExceptionHandler class»exceptionSelector:
WAExceptionHandler class»handleExceptionsDuring:context:
WAExceptionHandler class»handles:
WAExceptionHandler»handleExceptionsDuring:
WAExceptionHandler»handles:
WAExceptionHandler»initializeWithContext:
WAHeaderFields»checkValue:
WAHeaderFields»privateAt:put:
WAHeadingTag»initialize
WAHeadingTag»level1
WAHeadingTag»level
WAHeadingTag»tag
WAHtmlAttributes»encodeOn:
WAHtmlAttributes»privateAt:put:
WAHtmlCanvas»anchor
WAHtmlCanvas»heading:
WAHtmlCanvas»heading
WAHtmlCanvas»spaceEntity
WAHtmlDocument»scriptGenerator:
WAHtmlDocument»scriptGenerator
WAHtmlElement class»root:
WAHtmlElement»attributeAt:put:
WAHtmlElement»attributes
WAHtmlElement»encodeBeforeOn:
WAHtmlElement»encodeOn:
WAHtmlElement»initializeWithRoot:
WAHtmlElement»isClosed
WAHtmlRoot»add:
WAHtmlRoot»beXhtml10Strict
WAHtmlRoot»bodyAttributes
WAHtmlRoot»closeOn:
WAHtmlRoot»docType:
WAHtmlRoot»htmlAttributes
WAHtmlRoot»initialize
WAHtmlRoot»meta
WAHtmlRoot»openOn:
WAHtmlRoot»title:
WAHtmlRoot»writeElementsOn:
WAHtmlRoot»writeFootOn:
WAHtmlRoot»writeHeadOn:
WAHtmlRoot»writeScriptsOn:
WAHtmlRoot»writeStylesOn:
WAHttpVersion class»fromString:
WAHttpVersion class»major:minor:
WAHttpVersion class»readFrom:
WAHttpVersion»initializeWithMajor:minor:
WAInitialRequestVisitor class»request:
WAInitialRequestVisitor»initializeWithRequest:
WAInitialRequestVisitor»request
WAInitialRequestVisitor»visitPresenter:
WAKeyGenerator class»current
WAKeyGenerator»keyOfLength:
WALastAccessExpiryPolicy»
defaultConfiguration
WALastAccessExpiryPolicy»initialize
WALastAccessExpiryPolicy»isExpired:key:
WALastAccessExpiryPolicy»retrieved:key:
WALastAccessExpiryPolicy»stored:key:
WALastAccessExpiryPolicy»timeout
WALeastRecentlyUsedExpiryPolicy»
defaultConfiguration
WALeastRecentlyUsedExpiryPolicy»
initialize
WALeastRecentlyUsedExpiryPolicy»
isExpired:key:
WALeastRecentlyUsedExpiryPolicy»
maximumAge
WALeastRecentlyUsedExpiryPolicy»
removed:key:
WALeastRecentlyUsedExpiryPolicy»
retrieved:key:
WALeastRecentlyUsedExpiryPolicy»
stored:key:
WAMergedRequestFields class»on:
WAMergedRequestFields»allAt:
WAMergedRequestFields»at:ifAbsent:
WAMergedRequestFields»includesKey:
WAMergedRequestFields»initializeOn:
WAMergedRequestFields»keysDo:
WAMetaElement»content:
WAMetaElement»contentScriptType:
WAMetaElement»contentType:
WAMetaElement»encodeBeforeOn:
WAMetaElement»responseHeaderName:
WAMetaElement»tag
WAMimeType class»fromString:

WAMimeType class»main:sub:
WAMimeType class»textJavascript
WAMimeType class»textPlain
WAMimeType»charset:
WAMimeType»greaseString
WAMimeType»main:
WAMimeType»main
WAMimeType»parameters
WAMimeType»sub:
WAMimeType»sub
WAMutex»critical:
WAMutex»initialize
WAMutualExclusionFilter»handleFiltered:
WAMutualExclusionFilter»initialize
WAMutualExclusionFilter»shouldTerminate:
WANotifyRemovalAction»removed:key:
WAObject»application
WAObject»requestContext
WAObject»session
WAPainter»renderWithContext:
WAPainter»updateRoot:
WAPainter»updateUrl:
WAPainterVisitor»visitComponent:
WAPainterVisitor»visitDecorationsOfComponent:
WAPainterVisitor»visitPainter:
WAPainterVisitor»visitPresenter:
WAPathConsumer class»path:
WAPathConsumer»atEnd
WAPathConsumer»initializeWith:
WAPathConsumer»next
WAPresenter»childrenDo:
WAPresenter»children
WAPresenter»initialRequest:
WAPresenter»script
WAPresenter»style
WAPresenter»updateRoot:
WAPresenter»updateStates:
WAPresenterGuide class»client:
WAPresenterGuide»client
WAPresenterGuide»initializeWithClient:
WAPresenterGuide»visit:
WAPresenterGuide»visitPainter:
WARegistryConfiguration»parents
WARenderContext»actionBaseUrl:
WARenderContext»actionUrl:
WARenderContext»actionUrl
WARenderContext»callbacks
WARenderContext»defaultVisitor
WARenderContext»destroy
WARenderContext»document:
WARenderContext»document
WARenderContext»initialize
WARenderContext»resourceUrl:
WARenderContext»visitor:
WARenderContext»visitor
WARenderLoopConfiguration»parents
WARenderLoopContinuation»createActionContinuation
WARenderLoopContinuation»createRenderContinuation
WARenderLoopContinuation»presenter
WARenderLoopContinuation»toPresenterSendRoot:
WARenderLoopContinuation»updateRoot:
WARenderLoopContinuation»updateStates:
WARenderLoopContinuation»updateUrl:
WARenderLoopContinuation»withNotificationHandlerDo:
WARenderLoopMain»createRoot
WARenderLoopMain»prepareRoot:
WARenderLoopMain»rootClass
WARenderLoopMain»rootDecorationClasses
WARenderLoopMain»start
WARenderPhaseContinuation»createHtmlRootWithContext:
WARenderPhaseContinuation»createRenderContext
WARenderPhaseContinuation»handleRequest
WARenderPhaseContinuation»
processRendering:
WARenderVisitor class»context:
WARenderVisitor»initializeWithContext:
WARenderVisitor»renderContext
WARenderVisitor»visitPainter:
WARenderer class»context:
WARenderer»actionUrl
WARenderer»callbacks
WARenderer»context
WARenderer»document
WARenderer»flush
WARenderer»initializeWithContext:
WARenderer»render:
WARenderer»text:
WARequest class»method:uri:version:
WARequest»at:ifAbsent:
WARequest»cookiesAt:
WARequest»cookies
WARequest»destroy

WARequest»fields
WARequest»headerAt:ifAbsent:
WARequest»headerAt:
WARequest»
initializeWithMethod:uri:version:
WARequest»isGet
WARequest»isPrefetch
WARequest»isXmlHttpRequest
WARequest»method
WARequest»postFields
WARequest»queryFields
WARequest»setBody:
WARequest»setCookies:
WARequest»setHeaders:
WARequest»setPostFields:
WARequest»setRemoteAddress:
WARequest»uri
WARequest»url
WARequestContext class»
request:response:codec:
WARequestContext»application
WARequestContext»charSet
WARequestContext»codec
WARequestContext»consumer
WARequestContext»destroy
WARequestContext»handlers
WARequestContext»handler
WARequestContext»
initializeWithRequest:response:codec:
WARequestContext»newDocument
WARequestContext»push:during:
WARequestContext»request
WARequestContext»respond:
WARequestContext»respond
WARequestContext»responseGenerator
WARequestContext»response
WARequestContext»session
WARequestFilter»handleFiltered:
WARequestFilter»initialize
WARequestFilter»next
WARequestFilter»setNext:
WARequestFilter»updateStates:
WARequestHandler»addFilter:
WARequestHandler»addFilterLast:
WARequestHandler»basicUrl
WARequestHandler»configuration:
WARequestHandler»configuration
WARequestHandler»defaultConfiguration
WARequestHandler»documentClass
WARequestHandler»filters
WARequestHandler»filter
WARequestHandler»handle:
WARequestHandler»initialize
WARequestHandler»isApplication
WARequestHandler»isRoot
WARequestHandler»parent
WARequestHandler»preferenceAt:
WARequestHandler»responseGenerator
WARequestHandler»serverHostname
WARequestHandler»serverPath
WARequestHandler»serverPort
WARequestHandler»serverProtocol
WARequestHandler»setFilter:
WARequestHandler»setParent:
WARequestHandler»url
WAResponse class»messageForStatus:
WAResponse class»statusFound
WAResponse class»statusNotFound
WAResponse»contentType:
WAResponse»contentType
WAResponse»cookies
WAResponse»destroy
WAResponse»found
WAResponse»headerAt:ifAbsent:
WAResponse»headerAt:put:
WAResponse»headers
WAResponse»initializeOn:
WAResponse»initialize
WAResponse»location:
WAResponse»nextPutAll:
WAResponse»notFound
WAResponse»redirectTo:
WAResponse»status:message:
WAResponse»status:
WAResponse»status
WAResponseGenerator class»on:ShiftJIS
WAResponseGenerator»initializeOn:
WAResponseGenerator»notFound
WAResponseGenerator»requestContext
WAResponseGenerator»request
WAResponseGenerator»respond
WAResponseGenerator»response

WARoot class»context:
WARoot»setContext:
WAScriptGenerator»initialize
WAScriptGenerator»loadScripts
WAScriptGenerator»writeLoadScriptsOn:
WAScriptGenerator»writeScriptTag:on:
WAServerAdaptor class»defaultSmall
WAServerAdaptor class»manager:
WAServerAdaptor class»new
WAServerAdaptor class»port:
WAServerAdaptor class»startOn:
WAServerAdaptor»codec
WAServerAdaptor»contextFor:
WAServerAdaptor»defaultPort
WAServerAdaptor»defaultRequestHandler
WAServerAdaptor»handle:
WAServerAdaptor»handlePadding:
WAServerAdaptor»handleRequest:
WAServerAdaptor»initializeWithManager:
WAServerAdaptor»initialize
WAServerAdaptor»manager
WAServerAdaptor»port:
WAServerAdaptor»port
WAServerAdaptor»process:
WAServerAdaptor»requestFor:
WAServerAdaptor»requestHandler
WAServerAdaptor»responseFor:
WAServerAdaptor»start
WAServerManager class»default
WAServerManager class»initialize
WAServerManager class»shutDown
WAServerManager class»startUp
WAServerManager»adaptors
WAServerManager»canStart:
WAServerManager»register:
WAServerManager»start:
WASession»actionField
WASession»actionUrlForContinuation:
WASession»actionUrlForKey:
WASession»application
WASession»clearJumpTo
WASession»createCache
WASession»handleFiltered:
WASession»initializeFilters
WASession»initialize
WASession»isSession
WASession»presenter
WASession»properties
WASession»start
WASession»updateRoot:
WASession»updateStates:
WASession»updateUrl:
WASession»url
WASessionContinuation»basicValue
WASessionContinuation»captureAndInvoke
WASessionContinuation»captureState
WASessionContinuation»redirectToContinuation:
WASessionContinuation»registerForUrl:
WASessionContinuation»registerForUrl
WASessionContinuation»request
WASessionContinuation»respond:
WASessionContinuation»setStates:
WASessionContinuation»states
WASessionContinuation»updateStates:
WASessionContinuation»updateUrl:
WASessionContinuation»value
WASessionContinuation»withUnregisteredHandlerDo:
WASlime class»initialize
WASlime class»startUp
WASnapshot»initialize
WASnapshot»register:
WASnapshot»resetWASnapshot»restore
WATagBrush»after
WATagBrush»attributes
WATagBrush»before
WATagBrush»closeTag
WATagBrush»document
WATagBrush»isClosed
WATagBrush»openTag
WATagBrush»storeCallback:
WATagBrush»with:
WAUpdateRootVisitor class»root:WeakKey
WAUpdateRootVisitor»initializeWithRoot:
WAUpdateRootVisitor»root
WAUpdateRootVisitor»visitPainter:
WAUpdateStatesVisitor class»snapshot:
WAUpdateStatesVisitor»initializeWithSnapshot:
WAUpdateStatesVisitor»snapshot
WAUpdateStatesVisitor»visitPresenter:
WAUpdateUrlVisitor class»url:
WAUpdateUrlVisitor»initializeWithUrl:
WAUpdateUrlVisitor»url

WAUpdateUrlVisitor»visitPainter:
WAUrl class»absolute:
WAUrl class»decodePercent:
WAUrl»addField:value:
WAUrl»addField:
WAUrl»addToPath:
WAUrl»decode:
WAUrl»decodedWith:
WAUrl»encodeOn:
WAUrl»encodePathOn:
WAUrl»encodeQueryOn:
WAUrl»encodeSchemeAndAuthorityOn:
WAUrl»fragment
WAUrl»initializeFromString:
WAUrl»initialize
WAUrl»parsePath:
WAUrl»parseQuery:
WAUrl»password
WAUrl»path:
WAUrl»pathElementsIn:do:
WAUrl»pathString
WAUrl»path
WAUrl»postCopy
WAUrl»printOn:
WAUrl»queryFields:
WAUrl»queryFields
WAUrl»seasideUrl
WAUrl»slash:
WAUrl»subStringsIn:splitBy:do:
WAUrl»user
WAUrlEncoder class»on:codec:
WAUrlEncoder»nextPutAll:
WAUserConfiguration»addParent:
WAUserConfiguration»canAddParent:
WAUserConfiguration»expressionAt:ifAbsent:
WAUserConfiguration»initialize
WAUserConfiguration»
localAttributeAt:ifAbsent:
WAUserConfiguration»parents
WAValueExpression»
determineValueWithContext:
configuration:
WAValueExpression»value
WAValueHolder class»with:
WAValueHolder»contents:
WAValueHolder»contents
WAVisiblePresenterGuide»visitPresenter:
WAVisitor»visit:WAVisitor»start:
WAXmlDocument»closeTag:
WAXmlDocument»destroy
WAXmlDocument»
initializeWithStream:codec:
WAXmlDocument»
openTag:attributes:closed:
WAXmlDocument»
openTag:attributes:
WAXmlDocument»openTag:
WAXmlDocument»print:
WAXmlDocument»urlEncoder
WAXmlDocument»xmlEncoder
WAXmlEncoder»nextPutAll:
WeakAnnouncementSubscription class»
finalizationList
WeakAnnouncementSubscription class»
finalizeValues
WeakArray class»finalizationProcess
WeakArray class»initialize
WeakArray class»restartFinalizationProcess
WeakArray class»startUp:
WeakFinalizationList class»hasNewFinalization
WeakFinalizationList class»initialize
WeakFinalizationList class»startUp:
WeakFinalizationList»swapWithNil
WeakFinalizerItem class»new
WeakFinalizerItem»add:
WeakFinalizerItem»list:object:
WeakIdentityKeyDictionary»compare:to:
WeakIdentityKeyDictionary»startIndexFor:
WeakKeyDictionary»associationsDo:
WeakKeyDictionary»finalizeValues
WeakKeyDictionary»fullCheck
WeakKeyDictionary»grow
WeakKeyDictionary»initialize:
WeakKeyDictionary»noCheckAdd:
WeakKeyDictionary»overridingAt:put:
WeakKeyDictionary»rehash
WeakKeyDictionary»removeKey:ifAbsent:
WeakKeyDictionary»scanFor:
WeakKeyDictionary»scanForKeyOrNil:
WeakRegistry»add:executor:
WeakRegistry»add:
WeakRegistry»finalizeValues

WeakRegistry»protected:
WeakRegistry»remove:ifAbsent:
WeakSet»add:
WeakSet»do:
WeakSet»growTo:
WeakSet»grow
WeakSet»initialize:
WeakSet»like:
WeakSet»noCheckNoGrowFillFrom:
WeakSet»scanFor:
WeakSet»scanForEmptySlotFor:
Week class»nameOfDay:
WideString»at:
WideString»wordAt:
WriteStream»«
WriteStream»contents
WriteStream»cr
WriteStream»growTo:
WriteStream»nextPut:
WriteStream»nextPutAll:
WriteStream»on:
WriteStream»pastEndPut:
WriteStream»reset
WriteStream»size
WriteStream»space
ZdcAbstractSocketStream class»on:
ZdcAbstractSocketStream»SocketSendData:startingAt:count:
ZdcAbstractSocketStream»SocketWaitForData
ZdcAbstractSocketStream»autoFlush:
ZdcAbstractSocketStream»binary
ZdcAbstractSocketStream»bufferSize:
ZdcAbstractSocketStream»close
ZdcAbstractSocketStream»flush
ZdcAbstractSocketStream»initializeBuffers
ZdcAbstractSocketStream»initialize
ZdcAbstractSocketStream»isBinary
ZdcAbstractSocketStream»nextPut:
ZdcAbstractSocketStream»nextPutAll:
ZdcAbstractSocketStream»next
ZdcAbstractSocketStream»on:
ZdcAbstractSocketStream»peek
ZdcAbstractSocketStream»shouldSignal:
ZdcAbstractSocketStream»socketClose
ZdcAbstractSocketStream»
socketIsConnected
ZdcAbstractSocketStream»
socketIsDataAvailable
ZdcAbstractSocketStream»
socketReceiveDataInto:startingAt:count:
ZdcAbstractSocketStream»socket
ZdcAbstractSocketStream»timeout:
ZdcAbstractSocketStream»timeout
ZdcIOBuffer class»on:
ZdcIOBuffer class»onByteArrayOfSize:
ZdcIOBuffer»advanceWritePointer:
ZdcIOBuffer»availableForReading
ZdcIOBuffer»availableForWriting
ZdcIOBuffer»bufferSize
ZdcIOBuffer»buffer
ZdcIOBuffer»compact
ZdcIOBuffer»contentsStart
ZdcIOBuffer»freeSpaceStart
ZdcIOBuffer»isEmpty
ZdcIOBuffer»isFull
ZdcIOBuffer»next:putAll:startingAt:
ZdcIOBuffer»nextPut:
ZdcIOBuffer»next
ZdcIOBuffer»on:
ZdcIOBuffer»peek
ZdcIOBuffer»reset
ZdcSimpleSocketStream»atEnd
ZdcSimpleSocketStream»
fillBytes:startingAt:count:
ZdcSimpleSocketStream»
fillReadBufferNoWait
ZdcSimpleSocketStream»
fillReadBuffer
ZdcSimpleSocketStream»
flushBytes:startingAt:count:
ZdcSimpleSocketStream»
flushWriteBuffer
ZdcSimpleSocketStream»
isConnected
ZdcSocketStream»next:putAll:startingAt:
ZnBivalentWriteStream class»on:
ZnBivalentWriteStream»isBinary
ZnBivalentWriteStream»nextPut:
ZnBivalentWriteStream»nextPutAll:
ZnBivalentWriteStream»on:
ZnBivalentWriteStream»space
ZnByteArrayEntity class»bytes:
ZnByteArrayEntity class»designatedMimeType

ZnByteArrayEntity»bytes:
ZnByteArrayEntity»bytes
ZnByteArrayEntity»writeOn:
ZnCharacterEncoder class»newForEncoding:
ZnCharacterEncoder»beLenient
ZnCharacterEncoder»encodedByteCountForString:
ZnConstants class»defaultHTTPVersion
ZnConstants class»defaultMaximumEntitySize
ZnConstants class»defaultServerString
ZnConstants class»frameworkNameAndVersion
ZnConstants class»frameworkName
ZnConstants class»frameworkVersion
ZnConstants class»httpStatusCodes
ZnConstants class»knownHTTPMethods
ZnConstants class»knownHTTPVersions
ZnConstants class»maximumLineLength
ZnConstants class»remoteAddressHeader
ZnEntity class»byteArrayEntityClass
ZnEntity class»bytes:
ZnEntity class»new
ZnEntity class»stringEntityClass
ZnEntity class»text:
ZnEntity class»textCRLF:
ZnEntity class»type:length:
ZnEntity class»type:
ZnEntity»contentLength:
ZnEntity»contentLength
ZnEntity»contentType:
ZnEntity»contentType
ZnEntityReader»allowsReadingUpToEnd
ZnEntityReader»binary
ZnEntityReader»canReadContent
ZnEntityReader»hasContentLength
ZnEntityReader»headers:
ZnEntityReader»headers
ZnEntityReader»isChunked
ZnEntityReader»readEntity
ZnEntityReader»stream:
ZnEntityWriter»headers:
ZnEntityWriter»headers
ZnEntityWriter»isChunked
ZnEntityWriter»isGzipped
ZnEntityWriter»stream:
ZnEntityWriter»writeEntity:
ZnHeaders class»defaultResponseHeaders
ZnHeaders class»readFrom:
ZnHeaders»acceptEntityDescription:
ZnHeaders»at:add:
ZnHeaders»at:ifAbsent:
ZnHeaders»at:put:
ZnHeaders»clearContentLength
ZnHeaders»clearContentType
ZnHeaders»contentLength:
ZnHeaders»contentType:
ZnHeaders»headersDo:
ZnHeaders»headers
ZnHeaders»includesKey:
ZnHeaders»isDescribingEntity
ZnHeaders»isEmpty
ZnHeaders»normalizeHeaderKey:
ZnHeaders»readFrom:
ZnHeaders»readOneHeaderFrom:
ZnHeaders»removeKey:ifAbsent:
ZnHeaders»writeOn:
ZnLineReader class»on:
ZnLineReader»growBuffer
ZnLineReader»limit:
ZnLineReader»nextLine
ZnLineReader»on:
ZnLineReader»processNext
ZnLineReader»reset
ZnLineReader»store:
ZnLogSupport»debug:
ZnLogSupport»disable
ZnLogSupport»enabled:
ZnLogSupport»enabled
ZnLogSupport»info:
ZnLogSupport»initialize
ZnLogSupport»transaction:
ZnManagingMultiThreadedServer»closeConnections
ZnManagingMultiThreadedServer»closeSocketStream:
ZnManagingMultiThreadedServer»connections
ZnManagingMultiThreadedServer»lock
ZnManagingMultiThreadedServer»socketStreamOn:
ZnManagingMultiThreadedServer»stop:
ZnMessage class»readBinaryFrom:UTF8

ZnMessage»entity:
ZnMessage»entityReaderOn:
ZnMessage»entityWriterOn:
ZnMessage»entity
ZnMessage»hasEntity
ZnMessage»hasHeaders
ZnMessage»headers:
ZnMessage»headersDo:
ZnMessage»headers
ZnMessage»isConnectionClose
ZnMessage»readBinaryFrom:
ZnMessage»readHeaderFrom:
ZnMessage»setConnectionClose
ZnMessage»wantsConnectionClose
ZnMessage»writeOn:
ZnMimeType class»applicationOctetStream
ZnMimeType class»fromString:
ZnMimeType class»main:sub:parameters:
ZnMimeType class»main:sub:
ZnMimeType class»textPlain
ZnMimeType»=
ZnMimeType»asZnMimeType
ZnMimeType»charSet:
ZnMimeType»charSet
ZnMimeType»main:
ZnMimeType»main
ZnMimeType»parameterAt:ifAbsent:
ZnMimeType»parameters:
ZnMimeType»parameters
ZnMimeType»printOn:
ZnMimeType»setCharSetUTF8
ZnMimeType»sub:
ZnMimeType»sub
ZnMultiThreadedServer»augmentResponse:forRequest:
ZnMultiThreadedServer»closeSocketStream:
ZnMultiThreadedServer»exceptionSet:
ZnMultiThreadedServer»executeOneRequestResponseOn:
ZnMultiThreadedServer»executeRequestResponseLoopOn:
ZnMultiThreadedServer»listenLoop
ZnMultiThreadedServer»readRequestBadExceptionSet
ZnMultiThreadedServer»readRequestSafely:
ZnMultiThreadedServer»readRequestTerminationExceptionSet
ZnMultiThreadedServer»serveConnectionsOn:
ZnMultiThreadedServer»workerProcessName
ZnMultiThreadedServer»writeResponseBad:on:
ZnMultiThreadedServer»writeResponseSafely:on:
ZnMultiThreadedServer»writeResponseTerminationExceptionSet
ZnMultiValueDictionary»at:add:
ZnMultiValueDictionary»at:put:
ZnMultiValueDictionary»checkLimitForKey:
ZnMultiValueDictionary»defaultLimit
ZnMultiValueDictionary»initialize:
ZnMultiValueDictionary»keysAndValuesDo:
ZnMultiValueDictionary»limit
ZnNetworkingUtils class»defaultSocketStreamTimeout
ZnNetworkingUtils class»default
ZnNetworkingUtils class»ipAddressToString:
ZnNetworkingUtils class»listenBacklogSize
ZnNetworkingUtils class»serverSocketOn:
ZnNetworkingUtils class»socketBufferSize
ZnNetworkingUtils class»socketStreamOn:
ZnNetworkingUtils class»socketStreamTimeout
ZnNetworkingUtils»bufferSize
ZnNetworkingUtils»serverSocketOn:
ZnNetworkingUtils»setServerSocketOptions:
ZnNetworkingUtils»setSocketStreamParameters:
ZnNetworkingUtils»socketStreamClass
ZnNetworkingUtils»socketStreamOn:
ZnNetworkingUtils»timeout
ZnNullEncoder class»handlesEncoding:
ZnPercentEncoder»characterEncoder:
ZnPercentEncoder»characterEncoder
ZnPercentEncoder»decode:to:
ZnPercentEncoder»decode:
ZnPercentEncoder»encode:to:
ZnPercentEncoder»encode:
ZnPercentEncoder»safeSet:
ZnPercentEncoder»safeSet
ZnRequest»isHttp10
ZnRequest»method
ZnRequest»readHeaderFrom:
ZnRequest»requestLine:
ZnRequest»requestLine
ZnRequest»uri
ZnRequest»wantsConnectionClose
ZnRequestLine class»readFrom:
ZnRequestLine»isHttp10
ZnRequestLine»method:
ZnRequestLine»method
ZnRequestLine»readFrom:
ZnRequestLine»uri:
ZnRequestLine»uri

ZnRequestLine»version:
ZnRequestLine»version
ZnResourceMetaUtils class»
decodePercent:
ZnResourceMetaUtils class»
encodePercent:safeSet:encoding:
ZnResourceMetaUtils class»
parseQueryFrom:
ZnResourceMetaUtils class»
queryKeyValueSafeSet
ZnResourceMetaUtils class»
urlPathSafeSet
ZnResourceMetaUtils class»
writeQueryFields:on:
ZnResourceMetaUtils class»
writeQueryFields:withTextEncoding:on:
ZnResponse»setConnectionCloseFor:
ZnResponse»setKeepAliveFor:
ZnResponse»statusLine:
ZnResponse»statusLine
ZnResponse»useConnection:
ZnResponse»writeOn:
ZnSeasideServerAdaptorDelegate class»with:
ZnSeasideServerAdaptorDelegate»adaptor:
ZnSeasideServerAdaptorDelegate»adaptor
ZnSeasideServerAdaptorDelegate»
handleRequest:
ZnServer class»defaultServerClass
ZnServer class»initialize
ZnServer class»managedServers
ZnServer class»on:
ZnServer class»shutDown:
ZnServer class»startUp:
ZnServer class»unregister:
ZnServer»authenticator
ZnServer»bindingAddress
ZnServer»delegate:
ZnServer»delegate
ZnServer»maximumEntitySize
ZnServer»optionAt:ifAbsent:
ZnServer»optionAt:ifAbsentPut:
ZnServer»optionAt:put:
ZnServer»port:
ZnServer»port
ZnServer»reader:
ZnServer»reader
ZnServer»stop
ZnServer»unregister
ZnServer»useGzipCompressionAndChunking
ZnSignalProgress class»enabled
ZnSingleThreadedServer class»default
ZnSingleThreadedServer class»on:Latin1
ZnSingleThreadedServer»acceptWaitTimeout
ZnSingleThreadedServer»augmentResponse:forRequest:
ZnSingleThreadedServer»authenticateAndDelegateRequest:
ZnSingleThreadedServer»authenticateRequest:do:
ZnSingleThreadedServer»closeDelegate
ZnSingleThreadedServer»handleRequest:
ZnSingleThreadedServer»handleRequestProtected:
ZnSingleThreadedServer»initializeServerSocket
ZnSingleThreadedServer»isRunning
ZnSingleThreadedServer»logRequest:response:started:
ZnSingleThreadedServer»logRequest:
ZnSingleThreadedServer»logResponse:
ZnSingleThreadedServer»log
ZnSingleThreadedServer»noteAcceptWaitTimedOut
ZnSingleThreadedServer»periodicTasks
ZnSingleThreadedServer»process
ZnSingleThreadedServer»readRequest:
ZnSingleThreadedServer»releaseServerSocket
ZnSingleThreadedServer»serverProcessName
ZnSingleThreadedServer»serverSocket
ZnSingleThreadedServer»socketStreamOn:
ZnSingleThreadedServer»start
ZnSingleThreadedServer»stop:
ZnSingleThreadedServer»withMaximumEntitySizeDo:
ZnSingleThreadedServer»writeResponse:on:
ZnStatusLine class»badRequest
ZnStatusLine class»code:
ZnStatusLine»code:
ZnStatusLine»code
ZnStatusLine»reason
ZnStatusLine»version:
ZnStatusLine»version
ZnStatusLine»writeOn:
ZnStringEntity class»designatedMimeType
ZnStringEntity class»text:
ZnStringEntity»computeContentLength
ZnStringEntity»contentLength
ZnStringEntity»encoder:
ZnStringEntity»encoder
ZnStringEntity»hasEncoder

ZnStringEntity»initializeEncoder
ZnStringEntity»string:
ZnStringEntity»string
ZnStringEntity»writeOn:
ZnUTF8Encoder class»handlesEncoding:
ZnUTF8Encoder class»newForEncoding:
ZnUTF8Encoder»decodeBytes:
ZnUTF8Encoder»encodedByteCountFor:
ZnUTF8Encoder»
    findFirstNonASCIIIn:startingAt:
ZnUTF8Encoder»
    next:putAll:startingAt:toStream:
ZnUTF8Encoder»
    next:putAllASCII:startingAt:toStream:
ZnUTF8Encoder»
    next:putAllByteString:startingAt:
    toStream:
ZnUTF8Encoder»
    nextFromStream:
ZnUTF8Encoder»
    nextPut:toStream:
ZnUnknownHttpMethod class»method:
ZnUnknownHttpMethod»method:
ZnUrl class»fromString:
ZnUrl class»schemesNotUsingPath
ZnUrl»addPathSegment:
ZnUrl»decodePercent:
ZnUrl»encodePath:on:
ZnUrl»enforceKnownScheme
ZnUrl»hasFragment
ZnUrl»hasHost
ZnUrl»hasPath
ZnUrl»hasPort
ZnUrl»hasQuery
ZnUrl»hasScheme
ZnUrl»hasUsername
ZnUrl»isSchemeUsingPath
ZnUrl»parseFrom:defaultScheme:
ZnUrl»parseFrom:
ZnUrl»parsePath:
ZnUrl»printAuthorityOn:
ZnUrl»printOn:
ZnUrl»printPathOn:
ZnUrl»printPathQueryFragmentOn:
ZnUrl»printQueryOn:
ZnUrl»query:
ZnUrl»query
ZnUrl»scheme
ZnUtils class»httpDate:
ZnUtils class»httpDate
ZnUtils class»nextPutAll:on:
ZnUtils class»signalProgress:total:
ZnUtils class»streamingBufferSize
ZnZincServerAdaptor»basicStart
ZnZincServerAdaptor»configureDelegate
ZnZincServerAdaptor»configureServerForBinaryReading
ZnZincServerAdaptor»defaultCodec
ZnZincServerAdaptor»defaultDelegate
ZnZincServerAdaptor»defaultZnServer
ZnZincServerAdaptor»isRunning
ZnZincServerAdaptor»isStopped
ZnZincServerAdaptor»requestAddressFor:
ZnZincServerAdaptor»requestBodyFor:
ZnZincServerAdaptor»requestCookiesFor:
ZnZincServerAdaptor»requestFieldsFor:
ZnZincServerAdaptor»requestHeadersFor:
ZnZincServerAdaptor»requestMethodFor:
ZnZincServerAdaptor»requestUrlFor:
ZnZincServerAdaptor»requestVersionFor:
ZnZincServerAdaptor»responseFrom:
ZnZincServerAdaptor»server
ZnZincServerAdaptor»shutDown
ZnZincServerAdaptor»startUp

D.4 Seaside App B Extract

This section list the methods extracted from the nurtured Web application when using a seed containing all base libraries from Pharo. This list includes all methods installed from Seaside framework and the counter application. The list of methods part of the base library are excluded as it is the same list of the methods found in Pharo base library.

WAAccessIntervalReapingStrategy»defaultConfiguration
WAAccessIntervalReapingStrategy»initialize
WAAccessIntervalReapingStrategy»interval
WAAccessIntervalReapingStrategy»reap
WAAccessIntervalReapingStrategy»stored:key:
WAActionCallback»block:
WAActionCallback»evaluateWithArgument:
WAActionCallback»isEnabledFor:
WAActionCallback»signalRenderNotification
WAActionPhaseContinuation»continue
WAActionPhaseContinuation»handleRequest
WAActionPhaseContinuation»renderContext:
WAActionPhaseContinuation»renderContext
WAActionPhaseContinuation»runCallbacks
WAActionPhaseContinuation»shouldRedirect
WAAdmin class»defaultServerManager
WAAdmin class»serverAdaptors
WAAnchorTag»callback:
WAAnchorTag»tag
WAAnchorTag»url
WAAnchorTag»with:
WAApplication»contentType
WAApplication»doesHandlerSupportCookies:
WAApplication»handleDefault:
WAApplication»handleFiltered:
WAApplication»isApplication
WAApplication»isImplemented:
WAApplication»keyField
WAApplication»libraries
WAApplication»mainClass
WAApplication»mimeType
WAApplication»newSession
WAApplication»resourceBaseUrl
WAApplication»sessionClass
WAApplicationConfiguration»parents
WAAttributeSearchContext class»key:target:
WAAttributeSearchContext»at:ifPresent:
WAAttributeSearchContext»at:put:
WAAttributeSearchContext»attribute
WAAttributeSearchContext»cachedValues
WAAttributeSearchContext»findAttributeAndSelectAncestorsOf:
WAAttributeSearchContext»initializeWithKey
WAAttributeSearchContext»isAttributeInherited
WAAttributeSearchContext»isAttributeLocal
WAAttributeSearchContext»key
WABrush»initialize
WABrush»parent
WABrush»setParent:canvas:
WABrush»with:
WABufferedResponse class»on:
WABufferedResponse»contents
WABufferedResponse»destroy
WABufferedResponse»initializeOn:
WABufferedResponse»stream
WACache»at:ifAbsent:
WACache»expiryPolicy
WACache»initializeCollections
WACache»initializeMutex
WACache»initialize
WACache»keyAtValue:ifAbsent:
WACache»keyAtValue:
WACache»keySize
WACache»missStrategy
WACache»notifyRemoved:key:
WACache»notifyRetrieved:key:
WACache»notifyStored:key:
WACache»pluginsDo:
WACache»reapingStrategy
WACache»reap
WACache»removalAction
WACache»setExpiryPolicy:
WACache»setMissStrategy:
WACache»setReapingStrategy:
WACache»setRemovalAction:
WACache»store:
WACacheCapacityConfiguration»describeOn:
WACacheMissStrategy»missed:
WACachePlugin»configuration
WACachePlugin»defaultConfiguration
WACachePlugin»initialize
WACachePlugin»removed:key:
WACachePlugin»retrieved:key:
WACachePlugin»setCache:
WACachePlugin»stored:key:
WACacheReapingStrategy»reap
WACallback class»on:
WACallback»convertKey:
WACallback»evaluateWithFieldValues:
WACallback»key
WACallback»setKey:callbacks:

WACallback»valueForField:
WACallbackRegistry»advanceKey
WACallbackRegistry»handle:
WACallbackRegistry»increaseKey
WACallbackRegistry»initialize
WACallbackRegistry»nextKey
WACallbackRegistry»store:
WACanvas»brush:
WACanvas»flush
WACanvas»nest:
WACanvas»render:
WACanvas»text:
WAComponent»accept:
WAComponent»acceptDecorated:
WAComponent»decoration
WAComponent»initialize
WAComponent»updateStates:
WAConfigurationDescription»add:to:
WAConfigurationDescription»addAttribute:
WAConfigurationDescription»attributes
WAConfigurationDescription»expressions
WAConfigurationDescription»initialize
WAConfigurationDescription»integer:
WAConfiguredRequestFilter»configuration
WACounter»count:
WACounter»decrease
WACounter»increase
WACounter»initialize
WACounter»renderContentOn:
WACounter»states
WADefaultScriptGenerator»close:on:
WADefaultScriptGenerator»open:on:
WADevelopmentConfiguration»parents
WADispatcher class»default
WADispatcher»handleFiltered:named:
WADispatcher»handleFiltered:
WADispatcher»handlerAt:ifAbsent:
WADispatcher»handlerAt:with:
WADispatcher»handlers
WADispatcher»nameOfHandler:
WADispatcher»urlFor:
WADocument class»on:codec:
WADocument»close
WADocument»destroy
WADocument»initializeWithStream:codec:
WADocument»nextPut:
WADocument»nextPutAll:
WADocument»open:
WADynamicVariable class»use:during:
WADynamicVariable class»value
WAEncoder class»on:table:
WAEncoder class»on:
WAEncoder»initializeOn:table:
WAEncoder»nextPut:
WAErrorHandler class»exceptionSelector
WAExampleComponent»rendererClass
WAExceptionFilter»exceptionHandler
WAExceptionFilter»handleFiltered:
WAExceptionHandler class»context:
WAExceptionHandler class»exceptionSelector
WAExceptionHandler class»handleExceptionsDuring:context:
WAExceptionHandler class»handles:
WAExceptionHandler»handleExceptionsDuring:
WAExceptionHandler»handles:
WAExceptionHandler»initializeWithContext:
WAHeaderFields»checkValue:
WAHeaderFields»privateAt:put:
WAHeadingTag»initialize
WAHeadingTag»level1
WAHeadingTag»level
WAHeadingTag»tag
WAHtmlAttributes»encodeOn:
WAHtmlAttributes»privateAt:put:
WAHtmlCanvas»anchor
WAHtmlCanvas»heading:
WAHtmlCanvas»heading
WAHtmlCanvas»spaceEntity
WAHtmlDocument»scriptGenerator:
WAHtmlDocument»scriptGenerator
WAHtmlElement class»root:
WAHtmlElement»attributeAt:put:
WAHtmlElement»attributes
WAHtmlElement»encodeBeforeOn:
WAHtmlElement»encodeOn:
WAHtmlElement»initializeWithRoot:
WAHtmlElement»isClosed
WAHtmlRoot»add:
WAHtmlRoot»beXhtml10Strict
WAHtmlRoot»bodyAttributes
WAHtmlRoot»closeOn:
WAHtmlRoot»docType:
WAHtmlRoot»htmlAttributes

WAHtmlRoot»initialize
WAHtmlRoot»meta
WAHtmlRoot»openOn:
WAHtmlRoot»title:
WAHtmlRoot»writeElementsOn:
WAHtmlRoot»writeFootOn:
WAHtmlRoot»writeHeadOn:
WAHtmlRoot»writeScriptsOn:
WAHtmlRoot»writeStylesOn:
WAHttpVersion class»fromString:
WAHttpVersion class»major:minor:
WAHttpVersion class»readFrom:
WAHttpVersion»initializeWithMajor:minor:
WAInitialRequestVisitor class»request:
WAInitialRequestVisitor»
initializeWithRequest:
WAInitialRequestVisitor»request
WAInitialRequestVisitor»visitPresenter:
WAKeyGenerator class»current
WAKeyGenerator»keyOfLength:
WALastAccessExpiryPolicy»
defaultConfiguration
WALastAccessExpiryPolicy»initialize
WALastAccessExpiryPolicy»isExpired:key:
WALastAccessExpiryPolicy»retrieved:key:
WALastAccessExpiryPolicy»stored:key:
WALastAccessExpiryPolicy»timeout
WALeastRecentlyUsedExpiryPolicy»
defaultConfiguration
WALeastRecentlyUsedExpiryPolicy»
initialize
WALeastRecentlyUsedExpiryPolicy»
isExpired:key:
WALeastRecentlyUsedExpiryPolicy»
maximumAge
WALeastRecentlyUsedExpiryPolicy»
removed:key:
WALeastRecentlyUsedExpiryPolicy»
retrieved:key:
WALeastRecentlyUsedExpiryPolicy»
stored:key:
WAMergedRequestFields class»on:
WAMergedRequestFields»allAt:
WAMergedRequestFields»at:ifAbsent:
WAMergedRequestFields»includesKey:
WAMergedRequestFields»initializeOn:
WAMergedRequestFields»keysAndValuesDo:
WAMergedRequestFields»keysDo:
WAMetaElement»content:
WAMetaElement»contentScriptType:
WAMetaElement»contentType:
WAMetaElement»encodeBeforeOn:
WAMetaElement»responseHeaderName:
WAMetaElement»tag
WAMimeType class»fromString:
WAMimeType class»main:sub:
WAMimeType class»textJavascript
WAMimeType class»textPlain
WAMimeType»charset:
WAMimeType»greaseString
WAMimeType»main:
WAMimeType»main
WAMimeType»parameters
WAMimeType»sub:
WAMimeType»sub
WAMutex»critical:
WAMutex»initialize
WAMutualExclusionFilter»handleFiltered:
WAMutualExclusionFilter»initialize
WAMutualExclusionFilter»shouldTerminate:
WANotifyRemovalAction»removed:key:
WAObject»application
WAObject»requestContext
WAObject»session
WAPainter»renderWithContext:
WAPainter»updateRoot:
WAPainter»updateUrl:
WAPainterVisitor»visitComponent:
WAPainterVisitor»visitDecorationsOfComponent:
WAPainterVisitor»visitPainter:
WAPainterVisitor»visitPresenter:
WAPathConsumer class»path:
WAPathConsumer»atEnd
WAPathConsumer»initializeWith:
WAPathConsumer»next
WAPathConsumer»upToEnd
WAPresenter»childrenDo:
WAPresenter»children
WAPresenter»initialRequest:
WAPresenter»script
WAPresenter»style
WAPresenter»updateRoot:

WAPresenter»updateStates:
WAPresenterGuide class»client:
WAPresenterGuide»client
WAPresenterGuide»initializeWithClient:
WAPresenterGuide»visit:
WAPresenterGuide»visitPainter:
WARegistryConfiguration»parents
WARenderContext»actionBaseUrl:
WARenderContext»actionUrl:
WARenderContext»actionUrl
WARenderContext»callbacks
WARenderContext»defaultVisitor
WARenderContext»destroy
WARenderContext»document:
WARenderContext»document
WARenderContext»initialize
WARenderContext»resourceUrl:
WARenderContext»visitor:
WARenderContext»visitor
WARenderLoopConfiguration»parents
WARenderLoopContinuation»
createActionContinuation
WARenderLoopContinuation»
createRenderContinuation
WARenderLoopContinuation»
presenter
WARenderLoopContinuation»
toPresenterSendRoot:
WARenderLoopContinuation»
updateRoot:
WARenderLoopContinuation»
updateStates:
WARenderLoopContinuation»
updateUrl:
WARenderLoopContinuation»
withNotificationHandlerDo:
WARenderLoopMain»createRoot
WARenderLoopMain»prepareRoot:
WARenderLoopMain»rootClass
WARenderLoopMain»rootDecorationClasses
WARenderLoopMain»start
WARenderPhaseContinuation»
createHtmlRootWithContext:
WARenderPhaseContinuation»
createRenderContext
WARenderPhaseContinuation»
handleRequest
WARenderPhaseContinuation»
processRendering:
WARenderVisitor class»context:
WARenderVisitor»initializeWithContext:
WARenderVisitor»renderContext
WARenderVisitor»visitPainter:
WARenderer class»context:
WARenderer»actionUrl
WARenderer»callbacks
WARenderer»context
WARenderer»document
WARenderer»flush
WARenderer»initializeWithContext:
WARenderer»render:
WARenderer»text:
WARequest class»method:uri:version:
WARequest»at:ifAbsent:
WARequest»cookiesAt:
WARequest»cookies
WARequest»destroy
WARequest»fields
WARequest»headerAt:ifAbsent:
WARequest»headerAt:
WARequest»initializeWithMethod:uri:version:
WARequest»isGet
WARequest»isPrefetch
WARequest»isXmlHttpRequest
WARequest»method
WARequest»postFields
WARequest»queryFields
WARequest»setBody:
WARequest»setCookies:
WARequest»setHeaders:
WARequest»setPostFields:
WARequest»setRemoteAddress:
WARequest»uri
WARequest»url
WARequestContext class»request:response:codec:
WARequestContext»application
WARequestContext»charSet
WARequestContext»codec
WARequestContext»consumer
WARequestContext»destroy
WARequestContext»handlers
WARequestContext»handler

WARequestContext»initializeWithRequest:response:codec:
WARequestContext»newDocument
WARequestContext»push:during:
WARequestContext»registry
WARequestContext»request
WARequestContext»respond:
WARequestContext»respond
WARequestContext»responseGenerator
WARequestContext»response
WARequestContext»session
WARequestFilter»handleFiltered:
WARequestFilter»initialize
WARequestFilter»next
WARequestFilter»setNext:
WARequestFilter»updateStates:
WARequestHandler»addFilter:
WARequestHandler»addFilterLast:
WARequestHandler»basicUrl
WARequestHandler»configuration:
WARequestHandler»configuration
WARequestHandler»defaultConfiguration
WARequestHandler»documentClass
WARequestHandler»filters
WARequestHandler»filter
WARequestHandler»handle:
WARequestHandler»initialize
WARequestHandler»isApplication
WARequestHandler»isRoot
WARequestHandler»parent
WARequestHandler»preferenceAt:
WARequestHandler»responseGenerator
WARequestHandler»serverHostname
WARequestHandler»serverPath
WARequestHandler»serverPort
WARequestHandler»serverProtocol
WARequestHandler»setFilter:
WARequestHandler»setParent:
WARequestHandler»url
WAResponse class»messageForStatus:
WAResponse class»statusFound
WAResponse»contentType:
WAResponse»contentType
WAResponse»cookies
WAResponse»destroy
WAResponse»found
WAResponse»headerAt:ifAbsent:
WAResponse»headerAt:put:
WAResponse»headers
WAResponse»initializeOn:
WAResponse»initialize
WAResponse»location:
WAResponse»redirectTo:
WAResponse»status:message:
WAResponse»status:
WAResponse»status
WAResponseGenerator class»on:
WAResponseGenerator»expiredRegistryKey
WAResponseGenerator»initializeOn:
WAResponseGenerator»requestContext
WAResponseGenerator»request
WAResponseGenerator»respond
WAResponseGenerator»response
WARoot class»context:
WARoot»setContext:
WAScriptGenerator»initialize
WAScriptGenerator»loadScripts
WAScriptGenerator»writeLoadScriptsOn:
WAScriptGenerator»writeScriptTag:on:
WAServerAdaptor class»default
WAServerAdaptor class»manager:
WAServerAdaptor class»new
WAServerAdaptor class»port:
WAServerAdaptor class»startOn:
WAServerAdaptor»codec
WAServerAdaptor»contextFor:
WAServerAdaptor»defaultPort
WAServerAdaptor»defaultRequestHandler
WAServerAdaptor»handle:
WAServerAdaptor»handlePadding:
WAServerAdaptor»handleRequest:
WAServerAdaptor»initializeWithManager:
WAServerAdaptor»initialize
WAServerAdaptor»manager
WAServerAdaptor»port:
WAServerAdaptor»port
WAServerAdaptor»process:
WAServerAdaptor»requestFor:
WAServerAdaptor»requestHandler
WAServerAdaptor»responseFor:
WAServerAdaptor»start
WAServerManager class»default
WAServerManager»adaptors

WAServerManager»canStart:
WAServerManager»register:
WAServerManager»start:
WASession»actionField
WASession»actionUrlForContinuation:
WASession»actionUrlForKey:
WASession»application
WASession»clearJumpTo
WASession»createCache
WASession»handleFiltered:
WASession»initializeFilters
WASession»initialize
WASession»isSession
WASession»presenter
WASession»properties
WASession»start
WASession»unknownRequest
WASession»updateRoot:
WASession»updateStates:
WASession»updateUrl:
WASession»url
WASessionContinuation»basicValue
WASessionContinuation»captureAndInvoke
WASessionContinuation»captureState
WASessionContinuation»redirectToContinuation:
WASessionContinuation»registerForUrl:
WASessionContinuation»registerForUrl
WASessionContinuation»request
WASessionContinuation»respond:
WASessionContinuation»setStates:
WASessionContinuation»states
WASessionContinuation»updateStates:
WASessionContinuation»updateUrl:
WASessionContinuation»value
WASessionContinuation»withUnregisteredHandlerDo:
WASnapshot»initialize
WASnapshot»register:
WASnapshot»reset
WASnapshot»restore
WATagBrush»after
WATagBrush»attributes
WATagBrush»before
WATagBrush»closeTag
WATagBrush»document
WATagBrush»isClosed
WATagBrush»openTag
WATagBrush»storeCallback:
WATagBrush»with:
WATagCanvas»space
WATagCanvas»space
WAUnescapedDocument»initializeWithStream:codec:
WAUpdateRootVisitor class»root:
WAUpdateRootVisitor»initializeWithRoot:
WAUpdateRootVisitor»root
WAUpdateRootVisitor»visitPainter:
WAUpdateStatesVisitor class»snapshot:
WAUpdateStatesVisitor»initializeWithSnapshot:
WAUpdateStatesVisitor»snapshot
WAUpdateStatesVisitor»visitPresenter:
WAUpdateUrlVisitor class»url:
WAUpdateUrlVisitor»initializeWithUrl:
WAUpdateUrlVisitor»url
WAUpdateUrlVisitor»visitPainter:
WAUrl class»absolute:
WAUrl class»decodePercent:
WAUrl»addAllToPath:
WAUrl»addField:value:
WAUrl»addField:
WAUrl»addToPath:
WAUrl»decode:
WAUrl»decodedWith:
WAUrl»encodeOn:
WAUrl»encodePathOn:
WAUrl»encodeQueryOn:
WAUrl»encodeSchemeAndAuthorityOn:
WAUrl»fragment
WAUrl»initializeFromString:
WAUrl»initialize
WAUrl»isSeasideField:
WAUrl»parsePath:
WAUrl»parseQuery:
WAUrl»password
WAUrl»path:
WAUrl»pathElementsIn:do:
WAUrl»path
WAUrl»postCopy
WAUrl»printOn:
WAUrl»queryFields:
WAUrl»queryFields
WAUrl»seasideUrl
WAUrl»slash:
WAUrl»subStringsIn:splitBy:do:

WAUrl»user
WAUrlEncoder class»on:codec:
WAUrlEncoder»nextPutAll:
WAUserConfiguration»addParent:
WAUserConfiguration»canAddParent:
WAUserConfiguration»expressionAt:ifAbsent:
WAUserConfiguration»initialize
WAUserConfiguration»localAttributeAt:ifAbsent:
WAUserConfiguration»parents
WAValueExpression»determineValueWithContext:
WAValueExpression»value
WAValueHolder class»with:
WAValueHolder»contents:
WAValueHolder»contents
WAVisiblePresenterGuide»visitPresenter:
WAVisitor»start:
WAVisitor»visit:
WAXmlDocument»closeTag:
WAXmlDocument»destroy
WAXmlDocument»initializeWithStream:codec:
WAXmlDocument»openTag:attributes:closed:
WAXmlDocument»openTag:attributes:
WAXmlDocument»openTag:
WAXmlDocument»print:
WAXmlDocument»urlEncoder
WAXmlDocument»xmlEncoder
WAXmlEncoder»nextPutAll:
ZnSeasideServerAdaptorDelegate class»with:
ZnSeasideServerAdaptorDelegate»adaptor:
ZnSeasideServerAdaptorDelegate»adaptor
ZnSeasideServerAdaptorDelegate»handleRequest:
ZnZincServerAdaptor»basicStart
ZnZincServerAdaptor»configureDelegate
ZnZincServerAdaptor»configureServerForBinaryReading
ZnZincServerAdaptor»defaultCodec
ZnZincServerAdaptor»defaultDelegate
ZnZincServerAdaptor»defaultZnServer
ZnZincServerAdaptor»isStopped
ZnZincServerAdaptor»requestAddressFor:
ZnZincServerAdaptor»requestBodyFor:
ZnZincServerAdaptor»requestCookiesFor:
ZnZincServerAdaptor»requestFieldsFor:
ZnZincServerAdaptor»requestHeadersFor:
ZnZincServerAdaptor»requestMethodFor:
ZnZincServerAdaptor»requestUrlFor:
ZnZincServerAdaptor»requestVersionFor:
ZnZincServerAdaptor»responseFrom:
ZnZincServerAdaptor»server

Résumé

Un environnement d'exécution est l'ensemble des éléments logiciels qui représentent une application pendant son exécution. Les environnements d'exécution doivent être adaptables à différents contextes. Par exemple, nous pouvons envisager d'étendre un language de programmation pour améliorer la productivité des developpeurs. Aussi, nous pouvons envisager de réduire la consommation memoire des applications de manière transparente afin de les adapter à certaines contraintes d'exécution *e.g.*, des réseaux lents ou de la mémoire limités.

Nous proposons Espell, une infrastructure pour la virtualisation d'environnement d'execution de langages orienté-objets haut-niveau. Espell fournit une infrastructure généraliste pour la manipulation d'environnements d'exécution. Une représentation de 'premier-ordre' de l'environnement d'exécution orienté-objet fournit une interface haut-niveau qui permet la manipulation de ces environnements. Un hyperviseur utilise cette représentation de 'premier-ordre' et le manipule soit directement, soit en y exécutant des expressions arbitraires. Nous montrons que cet infrastructure supporte le *bootstrapping* (*i.e.*, l'amorçage ou initialisation circulaire) des languages et le *tailoring* (*i.e.*, la construction sur-mesure ou 'taille') d'environnement d'exécution. En utilisant l'amorçage nous initialisons un language orienté-objet haut-niveau qui est auto-décrit. Un langage amorcé profite des ses propres abstractions se montrant donc plus simple à étendre. La taille d'environnements d'exécution est une technique qui génère une application spécialisé en extrayant seulement le code utilisé pendant l'exécution d'un programme. Une application taillée inclut seulement les classes et méthodes qu'elle nécessite, et évite que des librairies et des frameworks externes surchargent inutilement la base de code.

Abstract

An application runtime is the set of software elements that represent an application during its execution. Application runtimes should be adaptable to different contexts. For example, on one hand we can think about extending a programming language to enhance the developers' productivity. On the other hand we can also think about transparently reducing the memory footprint of applications to make them fit in constrained resource scenarios *e.g.*, low networks or limited memory availability.

We propose Espell, a virtualization infrastructure for object-oriented high-level language runtimes. Espell provides a general purpose infrastructure to manipulate object-oriented runtimes in different situations. A first-class representation of an object-oriented runtime provides a high-level API for the manipulation of such runtime. A hypervisor uses this first-class object and manipulates it either directly or by executing arbitrary expressions into it. We show with our prototype that this infrastructure supports language *bootstrapping* and application runtime *tailoring*. Using bootstrapping we describe an object-oriented high-level language initialization in terms of itself. A bootstrapped language takes advantage of its own abstractions and is easier to extend. With application runtime tailoring we generate specialized applications by extracting the elements of a program that are used during execution. A tailored application encompasses only the classes and methods it needs and avoids the code bloat that appears from the usage of third-party libraries and frameworks.

Mots clé: application runtimes, object-oriented, virtualization, bootstrapping, tailoring.