N° d'ordre : 41342

THESE

présentée en vue
d'obtenir le grade de

DOCTEUR

en

Spécialité : informatique

par

Nikolaos Papoulias

**DOCTORAT DELIVRE CONJOINTEMENT
PAR MINES DOUAI ET L'UNIVERSITE DE LILLE 1**

Titre de la thèse :

Remote Debugging and Reflection in Resource Constrained Devices

Soutenue le 19/12/2013 à 10h devant le jury d'examen :

Président	*Roel WUYTS (Professeur – Université de Leuven)*
Directeur de thèse	*Stéphane DUCASSE (Directeur de recherche – INRIA Lille)*
Rapporteur	*Marianne HUCHARD (Professeur – Université Montpellier 2)*
Rapporteur	*Alain PLANTEC (Maître-Conf-HDR – Université de Bretagne Occ.)*
Examinateur	*Serge STINCKWICH (Maître-Conf – Université de Caen)*
co-Encadrant	*Noury BOURAQADI (Maître-Assistant – Mines de Douai)*
co-Encadrant	*Marcus DENKER (Chargé de recherche – INRIA Lille)*
co-Encadrant	*Luc FABRESSE (Maître-Assistant – Mines de Douai)*

Laboratoire(s) d'accueil : Dépt. IA, Mines Douai + RMoD INRIA Lille Nord de France

Ecole Doctorale SPI 072 (Lille I, Lille III, Artois, ULCO, UVHC, Centrale Lille)

This PhD dissertation was supported by
*"**Region Nord-Pas de Calais**"*

Contents

Acknowledgments

I would like to thank my supervisors for their support throughout the duration of my thesis. I truly believe that computer scientists should always remain - not only children - but also engineers at heart. This is the main reason why it has always been a pleasure working with you. Thank you Stef, thank you Noury, thank you Luc, thank you Marcus. This three-year adventure would also have not been possible without the kind support of the Nord-Pas-De-Calais region and EMD who funded my research. I am really greatfull Ch'ti, thank you.

Working at EMD and INRIA was such a full-feeling experience partly due to numerous colleagues and friends in both institutions with whom I shared many brainstrorming sessions and a lot of caffeine. Most notably: Mariano, Guillermo, Santiago, Ben and Nico but also Martin, Esteban, Igor, Damien, Jean-Baptiste and Camillo. I owe a lot of gratitude to Antoine Chammas who was always eager to help me in my numerous adventures with the french public sector ;) I am also deeply grateful to my family and friends back in Greece, who were constantly supportive and - as always - offered valuable advice in times of stress.

Finally I' d like to thank the miners of Douai for their rich history of struggle against social injustice. The goddess Europa, from whom a well known continent takes its name. But above all Adamantia - lord Byron's daughter - for her Bernouli sequence algorithm.

Εσύ ! Η κόρη Ποιητή και θυγατέρα Επαναστάτη.

Του Τρίτου άλματος Μητέρα, βύζαξε μας !

Κυρά στ' Αντικύθηρα , Δαντελωτή μου Αγάπη,

στο Μεσολόγγι Αθάνατος λογίζεται ο Πατέρας.

Για την κόρη του Βύρωνα,
Αδαμαντία.

Abstract

Building software for devices that cannot locally support development tools can be challenging. These devices have either limited computing power to run an IDE (*e.g.,* smartphones), lack appropriate input/output interfaces (display, keyboard, mouse) for programming (*e.g.,* mobile robots) or are simply unreachable for local development (*e.g.,* cloud-servers). In these situations developers need appropriate infrastructure to remotely develop and debug applications.

Yet remote debugging solutions can prove awkward to use due to their distributed nature. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spend for re-deploying applications while fixing bugs or improving functionality [ZeroTurnAround 2011]. Moreover current solutions lack facilities that would otherwise be available in a local setting because its difficult to reproduce them remotely (*e.g.,* object-centric debugging [Ressia 2012b]). This fact can impact the amount of experimentation during a remote debugging session - compared to a local setting.

In this dissertation in order to overcome these issues we first identify four desirable properties that an ideal solution for remote debugging should exhibit, namely: *interactiveness, instrumentation, distribution and security*. Interactiveness is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (*i.e.,* without stopping the application). Instrumentation is the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Distribution is the ability of a debugging solution to adapt its framework while debugging a remote target. Finally security refers to the availability of prerequisites for authentication and access restriction.

Given these properties we propose Mercury, a remote debugging model and architecture for reflective OO languages. Mercury supports interactiveness through a mirror-based remote meta-level that is causally connected to its target, instrumentation through reflective intercession by reifying the underlying execution environment, distribution through an adaptable middleware and security by decomposing and authenticating access to reflective facilities. We validate our proposal through a prototype implementation in the Pharo programming language using a diverse experimental setting of multiple constraint devices. We exemplify remote debugging techniques supported by Mercury's properties, such as *remote agile debugging* and *remote object instrumentation* and show how these can solve in practice the problems we have identified.

Keywords: Remote Debugging, Reflection, Mirrors, Interactiveness, Instrumentation, Distribution, Security, Agile Development

Résumé

La construction de logiciels pour des appareils qui ne peuvent pas accueillir localement des outils de développement peut être difficile. Ces appareils soit ont une puissance de calcul trop limitée pour exécuter un IDE (par exemple, smartphones), ou manquent d' interfaces d'entrée / sortie appropriées (écran, clavier , souris) pour la programmation (par exemple, les robots mobiles) ou sont tout simplement inaccessibles pour des développements locaux (par exemple cloud - serveurs). Dans ces situations, les développeurs ont besoin d'une infrastructure appropriée pour développer et déboguer des applications distantes.

Des solutions de débogage à distance sont parfois délicates à utiliser en raison de leur nature distribuée. Les études empiriques nous montrent que, en moyenne 10,5 minutes par heure de codage (plus de cinq semaines de travail de 40 heures par an) sont passées pour le re-déploiement d'applications pour corriger les bugs ou améliorer leur fonctionnalité [ZeroTurnAround 2011]. En plus, les solutions courantes manquent des aménagements qui seraient autrement disponibles dans un contexte local, car c'est difficile de les reproduire à distance (par exemple débogage objet-centré [Ressia 2012b]). Cet état influe sur la quantité d' expérimentation au cours d'une session de débogage à distance - par rapport à un contexte local.

Dans cette thèse, afin de surmonter ces problèmes, nous identifions d'abord quatre propriétés désirables qu'une solution idéale pour le débogage à distance doit présenter : *l'interactivité, l'instrumentation, la distribution et la sécurité*. L'interactivité est la capacité d'une solution de débogage à distance de mise à jour incrémentale de toutes les parties d'une application sans perdre le contexte de d'exécution (sans arrêter l'application). L'instrumentation est l'aptitude d'une solution de modifier la sémantique d'un processus en cours en vue d'aider le débogage. La distribution est la capacité d'une solution de débogage à adapter son cadre alors que le débogage d'une cible à distance. Enfin la sécurité fait référence à la disponibilité de conditions préalables pour l'authentification et la restriction d'accès.

Compte tenu de ces propriétés, nous proposons Mercury, un modèle de débogage à distance et une architecture pour des langues réflexifs à objets. Mercury ouvre (1) l'interactivité grâce à un méta-niveau à distance miroir basé sur un lien de causalité avec sa cible, (2) l'instrumentation à travers une intercession réflective basée sur la réification de l'environnement d'exécution sous-jacent, (3) la distribution grâce à un middleware adaptable et (4) la sécurité par la décomposition et l'authentification de l'accès aux aspects réflexifs. Nous validons notre proposition à travers un prototype dans le langage de programmation Pharo à l'aide d'un cadre expérimental diversifié de multiples dispositifs contraints. Nous illustrons des techniques de débogage à distance supportées par les propriétés de Mercury, tels que le *débogage agile distant* et *l'instrumentation objet à distance* et montrons comment ils peuvent résoudre dans la pratique, les problèmes que nous avons identifiés.

Mots clés: Débogage à distance, Reflexion, Miroirs, Interactivité, Instrumentation, Distribution, Sécurité, Développement Agile

CHAPTER 1

Introduction

Contents

At a Glance

This chapter introduces the domain and the context of our research. We explain the problems regarding debugging in the context of resource constraint devices. We summarize our approach and our proposed solutions. Finally we present the main contributions of this dissertation and give an overview for its structure.

1.1 Context: The programming cycle for resource constraint devices

Software is rarely deployed on the same machine it was written or debugged on. This is even mandatory, when building software for devices with resource constraints (e.g. smartphones) or ones with no input/output interfaces (keyboard, mouse or screen) for development (e.g. robots). In these cases debugging can be challenging because the target machine can be very different from the development one. Although emulators can help in this case, they are not always available, and are often of limited accuracy, especially when sensory input or actuators are involved.

1.2 Problem: Debugging a resource constraint device

In these situations developers have to remotely debug the target machine. Remote debugging tools fall into two main categories: those that incorporate post-mortem analysis (such as logging) and those that externally observe the state and execution flow of a running process through dedicated tools (i.e *remote debuggers*).

In the case of post-mortem analysis and logging the developer relies on the verbosity of the log. If it is too verbose, the developer might be overwhelmed with the amount of data. Conversely, limited logging may lead to several debugging cycles just for collecting data that will hint on some specific defect. This is due to the static nature of logs. Finally the cycle of re-compilation and re-deployment is time consuming, which makes debugging even more awkward.

When using remote debuggers, having the ability to introspect and modify a live execution (without loosing the context) is a major advantage compared to evaluating static logs. The hypothesis that the developer forms for a possible solution is much more informed in this case. In a lot of cases by just being able to follow execution and introspect or set variables in the target itself, one can be almost certain for the validity of a possible solution. Remote debugging is the most sensible solution in situations where targeted devices (such as smartphones or cloud-based servers) have different hardware or environment settings than development machines.

1.3 Shortcomings of Existing Approaches

Ideally, in OO languages developers should be able to evolve every organizational module and properties of the target application while remote debugging. Moreover they should be able to halt and inspect the running program not only at specific locations in the source code but also on every semantical event that involves objects. The remote debugging solution should depend on middleware that is extendable and adaptable even at runtime, in order to address the different communication needs of different targets and finally it should provide security constraints for both the target and the development machine. Unfortunately there is no existing approach that meets all these criteria in a satisfactory way. Current solutions lack facilities that would otherwise be available in a local setting or suffer from re-deployment issues. This is true for debugging solutions of all major OO languages in current use today (Java (JPDA) [Oracle 2013b] [Oracle 2013a], C# (.NET Debugger) [Microsoft 2012b], C++ and Objective-C (through Gdb) [Richard Stallman 2003]) and also true for dynamic languages with *live programming* support (such as Smalltalk and its debugging model [LaLonde 1990]), taking also into account bleeding-edge technological achievements [ZeroTurnAround 2012] and very recent research results [Würthinger 2010] [Ressia 2012b].

Given these shortcomings the following **research questions** concerning remote debugging are addressed in this dissertation:

1. *What are the properties of an ideal remote debugging solution ?*
2. *Given these properties which model for remote debugging can exhibit them ?*
3. *What are the trade-offs between this ideal model and a real world implementation ?*

1.4 Our Solution in a Nutshell

Thesis statement*. An ideal remote debugging solution should support:* ***interactiveness*** *through a mirror-based remote meta-level that is causally connected to its target,* ***instrumentation*** *through reflective intercession by reifying the underlying execution environment,* ***distribution*** *through an adaptable middleware and* ***security*** *by decomposing and authenticating access to reflective facilities.*

We are proposing a mirror-based model and an infrastructure for remote debugging. Our solution exhibits four desirable properties that we have identified as important for remote debugging, namely: *interactiveness*, *instrumentation*, *distribution* and *security*. Interactiveness is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (i.e without stopping the application). Instrumentation is the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Distribution is the ability of a debugging solution to adapt its framework while debugging a remote target. Finally security refers to the availability of prerequisites for security mechanisms in a remote debugging solution, such as authentication and access restriction.

We proposed the Mercury model and an architecture for remote debugging in reflective languages. Mercury supports interactiveness through a causal connection between the meta-level running on the developer machine, and the application to debug (the base-level) on the target device. The two levels are connected both computationally and structurally. It supports instrumentation through the reification of the underlying execution environment (virtual-machine) inside the run-time environment of the target (as an interpreter). Distribution is supported through an adaptable middleware [David 2002]. Finally it supports security in a remote debugging setting by organizing its reflective facilities into two different access groups for - respectively - introspection and intercession. We validated the applicability of our proposal through a prototype implementation in the Pharo language, and we illustrated it using concrete examples and a case study.

1.5 Contributions

The main contributions of this dissertation are:

1. The identification of four desirable properties than an ideal solution for remote debugging should exhibit, namely: *interactiveness*, *instrumentation*, *distribution* and *security*.
2. The definition of a model for remote debugging (*Mercury*) that exhibits these desirable properties.
3. A solution to the problem of Reflective-Data [Maes 1987b] in the context of mirrors [Bracha 2004] and its validation through a language prototype (*MetaTalk*[1]).
4. The reification of a previously illustrative notion (that of the *reflectogram* [Tanter 2003]) as an entity that controls the behavior of the meta-level at runtime.
5. A prototype implementation [2] of our model for remote debugging in the context of reflective languages.
6. The implementation of an adaptable middleware [David 2002] for supporting distribution under different communication contexts (*Seamless*)[3].
7. The implementation of a dedicated VM for Pharo (MetaStackVM) [4] that supports advanced intercession facilities.

1.6 Structure of the Dissertation

The dissertation is organized as follows:

Chapter 2 studies and provides definitions for the process of debugging. It identifies major properties and sub-properties of remote debugging solutions, that an ideal solution should exhibit. By using these properties state-of-the-art debugging solutions are evaluated and compared.

Chapter 3 provides definitions for reflection and remote reflection. It studies architectural alternatives for remote reflection. Finally it pinpoints open-issues regarding the use of reflection in the context of debugging.

[1]http://www.squeaksource.com/MetaTalk/
[2]http://ss3.gemstone.com/ss/Mercury-Prototype.html
[3]http://ss3.gemstone.com/ss/Seamless.html
[4]http://ss3.gemstone.com/ss/mSVM.html

Chapter 4 describes our solution to the problem of Reflective-Data [Maes 1987b] in the context of mirrors [Bracha 2004]. The model for a pluggable and state-full meta-kernel is presented which can be discarded when it is not being used (for example in-between debugging sessions).

Chapter 5 presents our solution for remote debugging. It details one by one the parts of our proposal that support the remote debugging properties which were identified in Chapter 2. Finally a comprehensive comparison of our solution with state-of-the-art is given.

Chapter 6 presents a prototype implementation of our proposed model for remote debugging and discusses engineering trade-offs that other implementors of our model should take into account.

Chapter 7 shows the intended usage of our model through working examples. Then two case-studies on *remote agile debugging* and *remote object instrumentation* using Mercury are presented. The case studies validate Mercury's properties in an experimental setting of three constraint devices, running real-world applications (smart-phone, tablet and remote server).

Chapter 8 concludes the dissertation by summarizing our work and presents future perspectives.

CHAPTER 2

Remote Debugging

Contents

At a Glance

In this chapter we study and provide definitions for the process of debugging. We then use these definitions to study different remote debugging approaches. Through this study we define four major properties of remote debugging solutions, namely: *interactiveness*, *instrumentation*, *distribution* and *security*. Finally, by using these properties we evaluate and compare existing solutions.

2.1 Debugging

While the term "debugging" itself is usually attributed to Grace Hopper creator of Cobol [Zeller 2005], debugging as a general diagnostic tool is applicable not only to programming languages but to any engineering process. The significance of debugging for software engineering in particular can be seen in scientific publications concerning effort estimation and project management. These studies support that on average, testing and debugging cover roughly 50 % percent of the development time [Beizer 1990] and that validation activities – such as debugging and verification – cover 50 % to 75 % of the total development cost [Brent Hailpern 2002].

In Figure 2.1 we present the three domains and the entities involved in software debugging. We will describe these domains and entities to give a working definition of debugging. Our definition depends loosely on those given by Zeller [Zeller 2005] and Sommerville [Sommerville 2001] respectively.

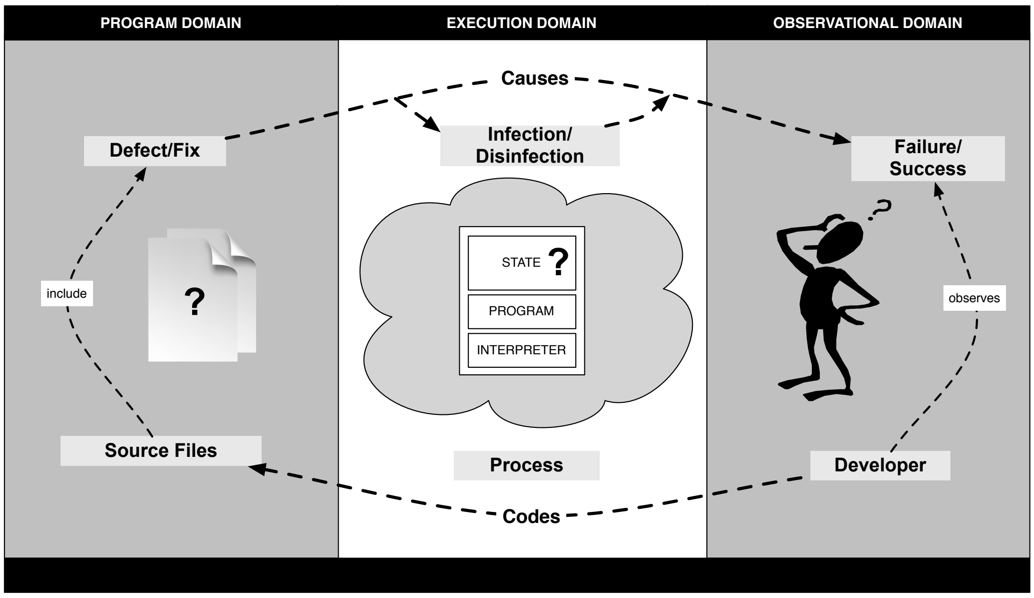


Figure 2.1: Domains and entities involved in the debugging process

The process of debugging involves three separate domains, namely the *program domain*, the *execution domain* and finally the *observational domain*.

The program domain includes program representations suitable for coding by a developer. These representations are usual textual models, but other models for coding do exist, as it is the case with visual programming. We designate these representations as source files.

The execution domain includes the representation of a running program (designated as process) in memory. A running process consists of a state component which includes its data and executional flow. A program component which includes the code produced through the compilation process of source files. And finally an interpreter component which designates the underlying execution mechanism of compiled code. The interpreter component can be a run-time software interpreter, a dedicated virtual-machine or an operating system scheduling machine-code.

The observational domain includes everything that is externally observable by a developer while a process is running. These observations can include the process' textual output to a standard device, messages from the operating system and in general *any side-effect* the running process has on the outside world.

During the programming cycle, debugging is usually initiated through direct observation by the programmer of a failure. We define a **failure** as an *unwanted executional behavior* in the observational domain. Conversely a **success** is the *observation of an execution in the observational domain without failures*.

A failure is caused by a series of infections in a process' state. An **infection** [Voas 1992] is an *unwanted computational state in the execution domain*. Conversely a **dis-infection** is *the absence of a previously existing infection*.

Similarly to a failure an infection has also an underlying cause. An infection is caused by a defect in a program's source files. A **defect** or bug [Beizer 1990, Humphrey 1999] is an *unintended mistake introduced by the programmer in the source representation of a program*. Conversely a **fix** is a *change in the source representation of a program that eliminates a defect*.

Given the above discussions we define debugging as follows:

Debugging *is a two phases process via which a programmer: i) relates a failure in the observational domain to a defect in the program's domain and ii) subsequently validates the elimination of a defect by applying a fix in the program's domain and relating it to a success in the observational domain.*

We call the first phase of debugging the **inference phase** where a defect or bug has to be inferred from an observed failure. While we designate the second phase as the **validation phase** where a possible fix is validated by an observed success.

Failures and defects as well as fixes and successes have a cause and effect relationship, which cannot be immediately inferred by the programmer. This is the inherent difficulty of the debugging process. For example a programmer may observe that a program crashed or misbehaved while trying to respond to a certain input. The cause though of this misbehavior may have originated several calls before the processing of the offending input itself by the program.

To infer a defect from a failure and devise a suitable fix for a success, both phases of debugging have to rely on *hypotheses and experimentations*. Zeller [Zeller 2005] draws the following analogy between the debugging process and the scientific method for the inference phase of debugging:

1. Observe a failure.

2. Invent an hypothesis as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make predictions.
4. Test the hypothesis by experiments and further observations.
5. If the experiment satisfies the predictions, refine the hypothesis.
6. If the experiment does not satisfy the predictions, create an alternate hypothesis.
7. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

A similar process can be devised for the validation phase of debugging where the developer makes an hypothesis over a possible fix. As we can see from steps 6 and 7 of the above process, *debugging is an intensive and iterative process*. This is the reason why debugging tools exist to assist the programmer and facilitate experimentation during both phases.

Debugging tools fall into two main categories: **a)** those that incorporate post-mortem analysis and **b)** those that externally observe the state and execution flow of a running process. It is the debugging tools of this second category that are usually designated as *debuggers*. We can define a debugger as follows:

Debugger *A debugger is an additional process of the execution domain used for run-time analysis. The debugger acts upon the process that is being debugged (*i.e.*, the debuggee), making a subset of the execution domain part of the observational domain.*

2.2 Remote Debugging

Software is rarely deployed on the same machine it was written or debugged on. This is even mandatory, when building software for devices with resource constraints (e.g. smartphones) or ones with no input/output interfaces (keyboard, mouse or screen) for development (e.g. robots). In these cases debugging can be challenging because the target machine can be very different from the development one. Although emulators can help in this case, they are not always available, and are often of limited accuracy, especially when sensory input or actuators are involved. Thus, developers have to use the target machine for debugging.

Remote Debugger *A remote debugger, is a debugger whose process runs on a different machine than that of the debuggee.*

A widely used local and remote debugger is the GNU debugger [Richard Stallman 2003] which supports the two phases of debugging (inference and validation) by providing the following facilities:

1. Start a program, specifying anything that might affect its behavior.

2. Make a program stop on specified conditions.
3. Examine what has happened, when a program has stopped.
4. Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

2.2.1 Remote Debugging Through Logging

Logging is the most prominent example of post-mortem analysis. Post-mortem analysis is the observation and reasoning process on a running program *after* its successful or failed execution. In terms of the notions that we introduced in the previous section we define logging as follows:

Logging *Logging is a post-mortem analysis technique which augments the observational domain with information from the execution domain via the injection of code in the program domain.*

Injecting code in the program domain can be done manually by the programmer, as in the case of *printf debugging* where the programmer manually intercepts program logic with logging methods that register information in the standard output. It can also be done semi-automatically through logging frameworks such as *Log4j* [Gupta 2007] and finally through compile-time facilities such as aspect frameworks like *AspectJ* [Kiczales 2001] which can hide this injection from the textual representation of a program and introduce it only in the final executable.

Concerning the debugging process logging can be used to expose *infections* and *disinfections* to the observational domain. By examining the logs produced during an execution the developer can devise informed hypothesis (using the information in the logs) about the causal connection between a failure and a defect. He can subsequently through the use of those logs incrementally validate a possible fix.

Figure 2.2 shows how the remote debugging process unfolds when it is achieved through logging. In the inference phase after having observed a failure the developer needs more information concerning the process state to invent, confirm or dispute an hypothesis. In step 1 (Coding and Compilation step of Figure 2.2) he injects his logging directives into the program's code. In step 2 the program is deployed and executed (step 3) in such a way so that the failure is reproduced. In step 4 the log is collected (from the standard output or elsewhere) and in step 5 the developer reasons upon the log and the observed execution trying to infer a defect from both the *observed failure* and the *possible infection* which is exposed through the logs.

Similarly in the validation phase after having applied a possible fix the programmer can use the logs to validate a disinfection on the program's state or to refine his fix.

Logging is said to be the most widespread debugging technique [Zeller 2005] since in its simplest form (of printf debugging) is accessible even to inexperienced developers and can have little to zero infrastructure requirements. The content however of the execution log is determined by decisions made at the coding and compilation step (step 1). Often,

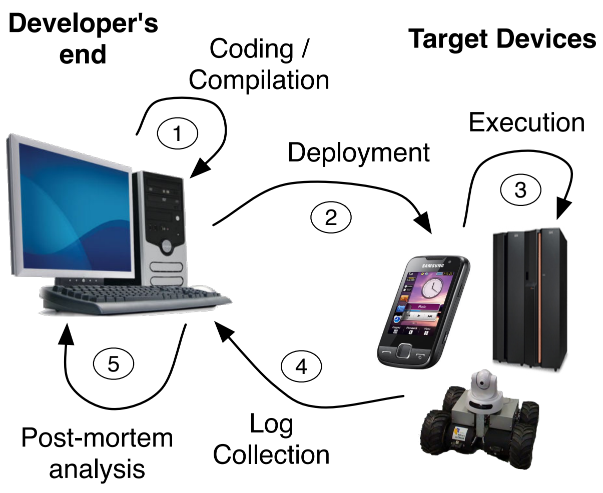


Figure 2.2: Debugging with *Post-Mortem* Analysis

developers don't know which information they need until the analysis stage. But, since the analysis is *post-mortem*, collecting missing information requires to go again through a whole cycle, after adapting the code or the compilation flags and options. On the other hand if the logging is too extensive the output will be cluttered making hard to distinguish useful information.

If the execution log is verbose enough but not cluttered the developer can form an hypothesis on what went wrong during the execution. To test this hypothesis the developer has to repeat again the process. He will develop a new version of the software, deploy it, run it on the target, and then re-collect the logs. Yet another post-mortem evaluation will confirm or refute the validity of the possible solution.

Review of this debugging solution. First, the developer relies on the verbosity of the log. If it is too verbose, the developer might be overwhelmed with the amount of data. Conversely, limited logging may lead to several debugging cycles just for collecting data that will hint on some specific defect. This is due to the static nature of logs. Finally the cycle for re-compilation and re-deployment is time consuming, which makes debugging even more awkward.

2.2.2 Using a Remote Debugger

In Figure 2.3 we show the different steps of a debugging process assisted by a remote debugger. In the inference phase after having observed a failure the developer needs more information concerning the process state to invent, confirm or dispute an hypothesis. In order to do that through a remote debugger *he has to set up his environment accordingly*. For example when executing the process in step 2 he has to instruct the underlying execution mechanism (designated as interpreter) to allow debugging and also deploy the debugging support.

During step 3 the execution of the target process is *interruptible*. This *interruption*

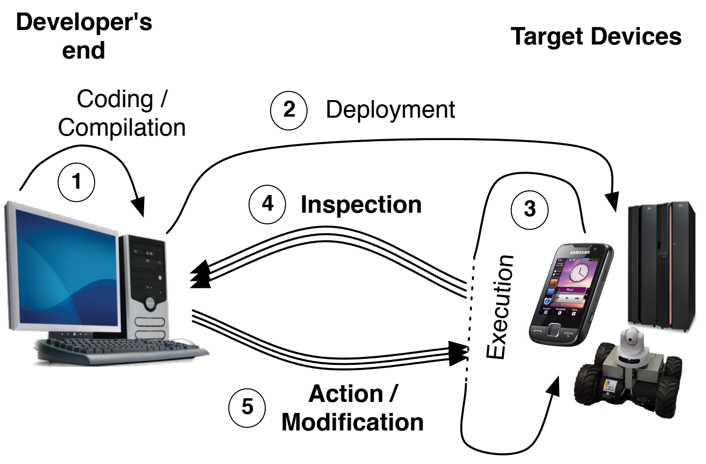


Figure 2.3: Remote debugging and interaction with a live execution

is either user-generated (the developer chooses to freeze the execution to inspect it) or is based on pre-determined execution events, like the raise of an exception. This is in contrast with Figure 2.2 where execution proceeds uninterrupted until it is finished, at which point the execution context is lost. Steps 4 and 5 represent the debugging loop. This loop takes place at execution time and in the presence of the execution context which can be inspected and modified. Step 4 represents the inspection phase, where information about the current execution context is retrieved from the target process. While in step 5 we depict the modification phase where the developer can a) provide further user-generated interruption points (breakpoints, watchpoints etc.) b) alter execution and its state (step, proceed, change the values of variables) and finally c) incrementally update parts of the code deployed in step 2 (save-and-continue, hot-code-swapping). Several loops can occur during the execution depending on the developers' actions (step, proceed, user-generated interruptions) and on execution events (exceptions, errors, etc.).

Similarly in the validation phase after having applied a possible fix the programmer can use this augmented observational domain (which now includes part of the execution domain that can be observed and changed through the remote debugger) to validate a disinfection on the program's state or to *incrementally* refine his fix.

Review of this debugging process. Having the ability to introspect and modify a live execution (without loosing the context) is a major advantage compared to evaluating static logs. The hypothesis that the developer forms for a possible solution is much more informed in this case. Indeed in a lot of cases by being able to follow execution and introspect or set variables in the target itself, one can be almost certain for the validity of a possible solution.

We thus conclude that using remote debuggers is the most sensible solution in situations where targeted devices (such as smartphones or cloud-based servers) have different hardware or environment settings than development machines.

2.3 Requirements for Remote Debugging Solutions

Despite their applicability for our problem domain, remote debuggers can also prove awkward to use due to their distributed nature. As an example, we can consider the cost of re-deployments in-between remote debugging sessions. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spent for re-deploying applications while fixing bugs or improving functionality [ZeroTurnAround 2011]. This means that the specific facilities that a remote debugging solution offers (e.g. for incremental updating or experimentation) during a remote debugging session can have a huge impact on productivity.

In this section, we present four desirable properties that an *ideal* remote debugging solution should exhibit. These properties are: *interactiveness*, *instrumentation*, *distribution*, and *security*. We introduce and discuss each property based on a typical software stack for supporting remote debugging, as depicted in Figure 2.4.

In Figure 2.4 we can see that the target device (on the right) that runs the debugged application must provide a middleware for communication and a run-time debugging support for examining processes, the execution stack, the system's organization, introspection of instance and local variables, etc. On the other hand, the developer machine must provide a middleware, debugging tools, and also a model of the running application that describes the application running on the target (*e.g.,* source code or breakpoints).

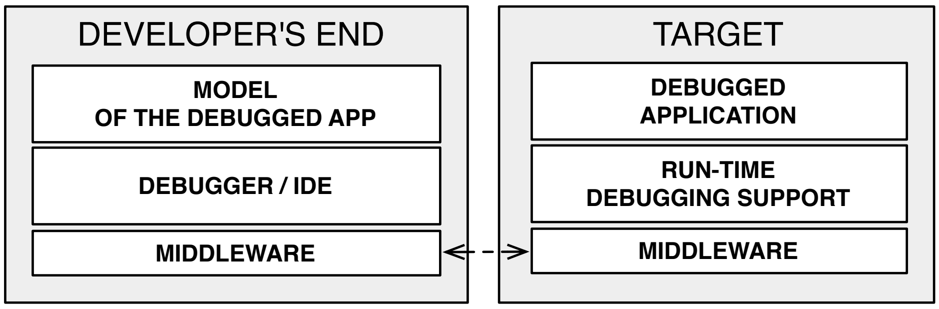


Figure 2.4: Software Entities Involved in Remote Debugging.

2.3.1 Interactiveness

We define interactiveness as the ability to dynamically inspect and change the target application code. By *dynamically* we mean that inspections and changes can be performed while the application is running.

In Figure 2.4, there is an implicit relationship between the model of the debugged application (on the developer's end), and the state of the debugged application (on the target). This relationship can be either *static* or *dynamic*, depending on whether a change in either one of them updates the other or not. When a remote debugging solution is interactive, this relationship is dynamic.

The fact that the target application does not need to be restarted in order to be debugged or evolved allows developers to:

- Track the origins of bugs and fix them without losing the execution context.
- Fix heisenbugs [Gray 1986], *i.e.,* bugs that are not easily reproducible.
- Increase productivity while debugging applications. Especially in situations where a full re-deployment is involved or in applications with a long startup time.
- Fix flaws [Zeller 2005] from within the debugger. Flaws are architectural bugs that are not associated with a specific location in the source file and require an architectural update (removing or adding new code) in order to be addressed.
- Debug critical applications (e.g server side applications) that cannot be restarted.

Ideally, in OO languages developers should be able to evolve every organizational module and properties of the target application while debugging. These changes should include:

Adding/Removing Packages The ability to introduce new packages (i.e named groups of classes) and remove existing ones.

Adding/Removing Classes The ability to introduce new classes and to remove existing ones.

Adding/Removing Superclasses The ability to edit a class hierarchy.

Adding/Removing Methods The ability to introduce new methods to a class and to edit or remove existing ones.

Adding/Removing Fields The ability to introduce new fields to a class or remove existing ones.

2.3.2 Instrumentation

With the term *instrumentation* we refer to the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Instrumentation is the underlying mechanism through which breakpoints and watchpoints are implemented. A debugging solution *instruments* the running process in order to halt at specific locations in the code, or when specific events occur (such as variable access) to either return control to the debugging environment or to perform predetermined checks and actions (such as breakpoint conditions).

Ideally in OO languages developers should be able to halt and inspect the running program both at specific locations in the source code and on specific semantical events that involve objects. In literature these events are referred to as *dynamic reification categories* [Redmond 2000]. These categories are a set of operations that can be thought of as *events* which are required for object execution [McAffer 1995] [Ressia 2010].

Taking into account these semantic events, instrumentation categories for debugging should at least include:

Statement Execution The ability to halt at a specific statement or line in the source code.

Method Execution The ability to halt at a specific method in the source code upon entry.

Class Instantiation The ability to halt at object creation of specific classes.

Class Field Read The ability to halt when a specific field of *any* instance of a class is read.

Class Field Write The ability to halt when a specific field of *any* instance of a class is written.

Object Read The ability to halt at *any* read attempt on a specific object.

Object Write The ability to halt at *any* write attempt on a specific object.

Object Send The ability to halt at any message send *from* a specific object.

Object Receive The ability to halt at any message send *to* a specific object.

Object as Argument The ability to halt whenever a specific object is passed as an argument.

Object Stored The ability to halt whenever a new reference to a specific object is stored.

Object Interaction The ability to halt whenever two specific objects interact in any way. This is a composite category as defined in [Ressia 2010]. It can be seen as a composition of object receive, send and argument categories.

2.3.3 Distribution

As seen in Figure 2.4 remote debugging requires a communication middleware. Ideally a debugging solution should depend on middleware that is extendable and adaptable even at runtime, in order to address the different communication needs of different targets. For example targets with different resources (memory, processing power, bandwidth) may require different serialization policies. While others such as server applications may require different security policies when they are debugged through an open network (see also subsection 2.3.4).

We can distinguish the following four categories of distribution support for debugging solutions, in ascending order of adaptability:

No-Distribution (-) The debugging solution does not support remote debugging.

Fixed-Middleware (+) The debugging solution supports remote debugging via a dedicated and fixed protocol which cannot be easily extended.

Extensible Middleware (++) The debugging solution supports remote debugging via a general solution for distributed computing (such as an object request broker) which can be extended, such as CORBA or DCOM.

Adaptable Middleware (+++) The debugging solution supports remote debugging via a general solution for distributed computing which can be extended and adapted at runtime [David 2002].

2.3.4 Security

During the development phase a debugging target needs to be accessible through the network to be debugged. This fact raises a security concern since debugging by its very purpose is a process demanding full access on the target's end. In cases where the target is accessible through an open network even during development (as in the case of cloud computing) a debugging solution should at least support *authentication* on the target's side.

Moreover on the developer's end, the debugging solution may be integrated inside an IDE with support for installing third-party plugins from possibly untrusted sources. For these cases a remote debugging solution should provide support for a security solution that is able to grant different access rights to different processes that want to use its facilities. As an example we can consider a third-party plugin in the IDE on the developer's side that handles process inspection. There is no need for this plugin to be granted access to the incremental updating facilities of the target.

We can distinguish the following *orthogonal* sub-properties of security in the context of debugging:

Internal (+) The debugging solution itself has security provisions.

External (+) Other frameworks or technologies are used (or can be used) in conjunction with the debugging framework in order to secure the session.

Target-Side (+) The debugging solution supports authentication on the target side.

Client-Side (+) The debugging solution supports access restrictions (for different processes or threads) on the client side.

An ideal solution should support all four of them (++++).

2.4 Evaluation of Existing Solutions

We now study existing debugging solutions of major OO languages in current use today (Java (JPDA) [Oracle 2013b] [Oracle 2013a], C# (.NET Debugger) [Microsoft 2012b], C++ and Objective-C (through Gdb) [Richard Stallman 2003]) as well as dynamic languages with *live programming* support (such as Smalltalk and its debugging model [LaLonde 1990]), taking also into account bleeding-edge technological achievements [ZeroTurnAround 2012] and very recent research results [Würthinger 2010] [Ressia 2012b].

2.4.1 JPDA

Java's debugging framework stack is JPDA [Oracle 2013b] and it consists of a mirror interface (JDI) [Oracle 2013a], [Bracha 2004], a communications protocol (JDWP) and the debugging support on the target as part of the virtual-machine's infrastructure (JVM TI). The application on the target machine must be specifically run with debugging support from the VM (the JVM TI) for any interaction between the client and the target to take place. JPDA does not provide facilities to interactively update the target other then the hot-swapping of pre-existing methods. The communication stress is handled by the low-level debugging communication protocol (JDWP), whose specification is statically defined. There are no security or authentication provisions in JPDA itself, but there exist general solutions for Java outside the framework for securing access to the target side (*e.g.,* in mobile platforms such as android).

2.4.2 JRebel and DCE

The DCE VM [Würthinger 2010] and Jrebel [ZeroTurnAround 2012] are both modifications for the Java virtual machine that support redefinition of loaded classes at runtime. Although these modifications of the underlying VM are not a solution for debugging themselves, they do provide incremental updating facilities for remote targets. These modifications if used in conjunction with the JPDA framework can support the property of instrumentation that we described in Section 2.3.

2.4.3 .NET

As with Java, the main remote debugging solution for .NET provided through visual studio [Microsoft 2012b] pre-purposes a dedicated debugging deployment. In the developer's end the model of the running application is again static, with the developer being responsible for providing the right sources and configuration files. In the case of .NET though the debugger can attach to a running remote process without loosing the context, provided that the static model for the application is available. Although the model in the developer's end is static, a limited form of updating is provided in the form of *edit-and-continue* [Microsoft 2012c] of pre-existing methods. There is no support for incremental updating of the target application with new packages, classes or methods. In terms of security the remote debugging solution for .NET integrates authentication mechanisms for both ends of the communication [Microsoft 2012a].

2.4.4 GDB

For Obj-C remote debugging is provided through the gnu-debugger [Richard Stallman 2003]. Gdb uses a dedicated process on the target machine called the *gdb-server* to attach to running processes. For full debugging support though the deployed application has to be specifically compiled and deployed with debugging meta-information embedded on the executable which cannot be discarded without re-deployment and loss of the running context. The model for the application on the developer's end is static and depends on the availability of source files. Gdb supports a limited form of updating through an *edit-and-continue* process of pre-existing methods by patching the executable on memory [Richard Stallman 2003]. There are no built-in provisions for security on the debugging solution itself. The communication scheme is statically defined through the gdb/mi (gdb machine interface) [Richard Stallman 2003].

2.4.5 Smalltalk

The most prominent example of an interactive debugger is the Smalltalk debugger [LaLonde 1990]. In Smalltalk the execution context after a failure is never lost since through reflection the debugger can readily be spawned as a separate process and access the environments' reifications for: *processes, exceptions, contexts* etc. Moreover it supports incremental updating in such a way that introducing new behavior through the debugger is not only possible but is actually advised [Black 2009]. Indeed incremental updating through debugging encourages and supports agile development processes, and more specifically Test Driven Development (TDD) [Abacus 2005]. In addition both the debugging and the reflecting facilities of Smalltalk are extensible. On the one hand the debugger model is written itself in Smalltalk. On the other hand the Smalltalk MOP is readily editable from within the system itself. Illustrative examples of MOP extensions in Smalltalk are given from Rivard in [Rivard 1996].

2.4.6 Bifrost

Finally in Smalltalk supporting advanced debugging techniques through instrumentation is illustrated in the Bifrost reflection framework [Ressia 2010] and through object-centric debugging [Ressia 2012b]. Bifrost is an extension to the Smalltalk MOP that relies on explicit meta-objects to provide sub-method [Denker 2007] and partial behavioral reflection [Tanter 2003]. Bifrost is implemented through dynamic re-compilation of methods. Method invocations are intercepted using the *reflective method* abstraction [Marschall 2006] and are subsequently recompiled using AST meta-objects that control the generated bytecode. With Bifrost intercession techniques such as the explicit interception of variable access, is made available at the instance level.

2.4.7 Comparison

In this Section we compare state-of-the-art debugging solutions in terms of *interactiveness, instrumentation, distribution and security*.

2.4.7.1 Interactiveness

In Table 2.1 we do a comparison in terms of interactiveness and its sub-properties as there were defined in Section 2.3:

As we see in Table 2.1 debugging environments of mainstream OO languages (JPDA, .Net Debugger, Gdb) do not support interactiveness with the exception of a *save-and-continue* facility for pre-existing methods. In the case of Gdb method hotswapping can lead to inconsistencies [Zeller 2005] since it is supported through memory patching, which is a blind process that replaces execution instructions in memory, without knowledge of the underlying semantics of the language. In the Java world recent developments (through

	JPDA	.NET	GDB	DCE	JREBEL	ST-80	BIFROST
Add/Rem Packages	×	×	×	✓	✓	✓	✓
Add/Rem Classes	×	×	×	✓	✓	✓	✓
Add/Rem IVs	×	×	×	✓	✓	✓	✓
Add/Rem Methods	×	×	×	✓	✓	✓	✓
Method (Body) HotSwapping	✓	✓	✓	✓	✓	✓	✓
Hierarchy Editing	×	×	×	✓	✓	✓	✓

Table 2.1: Interactiveness evaluation on state-of-the-art debugging solutions

Jrebel and DCE) provide full support for interactiveness as does Smalltalk and its extension Bifrost.

2.4.7.2 Instrumentation

In Table 2.2 we do a comparison in terms of instrumentation and its sub-properties as they were defined in Section 2.3. We have also included a last category marked as *condition/action* that describes whether in all instrumentation events the debugging solution can support user-generated checks and code in order to provide a more fine-grain control. As an example we can consider a conditional breakpoint that is able to execute user specified actions when triggered.

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST
Method Execution	✓	✓	✓	✓	✓	✓	✓
Statement Execution	✓	✓	✓	✓	✓	✓	✓
Field Read	✓	×	×	✓	✓	×	✓
Field Write	✓	×	×	✓	✓	×	✓
Object Read	×	✓	✓	×	×	×	✓
Object Write	×	✓	✓	×	×	×	✓
Object Send	×	×	×	×	×	×	✓
Object Receive	×	×	×	×	×	×	✓
Object as Argument	×	×	×	×	×	×	✓
Object Creation	×	×	×	×	×	×	✓
Object Interaction	×	×	×	×	×	×	✓
Object Stored	×	×	×	×	×	×	×
Condition/Action	×	×	✓	×	×	✓	✓

Table 2.2: Instrumentation evaluation on state-of-the-art debugging solutions

As we can see from our comparison, Bifrost is the front-runner of instrumentation with all other solutions supporting only plain breakpoints and watchpoints. Bifrost lacks an *Object Stored* event which is useful for following an object's reference propagation and counting. Finally both Bifrost and Gdb provide support for both conditions and actions on instrumentation events.

2.4.7.3 Distribution

In Table 2.3 we do a comparison in terms of distribution. Solutions are marked with - for not supporting distribution, + for supporting distribution through a fixed-middleware, ++ for an extensible middleware and +++ for an adaptable middleware.

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST
Distribution	+	++	+	+	+	-	-

Table 2.3: Distribution evaluation on state-of-the-art debugging solutions

As we can see in Table 5.4 no solution supports an adaptable middleware. The .NET debugging framework leads the comparison using a general purpose and extensible communication solution (DCOM) [Microsoft 2013]. We should note here that in the case of Smalltalk (which does not support the property of distribution), there were some efforts in the past to support remote development (including debugging) in Cincom Smalltalk, which were discontinued.

2.4.7.4 Security

In Table 2.4 we do a comparison in terms of security support while debugging as was described in Section 2.3:

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST
Built-in	×	✓	×	×	×	×	×
External	✓	✓	✓	✓	✓	×	×
Target-Side	✓	✓	✓	✓	✓	×	×
Developer-Side	✓	✓	×	✓	✓	×	×

Table 2.4: Security evaluation on state-of-the-art debugging solutions

As we can see in Table 2.4 only the .NET debugging framework has build-in provisions for security [Microsoft 2012a] for both the target and the developer side. In the Java world though (JPDA, DCE, JREBEL) there are other frameworks that are used in conjunction with JPDA in order to secure the debugging session such as the Java Security Manager [Oracle 2013c]. Gdb specifically warns developers not to use its remote debugging facilities in public networks [Richard Stallman 2003] and has no built-in provisions for access restrictions in the client side either. In this case the developer can only resort to external solutions such as a firewall or a VPN. For Smalltalk as far as the local debugging scenario is concerned (i.e no support for distribution) there are no security provisions.

2.4.7.5 Comparison overview

In Table 2.5 we present an overview of our comparison in terms of all properties that were described in Section 2.3:

Property	JPDA	.NET	GDB	DCE	JREBEL	SMALLTALK	BIFROST
Interactiveness	**+ (1/6)**	**+ (1/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**
Instrumentation	**+ (4/13)**	**+ (4/13)**	**+ (5/13)**	**+ (4/13)**	**+ (4/13)**	**+ (3/13)**	**+++ (12/13)**
Distribution	**+ (fixed)**	**++ (extensible)**	**+ (fixed)**	**+ (fixed)**	**+ (fixed)**	**- (no)**	**- (no)**
Security	**+++ (3/4)**	**++++ (4/4)**	**++ (2/4)**	**+++ (3/4)**	**+++ (3/4)**	**- (0/4)**	**- (0/4)**

Table 2.5: Summary comparison of state-of-the-art debugging solutions

As we can see from Table 2.5 debugging solutions based on reflection (such as Smalltalk and Bifrost in the local scenario) offer the most complete solutions in terms of interactiveness and instrumentation, but lack support for distribution and security. On the other hand solutions that do support these properties such as debugging environments of mainstream OO languages (JPDA, .Net Debugger, Gdb) and their extensions (Jrebel, DCE) lack support for either interactiveness or instrumentation (or in some cases both). There is no solution that meets all our criteria in a satisfactory way.

2.5 Summary

In this chapter we provided definitions for the processes of debugging and remote debugging as well as the notions that they involve. We distinguished remote debugging approaches into those that incorporate post-mortem analysis (such as logging) and those that make use of dedicated remote debugging frameworks that allow live inspection of the running process. We concluded that using remote debuggers is the most sensible solution in situations where targeted devices (such as smartphones or cloud-based servers) have different hardware or environment settings than development machines. We then defined four major properties of remote debugging solutions, namely: *interactiveness*, *instrumentation*, *distribution* and *security* as well as their sub-properties. Interactiveness is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (i.e without stopping the application). Instrumentation is the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Distribution is the ability of a debugging solution to adapt its framework while debugging a remote target. Finally security refers to the availability of prerequisites for security mechanisms in a remote debugging solution, such as authentication and access restriction. Then by using these properties we evaluated and compared state-of-the-art debugging solutions and concluded that none of them meets all of our criteria.

CHAPTER 3

Reflection for Remote Debugging: Architectural Alternatives

Contents

At a Glance

In this chapter we study and provide definitions for reflection and remote reflection. We use these definitions in order to assess the use of reflection for remote debugging. We study different design patterns that facilitate remote reflection, namely: *the remote proxy, the remote facade and mirrors* and discuss their strengths and shortcomings. Finally we pinpoint some open-issues regarding reflection in the context of debugging.

3.1 Reflection

P. Maes has proposed in the first chapter of her thesis [Maes 1987a], precise definitions to clearly characterize meta-programming and reflection. We refer here to these definitions and illustrate them in Figures 3.1 and 3.2:

- A *computational system* is something that *reasons* about and *acts* upon some part of the world, called the *domain* of the system.
- A computational system may also be *causally connected* to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other.
- A *meta-system* is a computational system that has as its domain another computational system, called its *object-system*. [...] A meta-system has a representation of its object-system in its data. Its program specifies *meta-computation* about the object-system and is therefore called a *meta-program*.

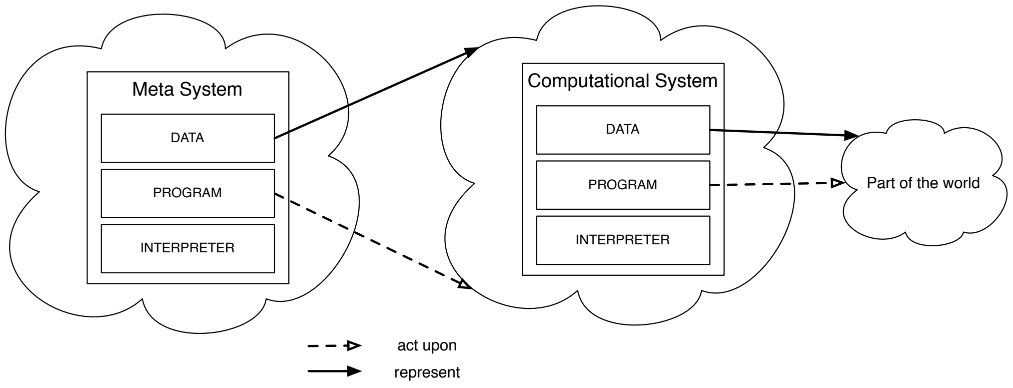


Figure 3.1: Relationships between a meta-system, a computational system and its domain

Reflection describes the ability of a system to reason and act upon itself. P. Maes [Maes 1987a] defines reflection in terms of meta-systems as follows:

- A *reflective system* is a causally connected meta-system that has as object-system itself. The data of a reflective system contain, besides the representation of some part of the external world, also a causally connected representation of itself, called *self-representation* of the system. [...] When a system is reasoning or acting upon itself, we speak of *reflective computation*. (see Figure 3.2).

It can be argued [Bracha 2010] that reflection is an inherited ability of the Von Neuman model of computation [von Neumann 1945]. This is a direct consequence of the fact that both code and data are being stored in computer memory as data (code/data duality). This gives the ability to a running process to manipulate its own code.

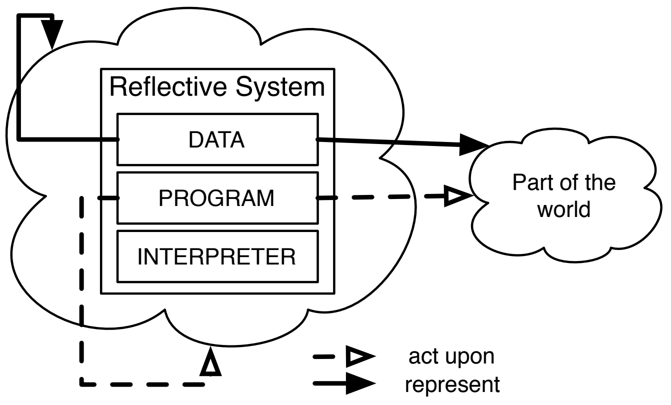


Figure 3.2: A reflective system

The notion itself however of reflection was formally introduced to programming language literature by Brian Cantwell Smith in 1982 (by means of the programming language 3-LISP [Smith 1982]). Smalltalk was to follow incorporating reflection to its design [Ingalls 1983].

Reflective facilities We now provide definitions and examples for different aspects of reflective systems combining compatible but different approaches in related literature: [Maes 1987b], [Ferber 1989], [Bracha 2004].

In OO reflective systems, reflection is concretized using a MOP (*Meta-Object Protocol*). A meta-object is a regular object that describes, reflects or defines the behavior of a notion of the language in question. The process of materialization of a notion of a language (such as an object, a class, a context or a method) as an object inside the language itself is called *reification*. The Smalltalk MOP which we will use to illustrate our definitions, is thoroughly described in [Rivard 1996].

Structural Reflection refers to the ability of a program to explicitly query and alter its internal state. Depending on which of the two operations (read or write) we are using we can further distinguish structural reflection into introspection and self-modification.

Introspection Query/Read *e.g.*

```
Object>>instVarAt: "answer an indexed variable value"
Object>>class "answer the receiver's class"
Object>>identityHash "answer the receiver's identity"
ProtoObject>>pointersTo "answer all objects in the system that hold a pointer to the receiver"
Behaviour>>allSelectors "answer all selectors understood by the receiver"
```

Self-Modification Alter/Write *e.g.*

```
Object>>instVarAt:put: "store a value into an indexed variable in the receiver"
Object>>become: "swap the pointer between the receiver and the argument"
Behaviour>>addSelectorSilently:withMethod: "add a method to the receiver"
Behaviour>>superclass: "change the superclass of the receiver"
Behaviour>>adoptInstance: "change the class of the argument to the receiver"
```

Behavioral Reflection refers to the ability of a program to dynamically execute code, alter its compilation process, and implicitly (on specific executional events) alter its semantics. Behavioral reflection can be sub-categorized into dynamic-execution facilities and intercession facilities. The latter can be further sub-divided into syntactic intercession (also known as compile-time reflection) and semantic intercession.

Dynamic Execution Execute *e.g.*

```
ProtoObject>>withArgs:executeMethod: "execute a method against the receiver"
Compiler>>evaluate: "compile and evaluate a string"
Object>>perform:withArguments: "send a message with a specific selector to the receiver"
```

Intercession

Syntactic Intercession

```
Compiler>>compile:in:notifying:ifFail: "compile a string"
"through overriding, example use: domain specific languages"
```

Semantic Intercession

```
    Object>>doesNotUnderstand: "handle a message not understood by the receiver"
    ProtoObject>>cannotInterpret: "handle a message when a nil method dictionary is
encountered"
```

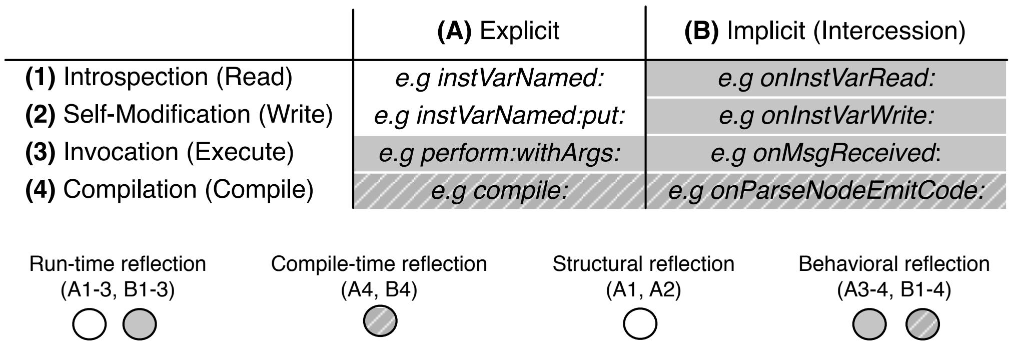
We summarize these definitions on Figure 3.3 where we present reflection in two dimensions. The horizontal dimension is the invocation dimension (i.e whether reflection is invoked explicitly in the program code, or implicitly by the execution environment) and the second is the action dimension (i.e distinguishing between different categories of reflective actions: read, write, execute and compile).

We also show that other definitions for reflection can be expressed in terms of those two. Such as the distinction between structural (white cells) and behavioral (grey cells) reflection, as well as the temporal characterization of reflection (run-time versus compile-time reflection).

Reflective systems can also be characterized according to: *Abstraction, Scoping and Pluggability*:

Abstraction: Low-level/High-level Reflection

Low-level reflection has as domain a program's representation in memory (object representation) and the semantics of the underlying execution mechanism (e.g



	(A) Explicit	**(B)** Implicit (Intercession)
(1) Introspection (Read)	*e.g instVarNamed:*	*e.g onInstVarRead:*
(2) Self-Modification (Write)	*e.g instVarNamed:put:*	*e.g onInstVarWrite:*
(3) Invocation (Execute)	*e.g perform:withArgs:*	*e.g onMsgReceived:*
(4) Compilation (Compile)	*e.g compile:*	*e.g onParseNodeEmitCode:*

Run-time reflection (A1-3, B1-3)
Compile-time reflection (A4, B4)
Structural reflection (A1, A2)
Behavioral reflection (A3-4, B1-4)

Figure 3.3: Relationship between different reflective categories as presented by [Maes 1987b], [Ferber 1989] and [Bracha 2004]

virtual-machine). High-level reflection has as domain the language model itself and the language semantics. An example of the distinction between the two is given below:

```
Object>>instVarAt: index "low—level access —— object as an array"
Object>>instVarNamed: aString "high—level access —— conceptual object"
```

Scoping: Scoped/Unscoped Reflection
Scoped or partial reflection refers to the ability of applying intercession and self-modification mechanisms to single objects, classes, namespaces etc. On the other hand, unscoped reflection applies reflection indiscriminately in an entire language (introducing new syntax or semantics by modifying a host compilier, modifying host class builders or class loaders e.t.c).

Pluggability: Pluggable/Cohesive Reflection
Reflection can also be defined as pluggable when there is language support for: a) multiple implementations for it and b) the ability to discard it entirely from the language. On the contrary reflection can be deemed cohesive when it has a strong coupling with the base system and the runtime of a language.

3.2 Reflection for Remote Debugging

When the system we depict in the process domain of Figure 2.1 is reflective, its own facilities can be used to inspect and modify the system itself thus facilitating debugging. In terms of the definitions that we introduced in Section 2.1 we can now define a reflective debugger as follows:

Reflective debugger *A reflective debugger is an internal sub-process of the process domain that is used for run-time analysis. It represents and acts upon the process* *domain as a whole through reflection, making the process state and its behavior part of the observational domain.*

Moreover in terms of the definitions we introduced for reflection in Section 3.1 we can now also define remote reflection as follows:

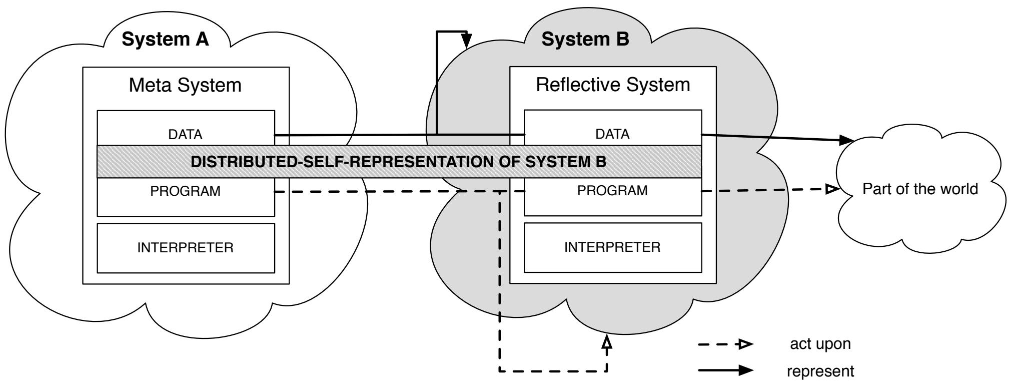


Figure 3.4: Remote Reflection

Remote reflection *is the ability of a reflective system (System B of Figure 3.4) to distribute the whole or part of its self-representation (designated with dashed-gray in Figure 3.4) to another meta-system (System A of Figure 3.4.)*

Debugging tools that rely on reflection are best suited to support experimentation during the debugging process. In fact as we saw in Chapter 2 the facilities that reflective debuggers offer **in a local setting** (such as the Smalltalk debugger and its extension through Bifrost) set the standard in terms of existing solutions for the properties of *interactiveness* and *instrumentation*.

In a remote setting though supporting these properties through reflection, as well as debugging facilities in general, introduces several challenges. In the following Sections we will discuss different design principles and patterns for remote reflection and describe these challenges.

3.3 Architectural Alternatives for Remote Reflection

In this Section we discuss remote reflection solutions from an architectural point of view. More specifically we look at core design patterns from literature that can facilitate remote reflection. These are, *the remote proxy pattern [Gamma 1995], the remote facade [Alpert 1998] and mirrors [Bracha 2004].*

3.3.1 Remote Proxy Overview

The most direct approach to remote reflection in OO languages can be achieved by means of the Proxy [Gamma 1995] design pattern. In Figure 3.5 we show how this pattern can be used to distribute reifications of the meta-level to another machine.

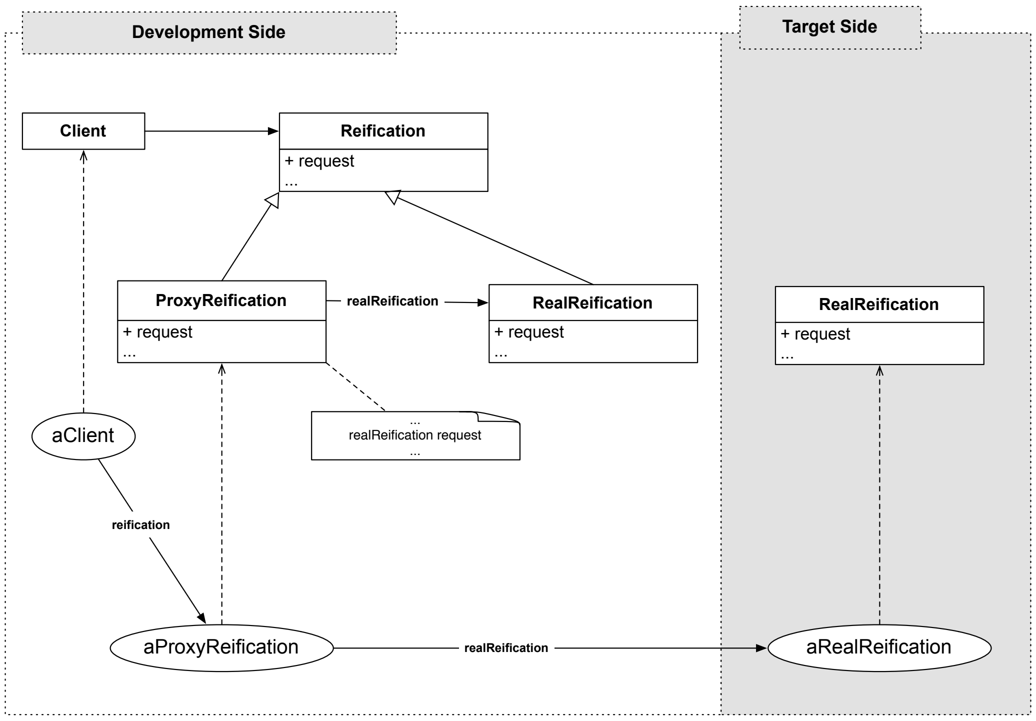


Figure 3.5: Using the Proxy pattern to support remote reflection

In Figure 3.5 we assume that the Reflective system and the Meta system of Figure 3.4 are respectively the target side and the development side of a remote debugging session. On the development side instances of class *Client* such as a reflective debugger depend upon reifications that conform to the interface that the abstract class Reification defines. The two subclasses of Reification namely ProxyReification and RealReification conform to the same API and their instances are indistinguishable from the point of view of the Client class. RealReification instances represent the actual meta-objects while instances of ProxyReification represent placeholders that encapsulate the real meta-objects.

The goal of the ProxyReification class is to introduce *one-level of indirection* between instances of Client and instances of RealReification to control message-passing. In our case this indirection is used to forward messages to an instance of RealReification in another machine (left part of Figure 3.5) thus enabling remote reflection. These kinds of proxies are usually referred to in literature as *remote proxies* [Alpert 1998] or *Ambassadors* [Coplien 1992].

3.3.2 Remote Proxy Design Challenges

Extensibility and Re-use One desirable side-effect of this direct proxyfication of the meta-level that we described above is that it enables re-use. Since both reification classes (ProxyReification and RealReification) conform to the same interface, the class Client can reflect on both the local and the remote environment. Furthermore since the reflective interface is defined separately on the abstract class *Reification* the solution can be extended with yet more concrete classes, without affecting the code of the class Client.

Distribution Despite its elegance and simplicity though the direct proxyfication of the meta-level is not a viable solution for remote reflection by itself. The reason is that the local reflective API in OO languages is unfit for transparent distribution raising issues of latency, concurrency and partial failure [Waldo 1994].

Let's take a real world example to illustrate some of the problems with transparent distribution for reflection in this context. In Script 1 a method of the class Client retrieves the class names of a remote target by iterating locally (on the development side) through the list of remote classes, while in Script 2 the same method is implemented by sending a dedicated message for class names to the other side, in which case the iteration over the loaded classes will take place on the target.

The method on Script 1 will send at least *n* messages to the target side (where n is the number of loaded classes on the target), since the iteration is taking place at the development side. The method on Script 2 will send only a single message to the other side and will return the list of class names immediately (since the iteration is happening on the target).

Of course even for Script 2 the resulting list may itself be a proxy. Meaning that if we try to print these class names (Script 3), we will not avoid iterating locally on the development side after all, producing an excessive amount of communication. Moreover for Script 2 we made the assumption that dedicated methods (such as className on Script2) exist on the target side. This assumption is not always true, since for most reflective languages the core-api is not distribution aware.

This example shows us that the use of local reflection on the target "as is" for debugging (through the use of the remote proxy pattern) does not scale well in terms of communication overhead in the remote debugging scenario.

Script 1: **Retrieving class names from a Smalltalk image by iteration**

```
Client>>classNamesFor: aSmalltalkImage

    ^ aSmalltalkImage environment allClasses collect: [:class | class name]
```

Script 2: **Directly retrieving class names from a Smalltalk image**

```
Client>>classNamesFor: aSmalltalkImage

    ^ aSmalltalkImage environment classNames
```

Script 3: **Printing class names from a Smalltalk image**

```
Client>>printClassNamesFor: aSmalltalkImage

    Transcript show: (self classNamesFor: aSmalltalkImage).
```

One possible solution to this problem is to devise a separate reflection API for remote targets that takes distribution into account. But maintaning two different sets of reflective APIs hinders re-use. For example in Figure 3.5 if the class Client was modeling a reflective debugger we would need two separate versions of it or a subclass to debug both locally and remotely, since the reflective API in the latter case would be different.

Identity When the meta-system that we depict in Figure 3.4 is itself reflective then remote meta-objects like *aProxyReification* of Figure 3.5 present an ambiguity in terms of the local reflective API on the development side. The problem stems from the fact that a remote meta-object is a local object itself that should respond to local reflection.

We designate this ambiguity as the identity problem of remote reflection. To illustrate the identity problem, Figure 3.6 presents two clients of reflection in the development side *clientA* and *clientB*. ClientA assumes that aProxyReification acts as a remote meta-object (e.g clientA is a reflective debugger that expects messages to be answered by *aRealReification*), while clientB assumes that aProxyReification acts as a local object (e.g clientB is a memory monitor that expects messages to be answered by *aProxyReification* itself). Both clients use reflection but expect different things from the proxy. When clientA asks the proxy for its class name the class of the remote reification should be returned, while when clientB sends the same message it expects the class of the proxy itself to be returned.

In the case of the identity problem too maintaining two different sets of reflective APIs one for local reflection and one for remote could solve the problem at the expense of code re-use.

Meta-Recursion In Figure 3.7 a remote meta-object is reflecting on a base level object on the target side that supports persistency (for e.g on a file or on a database) though reflective intercession. That is that the semantics for instance variable access for *aPersistentObject* have changed through reflection to synchronize its state with an external data storage.

Let us now assume that this was achieved by subclassing the host compiler on the target side to transform each read and write access of instance variables in the source code to meta-level calls. For example in this case each read access of instance variables in the class PersistentObject will be redirected to the meta-level method *instVarAt* which has been overridden from *Object* to provide the additional functionality.

In such cases debugging via remote reflection can be problematic when:

Debugging intercessed objects We want to reflect on aPersistentObject for debugging from the development side but without the side-effects introduced by the local reflective intercession that we introduced. For example when an object inspector is trying to read the state of aPersistentObject. In this case we would like the instVarAt method of Object to be called rather than the instVarAt method of PersistentObject.

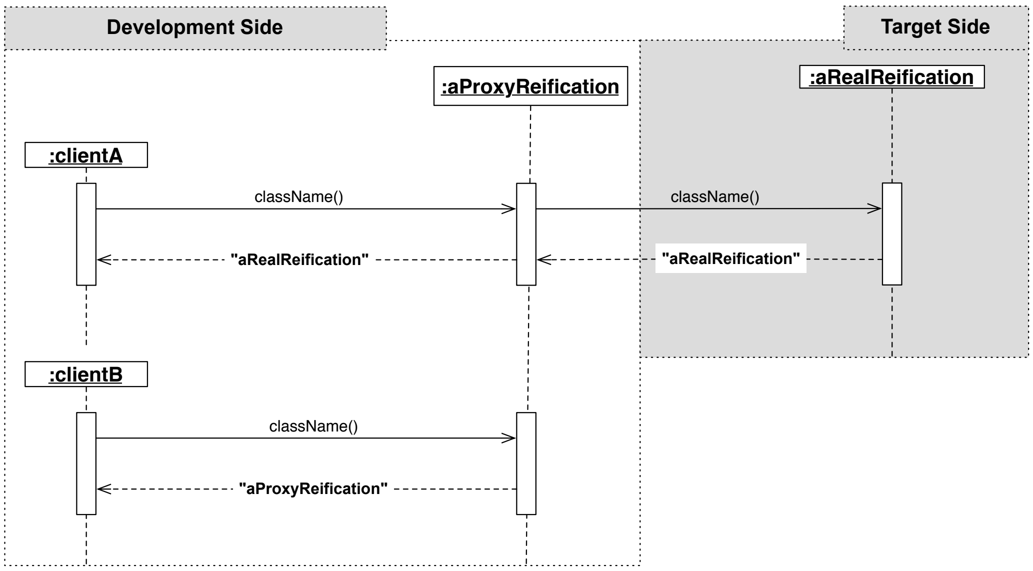


Figure 3.6: The identity problem

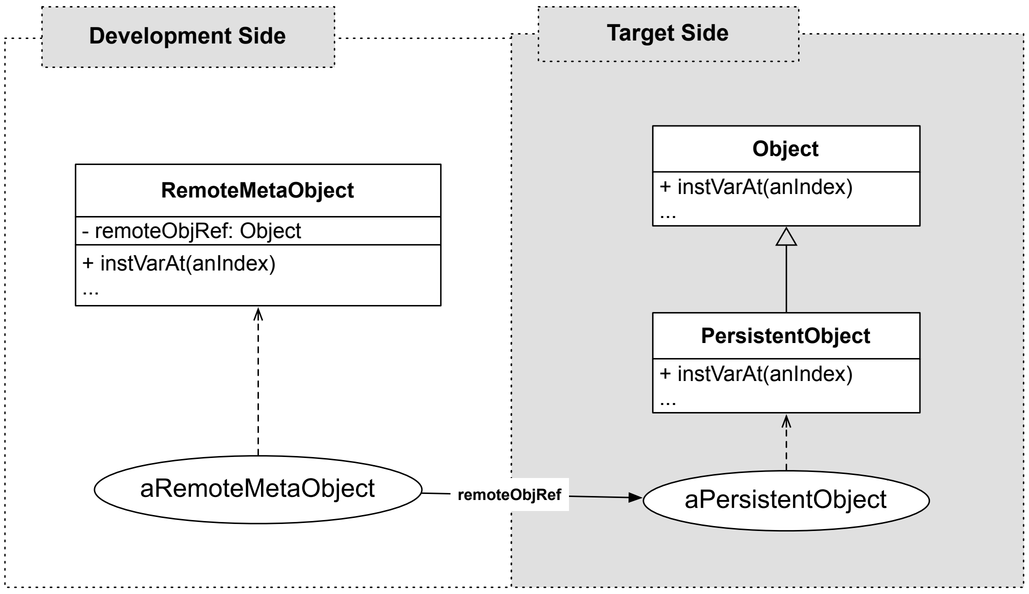


Figure 3.7: Reflecting on an intercessed object

Debugging reflective methods There is a bug on the reflective methods themselves (i.e the method instVarAt on PersistentObject) which is being called during remote debugging, recursively resulting in new exceptions.

Debugging reflective infrastructure Finally there is a bug on the host compiler itself that was introduced while implementing PersistentObject. In this case we are now unable to proceed since the faulty Compiler on the target cannot re-compile itself so that we can fix the problem.

In all of the above scenarios in one form or another we would like to use remote reflection from the development machine to debug part of the local reflection on the target. The problem arises when the remote reflection facilities on the development machine depend on the same part of local reflection on the target that we wish to debug. For this reason we designate this problem as the *meta-recursion problem* of remote reflection.

3.3.3 Remote Facade Overview

An alternative approach to remote reflection can be devised using a client-server model that depends on the Facade design pattern [Alpert 1998]. In this case the target side acts as the server (the Reflective System in Figure 3.4) and the development side acts as the client (the Meta System in Figure 3.4). This approach can overcome some of the problems we have described thus far but as we will see has its own drawbacks.

In Figure 3.8 the class *MetaLevelFacade* in both the development and the target side encapsulates access to all reflective facilities of the system by providing a *different* reflective API (than the default one) that hides all reifications of the meta-level. Clients of this class instead of accessing reflection through multiple different objects utilize the unique access point. This property of the Facade pattern has the effect of hiding all the details of reflection behind one common interface.

This unique access point can itself be proxied to support remote reflection as we illustrate with the class MetaLevelFacadeProxy. The difference here with direct proxyfication is that now instead of having multiple different proxies (one for each reified entity or object that is being reflected on the other side) there is one single point of access to the other side.

3.3.4 Remote Facade Design Challenges

Extensibility, Re-use At first glance by introducing a different API the facade solution seems to hinder re-use, since two different reflective APIs co-exist in the development side (the default one and the facade API). Nevertheless implementing a facade API for both the local and the remote case is fairly easy since in the case of the facade there is only one reification to duplicate. Clients such as a reflective debugger in this case can choose to use the facade API rather than local reflection directly when they want to promote re-use.

Distribution In terms of distribution the facade by having a different API than the default one can be used to facilitate communication through *procedural abstraction*. In essence

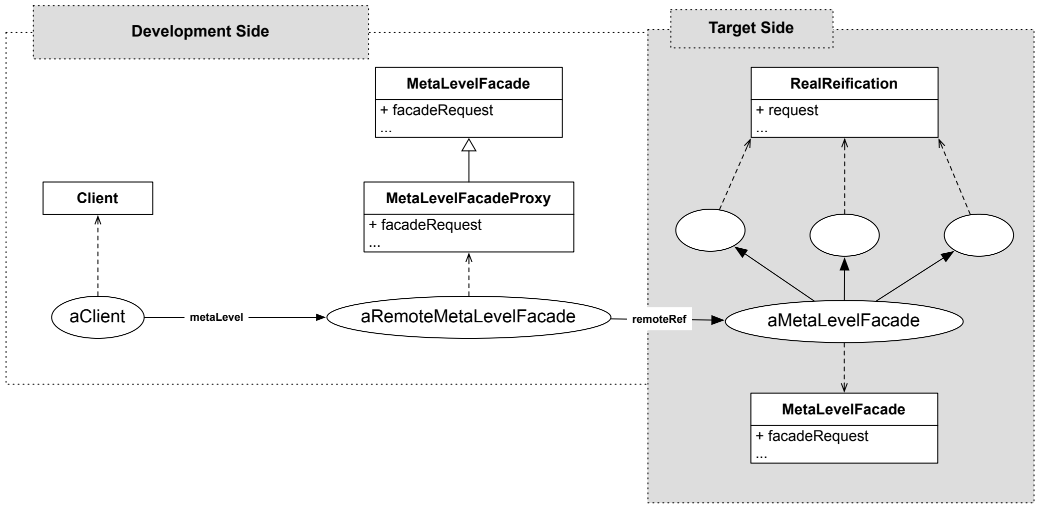


Figure 3.8: Remote reflection via a proxy facade

reflective operations involving multiple objects and messages can be now grouped into one single call minimizing the communication cost with the other side.

Identity Similarly if the facade API is carefully devised the facade itself can respond to local reflection as a local object and to the orthogonal reflective API that it implements.

Meta-recursion The facade pattern by itself cannot be used to solve the meta-recursion problem, since it ultimately depends on local reflection on the target. Nevertheless it introduces a level of indirection not only in the development side (as with direct proxification) but in the target side as well. This indirection on the target can be used to avoid some aspects of the meta-recursion problem especially in the case of intercessed objects by accessing a default (non-intercessed) implementation of reflective facilities (for e.g through the underlying virtual-machine).

Despite its advantages the facade solution when used by itself has a major drawback: **it collapses the object system of reflection into a single reification that is responsible for everything**. This fact has a direct impact on the facade API in terms of OO design.

To ensure encapsulation since no other reifications exist, return values of the facade API are either basic types or collections of basic types. Packages, classes, methods etc. are all accessed by name, and returned by name while individual objects have to be accessed by id since none of these entities have a corresponding reification in the system. The end result is a procedural API for reflection rather than an object-oriented one.

3.3.5 Mirrors Overview

Mirrors are defined explicitly by Bracha and Unghar [Bracha 2004] as *intermediary objects [...] that directly correspond to language structures and make reflective code independent* *of a particular implementation*. They further state that mirror-based systems should embody the following principles:

Encapsulation Meta-level facilities must encapsulate their implementation.

Stratification Meta-level facilities must be separated from base-level functionality.

Ontological Correspondence The ontology of meta- level facilities should correspond to the ontology of the language they manipulate.

In order to illustrate their proposal the authors give the following example, which we have translated into Smalltalk:

Script 1: Mixed base and meta-level functionality. Based on example from [Bracha 2004].

```
1 myCar := Car new.
2 numberOfDoors := myCar numberOfDoors.
3 carClass := myCar class.
4 carClassIvNames := carClass instVarNames.
5 anotherCar := carClass new.
6 carSuperclass := carClass superclass.
```

Script 2: The same example in a mirror-based system

```
1 myCar := Car new.
2 numberOfDoors := myCar numberOfDoors.
3 carMirror := Mirror on: myCar.
4 carClassMirror := carMirror class.
5 carClassIvNames := carClassMirror instVarNames.
6 carClassSuperMirror := carClassMirror superclass.
```

Script 1 illustrates reflection in a cohesive language kernel where base level and meta-level functionality co-exist in the same object. For example the object myCar in Script 1 responds to both base-level methods (e.g numberOfDoors) and meta-level (reflective) methods (e.g class). Bracha and Ungar designate this form as *traditional reflection*.

Script 2 illustrates another approach where the object myCar responds only to base level methods and the *jump* to the meta-level is made explicitly on line 3 via the creation of an *intermediate object*. This intermediate object (carMirror) responds to reflective functionally corresponding to the base-level object for which it was created (car). Bracha and Ungar designate this form as *mirror-based reflection*.

Bracha and Unghar make the following statements about mirrors:

- a) Mirrors make remote/distributed development easier

- b) Mirrors make deployment easier because reflection can be easily taken out or added, even dynamically

In this section in order to evaluate the mirror-based approach to remote reflection we will try to correspond Bracha and Ungar's proposal to known design patterns. By doing so we will see that **in terms of remote reflection** mirrors can be seen as an extension to both the proxy and the facade solutions described above.

3.3.6 Design patterns behind Mirrors

3.3.6.1 Explicit meta-object

The idea of a separate object that is responsible for reflective operations in OO languages was first introduced in 3-KRS [Maes 1987b]. 3-KRS introduces the notion of a *meta-object* as follows:

[...] Every object in the language is given a meta-object. A metaobject also has a pointer to its object. The structures contained in an object exclusively represent information about the domain entity that is represented by the object. The structures contained in the meta-object of the object hold all the reflective information that is available about the object. The meta-object holds information about the implementation and interpretation of the object

Meta-objects can be either implicit when they are invoked automatically by the underlying execution mechanism, or explicit when they are invoked on-demand by the programmer [Maes 1988]. From this point of view mirrors are explicit meta-objects. This is illustrated in Figure 3.9 where the method getClass() has been *factored out* from the API of base-level objects (such as anObject) to the explicit meta-object anObjectMirror. Bracha and Ungar designate this refactoring from the base-level to the meta-level as *functional decomposition* [Bracha 2004].

This re-factoring applies to languages where reflection is cohesive (coupled with the base-level) such as Java. In practical implementations though such as JDI, reflective methods are not removed from the base-level but the mirror implementation when loaded coexists with core reflection.

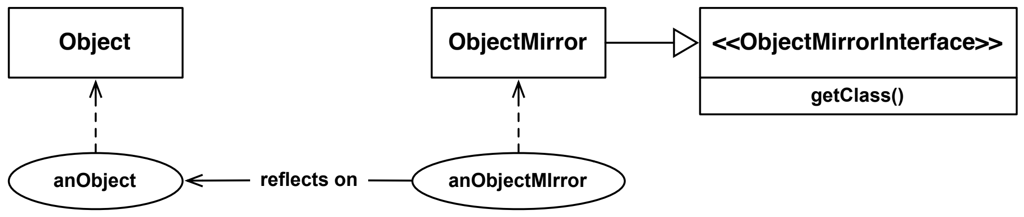


Figure 3.9: Mirrors are explicit meta-objects

In the case of remote reflection mirrors reside in the development side (left part of Figure 3.5) just like proxies do, acting as intermediate objects. The difference between mirrors and proxies in this case is that mirrors can have a different API from local reflection on the target side and may be independent from it.

3.3.6.2 Abstract class or interface

As we illustrate in Figure 3.9 mirrors conform to a specific reflective API that can be described through an interface or via an abstract class. This fact promotes extensibility and re-use as in the cases of the Proxy and the Facade pattern.

3.3.6.3 Factory

Mirrors also introduce the notion of a *reflective factory* which we illustrate in Figure 3.10. This mirror factory acts as an entry-point to different domains that can be reflected. In Figure 3.10 we give some possible examples. The mirror factory depending on which domain it was asked to reflect on (a local or remote environment, a source file, a core-dump of a crashed program on the disk etc.) will return the appropriate entry point mirror for that domain. As we described above this mirror regardless of which domain it represents will conform to the same API.

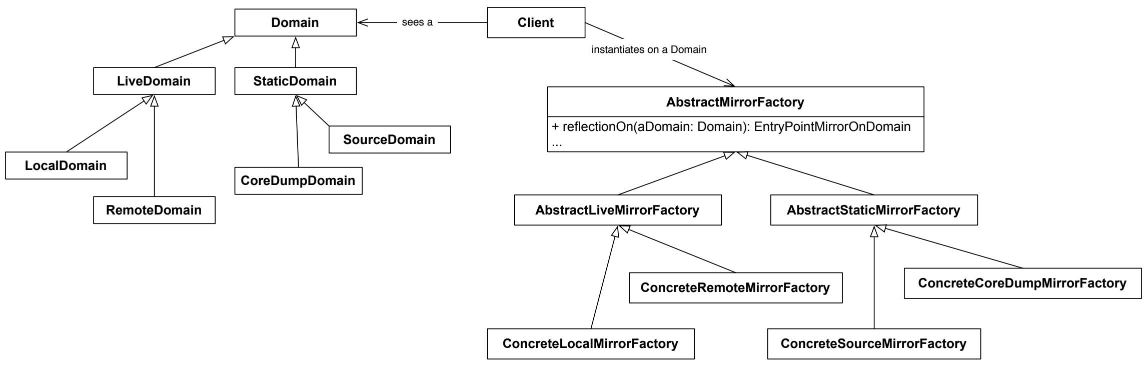

Domain
sees a
Client
instantiates on a Domain
LiveDomain
StaticDomain
AbstractMirrorFactory
+ reflectionOn(aDomain: Domain): EntryPointMirrorOnDomain
...
SourceDomain
LocalDomain
CoreDumpDomain
RemoteDomain
AbstractLiveMirrorFactory
AbstractStaticMirrorFactory
ConcreteRemoteMirrorFactory
ConcreteCoreDumpMirrorFactory
ConcreteLocalMirrorFactory
ConcreteSourceMirrorFactory

Figure 3.10: Reuse via a mirror factory

The mirror factory acts as the entry point to all reflective facilities (regardless of domain or implementation), promoting extensibility and re-use.

3.3.6.4 Facade & Bridge

Mirrors also make the distinction between low-level and high-level reflection by introducing the notions of: *low-level mirrors* and *virtual-machine mirrors*.

In terms of remote reflection the virtual machine mirror acts as the Facade proxy that we described in Section 3.3.3. All messages directed to mirrors which reflect on a remote environment will be directed via this virtual machine mirror to the other side. From this perspective mirrors act as OO wrappers around the Facade API.

Low-level mirrors on the other hand act as a Bridge [Alpert 1998] pattern between high-level mirrors and virtual-machine mirrors to decouple the mirror API from implementation

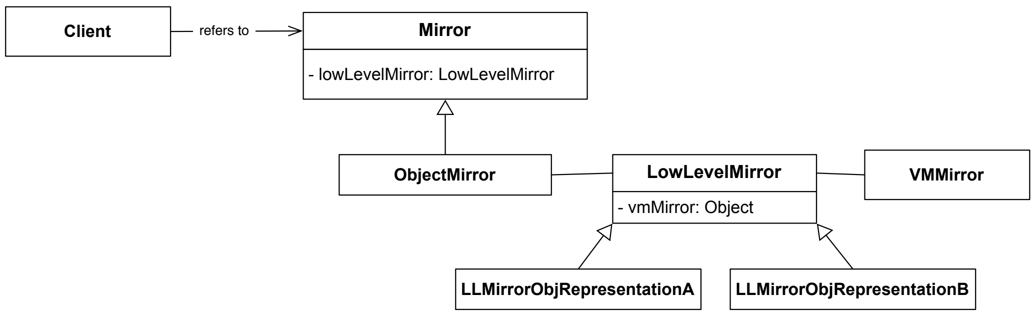

Client
refers to
Mirror
- lowLevelMirror: LowLevelMirror
ObjectMirror
LowLevelMirror
- vmMirror: Object
VMMirror
LLMirrorObjRepresentationA
LLMirrorObjRepresentationB

Figure 3.11: Distinguishing between low-level and high-level mirrors

details. One such detail for e.g can be the actual memory representation of an object on the other side.

3.3.7 Mirrors Design Challenges

Extensibility, Re-use Mirrors support both extensibility and re-use through the reflection factory that we described above and a unified interface for meta-objects of different domains.

Distribution In terms of distribution mirrors by introducing a different API than the default one can be used to facilitate communication. Of course it is the Facade (the virtual machine mirror) that acts as the final gateway to the other side and through which communication can be minimized via *procedural abstraction*. Mirrors overcome the main disadvantage of the plain client-server solution by introducing OO wrappers on top of this Facade.

Identity Similarly if the mirror API is carefully devised mirrors can respond to local reflection (as local objects) and to the orthogonal reflective API that they implement.

Meta-recursion If the implementation of the virtual-machine mirror is totally independent of local reflection on the other side, then mirrors can be used to solve the meta-recursion problem for local reflection by accessing a non-intercessed implementation of reflection.

Mirrors and The Problem of State A language kernel where reflection is cohesive, does not only mix base with meta-level operations but also base-level with meta-level state. Although mirrors advocate the distinction between base-level and meta-level operations (through functional decomposition) they do not address the problem of mixed state in cohesive reflection.

Bracha and Ungar suggest that their solution can reduce the footprint of applications by unplugging reflective facilities. Debugging frameworks (such as gdb [Richard Stallman 2003]) on the other hand reduce footprint on an executable by plugging and unplugging compile-time and debugging meta-information. So in the context of debugging the following question naturally arises regarding mirrors: *Can mirrors also decompose meta-information from a language kernel ?* If this kind of decomposition is possible then the property of stratification for mirrors that Bracha and Ungar describe should be extended to accommodate state.

In terms of distribution too, if mirrors represented the state (cached or otherwise) of the objects that they reflected on, one could in principle avoid communication with the other side in order to perform introspection (i.e to query on meta-state).

Mirrors and Intercession Finally mirrors can be considered more of an organizational model to reflection rather than an implementation one. For example although with a mirror-based model one can distinguish between introspection and intercession, the model itself does not suggest anything about *how intercession should be implemented, or what facilities should it support etc.*. Mirages [Mostinckx 2007] propose an extension to the mirror model to address this issue through implicit meta-objects. A related implementation for the programming language AmbientTalk, extends a contemporary API of mirrors on objects with the following three methods: a *doesNotUnderstand(selector)* protocol for message sends and two implicit hooks for object marshalling (pass() and resolve()) [Mostinckx 2009]. For supporting instrumentation though during debugging, mirrors should be able to support an even wider range of intercession facilities (see Chapter 2).

3.4 Summary

In Chapter 2 we saw that debugging tools relying on reflection in a local setting are best suited to support experimentation during the debugging process. In this chapter we studied and provided definitions for reflection and remote reflection in order to assess their use for remote debugging. A reflective system is a causally connected meta-system that has as object-system itself, while remote reflection is the ability of a reflective system to distribute the whole or part of its self-representation to another meta-system. We also studied different design patterns that facilitate remote reflection, namely: *the remote proxy, the remote facade and mirrors* and concluded that **in terms of remote reflection** mirrors present a more comprehensive solution which can be seen as an extension to both the remote proxy and the remote facade patterns. Finally we pinpointed two open-issues concerning mirrors in the context of debugging, regarding state and intercession.

CHAPTER 4

MetaTalk: A Mirror extension supporting Structural Decomposition

Contents

At a Glance

In this Chapter we present our solution to the problem of Reflective-Data [Maes 1987b] in the context of mirrors [Bracha 2004]. Mirrors are meta-level entities introduced to decouple reflection from the base-level system. On the one hand mirrors can reduce the footprint of applications in-between debugging sessions by unplugging reflective facilities. On the other hand they do not address the problem of reflective data so as to unplug meta-information. We argue that mirrors should support structural decomposition for reflection by factoring out language meta-information that is not used by the system's runtime. We validate our proposal through a language prototype: *MetaTalk*[1].

[1] http://www.squeaksource.com/MetaTalk/

4.1 Cohesive Language-Kernels

The main problem that mirrors address is the problem of a cohesive (mixed base and meta functionality) language kernel. This problem was first described by Pattie Maes [Maes 1987b] in terms of the reflective architectures of Smalltalk-72, Flavors and Smalltalk-80:

[...] In languages such as SMALLTALK-72 [Adele Goldberg 1976] and FLAVORS [Daniel Weinreb 1981]. An object not only contains information about the entity that is represented by the object, but also about the representation itself, i.e. about the object and its behavior. For example, in SMALLTALK, the class Person may contain a method to compute the age of a person as well as a method telling how a Person object should be printed. Also in FLAVORS, every flavor is given a set of methods which represent the reflective facilities a flavor can make usage of.

Pattie Maes [Maes 1987b] highlights two specific problems that cohesive language kernels introduce:

Extensibility *There are two problems with this way of providing reflective facilities. One is that languages always support only a fixed set of reflective facilities. Adding a new facility means changing the interpreter [...].*

Reflective Data *A second problem is that they mix object-level and reflective level which may possibly lead to obscurities. For example if we represent the concept of a book by means of an object, it may no longer be clear whether the slot with name "Author" represents the author of the book (i.e domain data) or the author of the object (i.e reflective data) [...].*

The first problem was partially solved in Smalltalk-80 by reifying classes inside the language and by introducing metaclasses, through which both the structure and the behavior of classes can be altered. More generally we can say that every meta-level that supports some form of intercession can solve this problem.

The second problem though (of reflective data) that Pattie Maes describes was never solved in Smalltalk-80 since:

- Classes in Smalltalk-80 still mix base-level and meta-level information.
- Due to the above fact metaclasses although structurally represent system information concerning their corresponding class, behaviorally they define both base-level (e.g Point class»#fromUser) and meta-level facilities (e.g Class»#addInstVar:).

The above situation has been shown to generate confusion about classes and metaclasses for developers [Borning 1987].

The solution that Pattie Maes proposed for solving the problem of reflective data was the introduction of entities called *meta-objects* in 3-KRS, which *[...] adopt a disciplined split between object-level and reflective level* and which **[...] hold all the reflective information that is available about that object**.

4.2 Mirrors as meta-objects

We now turn our attention to mirrors as a solution to cohesive reflection by comparing the proposal of Bracha and Ungar with that of Pattie Maes. In our view the two differ in the following two ways:

Pluggability Mirrors in contrast to meta-objects as defined by Pattie Maes are **pluggable** since they conform to the property of *stratification*. This means that the developer can choose to discard mirror-based meta-objects when they are not used, while in 3-KRS this is not possible.

Decomposition On the contrary, meta-objects as defined by Pattie Maes exhibit a stricter split between base-level and meta-level than mirrors. Meta-objects in 3-KRS apart from reflective behavior also decompose reflective state in order to solve the problem of reflective data that we described in the previous section.

In essence although with mirrors reflective functionality is pluggable, this functionality ultimately depends on some original sources of meta-information other than mirrors. These original sources may well reside in the base system of the language, in abstract syntax trees, in separate source files, in system dictionaries, in the object representation or hardwired inside the virtual-machine. This means that mirrors do not solve the reflective data problem that was described by Maes.

This fact was first described in [Malenfant 1991] for the initial implementation of mirrors (in Self) as follows:

> *Mirrors on the other hand, are Self objects but they don't store information on their own: they give a reading access to internal information about objects through a set of virtual machine primitives.*

The same fact was later generalized by Bracha and Ungar in [Bracha 2004], through their distinction between *functional decomposition* and classic OO decomposition:

> *The solution is to factor the reflective functionality of getClass out of the API of ordinary objects. This is exactly what mirrors do. This factoring implies a functional decomposition rather than a classic object-oriented one.*

Since mirrors describe only the functional decomposition of reflection from the base-system there is no decomposition of the meta-information itself. If a language provides a rich set of meta-information there will be an impact to the level of stratification that mirrors can offer.

This is the case of deeply reflective languages (such as Smalltalk) where there is a rich set of meta-information available from within the system. These meta-information can include: global resolution names, names of instance variables and methods, system categories and packaging information, the code for each module, sub-module in the language, abstract syntax trees, executional, profiling, tracing information etc.

4.3 Pluggable and state-full meta-objects

As we saw mirrors through stratification introduce the property of *pluggability* to meta-objects but do not propose a solution to the problem of *Reflective Data*. We argue that meta-objects should not only be pluggable (as mirrors are) but also state-full (as the meta-objects in 3-KRS). Meta-objects conforming to both properties could address the following issues:

Separation of concerns Decoupling reflective data from the base-level, presents a stricter separation of concerns, that can resolve the ambiguity of entities such as classes and metaclasses in cohesive language kernels.

Stratification Pluggable meta-objects that also decompose reflective state can discard not only reflective behavior (methods) but also reflective state (meta-information) when they are not used. In terms of both debugging and deployment it is desirable to be able to discard meta-information.

In order to support debugging, compilers such as gcc and javac embed compile-time meta-information inside the resulting executables. During deployment this meta-information is striped from executables in order to reduce memory footprint.

In cohesive language kernels with a rich set of both compile-time and run-time meta-information (such as Smalltalk) discarding these meta-information is not possible.

Reverse-engineering Reflective data also play a crucial role in reverse-engineering. Deployed executables with embedded meta-information are easier to reverse engineer.

Leaving a deployed executable open to reverse engineering, may not always be desirable. In this case also as far as cohesive languages are concerned, discarding meta-information is not an option.

Caching In terms of remote meta-objects the presence of state can reduce the communication footprint. In this case meta-objects with state can act as virtual proxies [Gamma 1995] to the other side which cache meta-information in order to postpone and reduce communication.

4.4 Extending mirrors to support Structural Decomposition

In analogy with the definition of Bracha and Ungar on functional decomposition [Bracha 2004] (concerning the factoring of reflective functionality out of the API of ordinary objects), we define structural decomposition as follows:

Structural Decomposition *is the factoring of reflective data out of the state of ordinary objects.*

Our goal with structural decomposition of reflection through mirrors is to extend the property of stratification to meta-data. With structural decomposition of reflection, one can effectively discard meta-data (see right part of Figure 4.1) when they are not used.

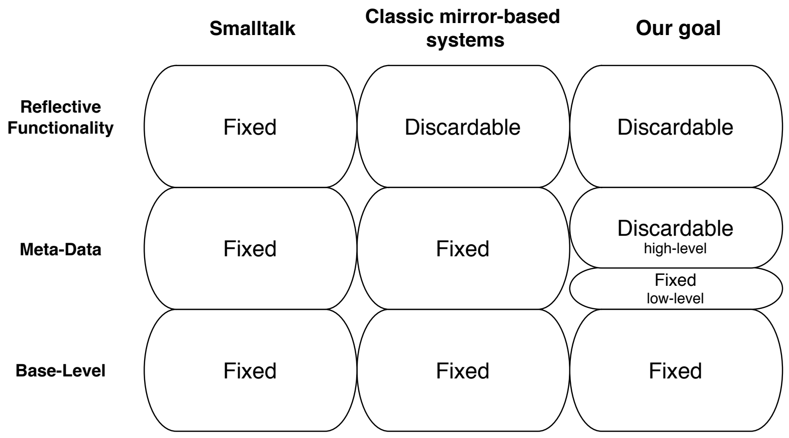


Figure 4.1: Our goal, towards structural decomposition with mirrors

Low-level vs High-level Meta-information Bracha and Ungar [Bracha 2004] also make the distinction between low-level and high-level mirrors. We discuss this distinction in the light of structural decomposition. Although structural decomposition of all meta-information is desirable, not all meta-information can be discarded. The run-time engine does require some meta-information to actually run the base-level. We qualify such meta-information as low-level. This information usually includes: class references, superclass references and unique indexes for each method.

We illustrate the difference between high and low-level meta-information in the context of Smalltalk. Meta-information such as class names, instance variable names, selectors, code, packaging information etc. are high-level, because they are not used by the virtual machine. Conversely, meta-information that is needed by the VM such as the superclass, the object format or the method dictionary of a class is low-level. The VM does make some assumptions on the storage point of these low-level meta-data. For example, the first instance variable of a class must be a reference to its superclass.

This is why, the class *Object* in Smalltalk cannot be extended with new instance variables. Indeed, the extension will shift the indices of instance variables of subclasses, including critical ones (e.g. the instance variable *superclass* in *Behavior*) which are assumed to be fixed by the VM.

To sum up we believe that all meta-information should be decomposed and thus materialized into mirrors. Furthermore there should be a distinction between low-level and high-level meta-information. Low-level meta-information cannot be discarded, since it is required by the run-time engine, but it should be decoupled from the object model. On the opposite all high-level meta-information should be materialized into pluggable mirrors.

4.5 Reference Model for the Structural Decomposition of Reflection

Aiming for simplicity and generality in our model, we chose to derive it from ObjVLisp [Cointe 1987]. We extended it by adding a meta-level with both an abstract and a concrete specification of mirrors.

The main underlying principles behind our reference model are:

Separation between the object model and the object representation. This separation is crucial because low-level meta-information, especially *class, superclass and methodDict references* should only be accessible by mirrors through the VM (object representation) but not from the base system of the language (object model).

In essence this means that there is no instance variable in the base-level describing meta-information of the language. The low-level meta-information that the VM requires is only part of the object representation (e.g stored in the object header) and thus is not accessible from within the language.

This decoupling of the object model from the object representation extends the property of stratification to low-level meta-information (which will only be accessible through mirrors and the VM). Furthermore it ensures the extensibility of the model, since the VM does not make any assumption about meta-information inside the object model.

Functional and Structural decomposition of reflection. All high-level meta - information and reflective functionality of our object model resides in the meta-level. This means that both functional decomposition is applied (as in legacy mirror-based systems), but also structural decomposition. Structural decomposition means not only that both low-level and high-level meta-information is accessed through mirrors, but also that mirrors are the storage entities of meta-information, allowing them to be discarded (removed from the system) when not needed. After discarding mirrors, low-level meta-information still resides in the object representation (and is used by the VM) but is not accessible from inside the language.

The separation between the object model and the object representation facilitates the application of structural decomposition. Through the clear separation between the object model and the object representation the only instance variables that describe meta-information reside in the meta-level and can be easily discarded.

Furthermore in our object model (Figure 4.2):

- Every object in the system has a Mirror, including classes and metaclasses.
- Mirrors are meta-level entities that hold all the meta-information of the respective object that they reflect on, and they are the sole provider of all reflective functionality on that object.

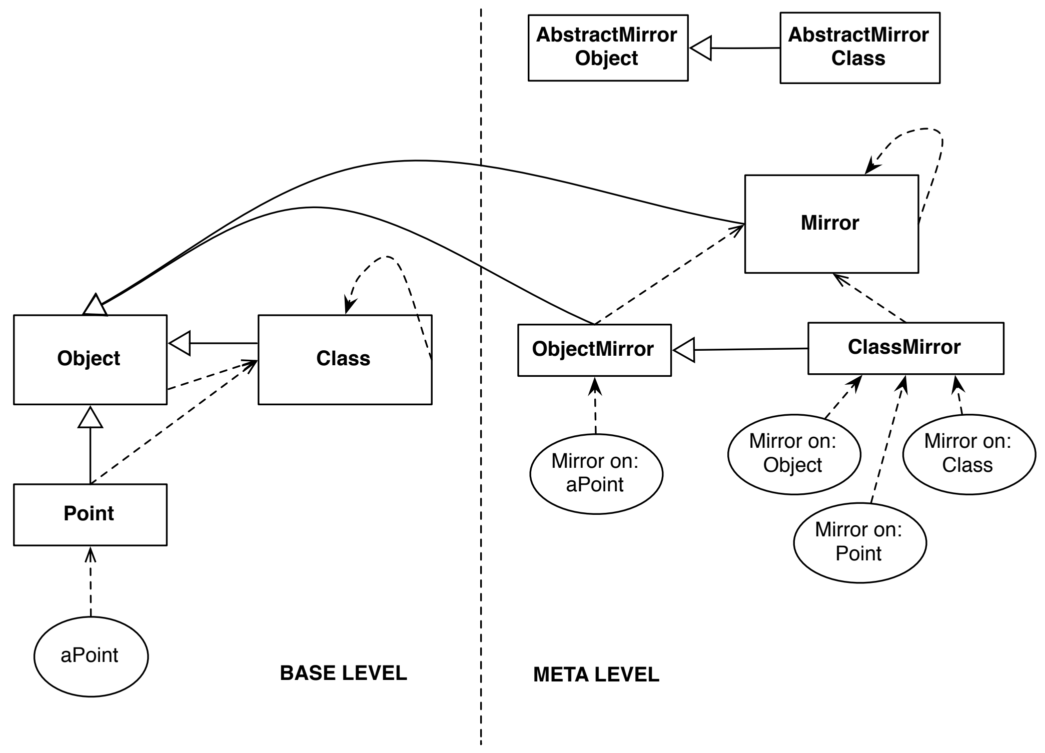


Figure 4.2: The MetaTalk Object Model

- All other entities in the language (including metaclasses) can provide reflective functionalities only explicitly through mirrors.
- Compilation by definition is taking place in the presence of meta-information. Only the meta-level has access to the compiler.
- Dynamic class definition can only be done through mirrors.
- The meta-level architecture corresponds to the language ontology, conforming to the ontological correspondence property described in [Bracha 2004] (e.g ClassMirror is a subclass of ObjectMirror since in the base-level classes are objects too).

In our object model the base-level is self-contained and can function independently. It can only instantiate terminal objects, without their meta-level counter parts. This allows the base system to operate separately and effectively prevents dynamic addition of behavior in the absence of Mirrors.

All reflective functionality and meta-information is accessible through mirrors. This information also includes low-level meta-information which is part of an object's representation but cannot be accessed from the base-level. The meta-level also provides an abstract description of mirrors that acts as a contract for different implementations. This contract describes the basic interface for the acquisition of Mirrors, which (as described in [Bracha 2004]) is done via dispatching on the object's class. In our object model depending on the implementation of the Mirror class, this can be either the actual mirror that holds the meta-information for the base-level object, or another mirror that delegates to the mirror holding the meta-information.

4.6 Implementation and Validation

In this section, we will discuss a prototype that validates the feasibility of our proposal.

4.6.1 Implementation

For validating the structural decomposition of meta-information in a mirror based reflection system we implemented the experimental class-based language MetaTalk. MetaTalk focuses on providing these characteristics to a dynamically typed OO language inspired by Smalltalk [Goldberg 1989] for the overall design, and by Resilient [Andersen 2004] for its declarative syntax.

MetaTalk follows the guidelines of our reference object model described in Section 4.5. It was fully implemented (object-representation, compiler, and virtual-machine) in the open-source, smalltalk-inspired environment Pharo [Black 2009]. Our compiler relies on *PetitParser* [Renggli 2010]. The code for our open-source language prototype can be found in the SqueakSource repository[2] and is released under the MIT license [3].

4.6.2 Example: Structural decomposition of core meta-information

Ideally by applying both functional and structural decomposition of reflection the following categories of resources can be freed when they are not used by the base-level:

Functional Decomposition:

1. All reflective methods of a language (In Smalltalk these are part of the language Kernel)

Structural Decomposition:

2. General system meta-information (SystemDictionary, SystemNavigation, SystemOrganizer etc)
3. All meta-information stored in classes (see Figure 4.3)
4. All reifications that are not used by the VM (Parser, Compiler, Decompiler, ProgramNode etc)
5. Standard clients of reflection, like programming tools (Browser, Inspector, Debugger etc)

Our prototype illustrates the ability to decompose the first three categories of resources, which concern the decomposition of reflection from a small kernel.

[2]http://www.squeaksource.com/MetaTalk/
[3]http://opensource.org/licenses/MIT

The most crucial part of this decomposition concerns meta-information stored in classes. *This part of the decomposition is crucial because the resulting system must simultaneously be self-describing as a whole (when the meta-level is present) but also allow the base-level to function independently without reflection.* In Figure 4.3 we show how we implemented this decomposition by factoring out high-level meta-information from the class to its corresponding mirror.

Concerning Figure 4.3:

- The object header of classes must be augmented to reference all low-level meta-information accessed by the VM. This step can ensure that the base-level can function independently from the meta-level, and that the VM does not make any assumptions about the object model of the language. In our prototype for example the augmented header (128bits) accommodates the references to *class, superclass and methodDict* respectively that our VM accesses.
- *There is no extra-cost for the augmentation of class headers since the low-level meta-information that it contains would otherwise be stored inside the object model.*
- The resources that are migrated (and can thus be discarded) *include the whole object graph that follows the meta-information references in the meta-level.*

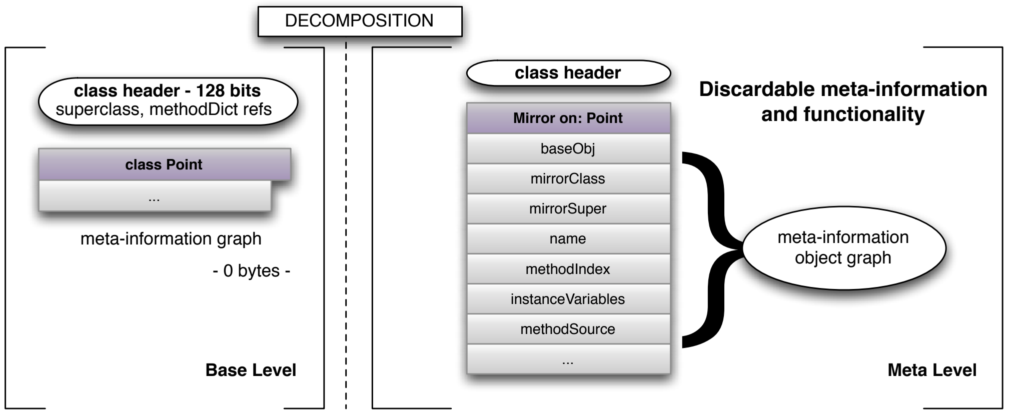


Figure 4.3: Example of high-level meta-information migration from the base to the meta-level in MetaTalk.

4.6.3 Validation

For the validation of structural decomposition one needs to test program execution both in the presence and in the absence of mirrors. Base-level functionality should be semantically identical in both cases. On the contrary, access to the meta-level should only be possible in the presence of mirrors. If base-level functionality is validated to be identical in both cases, this means that mirrors can be safely discarded when not needed. Moreover access to the meta-level should be validated to raise an error or an exception in the absence of mirrors.

```
MetaTalk>>> Validation process...

MetaTalk>>> Base−level functionality, in the presence of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta−level functionality, in the presence of mirrors:
MetaTalk>>> Mirror on: a Point3DX 1@0@30

MetaTalk>>> Base−level functionality, in the absence of mirrors:
MetaTalk>>> 1@0@30

MetaTalk>>> Meta−level functionality, in the absence of mirrors −− should signal an error by the vm.
...
<’metatalk−vm−exception: meta−level access is disabled’>
```

Figure 4.4: Standard output generated from steps 5 to 7 of the validation process.

```
newClass := ((Mirror on: Object) subClass: ’PointX’ instanceVariableNames: \{’x’ . ’y’.\}) baseObject.

(Mirror on: newClass) atMethod: ’initialize’ put: ’x := 0. y := 0.’.
(Mirror on: newClass) atMethod: ’x’ put: ’^ x.’.
(Mirror on: newClass) atMethod: ’y’ put: ’^ y.’.
(Mirror on: newClass) atMethod: ’x: aNumber’ put: ’x := aNumber asNumber.’.
(Mirror on: newClass) atMethod: ’y: aNumber’ put: ’y := aNumber asNumber.’.
(Mirror on: newClass) atMethod: ’asString’ put: ’^ x asString , ’’@’’ , y asString.’.

newSubClass := ((Mirror on: newClass) subClass: ’Point3DX’ instanceVariableNames: {’z’.}) baseObject.

(Mirror on: newSubClass) atMethod: ’initialize’ put: ’super initialize. z := 0.’.
(Mirror on: newSubClass) atMethod: ’z’ put: ’^ z.’.
(Mirror on: newSubClass) atMethod: ’z: aNumber’ put: ’z := aNumber.’.
(Mirror on: newSubClass) atMethod: ’asString’ put: ’^ super asString , ’’@’’ , z asString.’.
```

Figure 4.5: Step 4 of the validation process

Following this strategy for the validation of our prototype we took the following successive steps:

1. We compiled our kernel from sources, and validated its sound execution in the complete absence of the meta-level.

2. We then compiled our meta-kernel from sources, providing the system with its reflective functionality.

3. Subsequently we allowed global access to mirrors by invoking: *Baselevel reflect: true* , which is signaling the VM to permit mirrors to be pushed in the execution stack. From this point on and for the rest of the life of the system, no further compilation from sources can take place.

4. Then we dynamically created new classes, a superclass and a subclass through the meta-level (which is the only way to introduce at run-time new functionality to the system).

5. We tested the newly created classes for both their base and meta-level functionality (via mirrors).

6. Subsequently we forbade global access to mirrors by invoking: *BaseLevel reflect: false.*

7. Finally we repeated step 5 of the process, verifying that: (a) base-level functionality of the newly created classes was not by anyway altered by the absence of the meta-level, thus concluding that the meta-level could be safely discarded; (b) the subsequent attempt to access the meta-level signaled a terminal error by the vm.

```
'Validation process...' print.

'Base−level functionality, in the presence of mirrors:' print.
p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta−level functionality, in the presence of mirrors:' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: {30.}.
(Mirror on: p3D) perform: 'x:' withArguments: {1.}.
(Mirror on: p3D) print.

BaseLevel reflect: false.
'Base−level functionality, in the absence of mirrors:' print.

p3D := newSubClass new.
p3D z: 30.
p3D x: 1.
p3D print.

'Meta−level functionality, in the absence of mirrors −− should signal an error by the vm.' print.
p3D := newSubClass new.
(Mirror on: p3D) perform: 'z:' withArguments: {30.}.
(Mirror on: p3D) perform: 'x:' withArguments: {1.}.
(Mirror on: p3D) print.
```

Figure 4.6: Steps 5, 6 and 7 of the validation process.

Steps 4 through 7 are seen in Figures 4.5 and 4.6, respectively while the standard output generated in these steps can be found in Figure 4.4.

4.7 Summary

In this Chapter we saw that mirrors through stratification introduce the property of pluggability to meta-objects but do not propose a solution to the problem of Reflective Data. We argued that meta-objects should not only be pluggable (as mirrors are) but also state-full (as the meta- objects in 3-KRS). We showed that the property of stratification for mirrors, can be weak in cohesive kernels if structural decomposition is not taken into account. On the one hand mirrors can reduce the footprint of applications in-between debugging sessions by unplugging reflective facilities. On the other hand they do not address the problem of reflective data so as to unplug meta-information. We provided a solution with a reference model where mirrors are the initial source of meta-information. Finally we validated this solution through a language prototype supporting both functional and structural decomposition of reflection.

CHAPTER 5

Mercury: A Model for Remote Debugging in Reflective Languages

Contents

At a Glance

In this Chapter we present our solution for remote debugging. We start by defining the context of our proposal, discussing the connections with previous chapters. We then introduce one by one the various parts of our model in tandem with the properties that an ideal solution should exhibit. Finally we give a comprehensive comparison of our solution with state-of-the-art.

5.1 Introduction

In this chapter, we focus on remote debugging with reflective languages and we present a mirror-based model for remote debugging (following our discussion on Chapters 3 and 4) that exhibits the four desirable properties which we presented in Chapter 2 namely: *interactiveness*, *instrumentation*, *distribution* and *security*.

In Chapter 2, Figure 2.4 we gave an overview for the software entities involved in remote-debugging. We replicate this figure here for convenience (Figure 5.1) and follow its schema to give an overview of our solution. The model of the debugged application (left upper part of Figure 5.1) in our solution is dynamic (rather than static) and acts as a meta-level for target applications. This meta-level uses Mirrors [Bracha 2004] on the developer's side as to be causally connected with the debugged application and provide *interactiveness* support. The run-time debugging support on the target side reifies the underlying execution environment to support *instrumentation*. Our middleware follows a component architecture to be adaptable even during runtime. Finally from the security point of view, an authentication process takes place at the middleware level and different clients of our meta-level (such as different processes on an IDE) on the developer's end are subjected to different access restrictions by introducing a level of indirection (by means of a Mirror factory [Bracha 2004]) between the meta-level and its clients.

The following sections describe our solution based on this *live meta-level* and how it supports the four properties discussed in Section 2.3: *interactiveness*, *instrumentation*, *distribution*, and *security*.

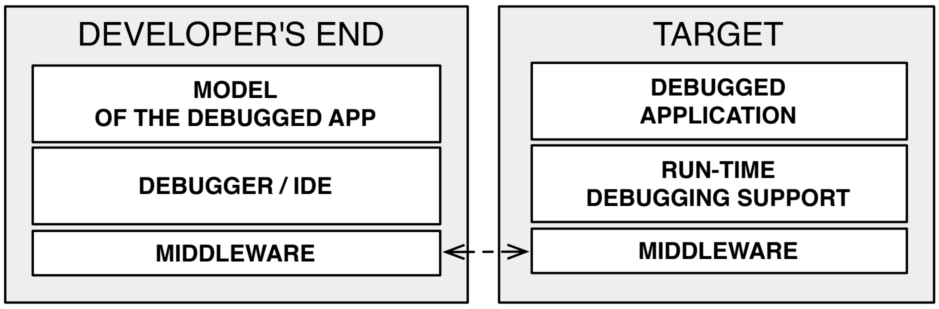


Figure 5.1: Software Entities Involved in Remote Debugging.

5.2 The Core Meta-Level

The core of our meta-level depends on two basic concepts:

The explicit meta-object [Maes 1987b] As we saw on Chapter 3 an explicit meta-object is a meta-object that is invoked only on-demand by the programmer. Our solution uses specific explicit meta-objects known as *Mirrors* (see also Section 3.3.5) defined by Bracha and Unghar as "*intermediary objects [...] that directly correspond to* *language structures and make reflective code independent of a particular implementation*" [Bracha 2004]. Mirrors through this decomposition between the reflective model and its implementation facilitate the reuse of a debugging solution for both local and remote targets.

The remote facade [Fowler 2005] The remote facade is a variant of the Facade pattern [Alpert 1998], which is used to "*provide a coarse-grained facade on fine-grained objects to improve efficiency over a network*" [Fowler 2005]. In essence the remote facade is the unique entry point on a remote target, hiding fine-grained details and providing a suitable API for communication.

In Figure 5.2 we depict our model. The meta-level located on the development machine (left part of Figure 5.2) is a set of mirrors that reflect on objects (e.g instance of the class Point) on the target side (right part of Figure 5.2). The target machine also includes support for reflection and debugging. This is the role of the package RTSupport that includes the RunTimeDebuggingSupport class (our remote facade).

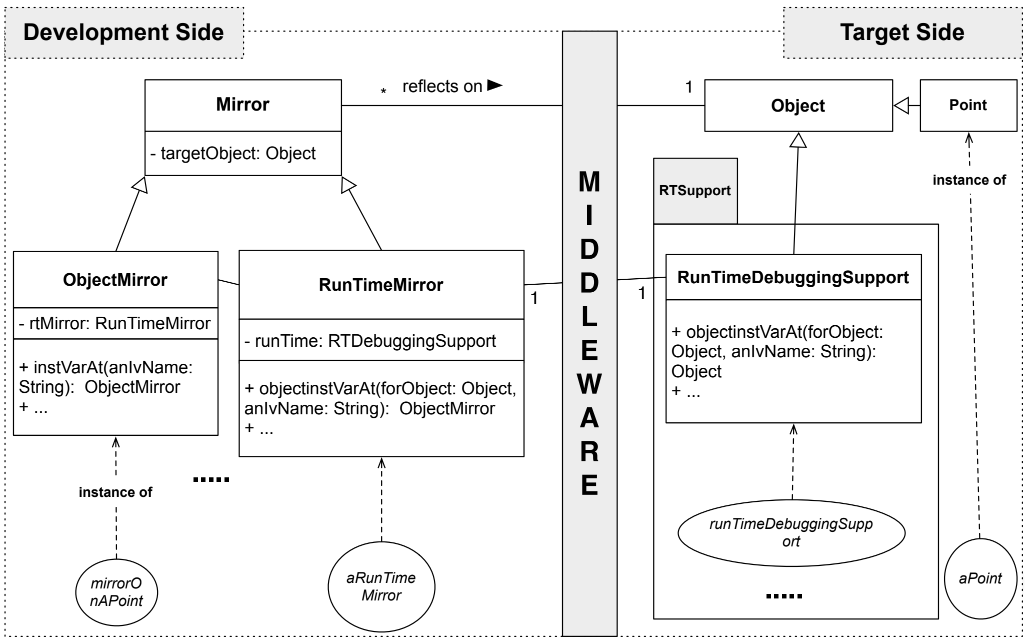


Figure 5.2: Our core model

On the left side of Figure 5.2, we depict the 3 core classes of our meta-level. The root is the Mirror class, that declares the targetObject field. So, every mirror holds a remote reference to one object on the target. Nevertheless, an object on the target can be reflected by multiple mirrors on the development side.

Both on the developer's end and on the target, a unique object is responsible to handle all communications to the other side. This object is an instance of RunTimeMirror on the developer's end and an instance of RunTimeDebuggingSupport on the target. On the developer's end, all mirrors can retrieve this object in their inherited field named rtMirror. The

API of the RunTimeMirror is completely equivalent to the one of RunTimeDebuggingSupport with one crucial difference: *each call to the target side results in a mirror or a collection of mirrors being returned to the developer's side*.

To show how communication and reflection is handled between the development machine and the target, consider the example of the mirror mirrorOnAPoint and its target object aPoint. Suppose that the developer wants to get the class of aPoint. To perform this operation, the IDE sends the getClass message to mirrorOnAPoint. As a result, mirrorOnAPoint sends the getClassOf: targetObject message to aRunTimeMirror passing as a parameter the remote reference that it holds. Then, aRunTimeMirror invokes through the middleware the getClassOf: targetObject method on runTimeDebuggingSuppport located on the target. The runTimeDebuggingSuppport retrieves the class Point and answers it back through the middleware. On the developer side, aRunTimeMirror receives a remote reference on the Point class, and creates a new mirror on the remote class. It is this mirror on the Point class that is returned back to mirrorOnAPoint.

5.3 Supporting Interactiveness

To support interactiveness, the model of the debugged application on the developer's end and the state of the debugged application on the target (cf. Figure 5.1) needs to be *causally connected*. This means that an arbitrary change in either one of them should update the other.

We describe how our model supports interactiveness through the class hierarchy and the API of our meta-level (starting from ObjectMirror). Figure 5.3 depicts 8 core classes of our meta-level which are divided into two groups: the ones that reify the structure of the debugged application (*structural reflection*) and the ones that reify the computation (*computational reflection*) [Ferber 1989, Maes 1987b].

In our model, both *structural reflection* and *computational reflection* are *causally connected* to the other side. For *structural reflection*, this means that the addition of a new package, a new class, method, etc. in the development side results in a structural update of the running application on the other side. These 8 core classes depicted in Figure 5.3 define an API that supports interactiveness. Instances of these classes reflect on remote objects on the target and all of their methods can be executed while the application is running.

ObjectMirror. An ObjectMirror enables retrieving information from the object reflected such as its class, reading/setting its fields or sending new messages to it and also changing its class (setClass).

EnvironmentMirror. It is the entry point mirror to the target application depicting the remote environment as a whole. Through the environment mirror globals are read/written, loaded packages are retrieved, interrupted processes and unhandled exceptions are accessed, code is evaluated (evaluate) and packages can be created, removed, or edited (newPackage, removePackage, etc.).

PackageMirror. A package mirror reflects on loaded packages on the target application. This mirror gives access to package's meta-information such as its name and the

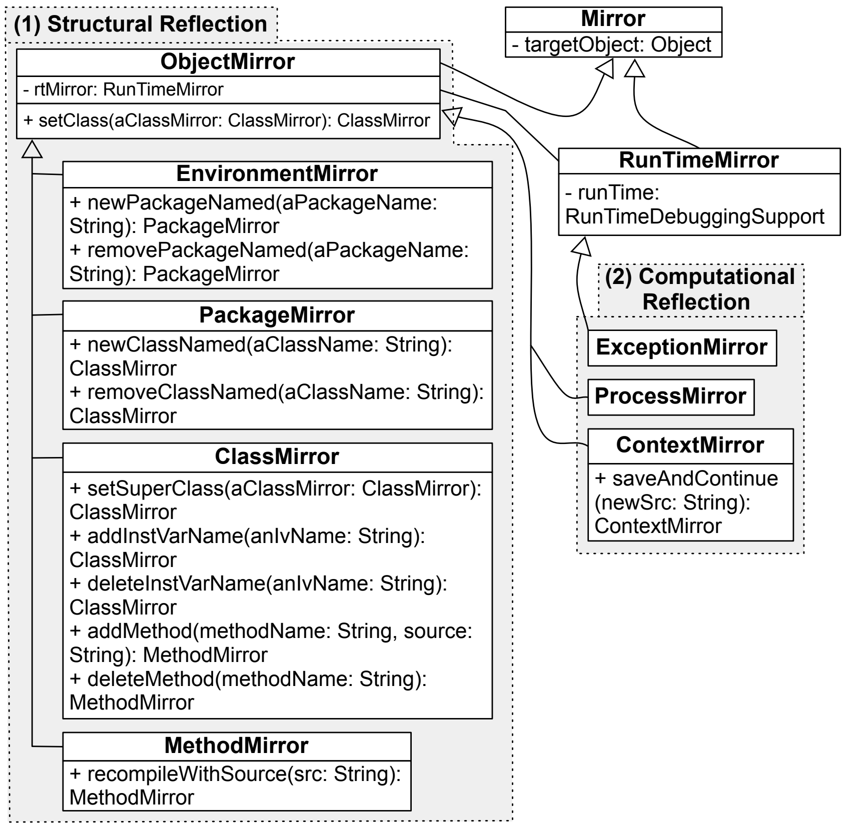

(1) Structural Reflection
ObjectMirror
- rtMirror: RunTimeMirror
+ setClass(aClassMirror: ClassMirror): ClassMirror
EnvironmentMirror
+ newPackageNamed(aPackageName: String): PackageMirror
+ removePackageNamed(aPackageName: String): PackageMirror
PackageMirror
+ newClassNamed(aClassName: String): ClassMirror
+ removeClassNamed(aClassName: String): ClassMirror
ClassMirror
+ setSuperClass(aClassMirror: ClassMirror): ClassMirror
+ addInstVarName(anIvName: String): ClassMirror
+ deleteInstVarName(anIvName: String): ClassMirror
+ addMethod(methodName: String, source: String): MethodMirror
+ deleteMethod(methodName: String): MethodMirror
MethodMirror
+ recompileWithSource(src: String): MethodMirror
Mirror
- targetObject: Object
RunTimeMirror
- runTime: RunTimeDebuggingSupport
(2) Computational Reflection
ExceptionMirror
ProcessMirror
ContextMirror
+ saveAndContinue (newSrc: String): ContextMirror

Figure 5.3: Core classes and API for supporting interactiveness

classes it contains. Classes can also be added or removed using the methods newClassNamed and removeClassNamed.

ClassMirror. Through a class mirror the name, superclass, fields, methods and enclosing package of the reflected class can be retrieved. The superclass can be changed, new instance variables and methods can be added/removed or edited (setSuperClass, addInstVarName, deleteInstVarName, addMethod, deleteMethod, etc.).

MethodMirror. Apart from retrieving the name, source or class membership of a Method, the developer can edit a method in place (recompileWithSource).

ProcessMirror. It allows one to retrieve meta-information on a process such as its stack and manipulate the execution flow.

ExceptionMirror. It is the reification of exceptions on the target. Through an exception mirror the description of an unhandled exception can be retrieved, as well as the process that it occurred and the offending execution context.

ContextMirror. It is the reification of a stack frame (context) on the target application. Through a contextMirror its process, method, receiver and sender can be retrieved, temporaries and arguments of the invocation can be read/written, its execution can be restarted but also the method that was invoked and created the context can be edited before continuing the execution (saveAndContinue).

5.4 Supporting Instrumentation

Instrumentation in our model is supported through *intercession*. Specifically the underlying execution environment is reified inside the run-time environment of the target as to be able to control the semantics of a running process.

The model of our solution uses the following patterns:

The observer [Alpert 1998] An observer defines a dependency between an object and its *dependents*, so that the dependents are notified for state changes on that object.

The implicit meta-object [Maes 1987b] Implicit meta-objects are meta-objects that are invoked automatically by the underlying execution mechanism.

Objects can be instrumented either to perform user-generated conditions and actions upon invocation of specific events (e.g RunTimeDebuggingSupport»objectOnReceive) or to halt the process on those specific events (e.g RunTimeDebuggingSupport»objectHaltOnReceive).

Figure 7.15 depicts the reification of the Interpreter (the underlying execution environment) which acts as our observer, connecting instances of Object (regular objects) to instances of ImplicitMetaObject (dependents). Whenever an event of interest is being applied to an object (such as a message send) the underlying execution mechanism invokes the Interpreter reification, which in turn notifies the ImplicitMetaObjects. The Interpreter resolves the relationship between objects and meta-objects through the MetaEnvironment, which acts as an environment dictionary for the meta-level. The MetaEnvironment provides a one-to-one mapping between objects and meta-objects.

Implicit meta-objects when notified, will invoke a callback (class Closure in Figure 7.15) which can be either a local callback or a remote callback from the developer's end. The RunTimeDebuggingSupport maintains a reference to the Interpreter reification in order to register these callbacks coming from mirrors on the developer's side.

5.5 Supporting Distribution

In order to support distribution via an adaptable middleware, we modeled our solution using the concept of the abstract Factory [Alpert 1998], through which families of related objects can be assembled and parametrized at runtime.

Figure 5.5 depicts the core classes of our model for distribution:

Middleware Deamon This abstract class defines methods for the creation and initialization of our middleware, acting as an Abstract Factory. It is also responsible for loading the RTSupport package (cf Figure 5.2) on the target upon the successful authentication of a client.

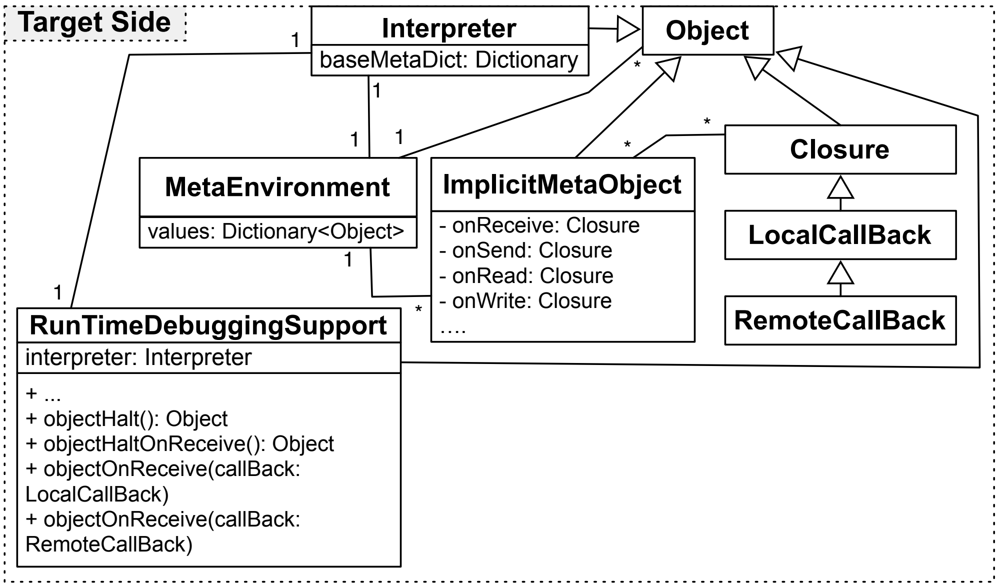


Figure 5.4: Core classes for instrumentation support in the target

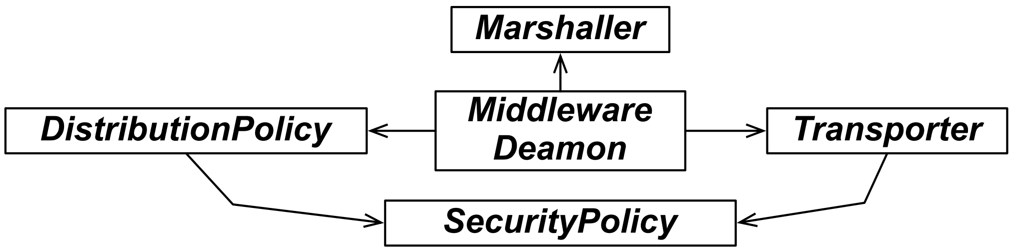


Figure 5.5: Core classes of our adaptable middleware

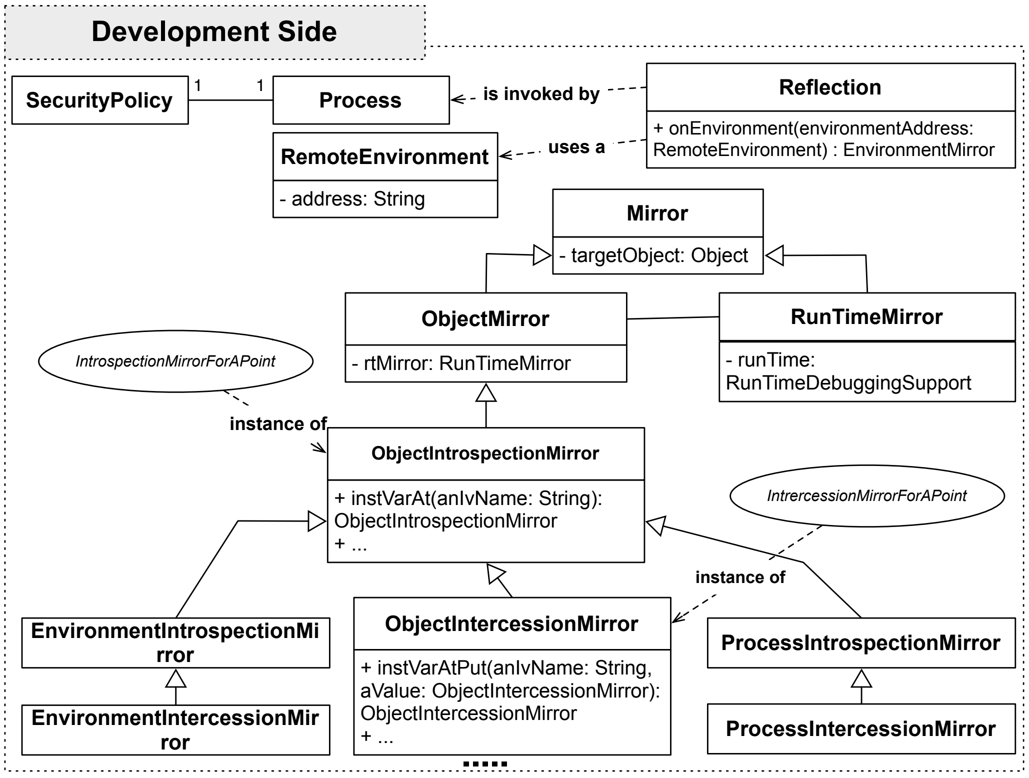
Transporter The concrete subclasses of this abstract class handle the actual communication between peers. Different transporters can support different communication protocols (e.g tcp, udp or web-sockets).

Marshaller The marshaller (through its concrete subclasses) is responsible for serializing and materializing information, passed through the connection. Different marshallers can support different transcoding algorithms to fit the needs of the debugging context (e.g serializing to xml, json or binary-form).

Distribution Policy This class (through its concrete subclasses) decides how specific objects or group of objects will be distributed among peers (ie processed when passed as parameters or results of remote messages). Options can include: full serialization, shallow serialization, proxying, etc..

Security Policy The concrete subclasses of this abstract class are responsible for authentication and for restricting access (either message sending or distribution) for specific instances or whole classes of objects.

5.6 Supporting Security



Development Side
SecurityPolicy
1
1
Process
is invoked by
Reflection
+ onEnvironment(environmentAddress: RemoteEnvironment) : EnvironmentMirror
uses a
RemoteEnvironment
- address: String
Mirror
- targetObject: Object
ObjectMirror
- rtMirror: RunTimeMirror
RunTimeMirror
- runTime: RunTimeDebuggingSupport
IntrospectionMirrorForAPoint
instance of
ObjectIntrospectionMirror
+ instVarAt(anIvName: String): ObjectIntrospectionMirror
+ ...
IntrercessionMirrorForAPoint
instance of
EnvironmentIntrospectionMirror
EnvironmentIntercessionMirror
ObjectIntercessionMirror
+ instVarAtPut(anIvName: String, aValue: ObjectIntercessionMirror): ObjectIntercessionMirror
+ ...
ProcessIntrospectionMirror
ProcessIntercessionMirror

Figure 5.6: Decomposing the meta-level hierarchy into introspection and intercession mirrors

Our model proposes the adoption of a pluggable debugging framework during development. From a security perspective this useful property can easily become a threat in an open network, if one can access and debug our applications. For this reason our model supports authentication and access-restriction mechanisms on both sides of the debugging process that can be used as basic building blocks for securing remote debugging sessions.

From the target's side the middlware is responsible for authenticating a debugging request. In the case the request is not authenticated, further access is denied and the RTSupport package (cf Figure 5.2) is not even loaded on the target. In the case of successful authentication, the target proceeds by dynamically loading the debugging infrastructure on the target (RTSupport package). From the developer's side, even if the target was granted full access to the other side, different processes or threads are exposed to different facilities.

To achieve these access restrictions, our meta-level is organized with *introspection* and *intercession* mirrors (cf. Figure 5.6). *Introspection* mirrors provide read access to their target objects and *Intercession* mirrors provide write-access or execution control to their target objects. Each time the meta-level is accessed a decision on which kind of mirror will be returned is decided depending on the security policy attached to a requesting process. This is achieved through the following steps:

1. The reflection factory is used to access the target application:

   Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'

2. The reflection factory attempts to connect to the otherside through the middleware and authenticate itself.

3. If the authentication is successful the target loads the run-time debugging support and exposes its API on the Reflection class.

4. The Reflection class then retrieves the SecurityPolicy (upper part of Figure 5.6) of the process that made the initial request.

5. According to the SecurityPolicy, the onEnvironement: method either returns a reference to an EnvironmentIntrospectionMirror (read-only access to the target application) or an EnvironmentIntercessionMirror (read/write-access to the other side).

6. Subsequent access to mirrors of packages, classes, methods, processes, objects, etc. through the initial EnvironmentMirror will have the same access-rights (for either introspection or intercession) as this initial reference.

5.7 Comparison with State Of The Art

In this Section we compare the state-of-the-art debugging solutions (which we discussed in 2) with our work in terms of *interactiveness, instrumentation, distribution and security*. For convenience we replicate here the comparison tables and commentary of Chapter 2, which are now augmented with the results of our our own work.

5.7.1 Interactiveness

	JPDA	.NET	GDB	DCE	JREBEL	ST-80	BIFROST	MERCURY
Add/Rem Packages	×	×	×	✓	✓	✓	✓	✓
Add/Rem Classes	×	×	×	✓	✓	✓	✓	✓
Add/Rem IVs	×	×	×	✓	✓	✓	✓	✓
Add/Rem Methods	×	×	×	✓	✓	✓	✓	✓
Method (Body) HotSwapping	✓	✓	✓	✓	✓	✓	✓	✓
Hierarchy Editing	×	×	×	✓	✓	✓	✓	✓

Table 5.1: Interactiveness – Comparison of our solution with state-of-the-art

As we see in Table 5.1 debugging environments of mainstream OO languages (JPDA, .Net Debugger, Gdb) do not support interactiveness with the exception of a *save-and-continue* facility for pre-existing methods. In the Java world recent developments (through

Jrebel and DCE) provide full support for interactiveness as does Smalltalk and its extension Bifrost. Our solution too which is based on Smalltalk provides full support for this property.

5.7.2 Instrumentation

In Table 5.2 we do a comparison in terms of instrumentation and its sub-properties as there were defined in Section 2.3. We have also included a last category marked as *condition/action* that describes whether in all instrumentation events the debugging solution can support user-generated checks and code in order to provide a more fine-grain control.

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST	MERCURY
Method Execution	✓	✓	✓	✓	✓	✓	✓	✓
Statement Execution	✓	✓	✓	✓	✓	✓	✓	✓
Field Read	✓	×	×	✓	✓	×	✓	✓
Field Write	✓	×	×	✓	✓	×	✓	✓
Object Read	×	✓	✓	×	×	×	✓	✓
Object Write	×	✓	✓	×	×	×	✓	✓
Object Send	×	×	×	×	×	×	✓	✓
Object Receive	×	×	×	×	×	×	✓	✓
Object as Argument	×	×	×	×	×	×	✓	✓
Object Creation	×	×	×	×	×	×	✓	✓
Object Interaction	×	×	×	×	×	×	✓	✓
Object Stored	×	×	×	×	×	×	×	✓
Condition/Action	×	×	✓	×	×	✓	✓	✓

Table 5.2: Instrumentation – Comparison of our solution with state-of-the-art

As we can see from our comparison, Bifrost and our solution are the front-runners of instrumentation with all other solutions supporting only non-OO breakpoints and watchpoints. In contrast with Bifrost we also support the *Object Stored* event which is usefull for following an object's reference propagation and counting. Finally as we can see our solution together with both Bifrost and Gdb provide support for both conditions and actions on instrumentation events.

Since both our solution and Bifrost are based on Smalltalk, we were also able to perform a micro-benchmark to compare the two, in terms of the overhead introduced by instrumentation. The benchmark is based on Tanter [Tanter 2003] and the Bifrost metrics are those reported in [Ressia 2012a]. The benchmark measures the slowdown introduced by each solution for one million messages send to a test object when a) no instrumentation is present b) instrumentation is loaded but is disabled for this specific object and c) instrumentation is enabled on the test object of the micro-benchmark.

	BIFROST	MERCURY
No instrumentation	1x	1x
Disabled instrumentation	1x	1x
Enabled instrumentation	35x	8x

Table 5.3: Instrumentation benchmark for Bifrost and Mercury

As we see in Table 5.3 for both solutions there is no overhead introduced when a specific object is not being instrumented, regardless of whether the solution is loaded into the environment. This is important for practical reasons so as to avoid slowing down the whole system while debugging. While instrumenting a specific object our solution introduces a significantly smaller overhead than Bifrost. We believe that this is due to the fact that our solution is based on the underlying virtual-machine rather than on byte-code manipulation as in the case of Bifrost.

On the other hand since Bifrost does not need additional support from the virtual machine it can be used with any smalltalk vm including those supporting just-in-time compilation.

5.7.3 Distribution

In Table 5.4 we do a comparison in terms of distribution. Solutions are marked with - for not supporting distribution, + for supporting distribution through a fixed-middleware, ++ for an extensible middleware and +++ for an adaptable middleware.

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST	MERCURY
Distribution	+	++	+	+	+	-	-	+++

Table 5.4: Distribution – Comparison of our solution with state-of-the-art

As we can see in Table 5.4 our solution is the only one that supports an adaptable middleware with the .net debugging framework following using a general purpose and extensible communication solution (DCOM) [Microsoft 2013]. We should note here that in the case of Smalltalk, there were some efforts in the past to support remote development (including debugging) in Cincom Smalltalk, which were discontinued.

5.7.4 Security

In Table 5.5 we do a comparison in terms of security support while debugging as was described in Section 2.3:

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST	MERCURY
Built-in	×	✓	×	×	×	×	×	✓
External	✓	✓	✓	✓	✓	×	×	✓
Target-Side	✓	✓	✓	✓	✓	×	×	✓
Developer-Side	✓	✓	×	✓	✓	×	×	✓

Table 5.5: Security – Comparison of our solution with state-of-the-art

As we can see in Table 5.5 only our solution and the .net debugging framework has build-in provisions for security [Microsoft 2012a] for both the target and the developer side. In the Java world though (JPDA, DCE, JREBEL) there are other frameworks that are used in conjunction with JPDA in order to secure the debugging session such as the Java Security Manager [Oracle 2013c]. Gdb specifically warns developers not to use its remote debugging facilities in public networks [Richard Stallman 2003] and has no built-in provisions for access restrictions in the client side either. In this case the developer can only resort to external solutions such as a firewall or a VPN. For Smalltalk as far as the local debugging scenario is concerned there are no security provisions.

5.7.5 Comparison overview

In Table 5.6 we present an overview of our comparison in terms of all properties in terms were described in Section 2.3:

Property	**JPDA**	**.NET**	**GDB**	**DCE**	**JREBEL**	**SMALLTALK**	**BIFROST**	**MERCURY**
Interactiveness	**+ (1/6)**	**+ (1/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**
Instrumentation	**+ (4/13)**	**+ (4/13)**	**+ (5/13)**	**+ (4/13)**	**+ (4/13)**	**+ (3/13)**	**+++ (12/13)**	**+++ (13/13)**
Distribution	**+ (fixed)**	**++ (extensible)**	**+ (fixed)**	**+ (fixed)**	**+ (fixed)**	**- (no)**	**- (no)**	**+++ (adaptable)**
Security	**+++ (3/4)**	**++++ (4/4)**	**++ (2/4)**	**+++ (3/4)**	**+++ (3/4)**	**- (0/4)**	**- (0/4)**	**++++ (4/4)**

Table 5.6: Summary – Comparison of our solution with state-of-the-art

As we can see from Table 5.6 our solution manages to cover all four properties of Section 2.3 being comparable to the Bifrost framework (in the local scenario) in terms of interactiveness and instrumentation. In our case though these properties are brought to remote debugging through an adaptable middleware that has built-in provisions for both ends of the debugging session.

5.8 Summary

In this Chapter we have proposed a mirror-based model and an infrastructure for remote debugging in reflective languages. Our solution exhibits the four desirable properties that we have discussed in Chapter 2, namely: *interactiveness*, *instrumentation*, *distribution* and *security*. Our solution *Mercury*, supports interactiveness through a causal connection between the meta-level running on the developer machine, and the application to debug (the base-level) on the target device. The two levels are connected both computationally and structurally. It supports instrumentation through the reification of the underlying execution environment (virtual-machine) inside the run-time environment of the target (as an interpreter). Distribution is supported through an adaptable middleware. Finally it supports security in a remote debugging setting by organizing its reflective facilities into two different access groups for - respectively - introspection and intercession. Subsequently, we gave a comprehensive comparison of our solution with state-of-the-art solutions. We concluded that in contrast with related work, our approach can in fact meet all the criteria that we have discussed in Chapter 2.

CHAPTER 6

Mercury: Implementation Details

Contents

At a Glance

In this Chapter we present a prototype implementation of our proposed model for remote debugging in reflective languages. We start by giving an overview of the core parts and technologies of the prototype, namely: the dedicated virtual-machine for the target (*metaStackVM*), the adaptable middleware (*seamless*), the meta-level and run-time debugging support (*mercury-core*) and an experimental debugging front-end (*alexandria*) for the framework. Each of these parts is then detailed separately to illustrate their connection to our model. Finally we discuss engineering trade-offs that other implementors of our model should take into account when implementing our solution.

6.1 Implementation Overview

We implemented a prototype[1] of the Mercury model (described in Chapter 5) in Pharo [Black 2009] and Slang [Ingalls 1997]. Pharo is a reflective, object-oriented and dynamically typed programming environment that is inspired by Smalltalk. Slang is a subset of the Smalltalk syntax with procedural semantics that can be easily translated to C. In Figure 6.1 we show the different constituents of our implementation.

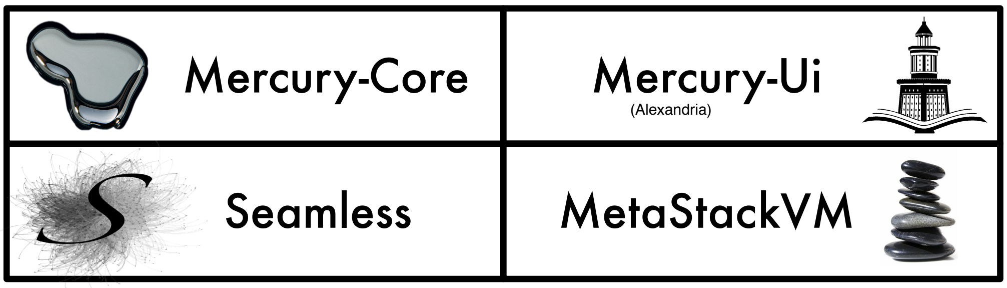


Figure 6.1: Core parts of Mercury's Prototype

MetaStackVM Is a dedicated virtual-machine for debugging targets, that extends Pharo's reflective facilities in order to support intercession.

Seamless Is our adaptable middleware that provides flexible communication facilities betweens peers during debugging sessions.

Mercury-Core Is the sub-project of Mercury that hosts the debugging meta-level and the debugging run-time support.

Mercury-UI Is a debugging front-end that exemplifies key functionalities of our solution.

All four part of our prototype implementation for Mercury are released under the MIT license [2].

6.2 MetaStackVM: Low-level Instrumentation support

Instrumentation in our prototype depends on a dedicated virtual machine: the metaStackVM [3]. We have implemented the metaStackVM by extending the standard Stack VM[4] of Pharo, in Slang [Ingalls 1997]. Despite having procedural semantics Slang provides some pseudo-OO abstractions. For example procedural module inclusion is presented in Slang as OO inheritance without polymorphism.

[1]http://ss3.gemstone.com/ss/Mercury-Prototype.html
[2]http://opensource.org/licenses/MIT
[3]http://ss3.gemstone.com/ss/mSVM.html
[4]https://ci.inria.fr/pharo/view/VM/job/PharoSVM/

6.2.1 Extending the Stack VM

On the upper part of Figure 6.2 we show the core entities of the standard Stack VM:

VMClass Hierarchy root of VM classes, provides common facilities related to source code translation.

ObjectMemory A direct-pointer object-oriented representation of program memory. Provides facilities for header access, garbage collection and so on.

StackInterpreter A byte-code interpreter. Pharo's particular implementation optimizes execution by mapping contexts to stack frames and lazily instantiating contexts.

On the lower part of Figure 6.2 we show our extensions for core vm classes in the metaStackVM:

MetaObjectMemory We extended the object memory of Pharo to be able to mark objects, whose semantics should be overridden. This is one way to efficiently implement partial reflection on objects (see definitions on Chapter 2) since the overhead of passing control to the meta-level is only paid for objects that have been marked.

Marking is done on the object header of instances which can be readily and efficiently be tested through single bitwise machine instructions on the vm level (methods getMetaLevelBit, setMetaLevelBit, unsetMetaLevelBit). Our implementation for bit handling on object headers is based on the approach for object tracing used by the Marea project [Mariano 2012].

MetaStackInterpreter We also extended the byte-code interpreter of the stack vm, overriding all instance-level reification categories (see listing on Chapter 2). These overrides check objects headers for their meta-level status and reifies semantic operations (such as message sending, field read, write e.t.c) into message objects which are then passed to the language environment (see Interpreter reification of Figure 7.15).

6.2.2 Implementation of Meta-Level Control

On the language side besides the entities we described on Chapter 5 (Figure 7.15) regarding instrumentation, we also implemented a dedicated meta-control solution. We saw on Chapter 3 that meta-level facilities need to deal with the problem of meta-recursion. In our implementation the meta-recursion problem may manifest itself when control is passed to the meta-level (i.e to the ConditionedMetaAction closure of Figure 6.3). We illustrate this problem by discussing the following code snippet:

Script 1: The meta-recursion problem

```
Interpreter on: MessageReceived for: anObject do: [:reifications |
```

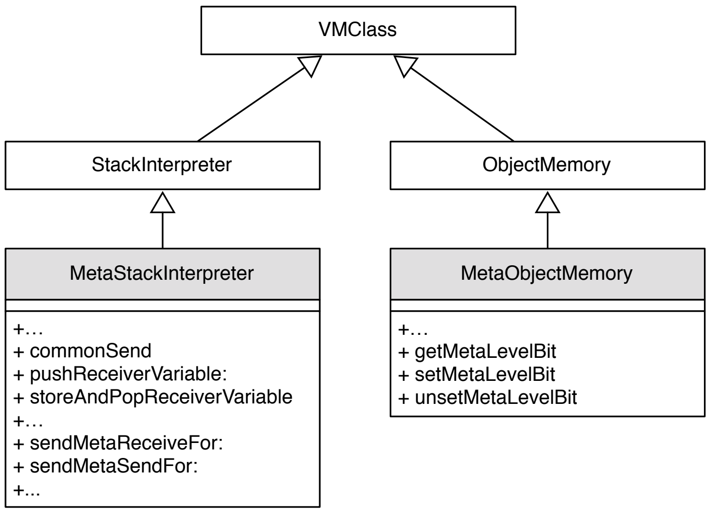


Figure 6.2: Extending the standard stack VM in Slang

```
2      anObject incrementMessageCounter.
3      anObject perform: reifications selector withArguments: reifications arguments.
4 ].
```

On Script 1 we are using the metaStackVM intercession facilities to implement message counting for the instance: *anObject*. On line 1 through the Interpreter reification (upper left class on Figure 6.3) we register a callback for the *MessageReceived* event on *anObject*. This registration in our implementation will create an entry on the class *MetaEnvironment* (right part of Figure 6.3), linking the object with a particular action for this event. This ultimately means that the object's header will be marked to override its base-level behavior. Upon enabling the meta-behavior on *anObject* the meta-level callback (lines 1 to 4 on Script 1 and class ConditionedMetaAction on Figure 6.3) will be automatically invoked by the vm upon every message send. As we can see this callback also receives an argument upon invocation with meta-information concerning the message send itself (i.e the selector of the message, its arguments e.t.c).

Then on line 2 *anObject* receives a message inside the meta-level callback to increment a message counter. One side of the meta-recursion problem can manifest itself in this very first call. If *anObject* is not un-marked before entering the meta-level then the messageSend *incrementMessageCounter* will itself be intercepted, resulting in an infinite recursion. The same is true for line 3 where we send the message *perform:withArguments:* to actually perform the initial message send. Oddly enough we may now want to re-enable the meta-level behavior either after line 3 (the more usual case) or **during** the invocation of the initial message send (if we want to also capture self sends for example).

On the context of debugging these problems may appear when we interrupt an execution for inspection. During execution we want instrumentation to be enabled on desired objects but during inspection we do not want to trigger new meta-events.

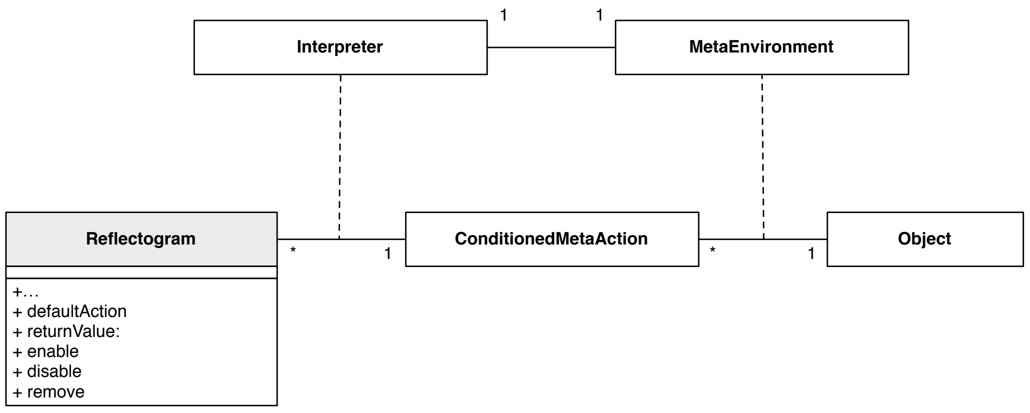


Figure 6.3: Reflectogram: Run-time meta-level control

In our implementation in order to address these issues we reify an object's reflectogram (left part of Figure 6.3), which is passed as a second argument to the meta-level. The notion of the reflectogram was introduced by Tanter et al. [Tanter 2003] as a conceptual illustration (see Figures 6.4, 6.5):

[...] A reflectogram illustrates the control flow between the base level and the metalevel during execution. Using full reflection (Fig. 1a), any operation at the base level is reified and therefore many –possibly useless– shifts occur. This does not occur with partial reflection (Fig. 1b).

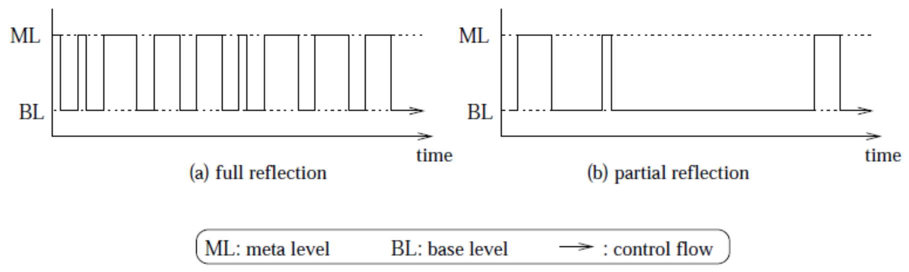


Figure 6.4: Tanter's reflectogram [Tanter 2003] depicting full (left) versus partial reflection

Denker et al. [Denker 2008] present an alternative view of the reflectogram to depict multiple meta-levels and the meta-recursion problem. Denker et al. propose the reification of the *metaContext* which represents the level in which a meta-jump occurs. The metaContext is an implicit entity of the meta-level, in the sense that the developer does not invoke it explicitly but rather executes code or binds meta-objects to specific meta-levels (see Scripts 2 and 3):

Script 2: Code execution with a metaContext [Denker 2008]

```
[ ... code executing on meta-1 ... ] valueWithMetaContext
```

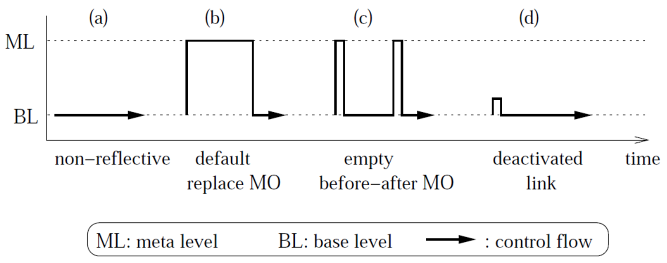


Figure 6.5: Tanter's reflectogram [Tanter 2003] depicting different kinds of meta-jumps.

```
2     [[ ... code executing on meta-2 ... ] valueWithMetaContext] valueWithMetaContext
```

Script 3: Binding to a specific meta-level [Denker 2008]

```
beepLink := Link new metaObject: Beeper.
beepLink selector: #beep;
beepLink level: 0. "a link that is active only when executing base level code"
```

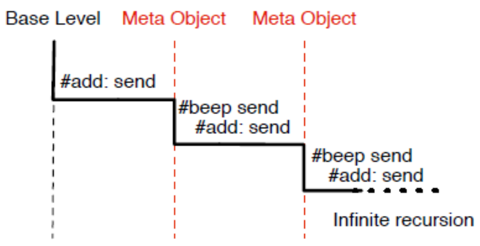


Figure 6.6: A reflectogram illustration from Denker [Denker 2008] depicting multiple meta-levels

Our reification of the conceptual notion of the reflectogram draws upon this idea of the metaContext. In our case though the reflectogram is an explicit entity (it is invoked by the programmer) in contrast to the metaContext (which is implicit). This enables us to deal with more aspects of meta-control such as:

- dynamically enabling/disabling/rebinding or removing meta-level behavior from within the meta-level (see methods enable, disable, remove, on:do: on Figure 6.3)
- dynamically controlling *temporal placement* (pre, post actions) of meta-jumps from within the meta-level (methods defaultAction and returnValue:)
- describing the meta-level per process (as in the metaContext reification) and per object (methods processMetaLevel and objectMetaLevel)

- provide common reflective methods for objects that are decomposed from the language kernel (methods object:at:, object:at:put:, object:perform:), to be able to override the reflectogram's own behavior (i.e performing reflective operations with the reflectogram enabled but without intercepting them).

To enable the use of the reflectogram in a meta-callback the developer has to register the callback with a second argument (see example on Script 4):

Script 4: The meta-recursion problems solved with the reflectogram

```
1 Interpreter on: MessageReceived for: anObject do: [:reifications :reflectogram |
2       reflectogram disable.
3       anObject incrementMessageCounter.
4       reflectogram enable.
5       reflectogram returnValue: reflectogram defaultAction.
6 ].
```

On Script 4 we register the same callback as in Script 1 but we are now using the reflectogram to solve the meta-recursion problems we discussed. Now on line 2 (just before incrementing the messageCounter) we disable the reflectogram so when on line 3 we will send a message to *anObject* we will not end up in an infinite meta-recursion loop. Moreover on line 4 we re-enable the reflectogram *before* the execution of the default action (line 5) (for which the meta-jump occured). This ensures that all message-sends (even self sends) will be intercepted when the default action is executed as if the instance was traced. If lines 4 and 5 were reversed (i.e if we enable the reflectogram after the defaultAction) then a typical proxy interception of message sends will have occurred. Finally as we see on line 5 the returnValue of the meta-callback can be explicitly set. The return value can be any valid expression (other than the defaultAction of the reflectogram) for cases when we want the meta-level not just intercepting but actually replacing the default behavior.

6.3 Seamless: a Framework for Adaptable Distribution

Seamless [5] is an extendable and adaptable framework for distributed computing. Its implementation follows the general model for an adaptable debugging middleware, which we described in Chapter 5 (see Figure 5.5). It has also been independently (from Mercury) used as a library in the Continuous integration services of Pharo[6]. We chose to implement our own middleware layer for Mercury since other available frameworks for Pharo (such as rST [7]), were neither adaptable nor robust enough for our use case.

[5]http://ss3.gemstone.com/ss/Seamless.html
[6]http://smalltalkhub.com/#!/~Pharo/ci
[7]http://www.squeaksource.com/rST/

6.3.1 Low-level communication infrastructure

On Figure 6.7, we provide an overview of the low-level communication infrastructure of Seamless:

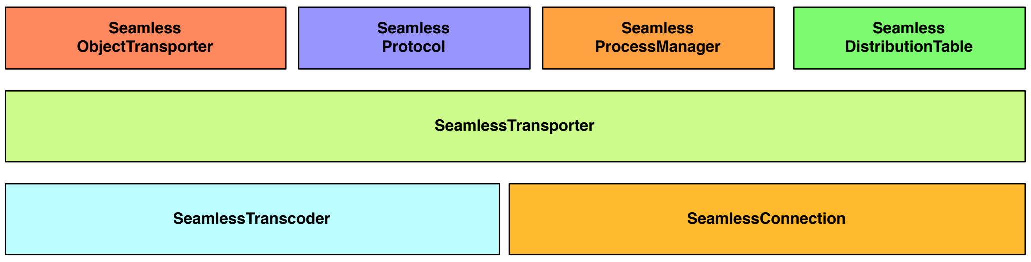


Figure 6.7: Low-level architectural overview of our adaptable middleware

SeamlessConnection Low-level bidirectional asynchronous communication abstraction. A seamless connection has the same api regardless of the underlying communication protocol or medium. Currently Seamless can operate both over plain sockets with its own protocol and over HTTP. The Mercury prototype uses plain sockets to increase speed and reduce bandwidth. Other options could include udp, web-sockets etc.

SeamlessTranscoder This is our marshaller which is responsible for serializing and materializing information, passed through the connection. As we saw in Chapter 5 different marshalers can support different transcoding algorithms to fit the needs of the debugging context. Currently Seamless operates by wrapping a fast binary marshaller: Fuel [Dias 2011], but also a simpler yet more verbose string marshaller has been tested. Other options could include serializing to xml, json etc.

SeamlessTransporter The concrete subclasses of this abstract class handles the actual communication between peers. Each concrete transporter knows about the communication channels supported by the peers (protocols, types of sockets etc.) and establishes an appropriate SeamlessConnection between them.

SeamlessObjectTransporter This is an OO abstraction for the transporter. Instead of sending or receiving bytes, plain-text, xml etc. higher-level components of the Seamless framework exchange objects through this class. These objects are instances of one the SeamlessProtocol classes.

SeamlessProtocol This is a whole hierarchy of classes, that defines an open object protocol. As we saw, connected peers with Seamless exchange objects (through the object transporter). These objects are instances of one of the SeamlessProtocol classes that contain both data (other objects) and meta-data (describing the semantics of the object exchange, message-passing infromation etc.).

SeamlessProcessManager While a SeamlessConnection is asynchronous by itself the ProcessManager can create its own blocking strategy by listening to asynchronous communications and suspending or resuming requesting processes on-demand. Mercury for example builds a synchronous communication channel on top of Seamless.

SeamlessDistributionTable This is an actual reference table keeping track of remote references, as well as local references of objects from other peers. Remote referencing in Seamless is also adaptable with two tested available implementations. The first one uses Ghost [Martinez Peck 2011] which is a uniform, light-weight and stratified general purpose proxy model, while the second one is more specialized and is based on shadow classes.

6.3.2 High-level communication orchestration

Figure 6.8 provides an overview of the high-level communication orchestration components of Seamless. These depend on the low-level communication infranstructure, through *SeamlessSession* which has a one-to-one relationship with *SeamlessConnection* on Figure 6.7.

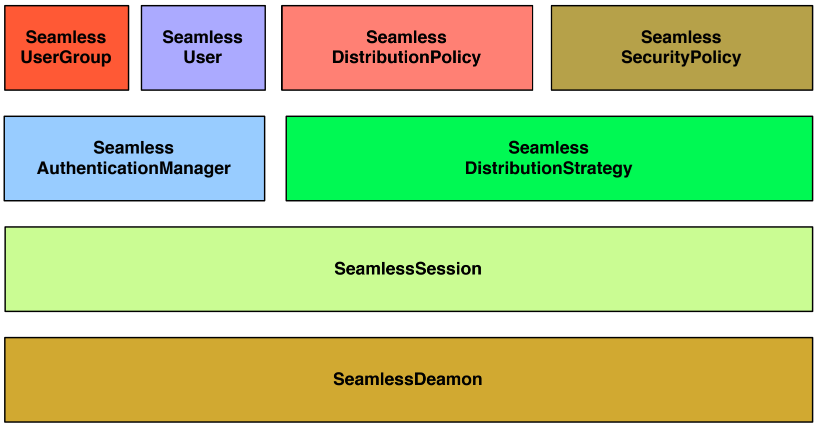

Seamless UserGroup
Seamless User
Seamless DistributionPolicy
Seamless SecurityPolicy
Seamless AuthenticationManager
Seamless DistributionStrategy
SeamlessSession
SeamlessDeamon

Figure 6.8: High-level architectural overview of our adaptable middleware

SeamlessDeamon As we saw in Chapter 5 this class defines methods for the orchestration (assembling) and initialization of our middleware, acting as an Abstract Factory. Deamons can handle multiple connections (see also SeamlessSessions) to different peers and each environment can have multiple deamons (with different adaptations) running parallely.

SeamlessSession This is a high-level view of a SeamlessConnection that is established upon successful authentication on both sides.

SeamlessAuthenticationManager A simple authentication manager (see users and groups below).

SeamlessUserGroup A user group is a named set of users that is associated with a specific distribution strategy.

SeamlessUser This is a standard/user password pair.

SeamlessDistributionStrategy A distribution strategy combines a distribution policy with a security policy (see below) to make decisions about whether or how communicaton will proceed.

SeamlessDistributionPolicy This class (through its concrete subclasses) decides how specific objects or group of objects will be distributed among peers. Options can include: full serialization, shallow serialization, proxying, etc..

SeamlessSecurityPolicy The concrete subclasses of this abstract class are responsible for authentication and for restricting access (either message sending or distribution) for specific instances or whole classes of objects.

6.4 Mercury-Core: Meta-level and Run-Time support

The Mercury-Core sub-project is itself divided into two separate configurations: the *Mercury-Core-Developer* configuration and the *Mercury-Core-Target*. The developer configuration (see left part of Figure 6.9) includes all prerequisites for the developer's end of a remote debugging session. The target configuration (righ part of Figure 6.9) includes the minimum support for a target that needs to be remotely debugged. This separation illustrates the stratification principle of mirrors which we described in Section 3.3.5. Scripts 5 and 6 present the configuration process on both ends.

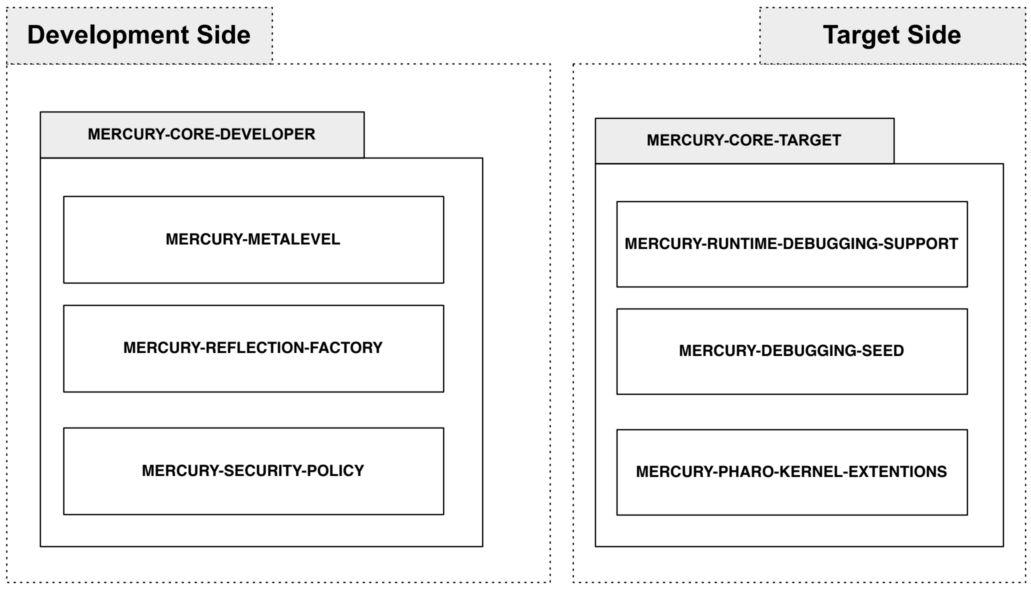


Figure 6.9: Organization of Mercury-Core on both sides of a debugging session

Script 5: Loading Mercury-Core on the developer's side

```
Gofer it
url: 'http://ss3.gemstone.com/ss/Mercury-Prototype';
package: 'ConfigurationOfMercuryPrototype';
load.
((Smalltalk at: #ConfigurationOfMercuryPrototype) project version: 'dev') load.
```

Script 6: Loading Mercury-Core on the target's side

```
Gofer it
url: 'http://ss3.gemstone.com/ss/Mercury-Prototype';
package: 'ConfigurationOfMercuryPrototype';
load.
((Smalltalk at: #ConfigurationOfMercuryPrototypeTarget) project version: 'dev') load.
```

On Figure 6.9 we also show the package organization for both configurations. The developer configuration includes the following packages:

Mercury-Metalevel This package includes the classes implementing the meta-level. Our implementation in this case follows exactly the model we described in Figures 5.2 and 5.3 on Chapter 5.

Mercury-Reflection-Factory This package contains the implementation for the mirror factory (see Figure 5.6 on Chapter 5). In the implementation we added one additional level of indirection, with the top-level reflection factory in the hierarchy delegating to more concrete factories that create the actual mirrors. Depending on the context the reflection factory may delegate to a concrete factory for local reflection, remote reflection, introspection, intercession and so on. From an implementation point of view this additional decomposition makes the framework easier to extend, thus the placement of all factory related classes in a separate package.

Mercury-Security-Policy This package contains the implementation of different security policies for meta-level access (see Figure 5.6) on the development side. In our implementation subclasses of SecurityPolicy are responsible for resolving the relation between the class of an object and its corresponding mirror class.

The configuration of the target includes the following packages:

Mercury-RunTime-Debugging-Support The run-time debugging support class (Figure 5.2) is included in this package, closely following the model we described in Chapter 5.

Mercury-Debugging-Seed This package includes the minimal support from the target's side that is needed to initiate the debugging process. The debugging seed (cf. Figure 6.10) supports minimal network capabilities via the middleware to receive new debugging requests (methods authenticate and loadDebuggingSupport) for a debugging session to be initiated.

In our implementation the debugging seed is also responsible for maintaining the running context of an application between debugging sessions (methods interrupted-Processes and unhandledExceptions). Through the debugging seed we can perform *test-runs* with our prototype. A *test-run* is the evaluation of an execution with as little interference from the debugging framework as possible. It allows analyzing the behavior of the application as near as possible to normal execution.

Debugging with a seed can be done either in a pull or push mode. In the pull mode, the application is deployed and started on the target together with the debugging seed set to be a server. The developer observes the external behavior of the system that is the target device and the application. For example, if the system is a robot, the developer will watch how the robot moves and reacts when detecting obstacles. Then, the developer connects to the seed server, loads the run-time debugging support and starts debugging.

In the push mode, the run-time debugging support is initially deployed with the application on the target. The developer then installs the seed on the target and instructs it to connect to the IDE when some condition occurs (e.g. some exception). Next, the run-time debugging support is unloaded from the target and the application is started. When the condition set by the developer occurs, the debugging seed reconnects to the developer machine. It loads back the run-time debugging support and thus triggers debugging.

Script 7 shows an example of a test-run. On line 2 the developer's side (via anEnvironmentMirror) registers a callback for new incoming exceptions and then on line 3 it invokes the method run instructing the target to run with minimal debugging support (i.e to unload via the debugging seed the debugging support) until a new exception occurs. If a new exception occurs during the test-run, the debugging support is reloaded and the developer's side is notified.

Script 7: Performing a test-run

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 anEnvironmentMirror onRemoteExceptionDo: [:aRemoteExceptionMirror | ... ].
3 anEnvironmentMirror run: 'Robot beginRandomWalk'.
```

Mercury-Pharo-Kernel-Extensions This is an implementation specific package. It includes extensions to key kernel classes of Pharo with behavior related to our framework. Especially for processes we extended the default reification of Pharo (class Process) to be initialized with a security policy for mercury.

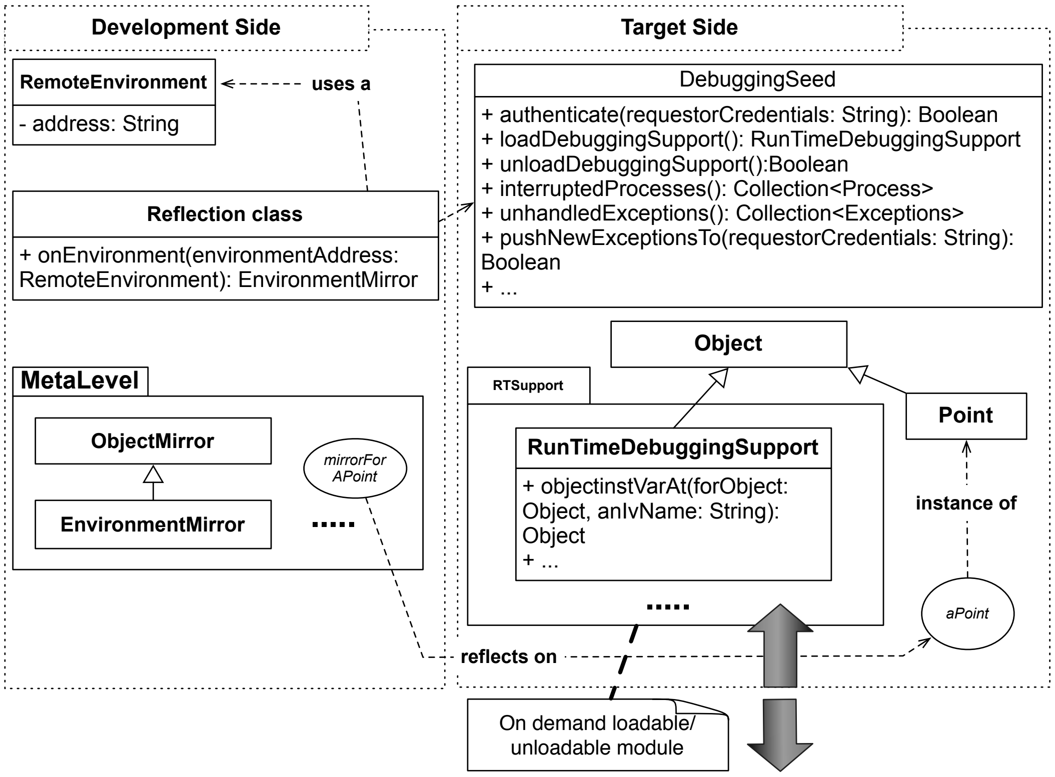

Development Side
Target Side
RemoteEnvironment
- address: String
uses a
Reflection class
+ onEnvironment(environmentAddress: RemoteEnvironment): EnvironmentMirror
DebuggingSeed
+ authenticate(requestorCredentials: String): Boolean
+ loadDebuggingSupport(): RunTimeDebuggingSupport
+ unloadDebuggingSupport():Boolean
+ interruptedProcesses(): Collection<Process>
+ unhandledExceptions(): Collection<Exceptions>
+ pushNewExceptionsTo(requestorCredentials: String): Boolean
+ ...
Object
RTSupport
Point
RunTimeDebuggingSupport
+ objectinstVarAt(forObject: Object, anIvName: String): Object
+ ...
instance of
aPoint
MetaLevel
ObjectMirror
EnvironmentMirror
mirrorFor APoint
.....
reflects on
On demand loadable/ unloadable module

Figure 6.10: Entities involved in the initialization of a debugging session

6.5 Alexandria: The Mercury Front-End

The experimental front-end (codenamed *Alexandria* [8]) of the Mercury prototype, follows a simple MVC pattern [Krasner 1988] which we depict on Figure 6.11.

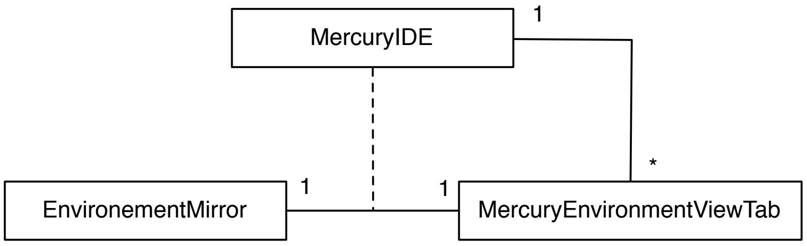


Figure 6.11: Alexandria MVC architecture

MercuryIDE The MercuryIDE class plays the role of the Controller. Since the mercury-ui can simultaneously debug multiple targets (local or remote) in different tabs it has a one-to-many relationship with our View class (MercuryEnvironmentViewTab).

[8]http://ss3.gemstone.com/ss/Mercury-Prototype.html

MercuryEnvironementViewTab This class is our View (see Figure 6.12), it has an implicit one-to-one relationship with the model (through the Controler).

EnvironementMirror The Model is the actual mirror hierarchy of the Mercury prototype whose root is the environment Mirror. Script 8 shows how we can open the Mercury ui on a particular target.

Script 8: Opening the Mercury ui on a particular target

```
MercuryIDE new
    openWithEnvironmentMirror: (Reflection on: RemoteEnvironment @ '127.0.0.1:8081').
```

6.6 Discussion: Implementation trade-offs

6.6.1 Supporting Interactiveness

Implementors of our model have essentially two options for supporting interactiveness through the RunTimeDebuggingSupport (depicted in the left side of Figure 5.2):

(a) Local reflection Local reflection on the target can be used to provide the corresponding API for interactiveness. This solution is applicable to languages that already provide a rich set of local reflective facilities. It is also a portable and extensible solution since the debugging support is written in the same language as the target application.

(b) Virtual Machine support Debugging support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with languages that do not support advanced reflective facilities on their own. It is also an attractive option for system debugging, in cases where core language reflection itself has to be debugged. This solution is less portable and extensible if it is not supported by the vendor of the target language.

In our prototype we used a combination of the two approaches mentioned above for interactiveness. Remote reflection on the instance level is separated from local reflection on the target and can thus support some limited form of system debugging. However, we also make use of local reflective facilities on the target for system-organization reflection (packaging meta-objects) and computational reflection (reifications of contexts and processes).

Our implementation currently depends on the compiler on the target. Ideally, the developers' end compiler should be used and the target should not host a compiler itself to further minimize the footprint.

Finally in the case of security checks the Mercury prototype extends the reification of Processes in Pharo to hold meta-information related to our framework. The implemented prototype though can still be vulnerable to an attack if local reflection on the developer's side and shared state between processes are used to circumvent our restrictions. This is acceptable in order to validate a solution for security in the context specifically of remote debugging but not in the general case of security for reflective languages which have deeper issues [Caromel 2001].

6.6.2 Supporting Instrumentation

To support instrumentation the following options can apply:

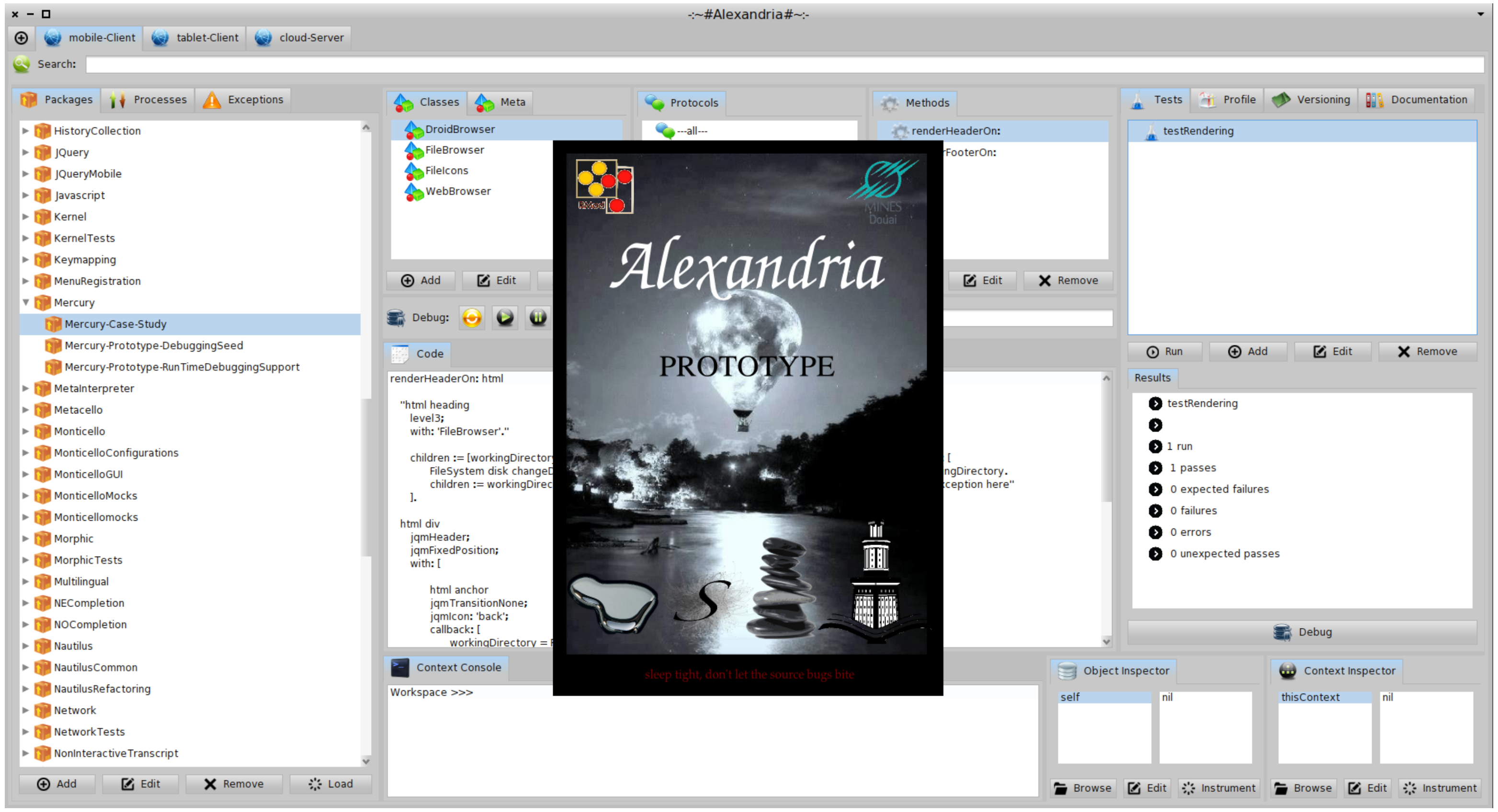
(a) Bytecode Manipulation The compiler can be used to re-compile part of the system to transparently introduce crosscuts that perform instrumentation checks (for message sending, field access, etc.). This solution has the disadvantage of instrumenting only static entities (such as classes or methods) and may perform poorly when specific objects (runtime entities) need to be instrumented. For example when instrumenting message sending on a specific object, all the methods of its class and its superclasses have to be re-compiled to introduce the crosscuts. On the other hand in the case of a self-hosted compiler this option favors portability.

(b) Virtual Machine support Instrumentation support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with instrumentation of run-time entities, since the checking can be performed on the object itself while it is being interpreted by the underlying execution environment. Portability may be an issue in this case if instrumentation is not supported by the vendor.

In our prototype as we saw in Section 6.2.1 we supported instrumentation by extending the stack-based virtual machine of Pharo. We chose to provide virtual-machine support since our focus was on instrumenting run-time rather than static entities. Furthermore we did not wish to have further dependencies on the compiler of the target.

6.7 Summary

In this Chapter we presented a prototype implementation of our proposed model for remote debugging in reflective languages. The Mercury prototype which is written in Pharo [Black 2009] and Slang [Ingalls 1997] consists of four parts: The dedicated virtual-machine for the target (*metaStackVM*) which was detailed both from the point of view of the underlying execution environment and from the language's side, where we saw how we solved in practice the meta-recursion problem. The adaptable middleware (*seamless*), for which we gave both a low-level view of the communication infrastructure and a higher-level overview for communication orchestration. The meta-level and run-time debugging support (*mercury-core*) for which we detailed its structural organization, installation and initialization through a dedicated debugging seed. Subsequently our experimental front-end (*alexandria*) was presented which follows an MVC pattern [Krasner 1988] with our mirror-based meta-level as the model. Finally we discussed engineering trade-offs for implementors of our model.



```
renderHeaderOn: html

  "html heading
    level3;
    with: 'FileBrowser'."

    children := [workingDirector
        FileSystem disk changeD
        children := workingDirec
    ].

  html div
    jqmHeader;
    jqmFixedPosition;
    with: [

        html anchor
        jqmTransitionNone;
        jqmIcon: 'back';
        callback: [
          workingDirectory = f
```

```
[
ngDirectory.
ception here"
```

```
testRendering

1 run
1 passes
0 expected failures
0 failures
0 errors
0 unexpected passes
```

```
Workspace >>>
```

Figure 6.12: Alexandria in action

CHAPTER 7

Mercury At Work

Contents

At a Glance

In this Chapter we show how Mercury can be used by developers to build remote debugging tools and front ends. We then continue by validating Mercury's properties in an experimental setting. Two case studies are considered involving remote debugging of multiple constraint devices. The first case study details how the property of interactiveness can be used to support a *remote agile debugging* paradigm. While the second shows how Mercury brings the idea of object-centric debugging in a distributed setting through *remote object instrumentation*.

7.1 Mercury Examples

In this section, we show how Mercury can be used in practice as a remote debugging framework to build front-ends and IDEs. The first three examples cover debugging basics (execution flow control, inspection and modification of objects, handling of remote exceptions) while the latter four exemplify the compliance of our framework with the debugging properties discussed in previous Sections.

Pharo Smalltalk in a nutshell. For readers not used to Pharo, message send uses whitespace instead of dot e.g. *receiver message*. Arguments are delimited by (:) in the message selector e.g. *object instVarAt: varName put: value*. The assignment operator is colon-equal (:=), strings are delimited using single quotes, and symbols start with the sharp sign (#). Collections use curly braces ({}) and dots (.) as separators e.g. *{1. 2. 3}*. Block closures are delimited by square brackets ([]) and use pipe (|) to separate arguments definition from the block's body; each argument is prefixed by (:) e.g. *[:arg1 :arg2 | arg1+arg2]*. Most of control structures are regular message sends e.g. aBoolean ifTrue: [...].

7.1.1 Inspecting remote environments and accessing objects

On line 1 of Script 1, the current process on the developer machine uses the mirror factory Reflection to access an environment on the remote target at address minesdouai.fr:8080. The mirror factory will return the appropriate meta-object for the remote environment according to the process' security policy. This will be an instance of one of the: EnvironmentIntrospectionMirror or EnvironmentIntercessionMirror classes depicted on Figure 5.6. All subsequent meta-objects returned will be of the same access-right level. On line 2 a package meta-object is accessed named: #Graphics-Primitives, then on line 3 a class meta-object inside this package named: #Point is retrieved. On line 4 a new instance of the class #Point is created on the target and its corresponding meta-object is returned on the developer's machine. Finally on line 5 the instance variable named x of this newly created object is set to a new value. If the process access-rights did not include intercession, that is if the reference on line 1 was an instance of an EnvironmentIntrospectionMirror meta-object, then the expression on line 5 would raise an exception.

Script 1: **Inspecting a remote environment and editing remote objects**

```
1 anEnvironmentMirror:= Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 aPackageMirror := anEnvironmentMirror packageNamed: #'Graphics-Primitives'.
3 aClassMirror := aPackageMirror classNamed: #Point.
4 anObjectMirror := aClassMirror newInstance.
5 anObjectMirror instVarAt: #x put: 100.
```

7.1.2 Handling remote exceptions

In Script 2, as before the remote environment is accessed through our mirror factory (line 1). Then on line 2 an expression is evaluated on the remote target: '3 / 0'. Then we show that if a remote exception occurs during the evaluation of an expression, a corresponding exception meta-object is returned and can be used by clients of our meta-level. In this case a LocalDebuggingClient class which makes use of our meta-level, is invoked with our remote exception meta-object.

Script 2: **Handling a remote exception**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 anEnvironmentMirror
3     evaluate: '3 / 0'
4     onRemoteExceptionDo: [:aRemoteExceptionMirror |
5          LocalDebuggingClient debug: aRemoteExceptionMirror]
```

7.1.3 Changing variables and controlling execution flow

On line 2 of Script 3, we access all interrupted processes (threads) on the remote target. These are all execution threads that have raised unhandled exceptions or have been interrupted for inspection by the developer. Then on line 3 we retrieve the top context on the stack of the first interrupted process. On line 4 we modify a temporary value inside that context. Then finally on line 5 we make the process to proceed with the interrupted execution.

Script 3: **Controlling execution flow**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 interruptedProcessesMirrors := anEnvironmentMirror interruptedProcesses.
3 aContextMirror := interruptedProcessMirrors first topContext
4 aTempObjMirror := aContextMirror tempNamed: 'x' put: '3'.
5 interruptedProcessesMirrors first proceed.
```

7.1.4 Incrementally changing the target's code and state

On Script 4, we show how a developer can introduce new behavior in the targeted application (interactiveness). On line 2 we introduce an empty new package through our environment meta-object. From line 3 to 5, we add a new class in this package with two instance variables and one method named hypothesis. We instantiate this new class (line 6) and send a message to this new instance (line 7) through its meta-object. As before (on Script 3) if our message-send raises an exception, a corresponding exception meta-object will be returned which can be used by clients of our meta-level.

Script 4: **Incrementally updating the remote target to test a bug**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 aPackageMirror := aRemoteEnvironment newPackageNamed: #NewPackage.
3 aClassMirror := aPackageMirror newClassNamed: #NewClass.
4 aClassMirror := aClassMirror ivs: { #x . #y}.
5 aClassMirror := aClassMirror addMethod: 'hypothesis: aNumber ...'
6 anObjectMirror := aClassMirror newInstance.
7 anObjectMirror perform: #hypothesis
8      withArguments: { 3 }
9      onRemoteExceptionDo: [:aRemoteExceptionMirror |
10          LocalDebuggingClient debug: aRemoteExceptionMirror]
```

7.1.5 Introducing breakpoints on execution events

In Script 5, we show how instrumentation can be used to alter semantics on the target application and provide facilities such as object-centric watchpoints. Especially in this example we show how halting on object creation can be achieved by conditioning the message receive event on a class.

On line 1 a class mirror is retrieved. On line 2 a remote callback is registered for instrumenting message sending on the remote class. This callback accepts one argument that provides meta-information about the event, such as the name of the method being invoked. On line 3 a condition and an action are set within the callback. Specifically when the message *new* (responsible for object creation) is sent to the remote class, the class that triggered the event (i.e *reifications trigger*) will cause the remote process to halt, effectively producing a watch-point on object creation.

Script 5: **Instrumentation of object creation**

```
1 aClassMirror := aPackageMirror classNamed: #Point.
2 aClassMirror onReceive: [:reifications |
3       reifications message selector = #new ifTrue: [reifications trigger halt] ]
```

7.1.6 Distribution

In Script 6, we show how we can adapt the middleware's serialization policy at runtime while debugging. On line 1 we retrieve a class mirror (on the class #Class) and then on line 2, we ask the class mirror for an object mirror on its instance variable: #localSelectors. The localSelectors instance variable is a Set holding all selectors (method names) defined locally in that class. Since this is a collection of basic instances (i.e symbols), further processing on it (like printing) would be more convenient if instead of a mirror (i.e the ivMirror in this case) we had a local copy. This is achieved on line 3 where we send the message *resolveLocally* to the ivMirror. This message (at run-time) instructs the middleware to override its serialization policy specifically for this instance and return a local copy of the underlying remote reference.

Script 6: **Adapting serialization at run-time**

```
1 aClassMirror := aRemoteEnvironment globalAt: #Class.
2 ivMirror := aClassMirror objInstVarNamed: #localSelectors.
3 localSet := ivMirror resolveLocally.
```

7.1.7 Security

In Script 7, we show how new sub-processes can be spawned on the developer's side with restricted access to the debugging framework. On line 1 to 3 of Script 7 we create a new process. The process is instructed to use the introspection reflection policy (message *forkWithIntrospectionReflectionPolicy*). This call will initialize the SecurityPolicy instance (left side of Figure 5.6) of the newly created sub-process. When the mirror factory receives a request to create a mirror for this sub-process (line 1) it will return a mirror (allowing only introspection) corresponding to the security policy that was set upon forking (line 3).

Script 7: **Restricting access to a sub-process**

```
[ aMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
  self assert: (aMirror class = EnvironmentIntrospectionMirror).
] forkWithIntrospectionReflectionPolicy.
```

7.2 Mercury Validation: Experimental Setting

For validating Mercury we prepared an experimental setting for simulating real-word debugging scenarios. We considered three different kinds of constraint devices as debugging targets. These devices (which can be seen in Figure 7.1) where chosen as illustrative examples of either:

- Targets that have different hardware or environment settings than development machines.
- Targets that are not locally or easily accessible.
- Targets that have resource constraints or no input/output interfaces for local development.

In these settings as we discussed in Chapter 1 remote debugging proves to be the most sensible solution - compared to other approaches - such as post-mortem analysis or emulators.

Our goal is to:

1. Verify the applicability of Mercury for these debugging targets.
2. Illustrate how a debugging session benefits from Mercury's properties which we discussed in Chapter 5. For this goal we study the following use-cases:
   (a) Combining agile development [Abacus 2005] with debugging **in a single remote debugging session** without the need of re-deployment.
   (b) Supporting both OO-centric [Ressia 2012b] and Stack-based debugging **in a remote setting** through remote object instrumentation.

7.2.1 Debugging Targets Set-Up

Figure 7.1 depicts the set-up of our experiment. In the lower part of the figure we show the development machine running our debugging front-end. The developer machine connects to our targets through two communication interfaces designated as *ETH and WIFI* for ethernet and wireless communication channels respectively.

Each tab depicted in the cropped screenshot at the center of the figure corresponds to the environmentMirror of a debugging target. New targets can be accessed programmatically (as we saw on our examples scripts) or interactivelly through the *add target Tab* (leftmost tab on our screenshot) by supplying the address and the port of a deployment as well as a target alias for the remote environment (as seen in Figure 7.2).

In the upper part of the figure we depict our debugging targets. Device A (Galaxy Nexus) is a smart-phone target connected to our development machine through wifi. Device B (Galaxy Tab) is a tablet target also connected through a wireless network, while Device C (HP Workstation) is a remote server to which we connect through the ethernet interface.

For our two android devices (phone and tablet) we also tested communication through a usb channel that establishes ethernet connections using port forwarding. [1]

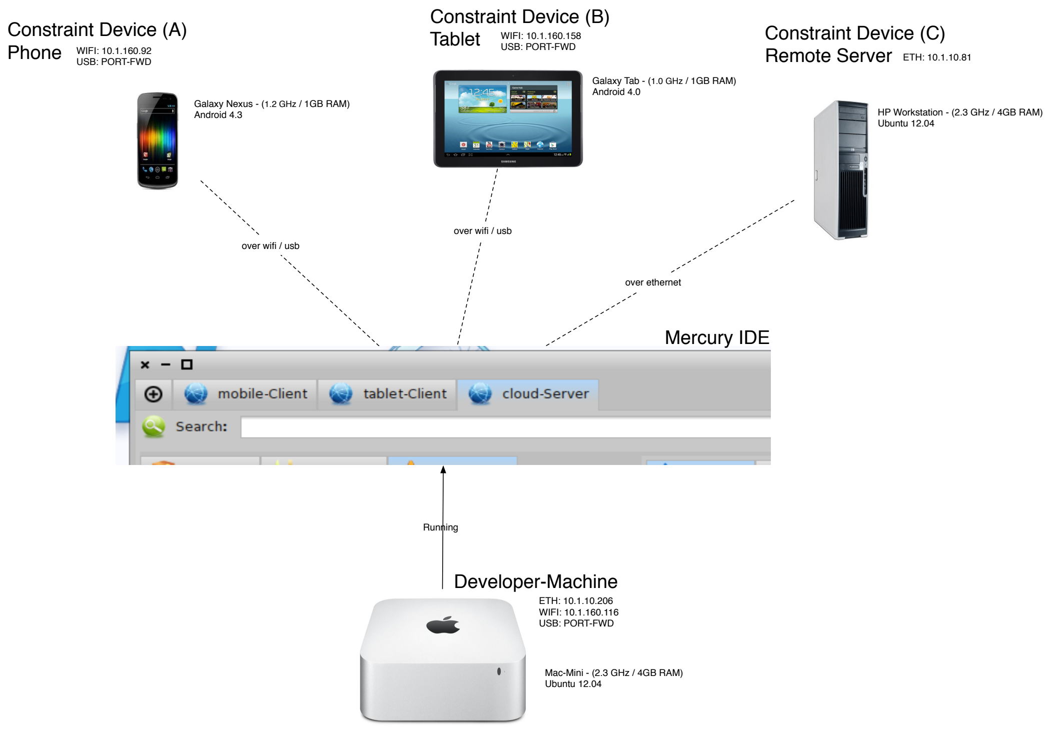


Figure 7.1: Experimental Set-up for our Debugging Targets

7.2.2 Remote Applications

7.2.2.1 The Droid and Cloud File Browsers

Figure 7.3 shows the Droid-Browser applicaton running on the tablet and phone devices (left and right part of Figure 7.3 respectively). The Droid-Browser is a local file browsing application that presents the option to upload local files stored on the device to the cloud (i.e the remote server of our experimental setting). The Droid-Browser is a local (to the device) web-application. This means that both the back-end (access to the local filesystem and serving of web-pages) as well as the front-end (rendering of web-pages) are running locally on the device.

On the other hand the Cloud-Browser is a normal web-application that presents the option to download files that where previously uploaded on the server. Its back-end (access

[1] http://developer.android.com/tools/help/adb.html

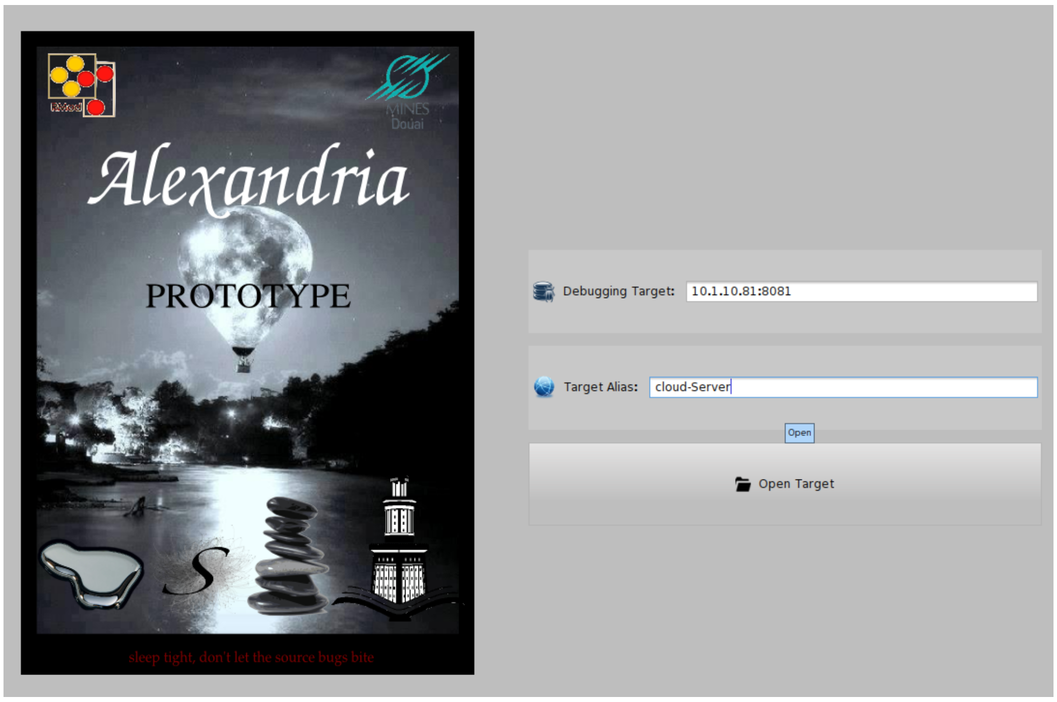

Alexandria
PROTOTYPE
MINES Douai
sleep tight, don't let the source bugs bite
Debugging Target: 10.1.10.81:8081
Target Alias: cloud-Server
Open
Open Target

Figure 7.2: Opening a remote target for debugging through the Mercury IDE

to the server's filesystem and serving of web-pages) is running on our remote server while the front-end is accesible through the web-browser of another machine (the developer's machine in our case). The Cloud-Browser front-end is shown in Figure 7.4.

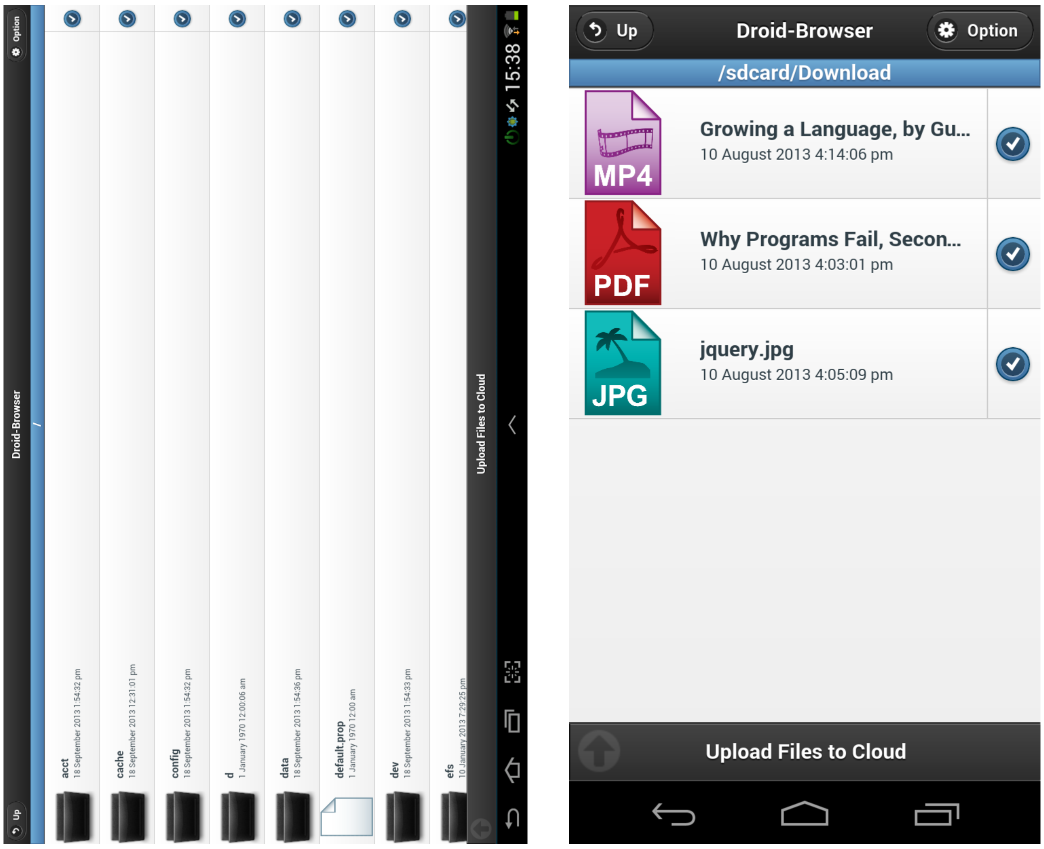
The two applications share part of their code for file browsing and serving of web-pages as seen in Figure 7.5. The core logic of both applications resides in two subclasses of a common ancestor class named FileBrowser. FileBrowser is itself a subclass of WAComponent which is part of the Seaside [Ducasse 2004] web-framework.

7.2.2.2 Software Stack on each Device

On the phone and tablet targets the deployed software stack is as follows:

- **Android OS** The operating system running on the two devices (versions 4.3 and 4.0 on phone and tablet respectively).
- **MetaStackVM - Android Port** A port of our dedicated vm supporting debugging instrumentation for the android platform.
- **Pharo** The 1.4 version (*summer edition*) of the Pharo environment.
- **SUnit**[2] The smalltalk testing framework (comes pre-loaded with the Pharo environment)

[2]http://sunit.sourceforge.net


Up Droid-Browser Option

/

acct
18 September 2013 1:54:32 pm

cache
18 September 2013 12:31:01 pm

config
18 September 2013 1:54:32 pm

d
1 January 1970 12:00:06 am

data
18 September 2013 1:54:36 pm

default.prop
1 January 1970 12:00 am

dev
18 September 2013 1:54:33 pm

efs
10 January 2013 7:29:25 pm

Upload Files to Cloud

15:38

Up Droid-Browser Option

/sdcard/Download

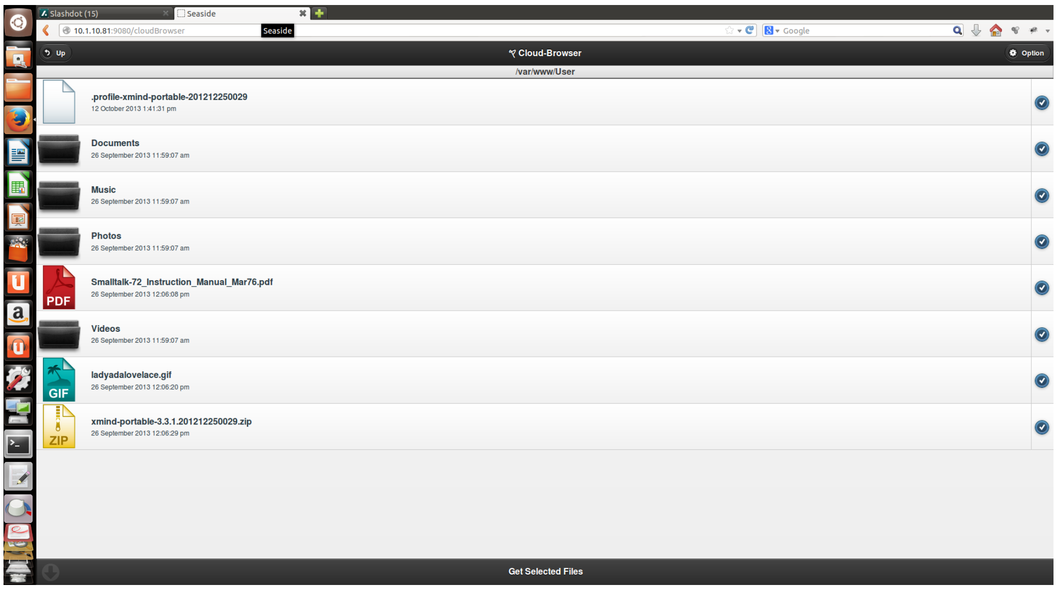
Growing a Language, by Gu...
10 August 2013 4:14:06 pm

Why Programs Fail, Secon...
10 August 2013 4:03:01 pm

jquery.jpg
10 August 2013 4:05:09 pm

Upload Files to Cloud

Figure 7.3: Droid-Browser: Our file browser for mobile devices



10.1.10.81:9080/cloudBrowser

Up Cloud-Browser Option

/var/www/User

.profile-xmind-portable-201212250029
12 October 2013 1:41:31 pm

Documents
26 September 2013 11:59:07 am

Music
26 September 2013 11:59:07 am

Photos
26 September 2013 11:59:07 am

Smalltalk-72_Instruction_Manual_Mar76.pdf
26 September 2013 12:06:08 pm

Videos
26 September 2013 11:59:07 am

ladyadalovelace.gif
26 September 2013 12:06:20 pm

xmind-portable-3.3.1.201212250029.zip
26 September 2013 12:06:29 pm

Get Selected Files

Figure 7.4: Cloud-Browser: Our file browser for the cloud

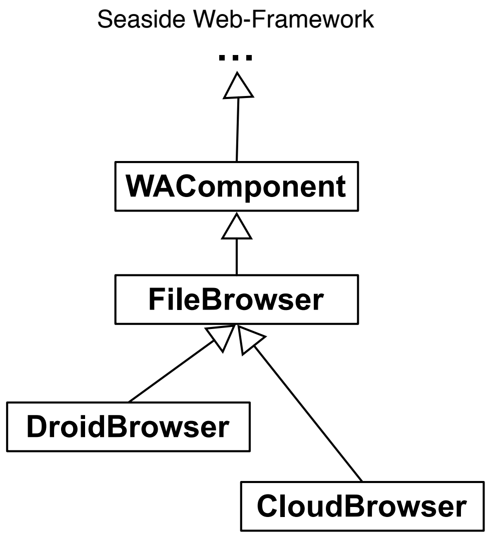


Figure 7.5: The Droid and Cloud browser apps implemented as Seaside components

- **Seaside** The Seaside [3] web-framework for Smalltalk (version 3.0).
- **Seaside-Jquery-Mobile** An integration library between Seaside and Jquery-Mobile.[4]
- **Droid-Browser** Our file browser for the android platform as a Seaside component.
- **PhoneGap** A stand-alone rendering solution for web-apps. [5]
- **Mercury-Core-Target** The Mercury-Core package for debugging targets as described in Figure 6.9.

On our server target the deployment was as follows:

- **Gnu/Linux** The Ubuntu distribution of the Gnu/Linux operating system (version 12.04).
- **MetaStackVM** Our dedicated vm supporting debugging instrumentation for Unix.
- **Pharo, SUnit, Seaside, Seaside-Jquery-Mobile** Same deployment as above.
- **Cloud-Browser** Our file browser for the the cloud.
- **Mercury-Core-Target** Same deployment as above

[3]http://www.seaside.st
[4]http://jquerymobile.seasidehosting.st
[5]http://phonegap.com/

On the developer machine the software deployment was the following:

- **Gnu/Linux** The Ubuntu distribution of the Gnu/Linux operating system (version 12.04).
- **Pharo** Same deployment as above running on the official vm for Pharo.
- **Mercury-Core-Developer** The Mercury-Core package for the developer's side as described in Figure 6.9.
- **Alexandria** Our experimental front-end.
- **Web-Browser** The firefox web-browser where the Cloud-Browser app is rendered.

7.2.3 Debugging Front-End Walkthrough

In Figure 7.6 we show an annotated screenshot of Mercury's front-end in debugging mode. The annotations depict the relationship between the different front-end widgets and the underlying mirror model of Mercury (see also Figure 5.3). Items are numbered 1 through 9 in clock-wise fashion.

On the upper-left corner (1) we see the exceptions panel presenting a list of unhandled exceptions on the target which is updated on the fly as new exception are raised. The package tab (2) (also shown selected in Figure 6.12) presenting the list of loaded packages on the target as well as a process tab (3) depicting the list of processes (green threads in the case of Pharo) that are running on the target.

The upper-center part of the figure (4 and 5) consists of panels for classes, methods and protocols (named sets of methods). Especially in the case of methods two panels are provided, one for viewing the methods of the selected class (or the class associated with a suspended context) and one for the methods of its corresponding Test class. Test classes and test methods can be also added and evaluated (6) dynamically as a consequence of our model supporting interactiveness. This interaction between testing and debugging in a remote setting will be the focus of our first case-study (Section 7.3).

In the lower-right corner we see two object inspectors (7) one for the currently selected context and one for each associated receiver object. Entries on these inspectors can be browsed and edited but also remotely intercessed as a consequence of our model supporting instrumentation. This will be the focus of our second case-study in Section 7.4.4.

In the lower-center part of the figure a source-code editor and a read-eval-print console can be seen. The console is associated with the environment mirror, while the source-code editor with the currently selected method mirror and context mirror. Above the source-code editor resides an execution control widget (for stepping, restarting, resuming, suspending execution e.t.c). Finally in the lower-left corner we can see the list of the suspended remote contexts (9) which can be navigated.

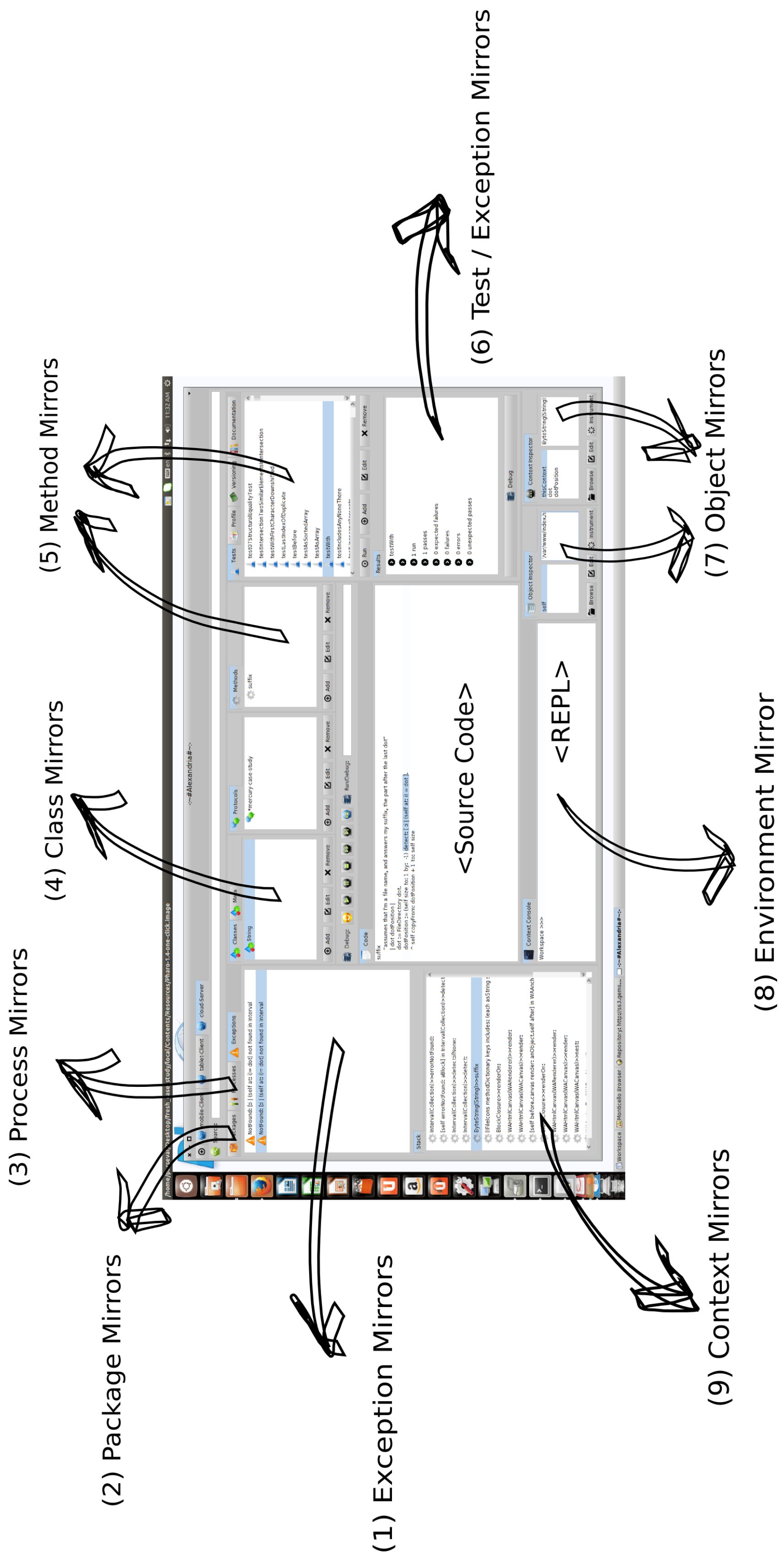


Figure 7.6: Mercury's mirrors depiction in the front-end

7.3 Case Study I: Remote Agile Debugging

7.3.1 Introduction

We discussed in Chapter 2 how the property of interactiveness can increase productivity in the case of remote targets by eliminating the need for re-deployment while fixing flaws (i.e., architectural bugs) or by enhancing the reproducibility of failures.

In this Section we consider another related use-case for interactiveness by combining Agile Development [Abacus 2005] with debugging **in a single remote debugging session** without the need of lengthy re-deployments. While doing so we illustrate how dynamically introducing tests while debugging enhances reproducibility by giving us the ability to investigate different flows of execution in parallel.

7.3.2 The Suffix Defect in a Remote Setting

For our use case we chose to investigate a variant of a well studied defect in literature for local debugging, involving a filename suffix mismatch [Black 2009]. Our study reproduces this defect in a remote setting using the experimental set-up discussed in the previous Sections, so as to validate Mercury's properties and discuss remote agile debugging and remote object instrumentation.

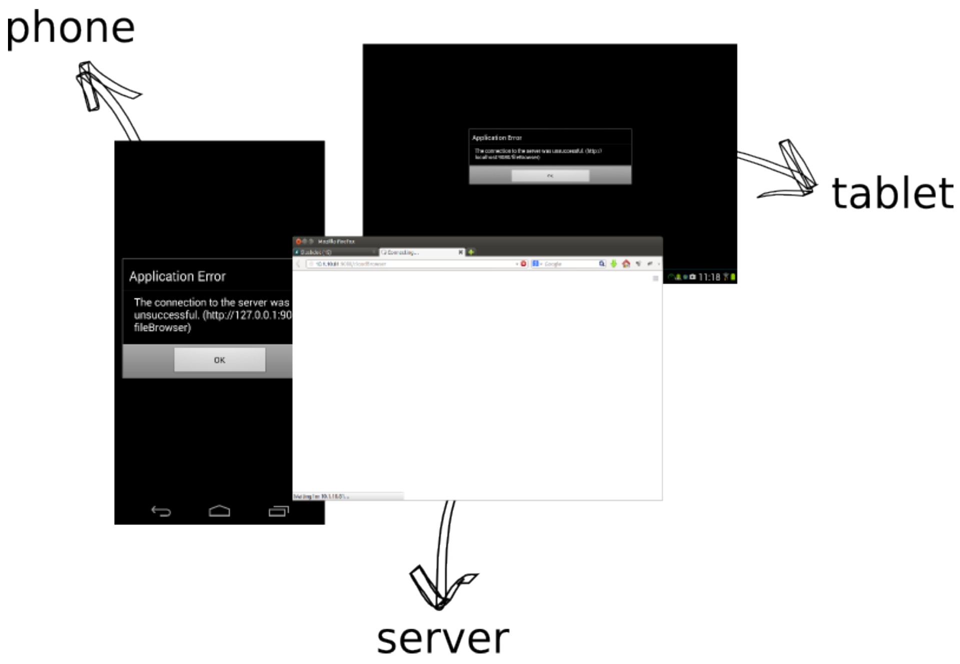
Our starting point is the deployment of the software described in Section 7.2.2.2 in all three devices, and the subsequent launch of the applications. Alas as seen in Figure 7.7 after launching the applications the defect in question manifests itself in all three targets. It is worth noting here that since the applications did not even properly start in our constraint devices, there can be no other meaningful feedback to the developer if he or she is not using a remote debugging solution.

We then connect through the debugging front-end of Mercury to our remote targets using the configuration depicted in Figure 7.1. In Figure7.8 we can see the remote unhandled error and the remote stack related with this initial failure on our mobile target. Our two other targets raised the same error. From the exception name and the remote stack we can deduce that a NotFound error was triggered from inside the #detect: method of class Collection, as seen in Script 8.

Script 8: **The method which raised the initial error**

```
detect: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Answer the first element for which aBlock evaluates to true."

    ^ self detect: aBlock ifNone: [self errorNotFound: aBlock]
```

After navigating the stack through our context mirrors (lower part of Figure 7.8), we come to the first context related with our application, which can be seen in Script 9. The method in Script 9 is an extension of our application for the system class String, which calculates the suffix of a given filename. Mercury informs us that the offending method call to #detect: originated from our code on line 3, by highlighting the corresponding source range.



Application Error

The connection to the server was unsuccessful. (http://127.0.0.1:90 fileBrowser)

Figure 7.7: Initial failure observed in Droid and Cloud-Browser apps after launch

Script 9: **Calculating a filename suffix**

```
String>>suffix
    | dot dotPosition |
    dot := FileDirectory dot.
    dotPosition := (self size to: 1 by:  -1) detect: [ :i | (self at: i) = dot ].
    ^ self copyFrom: dotPosition + 1 to: self size
```

In turn the method suffix was called while the Droid Browser was trying to render a corresponding icon for a file system entry according to its suffix, as seen in Script 10. The method #renderPathOn: in Script 10 belongs to the class FileBrowser (superclass of both Droid and CloudBrowser as seen in Figure 7.5) which seems to be the reason why all of our targets failed to render their ui. To validate this hypothesis we check the stack on all 3 targets and through our object and context browser (entry number 7 on Figure 7.6) we browse the offending file system entries for each case:

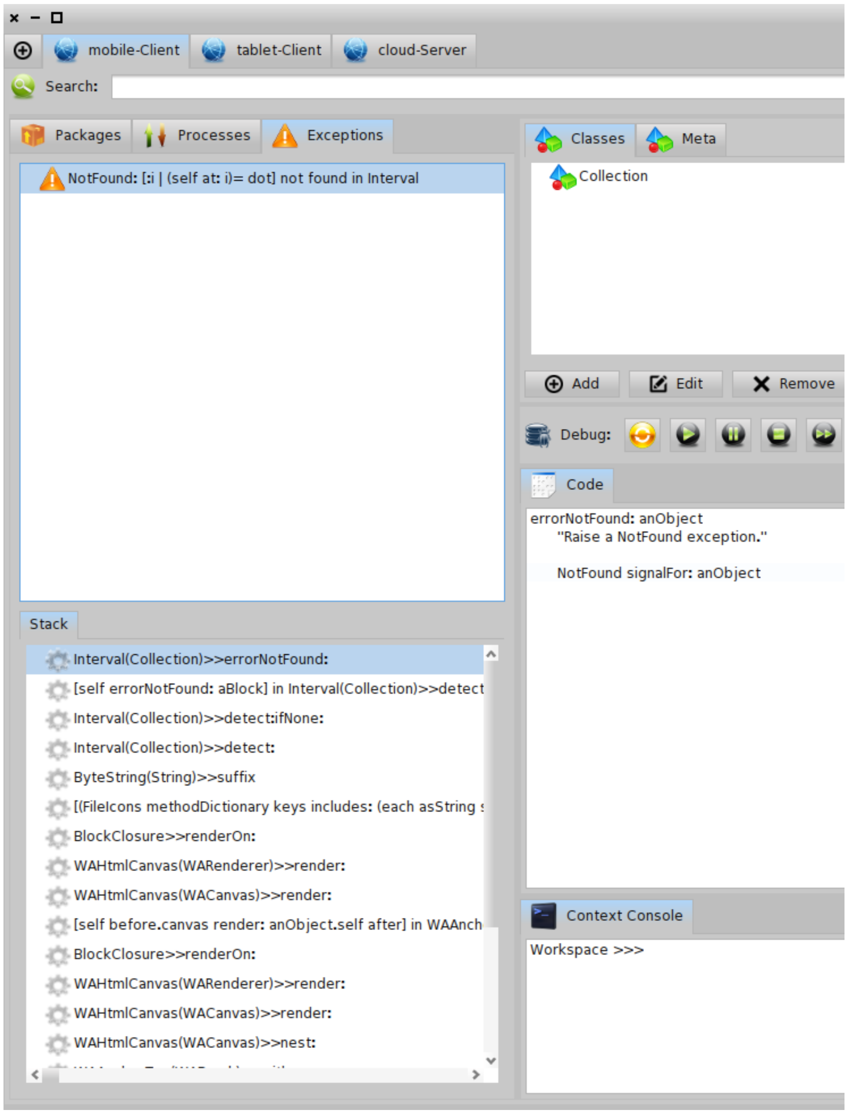
Phone: ’/charger’

Tablet: ’/default.prop’

Server: ’/var/www/User/.profile-xmind-portable-201212250029’

Script 10: **Icon rendering code calling the suffix method**

```
FileBrowser>>renderPathOn: html
[...]
    html image url: (FileIcons urlOf: (each asString suffix , ’Png’) asSymbol).
[...]
```



NotFound: [:i | (self at: i)= dot] not found in Interval

Collection

```
Interval(Collection)>>errorNotFound:
[self errorNotFound: aBlock] in Interval(Collection)>>detect
Interval(Collection)>>detect:ifNone:
Interval(Collection)>>detect:
ByteString(String)>>suffix
[(FileIcons methodDictionary keys includes: (each asString s
BlockClosure>>renderOn:
WAHtmlCanvas(WARenderer)>>render:
WAHtmlCanvas(WACanvas)>>render:
[self before.canvas render: anObject.self after] in WAAnch
BlockClosure>>renderOn:
WAHtmlCanvas(WARenderer)>>render:
WAHtmlCanvas(WACanvas)>>render:
WAHtmlCanvas(WACanvas)>>nest:
```

```
errorNotFound: anObject
    "Raise a NotFound exception."

    NotFound signalFor: anObject
```

```
Workspace >>>
```

Figure 7.8: Initial unhandled error observed in our Droid app running on the mobile target

The entries unfortunately tell us three different things: the suffix method fails both when it is invoked on a filename with no extension (as in the case of our smart-phone target) and on file paths that do have an extension (as in the case of our tablet target). Moreover in the case of the server target the failing filename is a longer file-path whose dot signifies something other than an extension (the fact that this is a hidden file on unix systems), which may be a contributing factor. Since the three devices failed on seemingly different input, we have to make sure that there is no device-specific error involved.

The first failing case (on the phone) by itself seems reasonable if we look at the code of the #detect: method on Script 8. For this case we can already form a hypothesis that all filenames with no extension will raise an error instead of returning an empty string. There is still though no apparent reason for the second and third failures.

7.3.3 Remote Agile Debugging through Interactiveness

Up until now we have seen a normal remote debugging session, where we were able to browse remote targets, navigate their stack and control execution. We will now see how we can use Mercury to dynamically introduce new code and tests while debugging without lengthy re-deployments of our applications.

By doing so we aim to achieve the following:

1. Re-produce the initial error multiple times in order to test different hypothesis without the need of re-deployment.
2. Simplify the offending context without re-starting the debugging session.
3. Maintain the state and suspended execution flow of the initial unhandled errors:
    (a) In order to cross-examine the initial failing state with new findings.
    (b) In case the initial errors are not easily reproducible (as is the case with heisenbugs [Gray 1986])

Our next step is shown in Figure 7.9. Since Mercury can dynamically evolve the target's code (interactiveness) we can remotely introduce new classes and methods for testing to all of our targets while debugging.

In this case we introduce the test class FileBrowserTest (right part of Figure 7.9) as a subclass of the class TestCase which is part of the SUnit framework on the target. Getting a class mirror on FileBrowserTest will allows us to incrementally run tests on our remote machines and debug their results, without ever quitting our current debugging session.

Our front-end has a dedicated panel for this process as shown in Figure 7.10 and 7.6 (sub-panel six on the right). The code of our FileBrowserTest class is given on Script 11. Our first method #suffixOf: is a helper method that replicates the behavior of the String»#suffix method of Script 9. Our second method #testSuffixWithDot invokes our helper method on a simple dotted filename and makes an assertion about the return value of this invocation (this case is similar to our initial error on our tablet). Method #test-SuffixWithoutDot makes an assertion for the case of a not-dotted filename (similar to our

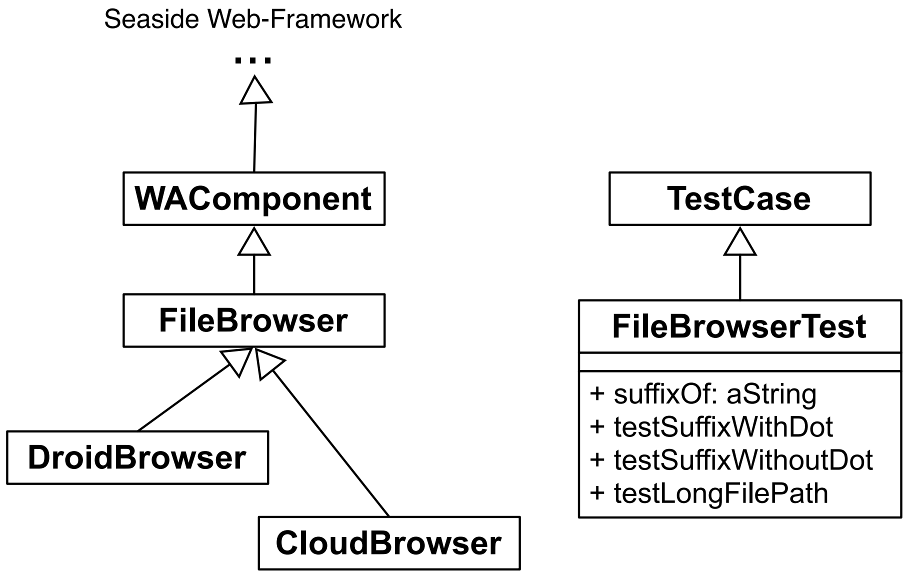


Figure 7.9: Interactivelly introducing the FileBrowserTest class and methods

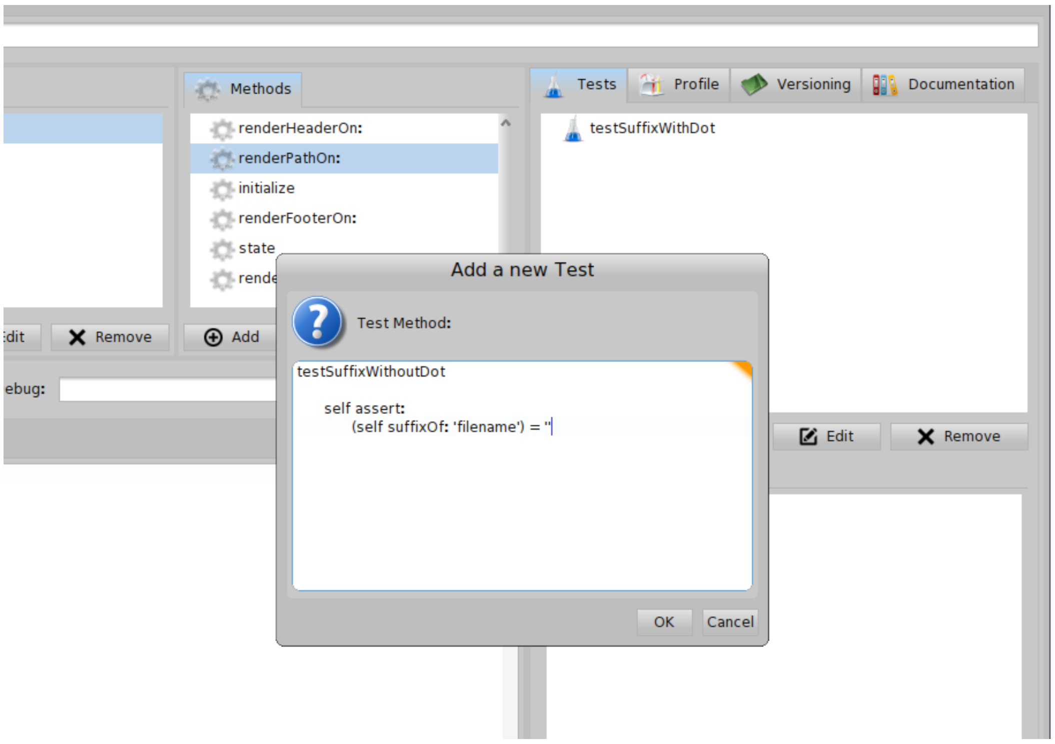
initial error for the smart-phone). Finally testHiddenFilePath tests a long hidden filename with its full path (similar to our initial error on the cloud server). Note here that all three tests will raise a new exception both when our helper method has a defect as well as in the case of a failed assertion.

Script 11: **Test methods**

```
FileBrowserTest>>suffixOf: aString
      "assumes that I'm a file name, and answers my suffix, the part after the last dot"
      | dot dotPosition |
      dot := FileDirectory dot.
      dotPosition := (aString size to: 1 by:  -1) detect: [ :i | (aString at: i) = dot ].
      ^ aString copyFrom: dotPosition + 1 to: aString size

FileBrowserTest>>testSuffixWithDot
      self assert:
          (self suffixOf: 'filename.ext') = 'ext'

FileBrowserTest>>testSuffixWithoutDot
      self assert:
          (self suffixOf: 'filename') = ''

FileBrowserTest>>testHiddenFilePath
      self assert:
         (self suffixOf: '/var/www/User/.a-looooooooong-hidden-filename') =
         'a-looooooooong-hidden-filename'
```

We add our test class and methods to all three targets, and run the tests. This way we will be able to determine if there is some device-specific cause for the error on one of the devices (e.g different representation of file-systems). This process is shown in Figure 7.11.

On Step 1 we run each test individually, on Step 2 we examine the results on the bottom left panel. If the results inform us of an error we can hit the debug button to examine and



```
testSuffixWithoutDot

    self assert:
        (self suffixOf: 'filename') = ''
```

Figure 7.10: Interactivelly adding a new test method through the UI

manipulate it (Step 3). Finally on Step 4 we can see that we are able to switch and cross-examine state and execution between the initial error and the re-produced errors from the test cases. Note here that it is possible to run and debug a single test multiple times, although in this case we run and debug each test only once.

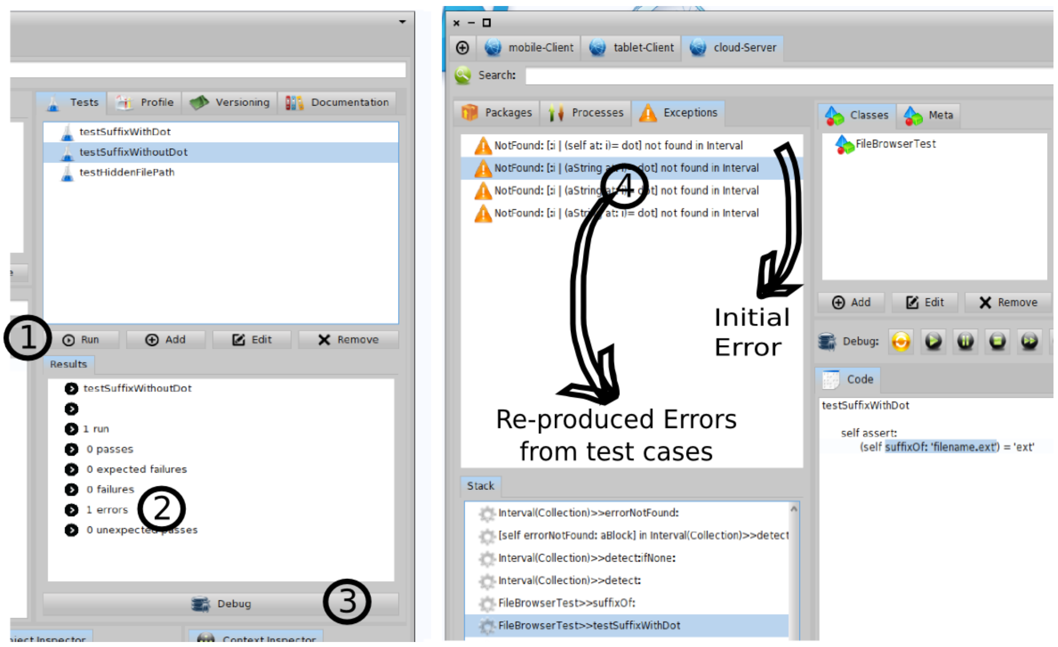
We repeat this process for all three devices and find that all tests fail on all three of our targets. By doing so we deduce that there is no device-specific cause underlying each case, although each one of the tests may be failing for different reasons.

7.3.3.1 Debugging Hypotheses and Fixes

At this point having 12 different threads of execution at our disposal spanning 3 different devices, we can start debugging our hypotheses.

In 7.3.2 we hypothesized that all strings without an extension will raise an error regardless of the underlying defect, by looking at the code of the #detect: method. We got more proof for the validity of this hypothesis from our second failing test (i.e #testSuffixWithoutDot) across all three devices.

Our expectation for the String»suffix method is to return an empty string in this case. So we can now device a possible fix. We need to introduce an error handler for this case which will return the empty string when the error is raised. We want of course to test this possible fix before applying it to the String»suffix method, especially because we are not expecting that the fix will solve the two other cases of the defect.



```
testSuffixWithDot
testSuffixWithoutDot
testHiddenFilePath
```

```
testSuffixWithoutDot

1 run
0 passes
0 expected failures
0 failures
1 errors
0 unexpected passes
```

```
NotFound: [:i | (self at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
```

```
FileBrowserTest
```

```
testSuffixWithDot

    self assert:
        (self suffixOf: 'filename.ext') = 'ext'
```

```
Interval(Collection)>>errorNotFound:
[self errorNotFound: aBlock] in Interval(Collection)>>detect
Interval(Collection)>>detect:ifNone:
Interval(Collection)>>detect:
FileBrowserTest>>suffixOf:
FileBrowserTest>>testSuffixWithDot
```

Figure 7.11: Remotely Running and Debugging multiple Test-Cases while maintaining the initial error

In order to do so we remotely update the FileBrowserTest»suffixOf: method as seen in Script 12. In line 4 of Script 12 we introduce an #on:do: exception handler that returns an empty string when the NotFound error is raised. Subsequently we re-run all tests. The results are shown in Figure 7.12.

Script 12: **Updating the suffixOf: method**

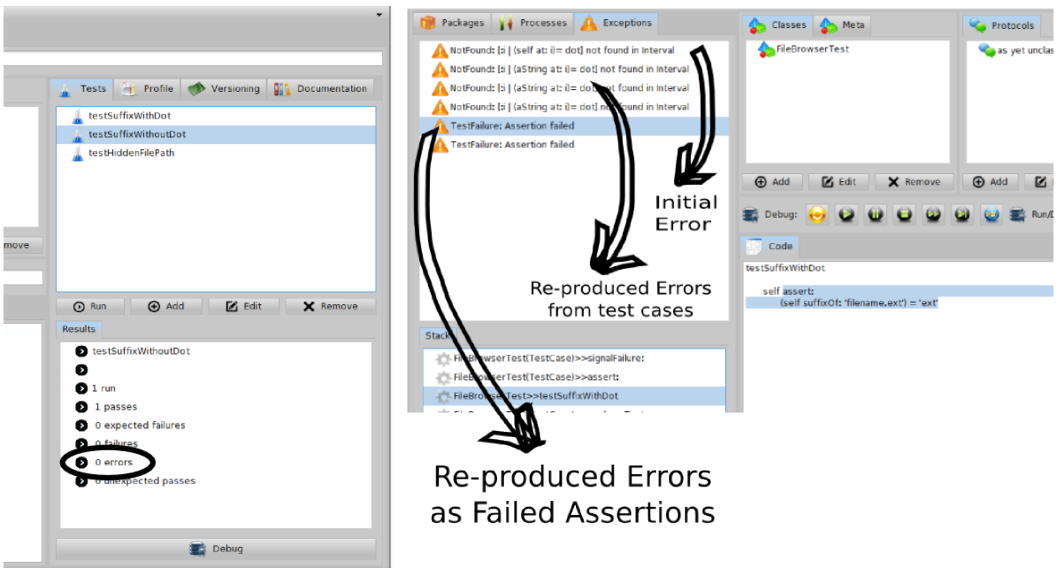
```
FileBrowserTest>>suffixOf: aString
1       "assumes that I'm a file name, and answers my suffix, the part after the last dot"
2       | dot dotPosition |
3       dot := FileDirectory dot.
4       dotPosition := [(aString size to: 1 by: -1) detect: [ :i | (aString at: i) = dot ]] on: NotFound do: [ ^ '' ].
5       ^ aString copyFrom: dotPosition + 1 to: aString size
```

In Figure 7.12 on the left we can see that our #testSuffixWithoutDot test now runs successfully, ensuring the applicability of our fix for this case. For the two other cases though as we expected the tests fail. After the introduction of the error handler in FileBrowserTest»suffixOf: the defect manifests itself as failed assertions on our two remaining tests. Debugging these last two failed assertions will be the focus of our second case study.

7.3.4 Results

Using our results shown in Figures 7.11 and 7.12 we can now verify that by being able to dynamically evolve the target's code (interactiveness) we were able to introduce and debug tests **without lengthy application re-starts or re-deployments**.

More specifically in a single remote debugging session: We were able to re-produce the initial error multiple times in order to test different hypothesis (**Goal 1**). We simplified



```
testSuffixWithDot
testSuffixWithoutDot
testHiddenFilePath
```

```
testSuffixWithoutDot

1 run
1 passes
0 expected failures
0 failures
0 errors
0 unexpected passes
```

```
NotFound: [:i | (self at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
NotFound: [:i | (aString at: i)= dot] not found in Interval
TestFailure: Assertion failed
TestFailure: Assertion failed
```

```
FileBrowserTest(TestCase)>>signalFailure:
FileBrowserTest(TestCase)>>assert:
FileBrowserTest>>testSuffixWithDot
```

```
FileBrowserTest
```

```
testSuffixWithDot

   self assert:
        (self suffixOf: 'filename.ext') = 'ext'
```

Figure 7.12: Successfully debugged first failing test (left). Defect now manifests itself as failed assertions (right).

the offending context (**Goal 2**), since we reproduced our defects independently of the Droid and CloudBrowser applications and the underlying Seaside web-framework. Moreover since we were able to introduce the helper method #suffixOf: we now do not need to experiment directly on the initial context (String»suffix). This way we can maintain the state and suspended execution flow of the initial errors unchanged (**Goal 3**) until we are certain about applying a fix.

As a consequence of Goal 3 we can now cross-examine the initial error with the new findings (**Goal 3a**). And finally in case the defect was not easily reproducible (heisenbug) we can now maintain it throughout the experimentation (**Goal 3b**).

7.4 Case Study II: Remote Object Instrumentation

7.4.1 Introduction

Recent research results on object-centric debugging [Ressia 2012b] suggests that traditional stack-based navigation is inadequate for certain kinds of questions that programmers ask while testing their hypotheses. Empirical studies on software evolution [Sillito 2006] support this argument, identifying cases where the relationship or the interaction between two or more objects at runtime are more relevant to the programmer than the examination of execution at specific lines of code.

In Chapter 5 we saw how the Mercury model supports the instrumentation of semantical events in a remote setting. Our goal now in this second case study is two-fold:

1. Verify that the remote object instrumentation facilities of Mercury can bring the idea of oo-centric debugging in a distributed setting.

2. Provide an example where Mercury uses the two paradigms (i.e stack-based and oo-centric debugging) in a complementary fashion.

7.4.2 The Hidden Path Hypothesis

We continue where we left off in our first case-study (Figure 7.12) with two out of three assertions still failing on all targets. We now turn our attention to the hidden path failure on the server that we discussed on Section 7.3.2. We would like to investigate whether the difference in the dot placement of a hidden file name is a contributing factor to our defect.

We begin navigating the stack of the corresponding failing assertion as seen in Figure 7.13. We would now like to examine the execution of the FileBrowserTest»suffixOf: method more closely. In order to do so we can use normal stack-based execution control (through the control-panel just above the source-editor seen in Figure 7.13). We *restart* the execution of the current context and then we *step-into* the suffixOf: method (Script 11), as seen in Figure 7.14.

In order to get to our point of interest though we now have to follow the iteration seen on line 4 of Script 11 (inside the block closure argument to the #detect: method). Getting inside the loop requires in total 10 control-flow commands from our initial execution point (in Figure 7.13) and for each additional iteration 3 commands more. A breakpoint inside the loop can reduce the number of commands we need to issue from the ui (to 1 command per iteration), but still the placement of the dot is such that we would need 30 iterations to get there (the loop iterates the string starting from the end). So even setting a breakpoint will be time consuming, especially if we need to reproduced and re-examine the failed assertion several times. In addition we do not know the specific inner working of the #detect: method and if we continue using a stack-based approach we would need to step-into the #detect method as well between iterations.

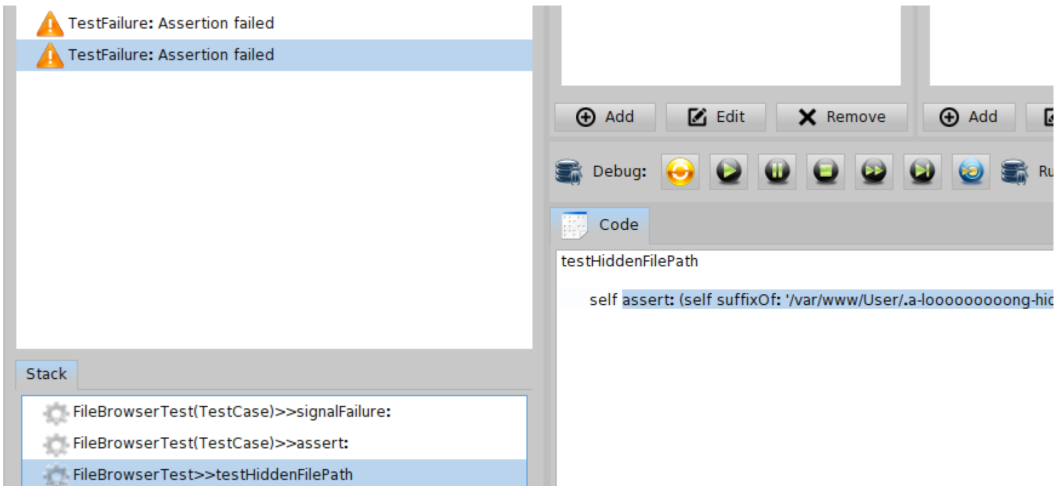

TestFailure: Assertion failed
TestFailure: Assertion failed
Stack
FileBrowserTest(TestCase)>>signalFailure:
FileBrowserTest(TestCase)>>assert:
FileBrowserTest>>testHiddenFilePath
testHiddenFilePath
self assert: (self suffixOf: '/var/www/User/.a-looooooooong-hic

Figure 7.13: Failed Assertion Stack for our Hidden Path Test

7.4.3 Combining Object and Stack Debugging in a Remote Setting

Stack-based debugging got us this far, but it is becoming cumbersome. We need to narrow the domain of our examination. What is needed here is either a conditional break-point or a watch-point breaking exactly at the required iteration. A generalized object-oriented version of such facilities as we saw was proposed by Ressia [Ressia 2012a]. We will now see how Mercury brings this object-centric debugging to a distributed setting through remote instrumentation.

We continue from our execution point seen in Figure 7.14, but instead of trying to navigate through the loop or inside the #detect: method we now invoke the remote instrumentation interface of Mercury (seen in Figure 7.15).

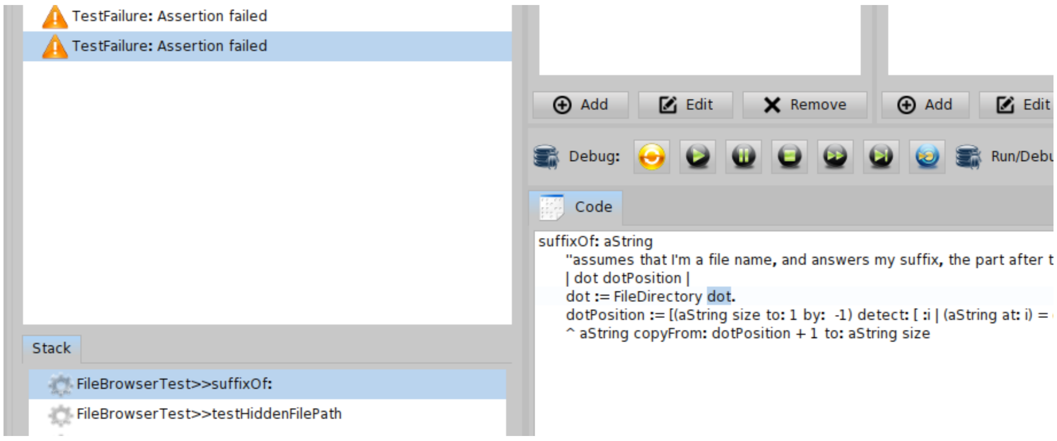
In the left part of Figure 7.15 we can see the failed assertion and its stack, the execution is currently suspended in the FileBrowserTest»suffixOf: method. In the right we can see the remote instrumentation ui. From the panel in Step 1 we can choose the semantical event which we wish to instrument (Object Interaction in our case). The text entry in Step 2 receives an expression whose result will be returned as a mirror in the developers machine. This mirror will serve as a target for the semantical event of Step 1. The text entry in Step 3 receives additional information related to the event which we wish to instrument. In the case of the *Object Interaction* event we need to supply an additional expression for calculating the mirror of the interacting object. Finally on Step 4 we can supply an optional condition for the meta-action we wish to perform and the meta-action itself which will be triggered upon the semantical event defined by Steps 1 through 3.

The code for implementing the instrumentation's meta-action can be seen in Script 13. On line 1 we can see (as was discussed in Chapter 6) that the meta-action receives two arguments. The first argument (named reifications on our Script) represents meta-information relating to the event (such as the object that triggered a particular event), while the second argument reifies the reflectogram, which controls meta-level execution. On line 3 we instruct the execution on the remote target to halt in a context of the object that triggered the event. While on lines 4 through 6 we instruct the reflectogram to perform the default action (for this semantical event) when execution resumes from the breakpoint of line 3.

Script 13: **Object-centric conditional watchpoint meta-action**

```
1 [:reifications :reflectogram |
2
3       reifications trigger halt.
4       reflectogram
5               override: true;
6               returnValue: reflectogram defaultAction.
7
8 ]
```

In a nutshell we have instructed Mercury through this process to halt execution the next time the dot inside the hidden filepath will interact in anyway with the dot object (representing the suffix separator defined by the FileSystem). This way through remote object instrumentation we have implemented a custom conditional watchpoint for our case with object-centric semantics. The results of this process are seen in Figure 7.16.


TestFailure: Assertion failed
TestFailure: Assertion failed
Stack
FileBrowserTest>>suffixOf:
FileBrowserTest>>testHiddenFilePath
Add
Edit
Remove
Add
Edit
Debug:
Run/Debu
Code

```
suffixOf: aString
    "assumes that I'm a file name, and answers my suffix, the part after t
    | dot dotPosition |
    dot := FileDirectory dot.
    dotPosition := [(aString size to: 1 by:  -1) detect: [ :i | (aString at: i) =
    ^ aString copyFrom: dotPosition + 1 to: aString size
```

Figure 7.14: Debugging the suffixOf: method

In the left part of Figure 7.16 we can see that the assertion's execution continued up until the semantical event that we defined above and then halted (Step 1). Specifically (as seen in the code editor of Figure 7.16) execution halted when the two objects we were targeting (the dot inside the filepath and the dot of the FileSystem interacted). The interaction took place while we were comparing the two objects (*(aString at: i) = dot*).

After examining the two objects we were targeting (Step 2) we pinpoint a mismatch. We are comparing two dots with different string representations (i.e $. and '.'). The defect causing our assertion to fail now becomes apparent: we are comparing a Character instance ($.) to a String instance ('.') which Pharo does not automatically cast. This may be the reason why the NotFound exception was triggering all along.

We test our hypothesis by restarting execution on our suspended context and changing the code of the suffixOf: method compared to Script 12 as follows:

Script 14: **Fix applied to the suffixOf: method**

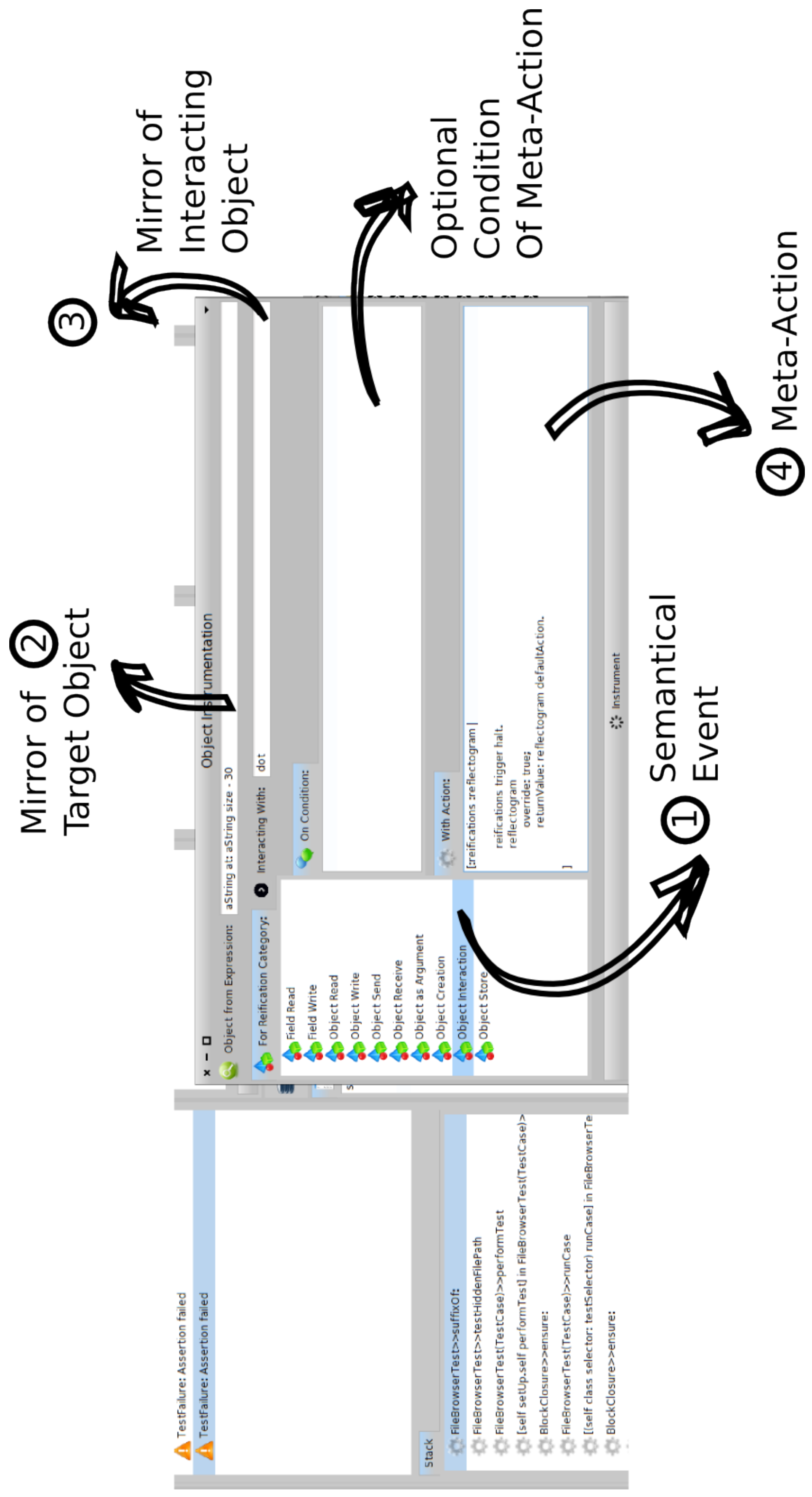
```
[...]
dot := FileDirectory dot first.
[...]
```

The change ensures that the filesystem's dot we are comparing to will also be a Character instance. After resuming execution our hypothesis is validated. Our second and third test cases execute without failed assertions. We can now apply our two fixes on the initial offending context. The final code of the suffix method reads as follows:

Script 15: **Fixes applied to the initial offending context**

```
String>>suffix
    "assumes that I'm a file name, and answers my suffix, the part after the last dot"
    | dot dotPosition |
    dot := FileDirectory dot first.
    dotPosition := [(self size to: 1 by:  -1) detect: [ :i | (self at: i) = dot ]] on: NotFound do: [^ ''].
    ^ self copyFrom: dotPosition + 1 to: self size
```

By resuming execution we now validate that the ui on all three targets is now rendering

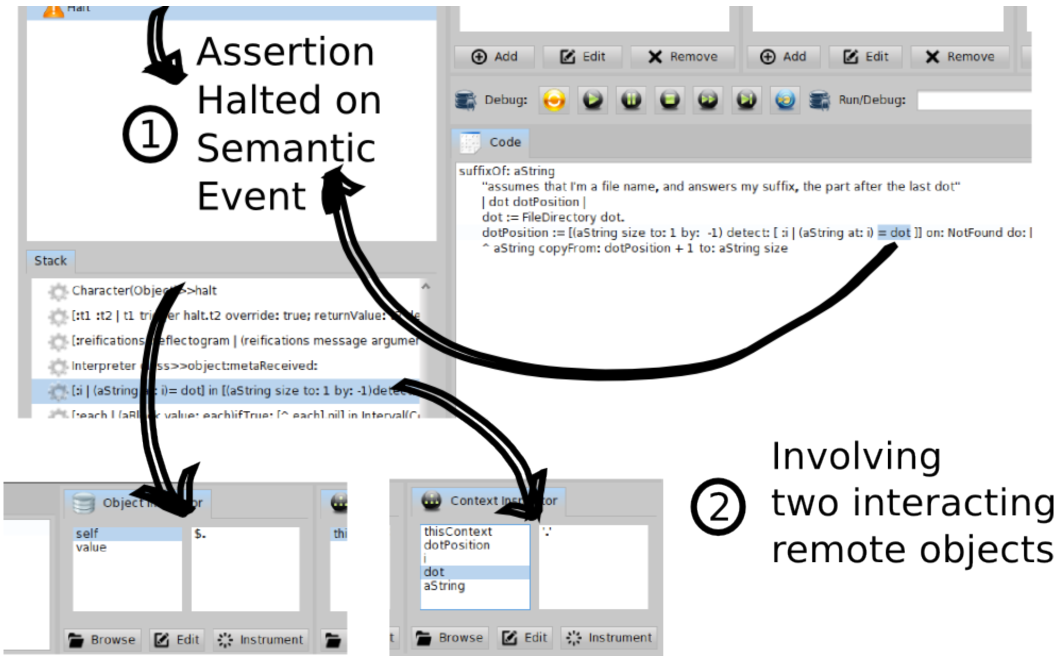


```
[:reifications :reflectogram |
    reifications trigger halt.
    reflectogram
        override: true;
        returnValue: reflectogram defaultAction.
]
```

```
TestFailure: Assertion failed
TestFailure: Assertion failed
```

```
FileBrowserTest>>suffixOf:
FileBrowserTest>>testHiddenFilePath
FileBrowserTest(TestCase)>>performTest
[self setUp.self performTest] in FileBrowserTest(TestCase)>
BlockClosure>>ensure:
FileBrowserTest(TestCase)>>runCase
[(self class selector: testSelector) runCase] in FileBrowserTe
BlockClosure>>ensure:
```

Figure 7.15: Remote Object Instrumentation



```
suffixOf: aString
    "assumes that I'm a file name, and answers my suffix, the part after the last dot"
    | dot dotPosition |
    dot := FileDirectory dot.
    dotPosition := [(aString size to: 1 by:  -1) detect: [ :i | (aString at: i) = dot ]] on: NotFound do: |
    ^ aString copyFrom: dotPosition + 1 to: aString size
```

```
Character(Object)>>halt
[:t1 :t2 | t1 trigger halt.t2 override: true; returnValue:
[:reifications reflectogram | (reifications message argumen
Interpreter class>>object:metaReceived:
[:i | (aString at: i)= dot] in [(aString size to: 1 by: -1)detect
[:each | (aBlock value: each)ifTrue: [^ each].nil] in Interval(C
```

```
self     $.
value
```

```
thisContext     '.'
dotPosition
i
dot
aString
```

Figure 7.16: Halting on Semantical Events

properly as seen in Figure 7.17. We show highlighted the filenames responsible for the initial defect.

7.4.4 Results

Using our results shown in Figures 7.15 and 7.16 we can now verify that through remote object instrumentation Mercury can support the idea of oo-centric debugging in a remote setting (**Goal 1**). More specifically we provided a real-world example where **in a single remote-debugging session** the two approaches (stack-based and oo-centric debugging) where used in complementary fashion (**Goal 2**) to provide a custom mechanism for object-centric conditional watchpoints.

Finally, it is worth noting that this second case study continued our remote agile debugging paradigm introduced in Section 7.3. Our hypotheses and candidate fixes involving remote object instrumentation where first tested and validated on simplified versions of the defect (i.e on our two failed assertions) before being applied to the initial offending context that was responsible for rendering the ui.

7.5 Summary

This Chapter provided hands-on examples for tool developer's wishing to integrate Mercury in their solutions. Seven code examples are detailed covering remote debugging basics (execution control-flow, object inspection, exception handling e.t.c) as well as more advanced techniques complying with the debugging properties that are discussed in Chap-

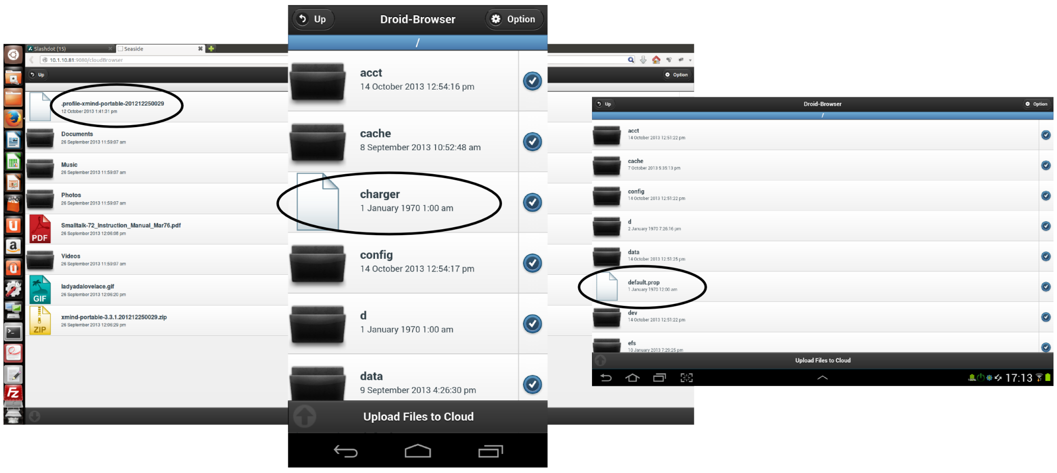


Figure 7.17: Debugged applications on Server, Mobile and Tablet targets

ter 5. Subsequently we detailed an experimental setting for the verification of Mercury's properties. Our experimental setting included three different constraint debugging targets (a smart-phone, a tablet and a remote server) and two case studies discussing *remote agile debugging* and *remote object instrumentation* respectively. Our first case study verified that Mercury - by being able to dynamically evolve the target's code (interactiveness) - is able to remotely introduce and debug tests **without lengthy application re-starts or re-deployments**. As a consequence Mercury can combine agile Development [Abacus 2005] with debugging **in a single remote debugging session**. Our second case study verified that Mercury through **remote object instrumentation** can bring the idea of object-centric debugging [Ressia 2012b] in a distributed setting and use it in conjunction with more traditional techniques (such as stack-based navigation) to enhance debugging experimentation on remote targets.

CHAPTER 8

Conclusion

Contents

At a Glance

This chapter concludes our dissertation by giving a summary of our study and contributions. Finally two future research perspectives for our work are presented, regarding debugging on the level of virtual machines and the convergence of developer driven debugging with automated debugging techniques.

8.1 Summary and Contributions

The context of this dissertation is remote debugging of resource constraint devices. We first provide definitions for the processes of debugging and remote debugging, as well as the notions that they involve. We use these definitions to distinguish remote debugging approaches into those that incorporate post-mortem analysis (such as logging) and those that make use of dedicated remote debugging frameworks that allow live inspection of the running process. We conclude that using remote debuggers is the most sensible solution in situations where targeted devices (such as smartphones or cloud-based servers) have different hardware or environment settings than development machines. Yet remote debugging solutions can prove awkward to use due to their distributed nature. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spent for re-deploying applications while fixing bugs or improving functionality [ZeroTurnAround 2011]. Moreover current solutions lack facilities that would otherwise be available in a local setting because it is difficult to reproduce them remotely (*e.g.,* object-centric debugging [Ressia 2012b]). This fact can impact the amount of experimentation during a remote debugging session - compared to a local setting.

To address these issues we identify four major properties that an ideal solution for remote debugging should exhibit, namely: interactiveness, instrumentation, distribution and security as well as their sub-properties. Interactiveness is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (i.e without stopping the application). Instrumentation is the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Distribution is the ability of a debugging solution to adapt its framework while debugging a remote target. Finally security refers to the availability of prerequisites for security mechanisms in a remote debugging solution, such as authentication and access restriction. Then by using these properties we evaluate and compare state-of-the-art debugging solutions and conclude that although debugging tools relying on reflection are best suited to support experimentation, none of the existing solutions meets all of our criteria in a satisfactory way.

We then continue by studying and providing definitions for reflection and remote reflection in order to assess the use of reflection for remote debugging. A reflective system is a causally connected meta-system that has as object-system itself, while remote reflection is the ability of a reflective system to distribute the whole or part of its self-representation to another meta-system. Consequently we compared different architectural alternatives that facilitate remote reflection, namely: the remote proxy, the remote facade and mirrors and conclude that in terms of remote reflection mirrors can be seen as an extension to both the remote proxy and the remote facade patterns. Finally we pinpoint two open-issues concerning mirrors in the context of debugging, regarding state and intercession.

On one hand mirrors can reduce the footprint of applications in-between debugging sessions by unplugging reflective facilities. On the other hand they do not address the problem of reflective data so as to unplug meta-information. We raise the question of structural decomposition of reflection and meta-information, in the context of mirror-based systems. We show that the property of stratification for mirrors, can be weak if structural decomposition is not taken into account. We provide a solution with a reference model where mirrors are the initial source of meta-information. Finally we validate this solution through a prototype supporting both functional and structural decomposition of reflection.

Continuing we propose a mirror-based model and an infrastructure for remote debugging in reflective languages. Our solution exhibits the four desirable properties that we have identified, namely: interactiveness, instrumentation, distribution and security. Our solution Mercury, supports interactiveness through a causal connection between the meta-level running on the developer machine, and the application to debug (the base-level) on the target device. The two levels are connected both computationally and structurally. Mercury supports instrumentation through the reification of the underlying execution environment (virtual-machine) inside the run-time environment of the target (as an interpreter). Distribution is supported through an adaptable middleware. Finally it supports security in a remote debugging setting by organizing its reflective facilities into two different access groups for - respectively - introspection and intercession. Subsequently, we give a comprehensive comparison of our solution with state-of-the-art. We conclude that in contrast with related work, our approach can in fact meet all the criteria that we have identified.

Next we present a prototype implementation of our proposed model for remote debugging in reflective languages. The Mercury prototype which is written in Pharo [Black 2009] and Slang [Ingalls 1997] consists of four core parts: the meta- level and run-time debugging support (mercury-core) for which we detail its structural organization, installation and initialization through a dedicated debugging seed. The adaptable middleware (seamless), for which we give both a low-level view of the communication infrastructure and a higher-level overview for communication orchestration. The dedicated virtual-machine for the target (metaStackVM) is detailed both from the point of view of the underlying execution environment and from the language's side, where we show how we solved in practice the meta-recursion problem. Then an experimental debugging front-end (alexandria) is presented which follows an MVC pattern [Krasner 1988] with our mirror-based meta-level as the model. Subsequently we discuss engineering trade-offs for implementors of our model.

Finally to validate our results we provide hands-on examples for tool developer's wishing to integrate Mercury in their solutions and detail an experimental setting for the verification of Mercury's properties. Our experimental setting includes three different constraint debugging targets (a smart-phone, a tablet and a remote server) and two case studies discussing remote agile debugging and remote object instrumentation respectively. We verified that Mercury - by being able to dynamically evolve the target's code (interactiveness) - is able to remotely introduce and debug tests without lengthy application re-starts or re-deployments. Thus combining agile Development [Abacus 2005] with debugging in a single remote debugging session. While with our second case study we verified that Mercury through remote object instrumentation can bring the idea of object-centric debugging [Ressia 2012b] in a distributed setting and use it in conjunction with more traditional techniques (such as stack-based navigation) to enhance debugging experimentation on remote targets.

Contributions The results of this dissertation can be utilized by language developers and researchers alike. We answer the following questions: *What are the properties of an ideal remote debugging solution ? Given these properties which model, design principles and patterns could be used to design such a solution ? What are the trade-offs for such an implementation ?*

We identify four desirable properties than an ideal solution for remote debugging should exhibit, namely: *interactiveness*, *instrumentation*, *distribution* and *security*. We describe a mirror-based model for remote debugging (*Mercury*) that exhibits these desirable properties and we detail a prototype implementation [1] in the context of reflective languages discussing implementation trade-offs.

Moreover we provide a solution to the problem of Reflective-Data [Maes 1987b] and illustrate our proposal through a language prototype (*MetaTalk* [2]) [Papoulias 2011]. For the problem of meta-recursion [Denker 2008] we provide a solution for our debugging framework through the reification of a previously illustrative notion (that of the *reflectogram* [Tanter 2003]) as an entity which controls the behavior of the meta-level at runtime.

Finally our work can be used as a reference for implementing an adaptable middleware [David 2002] that supports distribution under different communication contexts (*Seamless* [3]) and for extending a stack-based VM to support advanced intercession facilities (see MetaStackVM [4]).

8.2 Future Work

This section presents open questions that where not addressed in this study and future research perspectives that could extend our work.

8.2.1 Language and Virtual-Machine Debugging in the Same Model

When debugging software written for OO languages that run on top of a virtual machine (like Java, C# and Smalltalk) we *first* adopt a high-level perspective (that of the upper language environment) and *second* make an assumption for sound execution of the virtual machine itself.

There are cases however where adopting a high-level perspective of the environment is not informative enough for the problem at hand. Moreover the assumption about the sound execution of the virtual-machine - however justified for most cases - can be actually false. These scenarios include: core-language development, virtual-machine development, debugging memory sensitive algorithms (e.g efficient hashing in the presence of garbage collection), execution sensitive algorithms (i.e real-time applications), race conditions, multi-threading e.t.c.

Ideally in these cases the developer would use a combined debugging view of both the language and the virtual machine environment. In contemporary systems though since the

[1]http://ss3.gemstone.com/ss/Mercury-Prototype.html
[2]http://www.squeaksource.com/MetaTalk/
[3]http://ss3.gemstone.com/ss/Seamless.html
[4]http://ss3.gemstone.com/ss/mSVM.html

virtual machine is written in a low-level language that is different from the upper language environment, the two can be debugged only seperately from frameworks that do not interoperate.

One way to overcome this problem (while still maintaining a language running on top of a VM) was proposed by Ungar et al [Ungar 2005] and illustrated via the Klein VM project [5]. Klein VM is a metacircular virtual machine, which is effectively written in the same language which it is targeting (the Self language) with the exception of a small core written in a lower level language (C++). Since both the language and the VM are written using the same environment, Klein's reflective facilities can be used to debug on both the language and the underlying VM.

The solution though that Ungar et al propose cannot be used with existing languages. That is without rewriting existing virtual-machines from scratch within a meta-circular framework. Moreover Ungar et al conclude in their work that the efficiency of meta-circular virtual machines (compared with VMs written entirely in lower-level languages) is still an open issue.

Although we do not address this problem in this study, through our experience with the Mercury model and our prior work on high-level debugging abstractions for low-level languages [Papoulias 2010], we believe that a unified model for heterogeneous language systems (such as the low-level virtual-machine and high-level language environment system) is feasible without resorting to meta-circularity. Inter-language reflection which has been exemplified by Gybels et al. [Gybels 2006] is one possible alternative.

We plan to test this hypothesis by extending the Mercury prototype to be able to debug the virtual machine of remote targets.

8.2.2 Automated Debugging Techniques

This study focuses on debugging as a developer driven process. Automated debugging techniques such as automated test generation [Pasternak 2009] and delta-debugging [Zeller 2001] were outside of our research scope. A future direction for our work would be to investigate possible interactions of our framework with such approaches.

Andreas Zeller the inventor of delta-debugging and author of a well known debugging front-end himself (DDD [Zeller 2004]) sees developer driven debuggers as having a *toy-like quality* that can be distracting for developers:

Despite the functionality provided by a debugger, one should keep in mind that interactive debuggers have a certain toylike quality. That is, it is simply fascinating for the creator to see his or her program in action and to exercise total control. This can easily distract from solving the problem at hand [...]

On the other hand recent field studies [Parnin 2011] have put - the possibly less distracting - automatic debugging techniques to question, regarding their ability to solve real-world debugging scenarios.

Our approach in this study views developer driven debugging as an indispensable part of the programming process. We do not see debuggers as part time tools of experimentation. As we discussed on Chapter 2 incremental updating through debugging encourages

[5]http://kleinvm.sourceforge.net/

and supports agile development processes, and more specifically Test Driven Development (TDD) [Abacus 2005]. In the spirit of *live programming* we believe that introducing new behavior through the debugger should not only be possible but it should be actually advised [Black 2009].

From this perspective we advocate that automatic debugging techniques should be incorporated inside the developer driven debugging loop rather than simply supplement it. From a technological point of view our instrumentation framework (see metaStackVM on Chapter 6) can be used to support such an approach. This is also a future perspective that we would like to investigate.

List of Figures

List of Tables

Bibliography

[Abacus 2005] Alex Abacus, Mike Barker and Paul Freedman. *Using Test-Driven Software Development Tools*. IEEE Software, vol. 22, no. 2, pages 88–91, 2005. 35, 101, 108, 128

[Adele Goldberg 1976] Alan Kay Adele Goldberg. Smalltalk-72 instruction manual. Xerox Palo Alto Hesearch Center, Palo Alto, California, 1976. 58

[Alpert 1998] Sherman R. Alpert, Kyle Brown and Bobby Woolf. The design patterns Smalltalk companion. Addison Wesley, Boston, MA, USA, 1998. 44, 45, 49, 53, 71, 74

[Andersen 2004] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen and Mads Torgersen. *Design, Implementation, and Evaluation of the Resilient Smalltalk Embedded Platform*. In Proceedings of ESUG International Smalltalk Conference 2004, September 2004. 64

[Beizer 1990] Boris Beizer. Software testing techniques. Thomson Computer Press, 1990. 24, 25

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 35, 64, 82, 95, 108, 128

[Borning 1987] Alan Borning and Tim O'Shea. *Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language*. In J. Bézivin, J-M. Hullot, P. Cointe and H. Lieberman, editeurs, Proceedings ECOOP '87, volume 276 of *LNCS*, pages 1–10, Paris, France, June 1987. Springer-Verlag. 58

[Bracha 2004] 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. 20, 21, 33, 41, 43, 44, 50, 51, 52, 57, 59, 60, 61, 63, 70, 71, 129

[Bracha 2010] Gilad Bracha. *Linguistic Reflection via Mirrors. Talk at HPI Potsdam*. http://bracha.org/Site/Talks.html, 2010. 40

[Brent Hailpern 2002] Padmanabhan Santhanam Brent Hailpern. *Software Debugging, Testing, and Verification*. IBM Systems Journal, 2002. 24

[Caromel 2001] Denis Caromel and Julien Vayssière. *Reflections on MOPs, Components, and Java Security*. In ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, pages 256–274. Springer-Verlag, 2001. 94

[Cointe 1987] Pierre Cointe. *Metaclasses are First Class: the ObjVlisp Model*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 156–167, December 1987. 62

[Coplien 1992] James O. Coplien. Advanced C++: Programming styles and idioms. Addison Wesley, 1992. 45

[Daniel Weinreb 1981] David Moon Daniel Weinreb. Lisp machine manual. Symbolic Inc., Cambridge, Massachusetts, 1981. 58

[David 2002] Pierre-Charles David and Thomas Ledoux. *An Infrastructure for Adaptable Middleware*. In On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer Berlin Heidelberg, 2002. 19, 20, 33, 126

[Denker 2007] Marcus Denker, Stéphane Ducasse, Adrian Lienhard and Philippe Marschall. *Sub-Method Reflection*. In Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, volume 6/9, pages 231–251. ETH, October 2007. 35

[Denker 2008] 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. 85, 86, 126, 130

[Dias 2011] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse and Gabriela Arévalo. *Clustered Serialization with Fuel*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011), Edinburgh, Scotland, 2011. 88

[Ducasse 2004] Stéphane Ducasse, Adrian Lienhard and Lukas Renggli. *Seaside — a Multiple Control Flow Web Application Framework*. In Proceedings of 12th International Smalltalk Conference (ISC'04), pages 231–257, September 2004. 103

[Ferber 1989] Jacques Ferber. *Computational Reflection in Class-Based Object-Oriented Languages*. In Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, pages 317–326, October 1989. 41, 43, 72, 129

[Fowler 2005] Martin Fowler. Patterns of enterprise application architecture. Addison Wesley, 2005. 71

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional, 1995. 44, 60

[Goldberg 1989] Adele Goldberg and Dave Robson. Smalltalk-80: The language. Addison Wesley, 1989. 64

[Gray 1986] Jim Gray. *Why Do Computers Stop and What Can Be Done About It?* In Symposium on Reliability in Distributed Software and Database Systems, pages 3–12, 1986. 31, 111

[Gupta 2007] Samudra Gupta. Pro apache jog4j. second edition. APress, 2007. 27

[Gybels 2006] Kris Gybels, Roel Wuyts, Stéphane Ducasse and Maja D'Hondt. *Inter-Language Reflection — A Conceptual Model and Its Implementation.* Journal of Computer Languages, Systems and Structures, vol. 32, no. 2-3, pages 109–124, July 2006. 127

[Humphrey 1999] Watts S. Humphrey. *Bugs or Defects ?* Technical Report Vol. 2, Issue 1, 1999. 25

[Ingalls 1983] Daniel H. Ingalls. Smalltalk 80: Bits of history, chapitre The Evolution of the Smalltalk-80 Virtual Machine. Addison-Wesley, Reading, MA, 1983. 41

[Ingalls 1997] 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. 82, 95

[Kiczales 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. *Getting Started with AspectJ.* Communications of the ACM, 2001. 27

[Krasner 1988] G. E. Krasner and S. T. Pope. *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, vol. 1, no. 3, pages 26–49, August 1988. 93, 95

[LaLonde 1990] Wilf R. LaLonde and John R. Pugh. Inside smalltalk: vol. 1. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. 19, 33, 35

[Maes 1987a] Pattie Maes. *Computational Reflection.* PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Belgium, January 1987. 40

[Maes 1987b] Pattie Maes. *Concepts and Experiments in Computational Reflection.* In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 147–155, December 1987. 20, 21, 41, 43, 52, 57, 58, 70, 72, 74, 126, 129

[Maes 1988] Pattie Maes. *Issues in Computational Reflection.* In D. Nardi P. Maes, editeur, Meta-Level Architectures and Reflection, pages 21–35. Elsevier Science Publishers B.V. (North-Holland), 1988. 52

[Malenfant 1991] Jacques Malenfant, Christophe Dony and Pierre Cointe. *Reflection in Prototype-Based Object-Oriented Programming Languages.* In OOPSLA '91 Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programming, 1991. 59

[Mariano 2012] Martinez Peck Mariano. *Application-Level Virtual Memory for Object-Oriented Systems*. PhD thesis, Université de Lille, 2012. 83

[Marschall 2006] Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006. 35

[Martinez Peck 2011] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Efficient Proxies in Smalltalk*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11), Edinburgh, Scotland, 2011. 89

[McAffer 1995] Jeff McAffer. *Meta-level Programming with CodA*. In W. Olthoff, editeur, Proceedings ECOOP '95, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag. 31

[Microsoft 2012a] Microsoft. *Debugger Security, Visual Studio 2012*. http://msdn.microsoft.com/en-us/library/vstudio/ms242231.aspx, 2012. 34, 37, 79

[Microsoft 2012b] Microsoft. *How to: Set Up Remote Debugging, Visual Studio 2012*. http://msdn.microsoft.com/en-us/library/bt727f1t.aspx, 2012. 19, 33, 34

[Microsoft 2012c] Microsoft. *Supported Code Changes (C#), Visual Studio 2012*. http://msdn.microsoft.com/en-us/library/ms164927.aspx, 2012. 34

[Microsoft 2013] Microsoft. *Setting Up Remote Debugging, Visual Studio 2013*. http://msdn.microsoft.com/en-us/library/bt727f1t%28v=vs.71%29.aspx, 2013. 37, 79

[Mostinckx 2007] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont and Eric Tanter. *Mirages: Behavioral Intercession in a Mirror-based Architecture*. In Proceedings the ACM Dynamic Languages Symposium (DLS 2007), October 2007. 55

[Mostinckx 2009] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter and Wolfgang De Meuter. *Mirror-based reflection in AmbientTalk*. Softw. Pract. Exper., vol. 39, no. 7, pages 661–699, May 2009. 55

[Oracle 2013a] Oracle. *Java Debug Interface (JDI)*. http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html, 2013. 19, 33

[Oracle 2013b] Oracle. *Java Platform Debugger Architecture (JPDA)*. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/, 2013. 19, 33

[Oracle 2013c] Oracle. *The Security Manager*. http://docs.oracle.com/javase/7/docs/technotes/guides/security/index.html, 2013. 37, 79

[Papoulias 2010] Nick Papoulias. *High-Level Debugging Facilities and Interfaces: Design and Developement of a Debug-Oriented I.D.E.* In Pär Ågerfalk, Cornelia Boldyreff, JesúsM. González-Barahona, GregoryR. Madey and John Noll, editeurs, Open Source Software: New Horizons, volume 319 of *IFIP Advances in Information and Communication Technology*, pages 373–379. Springer Berlin Heidelberg, 2010. 127

[Papoulias 2011] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Towards Structural Decomposition of Reflection with Mirrors*. In Proceedings of International Workshop on Smalltalk Technologies (IWST'11), Edingburgh, United Kingdom, 2011. 126

[Parnin 2011] Chris Parnin and Alessandro Orso. *Are automated debugging techniques actually helping programmers?* In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM. 127

[Pasternak 2009] Benny Pasternak, Shmuel Tyszberowicz and Amiram Yehudai. *GenUTest: a unit test and mock aspect generation tool*. International Journal on Software Tools for Technology Transfer, vol. 11, no. 4, pages 273–290, 2009. 127

[Redmond 2000] Barry Redmond and Vinny Cahill. *Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java*. In Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures, 2000. 31

[Renggli 2010] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba and Oscar Nierstrasz. *Practical Dynamic Grammars for Dynamic Languages*. In 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, June 2010. 64

[Ressia 2010] Jorge Ressia, Lukas Renggli, Tudor Gîrba and Oscar Nierstrasz. *Run-Time Evolution through Explicit Meta-Objects*. In Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/. 31, 32, 35

[Ressia 2012a] Jorge Ressia. *Object-Centric Reflection*. PhD thesis, Institut fur Informatik und angewandte Mathematik, 2012. 78, 117

[Ressia 2012b] Jorge Ressia, Alexandre Bergel and Oscar Nierstrasz. *Object-Centric Debugging*. In Proceeding of the 34rd international conference on Software engineering, ICSE '12, 2012. xiii, xv, 19, 33, 35, 101, 115, 121, 124, 140

[Richard Stallman 2003] Stan Shebs Richard Stallman Roland Pesch. Debugging with gdb. Gnu Press, 2003. 19, 26, 33, 34, 35, 37, 54, 79

[Rivard 1996] Fred Rivard. *Smalltalk: a Reflective Language*. In Proceedings of REFLECTION '96, pages 21–38, April 1996. 35, 41

[Sillito 2006] J. Sillito, G.C. Murphy and K. De Volder. *Questions Programmers Ask During Software Evolution Tasks*. In Proceedings of the 14th International Symposium on Foundations on Software Engineering, SIGSOFT '06/FSE-14, pages 23–34. ACM, 2006. 115

[Smith 1982] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, 1982. 41

[Sommerville 2001] Ian Sommerville. Software engineering (6th ed.). Addison-Wesley, 2001. 24

[Tanter 2003] Éric Tanter, Jacques Noyé, Denis Caromel and Pierre Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification*. In Proceedings of OOPSLA '03, ACM SIGPLAN Notices, pages 27–46, nov 2003. 20, 35, 78, 85, 86, 126, 129

[Ungar 2005] David Ungar, Adam Spitz and Alex Ausch. *Constructing a metacircular Virtual machine in an exploratory programming environment*. In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pages 11–20, New York, NY, USA, 2005. ACM. 127

[Voas 1992] J.M. Voas. *PIE: a dynamic failure-based technique*. Software Engineering, IEEE Transactions on, 1992. 25

[von Neumann 1945] John von Neumann. *First Draft of a Report on the EDVAC*. IEEE CS Press Book, "The anatomy of a Microprocessor", 1945. 40

[Waldo 1994] J. Waldo, G. Wyant, A. Wollrath and S. Kendall. *A note on distributed computing*. Rapport technique, Sun Microsystems Labs, 1994. 46

[Würthinger 2010] Thomas Würthinger, Christian Wimmer and Lukas Stadler. *Dynamic code evolution for Java*. In Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10. ACM, 2010. 19, 33, 34

[Zeller 2001] Andreas Zeller. *Automated Debugging: Are We Close*. Computer, vol. 34, no. 11, pages 26–31, 2001. 127

[Zeller 2004] Andreas Zeller. *DDD - Data Display Debugger*. http://www.gnu.org/software/ddd/manual/, 2004. 127

[Zeller 2005] Andreas Zeller. Why programs fail: A guide to systematic debugging. Morgan Kaufmann, October 2005. 24, 25, 27, 31, 35

[ZeroTurnAround 2011] ZeroTurnAround. *Java EE Productivity Report 2011*. http://zeroturnaround.com/wp-content/uploads/2010/11/Java_EE_Productivity_Report_2011_finalv2.pdf, 2011. xiii, xv, 30, 124, 140

[ZeroTurnAround 2012] ZeroTurnAround. *What developers want: The end of application Re-deploys*. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012. 19, 33, 34

REMOTE DEBUGGING AND REFLECTION IN RESOURCE CONSTRAINED DEVICES

Building software for devices that cannot locally support development tools can be challenging. These devices have either limited computing power to run an IDE (*e.g.,* smartphones), lack appropriate input/output interfaces (display, keyboard, mouse) for programming (*e.g.,* mobile robots) or are simply unreachable for local development (*e.g.,* cloud-servers). In these situations developers need appropriate infrastructure to remotely develop and debug applications.

Yet remote debugging solutions can prove awkward to use due to their distributed nature. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spend for re-deploying applications while fixing bugs or improving functionality [ZeroTurnAround 2011]. Moreover current solutions lack facilities that would otherwise be available in a local setting because its difficult to reproduce them remotely (*e.g.,* object-centric debugging [Ressia 2012b]). This fact can impact the amount of experimentation during a remote debugging session - compared to a local setting.

In this dissertation in order to overcome these issues we first identify four desirable properties that an ideal solution for remote debugging should exhibit, namely: *interactiveness, instrumentation, distribution and security*. Interactiveness is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (*i.e.,* without stopping the application). Instrumentation is the ability of a debugging solution to alter the semantics of a running process in order to assist debugging. Distribution is the ability of a debugging solution to adapt its framework while debugging a remote target. Finally security refers to the availability of prerequisites for authentication and access restriction.

Given these properties we propose Mercury, a remote debugging model and architecture for reflective OO languages. Mercury supports interactiveness through a mirror-based remote meta-level that is causally connected to its target, instrumentation through reflective intercession by reifying the underlying execution environment, distribution through an adaptable middleware and security by decomposing and authenticating access to reflective facilities. We validate our proposal through a prototype implementation in the Pharo programming language using a diverse experimental setting of multiple constraint devices. We exemplify remote debugging techniques supported by Mercury's properties, such as *remote agile debugging* and *remote object instrumentation* and show how these can solve in practice the problems we have identified.

Keywords:

Remote Debugging, Reflection, Mirrors, Interactiveness, Instrumentation, Distribution, Security, Agile Development

LE DEBOGAGE A DISTANCE ET LA REFLEXION DANS LES DISPOSITIFS A RESSOURCES LIMITEES

La construction de logiciels pour des appareils qui ne peuvent pas accueillir localement des outils de développement peut être difficile. Ces appareils soit ont une puissance de calcul trop limitée pour exécuter un IDE (par exemple, smartphones), ou manquent d' interfaces d'entrée / sortie appropriées (écran, clavier , souris) pour la programmation (par exemple, les robots mobiles) ou sont tout simplement inaccessibles pour des développements locaux (par exemple cloud - serveurs). Dans ces situations, les développeurs ont besoin d'une infrastructure appropriée pour développer et déboguer des applications distantes.

Des solutions de débogage à distance sont parfois délicates à utiliser en raison de leur nature distribuée. Les études empiriques nous montrent que, en moyenne 10,5 minutes par heure de codage (plus de cinq semaines de travail de 40 heures par an) sont passées pour le re-déploiement d'applications pour corriger les bugs ou améliorer leur fonctionnalité [ZeroTurnAround 2011]. En plus, les solutions courantes manquent des aménagements qui seraient autrement disponibles dans un contexte local, car c'est difficile de les reproduire à distance (par exemple débogage objet-centré [Ressia 2012b]). Cet état influe sur la quantité d' expérimentation au cours d'une session de débogage à distance - par rapport à un contexte local.

Dans cette thèse, afin de surmonter ces problèmes, nous identifions d'abord quatre propriétés désirables qu'une solution idéale pour le débogage à distance doit présenter : *l'interactivité, l'instrumentation, la distribution et la sécurité*. L'interactivité est la capacité d'une solution de débogage à distance de mise à jour incrémentale de toutes les parties d'une application sans perdre le contexte de d'exécution (sans arrêter l'application). L'instrumentation est l'aptitude d'une solution de modifier la sémantique d'un processus en cours en vue d'aider le débogage. La distribution est la capacité d'une solution de débogage à adapter son cadre alors que le débogage d'une cible à distance. Enfin la sécurité fait référence à la disponibilité de conditions préalables pour l'authentification et la restriction d'accès.

Compte tenu de ces propriétés, nous proposons Mercury, un modèle de débogage à distance et une architecture pour des langues réflexifs à objets. Mercury ouvre (1) l'interactivité grâce à un méta-niveau à distance miroir basé sur un lien de causalité avec sa cible, (2) l'instrumentation à travers une intercession réflective basée sur la réification de l'environnement d'exécution sous-jacent, (3) la distribution grâce à un middleware adaptable et (4) la sécurité par la décomposition et l'authentification de l'accès aux aspects réflexifs. Nous validons notre proposition à travers un prototype dans le langage de programmation Pharo à l'aide d'un cadre expérimental diversifié de multiples dispositifs contraints. Nous illustrons des techniques de débogage à distance supportées par les propriétés de Mercury, tels que le *débogage agile distant* et *l'instrumentation objet à distance* et montrons comment ils peuvent résoudre dans la pratique, les problèmes que nous avons identifiés.

Mots clés:

Débogage à distance, Reflexion, Miroirs, Interactivité, Instrumentation, Distribution, Sécurité, Développement Agile