Rehwaldt, Nico
Exploring Run-time Behavior in Reversible Experiments

Exploring Run-time Behavior in Reversible Experiments

An Application of Worlds for Debugging

by

Nico Rehwaldt

A thesis submitted to the
Hasso-Plattner-Institute for Software Systems Engineering
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in Software Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Bastian Steinert

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

July 30, 2012

Abstract

Debuggers are often used to inspect running software systems in order to support the understanding of source code. They differ substantially from most tools that visualize run-time data as they make it possible to keep track of a running program as it executes. Thereby they encourage learning through experience. At the same time, conventional debuggers do not properly support incremental understanding. The reason is that they offer only limited options to safely re-explore a particular run-time behavior without restarting the program under observation.

Against that background, this work examines Worlds [44]—a language extension that allows for scoping of side effects in imperative programming languages. The thesis evaluates the concept of Worlds to isolate side effects caused during debugging. By doing so, the effects of prior explorations can be discarded and the examination of run-time behavior gets safe to be repeated until it yields the desired knowledge gain.

In the course of this work, we present the principle application of Worlds to enable debugging in reversible experiments. Furthermore, we propose a general purpose Worlds mechanism for Squeak/Smalltalk, which is necessary to evaluate the practical application of the concept for Smalltalk. In the course of the evaluation, we present a debugger that employs the mechanism to realize the safe re-examination of behavior inside a debugging session.

Zusammenfassung

Debugger ermöglichen es das Laufzeitverhalten eines Programms zu inspizieren und tragen damit mittelbar zum besseren Verständnis des zugrundeliegenden Quellcodes bei. Sie unterscheiden sich dabei grundsätzlich von anderen Werkzeugen die das Programmverstehen unterstützen, da sie es ermöglichen eine Programmausführung kontrolliert zu durchlaufen. Dadurch stärken sie das Verständnis von Programmen an Hand von Beispielen. Andererseits ist es schwierig, Programmverhalten mit Hilfe von Debuggern schrittweise zu verstehen, da konventionelle Debugger nur eingeschränkte Möglichkeiten bieten, lokal auftretendes Verhalten wiederholt zu betrachten.

Vor diesem Hintergrund betrachtet die vorliegende Arbeit Worlds [44], eine Erweiterung für objektorientierte Programmiersprachen die es ermöglicht, während der Programausführung auftretende Seiteneffekte zu kontrollieren. Im Rahmen von Debugging eingesetzt, erlaubt es Worlds die Effekte, die durch das Betrachten von Programmverhalten entstehen, einzufangen und zu verwerfen. Das ermöglicht eine wiederholte Ausführung von Programmteilen bis der zugrundeliegende Quellcode vollständig verstanden wurde.

Diese Arbeit stellt die prinzipielle Anwendung von Worlds im Rahmen von Debugging vor. Weiterhin präsentiert sie einen allgemein anwendbaren Worlds-Mechanismus für Squeak/Smalltalk, der es ermöglicht, die Anwendung von Worlds im Rahmen von Debugging praktisch zu erproben. Im Zuge der Evaluierung beschreibt sie einen Debugger, der die sichere Wiederausführung von Laufzeitverhalten basierend auf dem Standarddebugger für Squeak/Smalltalk und dem vorgestellten Worlds-Mechanismus ermöglicht.

Acknowledgments

First of all, I thank my girlfriend, Juliane for the thought experiments we conducted in the time I wrote this thesis. While these experiments helped me tremendously to grasp the full power of Worlds, I am thankful that all of them were properly isolated and left both, our apartment walls and our fridge, in an acceptable state.

Further, I would like to thank my supervisors Prof. Dr. Robert Hirschfeld and Bastian Steinert for the fruitful talks we had in the course of finding, evolving and finishing this thesis. I continue to appreciate their honest feedback and criticism and hope that some of their advices led to improvements in this work.

Thank is due to my family, too, which showed great interest in the thesis progress and kept my motivation up. I especially value Jan's efforts to read through this work to point out a seemingly endless number of small issues and to identify bigger rooms for improvements.

Finally, I would like to thank a whole bunch of other people for commenting on this thesis and making writing it enjoyable, first and foremost Tobias with his type setting tricks, the room C-E.4 folks and the Sweden crew.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	application programming interface
AST	abstract syntax tree
CLI	command line interface
COP	context-oriented programming
GDB	GNU Project Debugger
GUI	graphical user interface
IDE	integrated development environment
UI	user interface
UML	Unified Modelling Language
VM	virtual machine

1. Introduction

Comprehending a software system is crucial for both developing and maintaining it. To implement a certain feature, for instance, a developer must understand the program he wants to extend to figure out where and how the functionality is implemented best. The same applies for bug fixing, where a particular part of a program must be understood in order to locate a bug and eventually eliminate it. Not surprising, developers spend a considerable amount of their time with program comprehension [7, 19]. During that activity they often turn to debuggers to obtain detailed information about a program [21, 30].

Debuggers make it possible to interact with running programs by stepping through them in a user-controlled fashion. By doing so, debuggers enable it to track down the effects caused by executing certain program parts. This aids comprehension as it encourages learning through experience[1] when drawing connections between program behavior and code [17]. Arguably, it also supports a deeper kind of understanding than simply looking at static views such as source code [17, 20].

1.1. Motivation

During a debugging session, users need to *track changes* in the program state *live* and draw connections to particular parts of the executed code. The complexity of that process leads to the fact that run-time behavior is usually not understood immediately, rather knowledge about it is built up gradually until everything forms a big picture [27]. Navigation errors and incomplete perception entail that insights are often gained in retrospect, that is, when it is already too late. For instance, a user may notice the importance of a statement when he stepped over it only. Or he might suddenly understand the relevance of a variable which he failed to pay attention to before. To compensate for

[1] We may also refer to it as *experimental learning* [17].

these issues, run-time behavior often needs to be observed repeatedly until it is fully understood [22, 35][2].

Most conventional debuggers allow the re-execution of methods through stack rewinding. Often, however, that technique is not sufficient for the safe re-examination of behavior because it fails to revert global side effects caused by stepping through the program. The lack of other options forces developers to restart both program and debugging session whenever they want to repeatedly observe a particular behavior from a well-defined starting position. That in term is tedious and time consuming as the old execution context is lost and a new one—similar to it—has to be established manually.

Technically, approaches like back-in-time [22, 23] or record-and-replay [39] debuggers solve the issue by allowing the deterministic re-examination of run-time behavior. However, they face two issues which "renders them impractical for frequent use" [30]. At first, they are inherently *postmortem*, that is they allow re-examination of run-time behavior after the original program died or the execution context of interest is gone, only. Therefore, they fail to support local re-examination of behavior on top of already running programs [32]. Second, they have performance and scalability issues as huge amounts of run-time information have to be recorded [23, 32].

1.2. Thesis in a Nutshell

This master thesis examines Worlds [44]—a language extension that can scope side effects in imperative programming languages—in the context of debugging. It evaluates the application of the concept for the safe re-examination of behavior on top of conventional debuggers. In contrast to approaches like record-and-replay or back-in-time debuggers, which realize this through snapshotting or recording techniques, Worlds makes it possible by capturing and later discarding side effects of local program exploration. That in term makes the *online* exploration of run-time behavior during a debugging session safe to be repeated until it yields the desired knowledge gain.

In its core, this work has three contributions: First of all, it presents a experimentation model. The model describes the principal application of Worlds during debugging, which enables the exploration of run-time behavior in repeatable experiments. Second it introduces DWorlds, a general purpose Worlds

[2] The same observations apply to search results in source code or other forms of big amounts of information, where entities are often skimmed rather than fully understood [40].

implementation for Squeak [4]—an interactive programming and execution environment based on Smalltalk-80 [12]. That mechanism can be used to safely conduct large-scale experiments on arbitrary Smalltalk-written applications. Third, it describes the dwdbg, a debugger prototype for Squeak/Smalltalk that employs DWorlds and the experimentation model to isolate the side effects caused during parts of a debugging session. By doing so it, allows developers to safely re-explore program behavior by discarding the effects caused by prior explorations.

1.3. Outline

The remainder of this thesis are structured as follows: Chapter 2 lays the basis of this thesis by presenting background information on the topic. Building on these insights, chapter 3 introduces Worlds and applies the idea of Worlds to debugging. Furthermore, it justifies the need for a general purpose Worlds for Smalltalk to be applicable in the described scenario and enlists requirements for such an implementation. Chapter 4 presents DWorlds, the implementation of a generic Worlds for Smalltalk. Chapter 5 evaluates the application of DWorlds as a general purpose Worlds mechanism. In addition, it discusses the application of DWorlds in the dwdbg, a debugger which realizes the safe re-execution of behavior inside a debugging session. Chapter 6 depicts related work in the area. Last but not least, chapter 7 recapitulates the main findings of this work and sums up the topic.

2. Background

In this chapter we introduce the background necessary to dive into the topic. For that purpose we examine a number of areas adjacent to this work. In section 2.1 we dig into program comprehension and how to aid expert developers. Following up, section 2.2 takes a look at dynamic views and their application to aid understanding of software. Section 2.3 introduces debuggers and relates them to program comprehension and dynamic views. In addition, it characterizes a typical debugging session as a series of experiments. In section section 2.4 we conclude this chapter by presenting a light-weight approach to support the safe re-exploration of behavior inside a debugging session.

2.1. Program Comprehension

For developing or maintaining a software system, programmers need to have a sufficient understanding of the subject matter. *Program comprehension*, having said this, denotes both the ability and the process of gaining understanding of a software system. It comprises the capability to explain the functioning of a program and, thus, distinguishes from simply reading out source code. Biggerstaff et al. gave a concise definition of the term in [3]:

> [Program comprehension is the ability to] explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program [3].

The activity is an integral part of a software developers activity and stays very important in all phases during software development and maintenance [27]. In fact, studies have shown that programmers spend up to 60 percent of their time with understanding the program they are working on [7].

2.1.1. Comprehension Models

Research in the area has come up with various models which aim to explain how program comprehension works. Despite the differences in details, most models have a number of key points in common [27]. All approaches agree that programmers posses two kinds of knowledge: General knowledge and software knowledge. *General knowledge* is knowledge independent of a software such as knowledge about algorithms, patterns and procedures. *Software knowledge* on the other hand represents the level of understanding of software-specific details. The process of understanding matches both kinds of knowledge to a *mental model* of a program until the developer feels that he understood the program sufficiently well [27].

2.1.2. Supporting Expert Developers

Strategies to gain software knowledge and create a mental model of an application differ between novice programmers and expert developers. Mayrhauser and Vans report that experts are likely to perform shallow reasoning when building up mental models [27]. For instance, they are often seen to skim source code rather than analyzing it in depth[1]. In contrast, novice programmers are more likely to carry out deep reasoning and thereby try to figure out the relationship between objects (e.g. code artifacts) in a detailed analysis [27].

Concerning the differences between experts and novices Mayrhauser and Vans continue to note:

> Experts approach problem comprehension with flexibility. They discard questionable hypotheses and assumptions much more quickly than novices do, and they tend to generate a breadth-first view of the program. As more information becomes available, they refine their hypotheses [27].

That means, expert developers can be supported by providing them with the right information to quickly strengthen or discard hypotheses. One valuable source of such information is actual data taken from a running program.

[1] An observation which Starke et al. could confirm in exploratory studies on source code search and skimming [40].

2.2. Running Programs as Sources of Insights

Dynamic views present data available at run-time. Thus, they provide information which might be hard to extract from static representations such as source code. Often they enrich static views to increase their value for the understanding of particular aspects of a program. The reasons why developers employ dynamic views to help understanding programs are three-fold:

- Most information about the behavior of a program can be extracted from source code. Often though, the ability of doing so is limited by time constraints as well as the human capacity to remember and/or build internal mental models [8].
- Short-cutting the process of source code comprehension programmers often accept incomplete knowledge [27, 40]. At the same time, they create assumptions based on their existing knowledge and experience. Dynamic views offer means to strengthen or disprove these assumptions and thus "eliminate space for speculation" [30].
- Dynamic views provide examples of run-time behavior for inherently abstract static representations such as source code. They act as external presentations for the source code and thereby contribute to its understanding [42, 34].

Consequently, the need for dynamic views to support program comprehension is clearly recognized in literature [7, 8, 30, 32, 34, 41]. Research in the field resulted in various approaches to visualize run-time data in models [2, 8, 37, 43], directly embed run-time information into a integrated development environment (IDE) [30, 34] or use special tools such as the debugger for live inspection [32, 41].

2.2.1. Dynamic Views in the IDE

The work by Röthlisberger et al. showed how source code could be enriched with run-time information [34]. User studies they conducted suggest that run-time data embedded directly into source code accelerated the understanding of the software at hand, e.g. because users knew exactly, which methods were invoked during a particular execution of a program [34].

Perscheid et al. argue that tools to provide dynamic views should "allow for a feeling of immediacy to encourage frequent use" [30]. Hence, these

tools must be tightly integrated into an IDE, easily accessible and responsive (e.g. with a short start up time) in order to be useful. To provide dynamic views with immediacy characteristics, Perscheid et al. encourage the use of a two-phased analysis of collected data. Implementing that idea they present a tool for visualizing run-time data that shows satisfactory results in both responsiveness and level of detail provided [30].

2.2.2. Collecting Run-time Data

Information visualized in dynamic views must be collected from a running program. Different approaches to do so have been presented in related work on the topic. Often employed to collect run-time data are *instrumentation* techniques [2, 30, 34]. Other approaches rely on *reflection* [2] or use a mixture of both techniques [37]. All tools require some sort of user intervention to either *manually* [2, 8, 34] or *semi-automatically* [30] to collect run-time data before a dynamic view can be presented to the user.

2.2.3. Online vs. Postmortem

Whether the program whose run-time data is visualized is still alive when the data is presented to the user classifies approaches as online or postmortem [2]. *Online* approaches collect, analyze and visualize data at run-time and allow interaction with the still running program. They are—due to their nature—specific to the current run of a software system. *Postmortem* approaches instead kick in when the program already died. They are not a priori specific to a certain run of a program of interest. Therefore, postmortem approaches are challenged to carry out *typical* or *appropriate* runs of a program to gather data that represents meaningful and relevant results. Most tools that provide postmortem dynamic views leave it up to the user to decide what a typical run is and refrain from gathering run-time data through different runs of a software. Some, in contrast, facilitate running unit tests to collect the data semi-automatically [30, 41].

2.3. Debugging

Formally, the term *debugging* refers to "the process of finding and reducing the number of bugs in a computer program, thus making it behave as expected"[2].

[2]Origin of statement unknown; used unquoted in many places on the internet and in [23]

More broadly, however, it can be seen as "the process of interacting with a running software system to test and understand its [...] behavior" [32]. That is why programmers use *debuggers* both for the sake of bug fixing and comprehending a program [21, 32, 41].

By means of user-defined break points—typically put in the source code of a program—a debugger interrupts a running program and enables the user to walk through the remainder of the execution in a step by step fashion.

2.3.1. Debuggers as Dynamic Views

Conventional debuggers usually show the run-time state of a program along with source code or another static representation associated with the current execution context. As such, they provide a dynamic view on a software system. In two aspects, though, they stand out against most other approaches towards dynamic views: They are extensible tools to inspect and interact with running programs and rely on the user to comprehend these through trying out and experience.

Extensible Online View

The program being debugged is alive when the interaction with it through a debugger happens. Changes in the program state are immediately reflected in the debugger's user interface (UI). That said, conventional debuggers offer an online view on the running software system.

While debuggers were originally command line interfaces (CLIs) [29], most modern IDEs offer graphical interfaces to debuggers which tightly integrate in the development environment. Typically, the view on the running program presented by debuggers is not fixed but can be extended by a user to his or her special needs. Often for example, debuggers initially show a simple view on the current execution context only. That view, however, allows a user to dig deeper into the program state to display and track variables he regards relevant for the understanding of the debugging subject. That is possible not least because the user can always resort to the running program to directly interact with it and thus figure out special pieces of information.

Built for Interaction

Most other approaches towards dynamic views focus solely on providing more or less elaborated views on run-time data. Debuggers, in contrast, are

built for interaction with a running program. They allow to experience running programs by stepping through them rather than simply looking at them. That in term encourages experimental learning [17] and supports a deeper kinds of understanding [20].

Interestingly enough, programmers can even change program state manually during a debugging session and thus are allowed to directly interfere with the program execution. They would do that to compensate errors in the running program or to test follow up execution under different preconditions. One use case of that activity is playing with a variable to observe how changing it affects the actual execution of the program under observation.

2.3.2. Supporting Program Exploration

During a debugging session programmers usually inspect a particular code section or behavior in order to understand it. Thereby they have particular questions in mind. These questions can range from a general concerns, for instance "What does that code do?", to very specific information needs such as "When did that variable get the wrong value?" [36]. Due to the nature of comprehension, often new questions arise in the process that need to be answered accordingly, too [27]. That said, debugging for program comprehension equals a series of *exploratory experiments* on a running program with the goal to answer a number of particular of questions [32].

From a given starting point—i.e. a specific execution context and a well-defined global state—a user starts an experiment with one or more questions in mind. He explores the program behavior by stepping through it. Whether he is able to answer or discard his questions denotes the success of the experiment. That said exploratory experiments can fail for two reasons. At first, a developer may fail to understand the executed behavior or might have understood it only to a certain extend. Usually that is the case if he needs to build up knowledge about a particular behavior incrementally before he is able to understand it sufficiently well [27]. Second, new questions might have come up during the exploration and the programmer realizes that these new questions have to be answered first in order to get the answer to the actual question.

Both cases lead to the fact that explorations need to be repeated or new explorations have to be started. Before that is possible the user needs to undo the remainder of the old exploration. These remainders include the global side effects caused by stepping through the program as well as the execu-

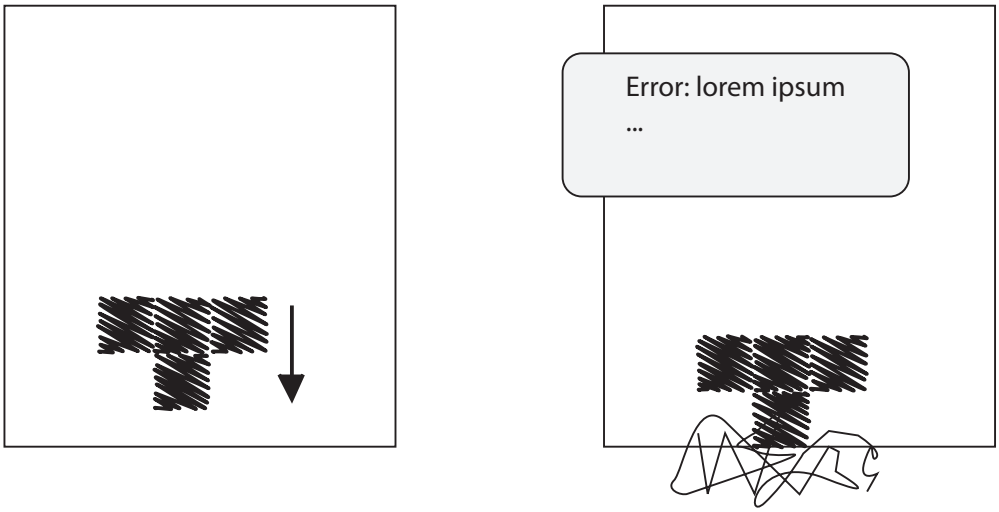

Figure 2.1.: Observable but hard to reproduce program state in a game that leads to an error when the T-brick reaches the ground

tion context which probably changed. The incapability of most debuggers to undo the side effects leads to the fact that program and debugger have to be restarted to re-establish the proper basis for exploration.

Local and Global Explorations

When talking about understanding run-time behavior we distinguish local and global explorations. *Local explorations* are tied to a particular execution context and program state. They are conducted to understand how a distinct part of a program works and how that actually affects the application. Examples for a local exploration is the—possibly repeated—stepping through a method in order to understand what it does. In contrast, *global explorations* are not tied to a single execution context. Instead, they base on a particular perceivable program state, only and aim to understand the upcoming parts of the program execution. Global explorations are often conducted in the area of graphical user interface (GUI) applications where the impact of external events (e.g. user interactions) or internal stepping mechanisms must be understood. An example for that is the implementation of a well known game where a T-brick reaching the ground triggers an error (cf. figure 2.1). While the behavior is easy to perceive, it may be hard to track down and reproduce the error.

2.4. A Light-weight Approach to Safe Experimentation

Expert developers usually have a good feeling about which pieces of a program are important for a specific use case and thus may need in-depth examination [27, 40]. Based on that knowledge, light-weight approaches can be employed to makes explicitly indicated exploratory experiments repeatable. One way to do so is *checkpoint debugging* which gives the user a chance to specify points in a debugging session he can can revert to later. The GNU Project Debugger (GDB)[3] supports it and the GDB user documentation describes concisely what checkpointing is:

> Returning to a checkpoint effectively undoes everything that has happened in the program since the checkpoint was saved. [...] Effectively, it is like going back in time to the moment when the checkpoint was saved.
>
> Thus, if you're stepping thru a program and you think you're getting close to the point where things go wrong, you can save a checkpoint. Then, if you accidentally go too far and miss the critical statement, instead of having to restart your program from the beginning, you can just go back to the checkpoint and start again from there.
>
> This can be especially useful if it takes a lot of time or steps to reach the point where you think the bug occurs [10].

Checkpoint debugging cannot guarantee *perfect determinism* when examining behavior repeatedly. The reason is that program parts have to be re-executed on top of checkpoints to repeatedly inspect the run-time behavior. As a result components like time, external uncontrollable resources (i.e. some web service) or the presence of multi-threading can screw with the program execution and lead to different results in consecutive runs. Still, it is often sufficient for behavioral re-examination. The reason is that it bases the re-execution on the same prerequisites (i.e. program state) and thus allows the user to recognize important beacons such as previously discovered important variable changes from one state to the other. We say that re-examination of behavior based on checkpointing is *quasi deterministic*, that is deterministic for the majority of every day use cases.

[3] `http://www.gnu.org/software/gdb/`

Debuggers supporting checkpointing face two issues, (1) saving the execution context to return to it later and (2) preserving the program state at a given point in time so that it can be restored later and re-execution of the behavior can be achieved. The GDB uses platform-specific features to tackle these issues. On Linux systems for instance it realizes checkpointing using a platform specific snapshotting technique. On some platforms, in contrast, it does not support checkpointing due to the lack of platform features to implement it [11].

⁂

The remainder of this work examines if something similar to checkpoint debugging can be implemented platform independent given the ability to control the scope of side effects. Worlds [44], an interesting extension to object-oriented programming languages provides exactly that functionality. Rather than snapshotting the program state at a given point in time, it could allow us to execute program behavior during a debugging session inside an experiment whose effects can easily be discarded. Worlds serves as a basis for our study which starts in the following chapter.

3. Experimenting with Worlds

Warth et al. introduce Worlds as a language construct that "reifies the notion of program state and allows programmers to limit the scope of side effects" [44]. Particularly interesting about Worlds is that it can capture effects such as state changes caused by parts of a program's execution. Reflecting on execution results a program can decide whether the effects should become globally visible or can safely be forgotten. This unique ability renders Worlds a clean and simple mechanism to safely perform speculations and experiments within iterative programs. In the context of debugging, Worlds can be an interesting tool to look at, too. Because of its ability to scope side effects it can be employed to encapsulate certain parts of a debugging session in exploratory experiments whose effects can be easily reverted.

This chapter examines the application of Worlds in the debugging domain. To start with section 3.1 briefly introduces the language mechanism as presented by Warth et al. in [44]. Following up, section 3.2 shows how the concept of Worlds can be used in principle to make exploratory experiments during a debugging session reversible. Section 3.3 evaluates the applicability of the current Worlds implementation for Smalltalk to be employed in the described scenario. Based on these findings, section 3.4 motivates the need for a generic Worlds for Smalltalk that enables the scoping side effects caused by arbitrary Smalltalk-based applications.

3.1. Worlds in Brief

To start with, this section gives an overview about the goals of Worlds [44], depicts the basic concepts behind it and provides examples how it feels being embedded in the Smalltalk programming language. Furthermore, it presents the details needed to understand and evaluate the current implementation of Worlds for Squeak/Smalltalk.

3.1.1. Goals and Applications

One of the major drivers for Worlds is the need to safely perform speculations and experiments within iterative programs. It does so by offering a "clean and flexible mechanism for controlling the scope of side effects" thereby implementing a simple way of "doing and undoing" [44].

Worlds becomes especially useful whenever the failure of particular actions is anticipated and needs to be compensated or rolled back through some sort of cleanup operation. That comprises a number of applications like trying to randomly find solutions for difficult problems or working with unreliable services. Furthermore, it fits use cases such as user-triggered undo or transactional memory [13, 35] semantics.

3.1.2. Underlying Concepts

Central to Worlds is the *world* which is the scope object state is perceived and parts of a program can be executed in. While having the same identity an object may have a different state depending on in which world[1] it is inspected (cf. figure 3.1). Conceptually there does not exist something like non-world execution, because an instance of a world—top-level world representing the global state—is active per default. In practice, we distinguish between the execution in the top-level world, which yields the normal unscoped program behavior and the execution in nested worlds that scopes side effects. In the following we refer to the latter as in-worlds execution while we regard the top-level world as the *root* or *global scope*. Beginning from the top-level world, worlds can be arbitrarily nested to allow independent experiments based on the parent scope. By nesting worlds, two worlds span a parent-child relationship in which the child world initially inherits the state of the parent.

To allow safe experimentation Worlds ensures two important properties: Isolation and consistency.

Properties for Safe Experimentation

Similar to transactions in software transactional memory [13, 31, 35] worlds *isolate* the behavior executed in them [44]. In the context of Worlds that means that (1) side effects are captured in a world so that program parts executed

[1] Mind the difference when reading on: *Worlds* denotes the language construct described in [44], a *world* is the environment proclaimed by Worlds and *worlds* is the multiple of a world.

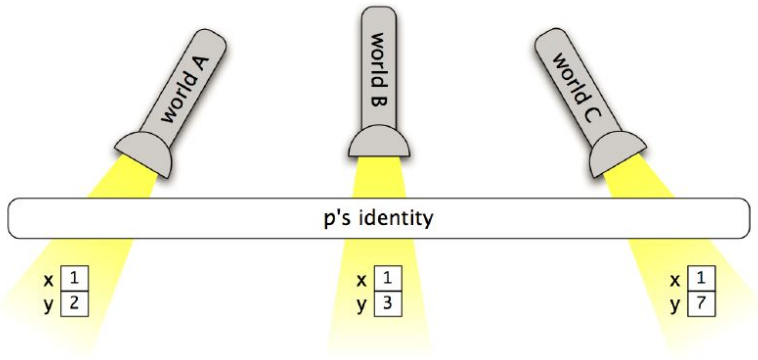

Figure 3.1.: An object inspected in different worlds (taken from [44]).

outside it are not affected and that (2) execution outside the world does not affect the execution of worlds scoped program parts. Ensuring the isolation property leads to two important insights: First, worlds preserve a consistent view on object state. Once read, the state of an object in the scope of a world will not change unless it is changed from inside the world[2]. Second, side effects produced by execution inside a world are not visible to the outside until the world state is explicitly *committed*.

The other property Worlds adheres to is *consistency* which is also known from transactions [31, 44]. Consistency ensures that the side effects captured in the world are either committed as a whole or no change is made when a world should be committed. This makes sure that the parent scope of a world is never left in an inconsistent state.

Ensuring Consistency

To ensure consistency in Worlds, worlds can only be committed when they are *up-to-date* with regard to their parent, i.e. the parent of a world did not change in the meantime. For a parent world p and its child c it is more specifically defined in the following way: For every object state s_c accessed in c the corresponding value s_p in p did not change since s_c was first accessed. This property makes sure that committing a world cannot override changes in

[2]Warth et al. call that the "no surprises" property

the parent world unless whatever was executed in it is independent[3] of these changes[4].

Worlds employs an optimistic concurrency control protocol and checks the *up-to-date-ness* during the commit phase only [31]. By doing so it makes it possible to run multiple experiments in parallel—independent to each other, the state of the parent and of whether a world should ever be committed or not. Upon commit of a world a check is performed which makes sure that the world is still up-to-date, thereby making sure that committing it does not put the parent in an inconsistent state [44]. Worlds are committed in a uninterrupted manner so that only one commit can be performed at a time and global consistency is ensured.

The Worlds API

A world w in Worlds is a first-class entity and as such reifies program state. To let the magic happen, it offers the following abstract operations:

$\texttt{sprout}(w) \rightarrow w_{\texttt{child}}$ Create and returns a child world $w_{\texttt{child}}$ of w that inherits all captured side effects. Preserving the isolation property, changes in the child will not be visible in w before it got committed.

$\texttt{eval}(w, \texttt{block}) \rightarrow \texttt{anything}$ Evaluates `block` (a number of statements) in the scope of the world w. Side effects produced by the block execution get captured in w and the result of the execution is returned.

$\texttt{commit}(w)$ Commit the side effects in world w into the parent world, thereby making sure that the consistency property is not violated.

Interested readers may have noticed that there is no `abort` or `rollback` operation as known from transactions. The reason is a conceptional one: Rather than manually removing a world and thus cleaning it up, it is forgotten and cleaned up automatically (e.g. by the garbage collector when it is no longer referenced).

[3] i.e. accessing different objects

[4] Up-to-date-ness in term is a common consistency semantic found in versioning control systems as well as the implementation of transactional memory for Smalltalk presented by Renggli and Nierstrasz [31].

3.1.3. Look and Feel

While the previous subsections hopefully laid the foundation needed to understand Worlds as a concept, we will now show how it actually feels using it. Therefore, we discuss a small example of how Worlds looks like when being embedded in Smalltalk.

The basis of our example is shown in listing 3.1. Two characters exist in the global scope: `waldi`, a pet and `klaus`, a person. Both do not yet know each other. Given both `waldi` and `klaus`, we want to use Worlds to experiment with their relationship. We obtain the active world `w1` using a factory function as shown in listing 3.2. As it represents currently active scope we need to create a child world `w2` to experiment in.

```
| klaus waldi w1 w2 |

waldi := Pet new.
waldi name: 'Waldi'.

klaus := Person new.
klaus name: 'Klaus'.
```

Listing 3.1: Klaus and Waldi not knowing each other

Within `w2` we make `klaus` and `waldi` get to know each other whereas we take the opportunity to change `klaus`' name to Achim, too. Checking the name of `klaus` we can assert that `klaus` is in fact called Achim in the scope of `w2`.

```
"Obtain the current world"
w1 := DWorld current.

"Create a child world"
w2 := w1 sprout.

w2 eval: [
  klaus
    pet: waldi;
    name: 'Achim'.

  klaus name. "Achim"
].
```

Listing 3.2: Experimenting with relationships and names

Outside our experiment we can confirm that nothing changed (i.e. the isolation property has been obeyed). Neither has `klaus` been renamed nor did he get to know `waldi` (cf. listing 3.3). Things change, however, when the experiment executed in `w2` gets committed. In that case both know each other and `klaus` indeed has to apply for a new passport.

```
"Changes local to the world are not visible outside of it"
klaus name. "Klaus"
klaus pet. "nil"

"Unless the world is committed"
w2 commit.

klaus name. "Achim"
klaus pet name. "Waldi"
```

Listing 3.3: Unless committed, an experiment remains speculation

But what happens if we change `klaus` name in `w1` right before committing the experiment? Taking the consistency rules mentioned in section 3.1.2 into account, experiments are not allowed to override changes in the parent scope. Therefore, trying to commit `w2` would signal an error instead of leaving our top-level world in a possibly inconsistent state (refer to listing 3.4).

```
w2 eval: [ "our experiment" ].

"Change name outside the experiment"
klaus name: 'Sven'.

"Will signal serialization error"
"(klaus' name changed in both w1 and w2)"
w2 commit.

"While w2 was not committed"
klaus pet. "nil"
```

Listing 3.4: Programmer in trouble

This example should have given a basic impression how it feels like to work with Worlds. Following up we will dig a bit deeper into Worlds for Squeak/Smalltalk and highlight a few implementation-specific specialties.

3.1.4. Inner Workings of Worlds for Smalltalk

Warth et al. realized Worlds prototypes for both JavaScript and Squeak/Smalltalk and describe their Squeak implementation as the "more performant prototype of Worlds" [44]. We will now focus on Worlds for Squeak/Smalltalk and depict the most important details about the implementation.

Accessing World-specific Object State

Worlds for Squeak implements world-aware objects in subclasses of `WObject`. For the class and its subclasses Worlds changes instance variable lookup and store semantics to achieve world scoped instance variable access [44].

In instances of `WObject` and its subclasses state access is redirected to the objects state in the currently active world. To realize this behavior `WObject` overrides the methods `#instVarAt:`, `#instVarAt:put:`, `#at:` and `#at:put:` which are responsible for variable reads, writes as well as indexed field accesses. The standard Squeak compiler inlines instance variable access in methods rather than using the generic accessors `#instVarAt*`. To fix this, Worlds ships with a special compiler. This compiler transforms methods in subclasses of `WObject` to use the `#instVarAt*` methods rather than direct instance variable access and thus makes the access Worlds-friendly. As the transformation happens before the compilation to byte-code, it is not visible in the sources of transformed methods (cf. Listings 3.5 and 3.6).

```
Person#meAndMyPet

 ^ self name,
   ' and ',
   self pet name.
```

Listing 3.5: Method source

```
Person#meAndMyPet

 ^ (self instVarAt: 0),
   ' and ',
   (self instVarAt: 1) name.
```

Listing 3.6: Method transformation applied by the Worlds compiler

Implementing Worlds features on top of `WObject` limits the application of Worlds to classes based on `WObject`. As a result applications must be rebased on `WObject` if they want to benefit from Worlds-specific features. Warth et al.'s Worlds implementation for Squeak/Smalltalk ships with some basic Worlds-aware classes. These comprise reimplementations of `Array`, `OrderedCollection` and `Dictionary` on top of `WObject`.

Evaluating Code

A world is implemented by the `WWorld` class and holds records to object state accessed in it. To allow parallel usage of Worlds in multiple processes the current `WWorld` is stored local to the running process in a process-specific variable [44]. When a piece of code should be evaluated in the scope of a world it is passed to `WWorld#eval:` as a block (cf. Section 3.1.2). Upon invocation of `#eval:` the world stores itself on the process, evaluates the block and restores the previously active world for the active process.

In order to be able to ensure isolation when evaluating code inside a world, each `WWorld` caches object state with copy-on-read semantics. That is, whenever an instance variable or indexed field is accessed for the first time in a world, it will be looked up in the parent and copied to the child as the new working value. During consecutive access the working value is returned and no lookup is needed.

Committing

To ensure consistency upon commit a world keeps two versions for each object attribute or indexed variable accessed in it: The working value representing its current state and the original value at the time it was first accessed in the world. Upon commit, the world performs a *serialization check* [44] and makes sure that the original value and the working value in the parent world did not diverge. In case they diverged the world is no longer up-to-date and the commit operation is aborted with an error.

⁂

This section gave an overview about Worlds, both regarding the underlying concept and the implementation for Squeak/Smalltalk. The following section will describe the principal application of Worlds to make exploratory experiments reversible.

3.2. Aiding Exploration

We propose the usage of Worlds in exploratory experiments that are conducted during a debugging session. It allows us to encapsulate the side effects cased by the program exploration during experiments. Eventually, that makes

backtracking of experiments both regarding program state and execution context possible and allows for the safe re-examination of behavior inside a debugging session.

3.2.1. Experimentation Model

In the following we define the model that enables the safe experimentation during a debugging session[5]. The center of the model is an exploratory experiment e which we define as a tuple $(w_{previous}, c, w_{base}, w_{current})$. The contents of the tuple are formalized as follows: $w_{previous}$ is the active world and c is the execution context at the time the experiment was initiated. w_{base} is the base world of the experiment and an immediate child of the active world at the time the experiment was initiated. Thus it is either a child of the top-level world or the previously active world. $w_{current}$ is an immediate child of w_{base} which is used to capture the side effects during the execution.

Experimental Operations

Six abstract operations in addition to these introduced in section 3.1.2 form the basis to work with exploratory experiments during a debugging session:

$currentWorld() \rightarrow w$ Returns the currently active world w at the time the operation was called. Defaults to the top-level world in case no other world is active.

$create() \rightarrow e$ This operation initiates a new experiment. The experiment $e = (w_{previous}, c, w_{base}, w_{current})$ gets created in a way that $w_{previous} = currentWorld()$ is the currently active world, c is the current execution context, $w_{base} = sprout(w_{previous})$ is a child of the active world and $w_{current} = nil$. The world w_{base} captures the state of the parent experiment at the time the experiment was created.

$activate(e) \rightarrow e$ Activates the experiment so that all follow up interactions with the program under observation are scoped to the experiments' active world $w_{current}$, that is $currentWorld() = w_{current}$. The world is active until the experiment gets discarded, another experiment is activated or the world

[5]Note that the model targets experimentation with single-threaded applications only

change is triggered programmatically (for instance because the program being explored uses the Worlds mechanism itself).

$reset(e) \rightarrow e$ Resets and possibly re-starts the experiment e. Thereby it sets the current execution context to c and recreates the current world as $w_{current} = sprout(w_{base})$.

After resetting an experiment the execution in the debugger continues at the point where the experiment was originally created. Additionally the experiment runs in a new world that is derived from the experiments base world. That discards changes in previous runs of the experiment silently. Deriving $w_{current}$—the actual world of the experiment—from w_{base} makes sure that the experiment will always start with the same prerequisites, i.e. is independent of the parent experiment.

$discard(e) \rightarrow e$ Discards the experiment and restores execution context c and the world $w_{previous}$ to continue in the execution where the experiment was started.

$commit(e) \rightarrow e$ Commits the side effects caused during an experiment when the experiment is completed successfully. Restores the world $w_{previous}$ to continue in the parent experiment at the point this experiment was ended.

Based on these six operations, starting a new experiment can be modeled as a combination of creating the experiment as well as resetting and activating it. Thus it can be defined as $start() = activate(reset(create()))$.

Preliminary Discussion

In the model, exploratory experiments and worlds are tightly connected to each other. An experiment contains a current world that captures the side effects while the experiment is active. To ensure independence of the experiment with regard to the parent, the experiment is associated with a base world, too, from which the side-effect capturing, current world of an experiment is derived. That base world is sprouted once from the world that is active at the time the experiment is created. The fact that an experiment does not base on a possibly existing parent experiment but on the currently active world ensures that applications can use the Worlds-mechanism themselves.

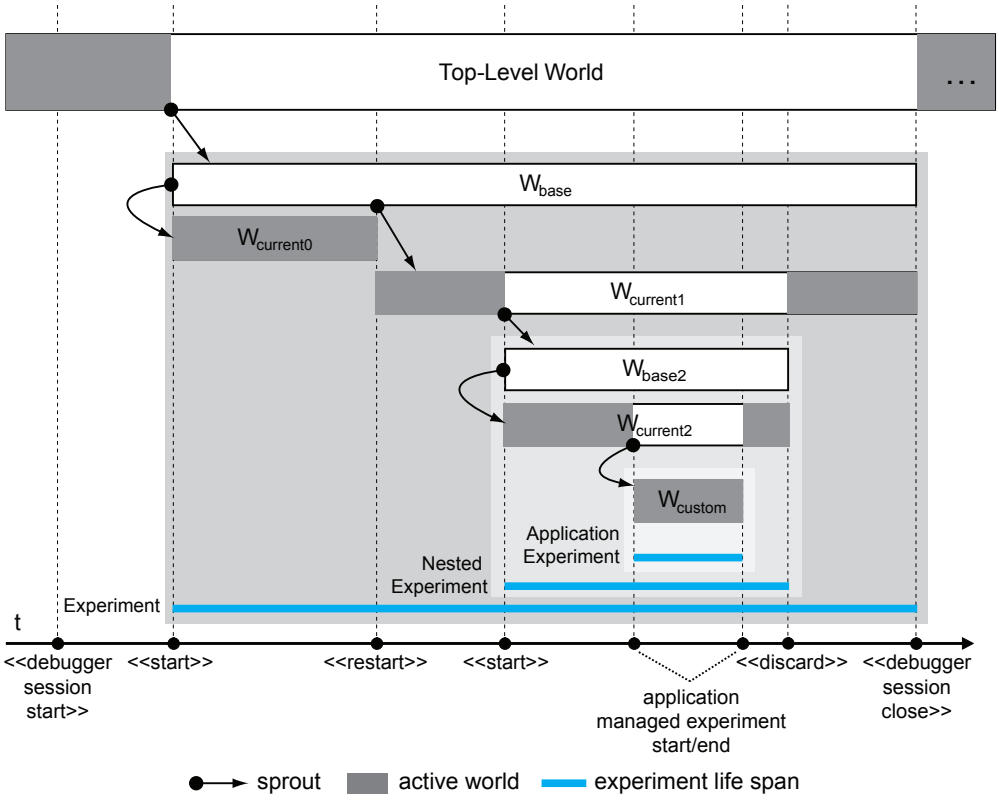

Figure 3.2.: Worlds and experiments during a fictive debugging session

Figure 3.2 depicts the relationship between experiments and active Worlds during a fictive debugging session. It shows two experiments that were conducted during a debugging session and highlights which world was active at each point in time. Marked on the time line are important events that occur during the session. At the beginning of the debugging session no experiment is active. That makes objects scope to the top-level world, an always existing world that represents the global context. At the time a user wants to understand a method in detail, he starts a new experiment (denoted by the first «`start`» event). That conserves the current state of the top-level world in the experiments base world. Additionally, it activates a newly created child of that base world. Some time later, he wants to safely re-examine the method and thus restarts the experiment (cf. «`restart`» event). That replaces the currently active world with a new child of the experiments base world. The world gets temporarily disabled as the user starts a nested experiment (second «`start`»

event) and gets re-activated only when the nested experiment is discarded («discard» event).

Application-local experiments embed naturally in the described model. Whenever an application executes code in the scope of a new world that world is a child of the currently active world, which in term could be the current world of an exploratory experiment. Resetting or discarding the exploratory experiment discards nested application level experiments, too.

3.2.2. Example Scenarios

In the following, we go through a number of examples to show how Worlds for debugging, especially the encapsulation of certain debugging steps in reversible explorations, can benefit particular real-world debugging scenarios.

Understanding Program Behavior

As mentioned previously, understanding a program from source code alone is a time-consuming and complex activity. Therefore, developers often employ debuggers to get a dynamic view on the program. That view helps them to explore relationships between objects and gain other kinds of information that is hard to extract from source code. Our first example depicts, how the debugger can be used to understand a program or better how it can help to find the right place to extend the program with certain features.

During the work on a source-code generation tool for Squeak/Smalltalk, it got apparent that the RBParser—a tool which creates an abstract syntax tree (AST) from a source text of a method—failed to recognize binary methods when they were prefixed with alphanumeric characters. That made it impossible for the parser to recognize source texts such as the one shown in listing 3.7. The question was, how could the parser be best extended to recognize these special characters?

```
Number#__special__/ aNumber

  ^ 'I would have executed a special binary method'
```

Listing 3.7: A piece of Smalltalk source code, unparsable by recent versions of RBParser for Squeak/Smalltalk 4.x (the code was the subject of a automatic selector rewrite operation).

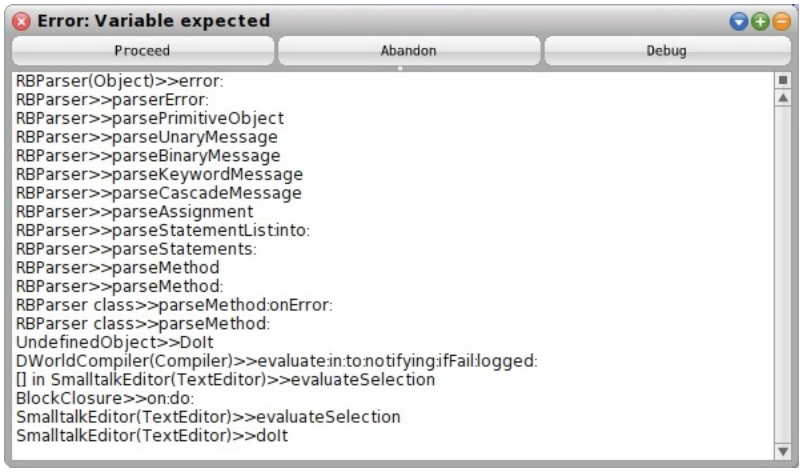

Figure 3.3.: Stack trace of a parse error

A quick look at the stack trace, depicting the origin of a cryptic "variable expected" error, revealed that the entry point for the parser is the class side method `RBParser class#parseMethod:`. That method somehow delegates to the instance side method `RBParser#parseMethod`. Eventually, the parser would fail trying to parse a primitive object in `RBParser#parsePrimitiveObject` (cf. figure 3.3).

Despite debugger support, best efforts and good knowledge about Smalltalk in general, it took roughly 15 minutes to understand the problem and additional ten minutes to locate the position in the parsing infrastructure where the extension could be implemented best. The reasons for that are two-fold.

Some methods consume tokens from the source stream, i.e. they change the internal state of the parser while they execute (cf. listing 3.8, lines 3 and 8). Simply re-executing these methods via stack-rewinding is not possible, as that creates a mismatch between execution context and state. For example, the method `RBParser#parseUnaryMessage` shown in listing 3.8 would, when being re-executed, expect a primitive token n when initially parsing the first source node. The internal state would already offer the next token $n + 1$ and the mismatch messes up all subsequent observations. Often, it is even impossible to compensate that mismatch because the state changes happen undetected to the user. Once again looking at listing 3.8, a developer would not notice the changes in the internal state of the parser unless he steps into the method `#parsePrimitiveObject`, invoked in line three of the listing.

```
parseUnaryMessage
  | node |
  node := self parsePrimitiveObject.
  self addCommentsTo: node.
  [currentToken isLiteralToken
     ifTrue: [self patchLiteralMessage].
  currentToken isIdentifier]
     whileTrue: [node := self parseUnaryMessageWith: node].
  self addCommentsTo: node.
  ^node
```

Listing 3.8: A method of the parser that changes the parser's internal state

The result is that the debugging session has to be frequently re-started in order to safely re-examine the parsing behavior. That also implies that the execution context, e.g. "beginning of method x, when conditions $y_1..y_n$ apply", has to be re-established. Depending on how often method x is called and how easy it is to check the conditions $y_1..y_n$ that can be a tedious and time consuming activity.

Worlds for debugging helps tremendously in the described example. At first, it allows us to examine methods like `RBParser#parseMethod` repeatedly and in a safe and reversible manner, without the need to re-start the debugging session. Furthermore, it makes it possible to structure the session into a series of nested experiments like "understanding class side `#parseMethod:`", "understanding `#parseMethod`", "understanding failure in `#parsePrimitiveObject`" and so forth. Each of the experiments could easily be backtracked whenever new insights lead to the violation of assumptions and thus render the experiments useless.

One of those assumptions in the described scenario is that the parser was the cause of trouble. In fact, it was not the parser that had to be extended but an upstream `RBScanner` which tokenizes the source stream. Later on, the parser starts to work on the already corrupted token stream and eventually recognizes illegal tokens.

Debuggers as Rapid Prototyping Tools

Ressia et al. note that debuggers are often used in test-driven development to identify which parts of the program need to be implemented next [32]. While not all debuggers are good at that activity, the Squeak/Smalltalk debugger arguably is, because it facilitates the on the fly definition of missing methods

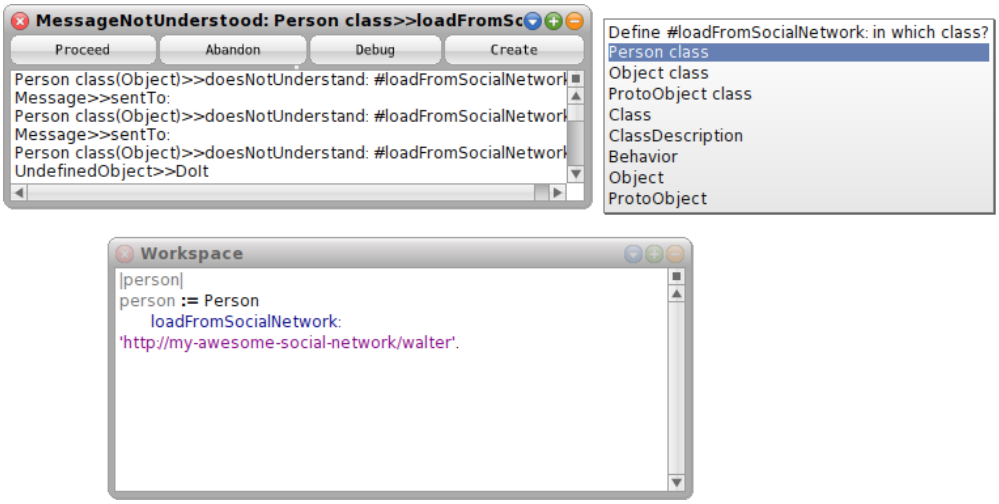

Figure 3.4.: Method refinement in the Squeak/Smalltalk's debugger

and their refinement during a debugging session. That makes it also a viable tool for rapid prototyping.

The goal of a programming session is to extend the class `Person` with a facility to instantiate the model from information publicly available in a social network. A class side method should implement that behavior. Rather than explicitly implementing the method, the user evaluates code that accesses the method. The access of a undefined method triggers a method not understood exception and pops up the debugger. As shown in figure 3.4, the developer can use the tools provided by Squeak/Smalltalk to define the method on the fly and jump right into debugging it. Inside the debugging session he is able to refine the method definition. Thereby he can resort to tools such as the inline definition of new classes and instance variables (cf. figure 3.5). Testing the code comes integrated in the development process, as the method can be executed and thus checked right after its definition was changed.

As we have seen in the previous example, the usage of Worlds for debugging can eliminate the need to restart a debugging session, as long as experiments during the session are prepared accordingly. Consequently, continuous experimentation with a running program works well together with the rapid prototyping capabilities of the Squeak/Smalltalk debugger, because even less time is wasted with restarting debugging sessions. When combining safe experimentation using Worlds with a platforms capability to evolve pro-

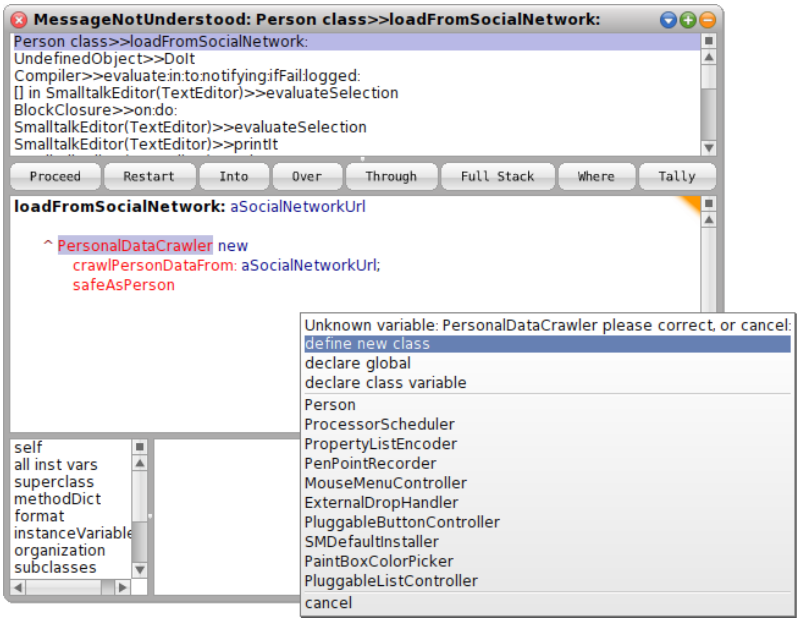

Figure 3.5.: Method definition during rapid prototyping in Squeak/Smalltalk

grams during a debugging session, a debugger becomes an always running tool for the simultaneous development and inspection of run-time behavior.

3.3. Practical Limitations

The previous section presented the principal application of Worlds to make exploratory experiments during a debugging session reversible. In practice the application of the current Worlds implementation for Squeak/Smalltalk to the described scenarios is limited for several reasons.

Worlds for Squeak/Smalltalk builds Worlds support on top of a special class hierarchy. Because of that the ability to scope side effects is limited to a small number of classes (cf. listing 3.9). However, a Worlds implementation applicable to the debugging scenario must allow large-scale experiments on arbitrary objects. That said, it must offer a general purpose Worlds mechanism rather than a specifically bedded version (compare figure 3.6 on page 32).

A general purpose implementation faces a number of issues for which the current Worlds implementation does not have an answers to. At first, there is the question of how to implement a Worlds dispatch—the mechanism that dispatches instance variable access through the currently active world—in a

```
| warray array w1 |

warray := WArray with: #foo. "array containing #foo"
array := Array with: #foo. "a different array containing #foo"

w1 := DWorld current sprout. "world to experiment in"

w1 eval: [
  warray at: 1 put: #bar.
  array at: 1 put: #bar.
].

warray at: 1. "#foo"

"Array is not a world-scoped class"
array at: 1. "#bar"
```

Listing 3.9: Not really safe experiments in the original Worlds for Smalltalk

manner, so that it works for any object. In the current Worlds implementation, state access is always dispatched and thus incurs a performance penalty. For a general purpose implementation, it might be desirable to dispatch state access only when it is actually needed, i.e. when an object is currently accessed in the context of a side effect scoping world.

Second, there is the question of how to implement the Worlds core in the presence of a global Worlds dispatch. A number of classes are required to implement the Worlds mechanism as such, including world-scoped instance variable access, logic for sprouting and committing. That core must not use the Worlds dispatch or else infinite recursions will occur. Warth et al. circumvented the problem by limiting Worlds features to a small amount of classes. Thereby they where able to implement the Worlds core using the other 99 percent of the remaining classes in the Squeak/Smalltalk eco system. A general purpose Worlds for Smalltalk must instrument core classes such as `Array` and `Object`, too (again, refer to figure 3.6). That makes the naive implementation of Worlds using these classes not an opinion.

New challenges such as customizability, e.g. to work with primitively implemented behavior[6] or irreversible operations, arise in the context of a general purpose Worlds. The current Worlds implementation has no answers to both and as a result it does neither employ a safe and generic way to handle primitive methods, nor does it deal with irreversible operations.

[6] Behavior that is implemented in the virtual machine (VM) rather than in Smalltalk code

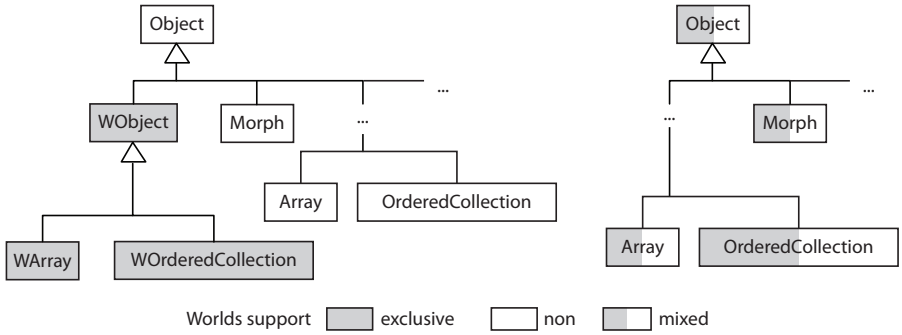

Figure 3.6.: Worlds-support as special feature versus general ability

Last but not least, the current Worlds implementation for Smalltalk does not support a number of scenarios, first and foremost the inspection of in-worlds object state through conventional dynamic views and the safe experimentation with graphical applications.

3.4. Towards a General Purpose Worlds

The limitations of the current Worlds for Smalltalk make it impossible to use it in the proposed scenario. Needed is a general purpose Worlds mechanism that enables the usage of Worlds in large-scale experiments on arbitrary classes (cf. figure 3.6). This section introduces some of the major requirements for such a mechanism.

3.4.1. Generic Worlds Dispatch

One basic feature of a general purpose Worlds is a Worlds dispatch that looks up object state in the context of a world. A few requirements make sure the dispatch seamlessly embeds into the Smalltalk ecosystem and stays compatible with existing applications.

Pluggability The majority of applications should be Worlds-ready out of the box. That means that the Worlds implementation must be *easily pluggable* and should pose *no special requirements* on existing applications (e.g. usage of accessors or adherence to a special naming scheme). Pluggability ensures that Worlds is a generally usable mechanism and limits the complexity which is added to applications in order to employ Worlds.

Customizability While most applications should work in Worlds out of the box, *customizations* to the implementation must be possible to meet special needs of certain applications. These needs comprise (1) safe ways to work with primitive methods inside a world, (2) performance optimizations for state lookup as well as (3) handling of irreversible operations, e.g. file system access.

Transparency In order not to impinge on program understanding and maintainability, the Worlds mechanism must be implemented *transparent to the user*. That means that only the Worlds application programming interface (API) should be exposed in application code while the internal mechanism is hidden. For understanding a program at run-time this means that stepping through it should show similar execution semantics in the presence and absence of a world scoping.

Separation of Concerns To make both, the applications and the Worlds mechanism, easier maintainable a Worlds implementation should aim for separation of Worlds-specific concerns (e.g. state lookup in the scope of a world) and non-worlds concerns (e.g. normal state lookup).

Performance While not being a primary focus of this thesis, performance still plays a key role for a Worlds mechanism. An ideal Worlds mechanism should not impact system performance outside the scope of a nested world at all. At the same time, the performance impact caused by world-scoping should be kept minimal so that the experimenting with running programs can still be done with an acceptable speed.

The requirements mentioned here ensure that the Worlds mechanism is "general applicable and easy to employ". In the following subsections we will explore two scenarios, which are of high importance for a general purpose Worlds: Tool support and working with GUI applications.

3.4.2. Tool Support

Squeak/Smalltalk offers a number of tools that show a run-time view on objects in a running program and thereby allow both inspection of and interaction with these objects. The tools are usually invoked directly from application code or from within an already existing dynamic view. Figure 3.7 shows the

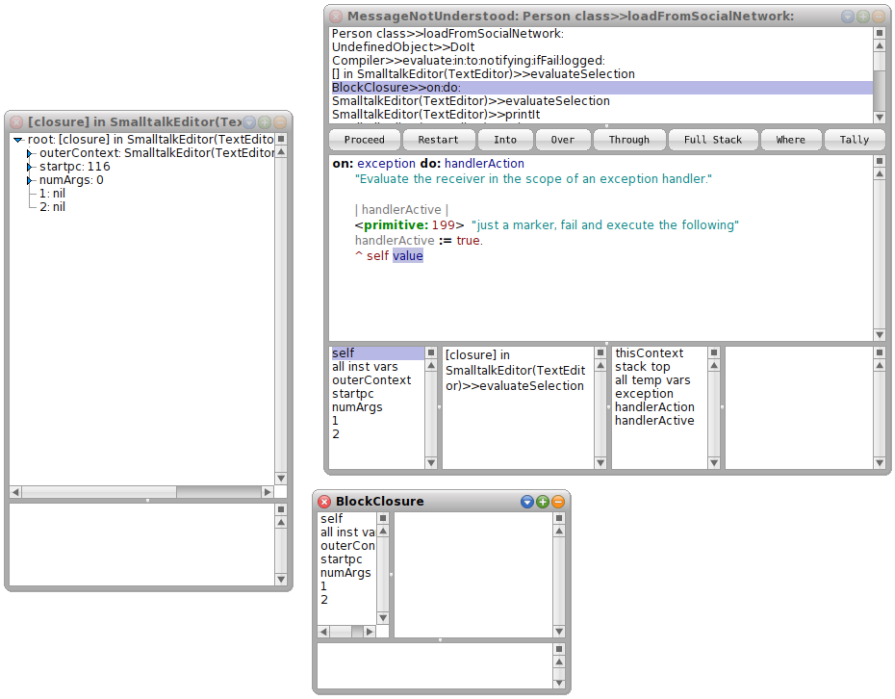

Figure 3.7.: Common tools for run-time program inspection in Squeak/Smalltalk: `Debugger` (top right), `Inspector` (bottom) and `Explorer` (top left).

three most prominent dynamic views in Squeak—debugger, inspector and explorer. All three tools contribute to the understanding of running programs as they display the run-time state of application objects. Furthermore, they allow user-object interaction by evaluating Smalltalk code in the context of displayed objects.

Listing 3.10 shows the usage of these development tools in a Worlds context. `Klaus`—a person we already got to know in section 3.1—is experimented with. In the scope of `w1`, a new world, `klaus`' name is changed. To ensure the name change worked, `klaus` is inspected and the program is halted. Both, the inspector showing `klaus` and the debugger opening up, should not be part of the experiment but rather be regarded as system tools operating outside of experiments. At the same time, they should show objects in a world with their world-specific object state and allow interaction with the objects local to the current world.

These observations lead us to two specific requirements for a general purpose Worlds mechanism for Smalltalk:

```
| klaus w1 |

klaus := Person new.
klaus name: 'Klaus'.

"Child of the current world (to experiment in)"
w1 := DWorld current sprout.

w1 eval: [
  klaus name: 'Achim'.

  "Inspect klaus in the scope of the current world"
  klaus inspect.

  "Halt the program to inspect klaus"
  klaus halt.
].
```

Listing 3.10: Inspecting `klaus` in the scope of a world

(1) The mechanism must allow it to invoke system tools such as the debugger, inspector and explorer from inside an experiment. At the same time it must ensure that these *tools are running outside of the experiment*, i.e. are not world-scoped.

(2) System tools must be able to interact with objects in the scope of the objects world. That is required to (a) visualize the object state and (b) allow world-scoped object interaction.

    In order to allow interaction with system tools and world-scoped objects a general purpose Worlds mechanism must offer a facility to *locally scope interaction with an object to a particular world* so that a world-scoped object can safely be inspected from outside the world it was scoped to.

Fulfilling these two requirements makes it possible to employ the common run-time inspection tools outside experiments. At the same time world-scoped objects can be inspected with their in-worlds state. Attaching the world-scoping directly to an in-worlds object makes it possible to inspect, explore or debug worlds-scoped objects without the need to adapt existing tools. With that in mind we will now examine how Worlds can be used in the context of GUI programs.

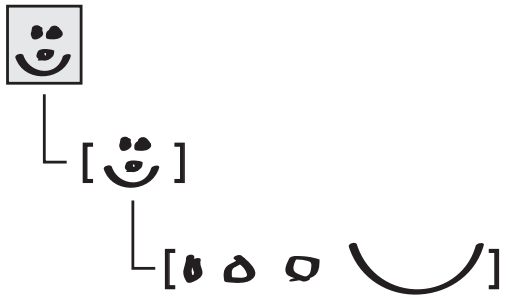

Figure 3.8.: A smiley morph and its child morphs forming a tree structure

3.4.3. Support for GUI Applications

Squeak/Smalltalk is a living system that contains everything from source code, methods and classes over development tools to the actual programs being run. Created applications do not run standalone but rather integrated in the Smalltalk environment [4, 12]. For that reason it is no surprise that graphical applications run tightly integrated into the system, too.

To create graphical applications in Squeak/Smalltalk a user probably relies on *Morphic*, the Squeak user interface framework [26]. Central to Morphic is a *morph* which is the Morphic abstraction for a single graphical element. Each morph has global bounds, a color and a number of extension attributes. Furthermore it has an owner (another morph that contains it) and a number of submorphs. Owner and morph as well as morph and submorphs span a parent-child relationship. That results in a tree structure originating at a morph without owner (cf. figure 3.8). To make morphs interactive they can handle input as well as user interface events such as drag-and-drop, mouse move or key press and react to these events accordingly. Additionally morphs can receive regular stepping impulses to realize periodic recomputations such as the movement of submorphs.

The root morph in the hierarchy of displayed morphs is the *Morphic world*, an instance of `PasteUpMorph`. It is always displayed and exposed to all objects in the system as a global `World` variable. A single user interface process is responsible to continuously *cycle* the Morphic world (see `Project class#spawnNewProcess`). During each cycle, the world will handle captured input events, invoke step methods on interested child morphs and redraw the user interface (cf. `WorldState#doOneCycleNowFor:`).

In order for a morph to be displayed in Squeak/Smalltalk it must be explicitly added to the active morphic world. That happens through `Morph#open`

`InWorld` which internally adds the morph to the worlds' list of submorphs and registers the morphs' desired stepping (if any). The addition of a morph to the Morphic world is persistent until the morph is explicitly removed using `Morph#delete`.

Morph and owner are tightly connected in a bi-directional relationship. In that relationship both sides are free to initiate the communication. A morph for instance notifies the owner whenever its appearance in terms of color, positioning, size or child morph appearance changed. Additionally it communicates with the owner to realize a number of other functions such as deleting itself from the Morphic world. The owner on the other hand communicates with a morph mainly during a cycle. It starts the communication with submorphs to dispatch events, invokes step methods on submorphs and to initiate their redraw during a world cycle. In addition the owner can also communicate with the child out-of-band, e.g. to dispatch deferred change notifications. Special care has to be taken when we want to enable Worlds-scoped behavior in Morphic. Wherefore, we will depict in the following example.

Listing 3.11 depicts the usage of Morphic in the context of Worlds. It shows a piece of code that creates a morph `m`. It adds `m` to the morphic world, thus making it visible to the user. Later, the world `w1` is used to conduct an experiment in which a yellow colored morph `m1` is added to the morphic world. Some time in the future the morphs `m` and `m1` get deleted so that they are no longer displayed.

What is the visual result of that code section? The morph `m` gets added to the morphic world in the global state and thus is visible to the user until it is deleted. The yellow `m1` morph, though, is added in a local experiment. Changes to the experiment are only visible inside the `w1`. As the world is not active during `UI` cycles `m1` remains unrecognized. In fact, the yellow morph is never really added to the morphic world and thus at no point in time displayed to the user.

To actually display `m1` as a yellow colored morph it must be installed into the morphic world as a *permanent local experiment* (cf. figure 3.9). This ensures that the communication of the paste up morph with `m1` is always carried out in the scope of `w1`. At the same time, it must guard the morphic world to ensure that communication with it is carried out in the root scope. Only this way world-scoped morphs can safely add and delete themselves as well as ulterior communicate with a root-scoped owner.

```
| m w1 m1 |

m := Morph new.
m position: 50@50.

"Display the morph in a world"
m openInWorld.

"Obtain a fresh world to experiment in"
w1 := DWorld current sprout.

"Locally experimenting with another morph"
w1 eval: [
  m1 := Morph new.
  m1 position: 20@20.

  m1 color: Color yellow.

  "Only visible in experiment"
  m1 openInWorld.
].

"A long time later... delete the morphs"
m delete.
w1 eval: [
   m1 delete.
].
```

Listing 3.11: Experimenting with a morph

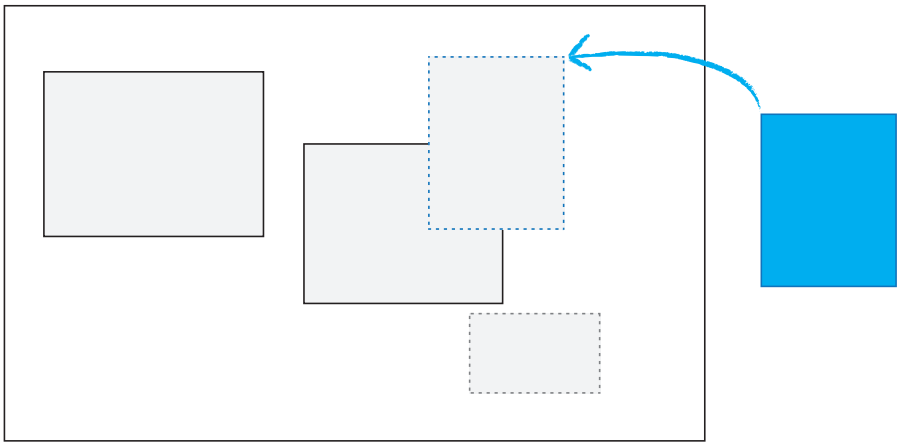

Figure 3.9.: Installing world-scoped morphs into the morphic world to allow morph-local experiments.

That leads us to our next requirements for a Worlds mechanism, that enable it to experiment with morphic programs or, more general, make it safe to interact between in-worlds and normal objects:

(3) In order to make worlds scoped state visible to the user, there must be a way to *persist the worlds scoping of particular objects*, i.e. of morphs, so that it survives multiple processes or the next redraw cycle.

    That allows particular objects to permanently interact with other objects in the scope of their world and thereby expose world-scoped state and behavior.

(4) At the same time, there must be a mechanism to *guard global resources* such as the paste up morph so that interaction with them is always conducted outside of an experiment.

Fulfilling the requirements facilitates the safe propagation of stepping impulses, redraw requests and events from a "normal" parent morph to a possibly world-scoped child morph. Furthermore, it enables the communication between a experimental child and a "normal" parent, e.g. to realize the removal of the child morph from the world.

Looking closely back to our previous requirements it is worth noting that requirements (3) and (4) are—when abstracting Morphic specialties—the more general cases of the requirements (1) and (2) because they allow the explicit scoping of arbitrary objects.

3.4.4. Summing up

In this subsection we will quickly summarize the requirements found for a general purpose Worlds mechanism.

Generic Worlds dispatch

We identified a number of requirements which are important for a generic Worlds dispatch because they ensure that it is easy to employ and to maintain. These are pluggability, customizability, transparency, separation of concerns and performance. In the context of a general applicable Worlds mechanism the requirements assure that the developed mechanism is easy to roll out and understand, adaptable and employable out of the box in a wide range of applications.

Scoping in Execution Context

From the original Worlds implementation we inherit the requirement that Worlds-scoping should be *limited in execution context*. That is, capturing side effects using Worlds always happens process locally inside blocks passed to `World#eval:`. That makes sure that the range of a in-worlds execution is limited and that the concept of Worlds remains easy to grasp for a developer.

Scoping and Non-Scoping in Space

We had to relax scoping in execution context to enable the usage of worlds in conjunction with development tools and morphic applications. Both cases required it to persist local world-scoping on objects in order to be able to interact with the objects from outside their world. We call that *scoping in space* (i.e. object space) as it allows for the application of worlds beyond the boundaries of execution context.

Safely working with global resources such as the morphic world or development tools posed the additional requirement to explicitly exclude these resources from the participation in world-supported experiments. Thus we proposed explicit *non-scoping in space*[7] to safely work with these resources from within world-scoped execution.

⁂

In this chapter we presented Worlds, a language construct which reifies the notion of state and enables the safe experimentation within imperative object-oriented programming languages. Further, we showed how debugging in exploratory experiments can be supported using Worlds by enabling it to safely revert the effects caused by an exploration. We discussed issues of the current implementation of Worlds for Smalltalk which limit its application to debugging and proposed a general purpose Worlds to help out.

Building on the findings the next chapter introduces DWorlds, our implementation of a general applicable Worlds mechanism.

[7] Which can also be seen as the explicit scoping to the top-level world

4. A General Purpose Worlds

The original Worlds implementation for Squeak/Smalltalk is a research prototype which makes it possible to write simple, partially worlds-enabled applications. While it has been extended in some case studies, for instance to implement the undo functions in a graphical editor [44], it is far from being a general purpose language extension. The reason is that the implementation builds support for world-scoping on top of a special class hierarchy. Standard classes, such as `Collection` or `Morph`, have to be re-based onto that class hierarchy to give them the ability to participate in world-scoped executions. Thus, making entire applications Worlds-compatible in Warth et al.'s Worlds for Smalltalk requires major changes in already existing class hierarchies. That is cumbersome and poses the risk of breaking existing behavior.

In this chapter we present DWorlds, an implementation of Worlds that provides Worlds-scoping as a pluggable, general purpose language feature. We highlight the key aspects of the implementation such as the general purpose Worlds dispatch (section 4.1) and the coexistence of normal and in-worlds behavior (section 4.2). The latter allows us to implement the core of the mechanism in the presence of a global Worlds dispatch (section 4.3). Furthermore, we propose both a technique and an algorithm which enable the explicit spatial scoping and non-scoping of objects (section 4.4).

4.1. Worlds Dispatch

To support worlds scoping of arbitrary objects, an implementation must implement a *Worlds dispatch*, that is a mechanism to redirect access to object state through the current world. In the following we present a language level solution to the Worlds dispatch which employs the reflective facilities of Smalltalk to change object state access in a Worlds compatible manner. To start with, however, we will quickly introduce the facilities Smalltalk provides to access state.

4.1.1. Object State in Smalltalk

Basically, Smalltalk facilitates access to object state through primitives as well as a number of built in primitive methods[1]. Thereby it distinguishes between two kinds of object state: Instance variables and indexed fields. Whether an object holds instance variables, indexed fields or both depends on its class definition (cf. listing 4.1).

```
Object subclass: #Person
  instanceVariableNames: 'name age pet'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'DWorlds-Tests'

ArrayedCollection variableSubclass: #Array
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: 'TextConstants'
  category: 'Collections-Sequenceable'
```

Listing 4.1: Definitions for fixed length `Person` and variable length `Array` classes

The majority of classes including `Morph`, `Inspector` and the `Person` class shown in listing 4.1 are *fixed length* classes, which means, their instances hold a fixed number of instance variables. Smalltalk exposes instance variables state via the reflective methods `Object#instVarAt:` and `Object#instVarAt:put:`. Additionally, it employs direct inlining of instance variable reads and writes in methods (see listing A.1 in Appendix A). *Variable length* classes hold a number of indexed fields in addition to whatever instance variables they contain (see definition of `Array` in listing 4.1). Accessing indexed fields is realized through the methods `Object#basicAt:`, `Object#basicAt:put:`, `Object#at:` and `Object#at:put:`, respectively (refer to listing A.2 in Appendix A).

Summing up, six built in state accessors as well as direct instance variable reads and writes make up the state access in Smalltalk.

4.1.2. Dispatching State Access

To implement a Worlds-compatible state lookup for the built state accessors, the methods must be altered to function in a worlds-aware manner. A naive

[1] Primitives offer a way to execute low level operations in Smalltalk directly by the VM rather than evaluating them in a method. To get to know more about primitives and Smalltalk open a Squeak image of your choice and browse `Object class#whatIsAPrimitive`.

approach to realize that, is to create an indirection which dispatches the method invocation to the objects record in the currently active world. Listing 4.2 exemplifies this approach for the method `Object#instanceVarAt:`. As shown in the listing, the original methods `Object#instVarAt:` got renamed to `#originalInstVar At:`. Additionally, the method `#instVarAt:` invokes `#originalInstVarAt:` either on the current world's record for the object or the object itself in case no world is active.

```
Object#originalInstVarAt: index

  <primitive: 73>
  "Access beyond fixed variables."
  ^ self basicAt: index - self class instSize


Object#instVarAt: index
  "Dispatching to the working copy of myself in the current
   world if a current world exists"

  (DWorld current) ifNotNilDo: [ :w |
    ^ (w changeFor: self)
       workingCopy originalInstVarAt: index ].

  ^ self originalInstVarAt: index
```

Listing 4.2: Rewritten `Object#instVarAt:` which dispatches instance variable reads through the currently active world (if any)

Direct instance variable access is not dispatched through methods. Instead, it is inlined by the Smalltalk compiler and delegates directly to the underlying virtual machine. As a result, it cannot be intercepted on the language level. Demanding instance variable access via an indirection (i.e. through accessors) would in theory be a solution. In practice though, it would require a major rewrite existing functionality as direct instance variable access is widely used by most classes in Squeak/Smalltalk[2].

To still be able to dispatch direct instance variable access, the Squeak compiler needs to be extended to rewrite direct instance variable access as method invocations. That is basically what Warth et al. do and what we already described in section 3.1.4. A slight difference, however, exists: A general purpose

[2] The `Morph` class for instance—the core of the Morphic UI framework—defines seven instance variables which are directly accessed 330 times. Numbers taken from a Squeak/Smalltalk 4.1 image

Worlds compiler must effectively transform most classes in the system rather than cherry picking only a few of them. That leads to the fact that the proposed solution would not work when being rolled out on a global scale.

4.2. Coexistence of In-Worlds and Normal Behavior

For our implementation of Worlds we employ[3] an approach presented by Renggli and Nierstrasz in [31]. It enables the coexistence of transformed and untransformed code. Thereby it allows us to execute worlds-specific behavior in the event of an nested-world execution, while using the original behavior if the program execution happens in the top-level world.

4.2.1. The DWorlds Compiler

The core of the implementation is the `DWorldCompiler` compiler, a special compiler which recompiles methods in the Squeak/Smalltalk image to a special nested-world form. We refer to these methods as *in-worlds* methods and the behavior executed by them as *in-worlds* behavior to separate them from the original methods and normal behavior[4].

The compiler produces transformed in-worlds methods, having method names prefixed with a `__dw__` and thus distinguishable from their normal counter parts. All message names used in an in-world method get transformed to their in-world equivalent. That makes sure in-worlds execution is transitive. Additionally direct instance variable access is dispatched to the in-worlds form of `#instVarAt:` and `#instVarAt:put:` (refer to listings 4.3 and 4.4). As a result, in-worlds methods dispatch state access to the currently active world.

In addition to the generation of in-worlds methods the compiler transforms source code inside `DWorld#eval:` blocks of normal methods applying the same transformations as in in-worlds methods. As it can be seen in listing 4.5, it thereby generates the entry points that start the worlds-scoped evaluation.

[3] In fact, our implementation of Worlds is not related to the original Worlds implementation at all. Instead it is in parts based on Stefan Marr's port of the original transactional memory for Smalltalk implementation. Marr's implementation can be found at `http://ss3.gemstone.com/ss/LRSTM.html` while the version by Renggli and Nierstrasz is located at `http://source.lukas-renggli.ch/transactional.html`.

[4] That is not 100 percent correct, as a top-level world is—as previously mentioned—always active. Yet, it makes it easier to distinguish normal execution and execution with world-scoping semantics.

```
meAndMyPet

  ^ name,
  ' and ',
  pet name.
```

Listing 4.3: Source of `#meAndMyPet`

```
__dw__meAndMyPet

  ^ (self __dw__instVarAt: 0) __dw__,
  ' and ' __dw__,
  (self __dw__instVarAt: 1) __dw__name.
```

Listing 4.4: Method `#meAndMyPet` after transformation by the DWorlds compiler

```
meAndMyPetPartlyEvaluateInWorld

  | world myName |
  world := DWorld current sprout.
  myName := name.

  world eval: [
    ^ myName __dw__,
    ' and ' __dw__,
    (self __dw__instVarAt: 1) __dw__name.
  ].
```

Listing 4.5: A normal method which got only statements in a `DWorld#eval:` block transformed by the `DWorldCompiler` compiler

Usually, the Squeak/Smalltalk compiler gets invoked whenever methods are changed or code is evaluated. Upon invocation it delegates these requests to a parser, which parses the source code into a abstract syntax tree (AST) and returns the top-level node of the tree (see figure 4.1). From that tree, a compiled method is generated which can either be executed or installed in the respective slot of a class' method dictionary.

The `DWorldCompiler` utilizes the basic compiler infrastructure provided by Squeak/Smalltalk but extends it with an additional transformation step before the actual AST representation of the source code is returned. To realize this it uses the refactoring browser compilation and transformation tools [33]. In the course of that thesis, these had to be resurrected to work with newer versions of Squeak/Smalltalk. `DWorldTransformer`, a subclass of `RBSemantic-`

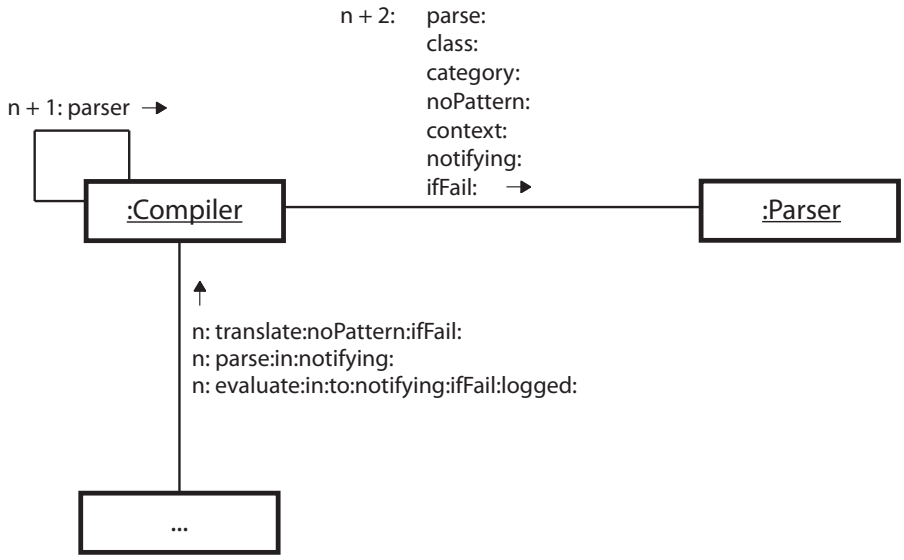

Figure 4.1.: Relationship between parser, compiler and other classes in Squeak/Smalltalk visualized in a UML collaboration diagram

`Annotator` is responsible for the AST transformation. It is a program node visitor which traverses the different elements of the abstract syntax tree. Various methods in the class are responsible to transform particular AST elements into their in-worlds specific form if needed. The method `acceptMethodNode:` shown in listing 4.6, for instance, accepts the root method node to give in-worlds methods to their in-worlds names.

```
acceptMethodNode: aNode
  super acceptMethodNode: aNode.
  self fullTransformation ifTrue: [
    aNode selector: (aNode selector asDWorldsSelector)
  ].
```

Listing 4.6: Method in the `DWorldTransformer` realizing the transformation of in-worlds method names

The transformer operates in two modes, a full and a local transformation mode. In the full transformation mode it transforms all method elements to create an in-worlds method. In the local transformation mode it performs the transformation on the arguments of `DWorld#eval:`, that is world-scoped blocks, only.

Table 4.1.: Annotations to control the generation of in-worlds methods

annotation	description
`<atomicDoNotTransform>`	use this method as an in-worlds method and do not transform it
`<atomic: aSym>`	use the method with the symbol `aSym` in place of this method and transform it
`<atomicUseUntransformed: aSym>`	use the method with the symbol `aSym` in place of this method and perform no transformation

4.2.2. Customizing In-Worlds Behavior

The generation of in-world methods can be customized through source code annotations[5] in their untransformed originals. The annotations serve two purposes: Limiting the scope of in-worlds execution and providing alternative in-worlds behavior. Table 4.1 summarizes the three available annotations.

The annotation `<atomicDoNotTransform>` instructs the compiler to install the original untransformed method in place of the in-worlds method. Thereby it effectively limits the range of the in-world execution. It is primarily used to guard the Worlds core, including its API methods, to make sure that it is never accessed in the scope of a World (cf. listing 4.7).

```
atomicInstVarAt: anInteger
  <atomicDoNotTransform>
  ^ self workingCopyInWorld instVarAt: anInteger
```

Listing 4.7: Usage of the `<atomicDoNotTransform>` annotation to guard the in-worlds instance variable lookup

In contrast, the method `<atomic: aSym>` instructs the compiler to use an alternative implementation for the in-worlds execution of a particular method. Thereby it allows for changing the implementation of a particular method in the scope of a side effect capturing world. This annotation is particularly useful to implement the Worlds dispatch for state accessors such as `#instVarAt:` or `#at:put:`. Rather than weaving Worlds functionality into these methods, the methods get annotated to realize a different behavior in the scope of a nested world. The method `#instVarAt:` for example gets replaced by `#atomicInstVarAt:` in the event of an in-worlds execution (see listing 4.8).

[5] Or pragmas following the Smalltalk nomenclature

```
instVarAt: index
  "Primitive. Answer a fixed variable in an object [...]"

  <primitive: 73>
  <atomic: #atomicInstVarAt:>
  "Access beyond fixed variables."
  ^self basicAt: index - self class instSize
```

Listing 4.8: Usage of the `<atomic:>` to assign an alternative instance variable lookup method for the in-worlds execution

The third annotation is `<atomicUseUntransformed: aSym>`. It is rarely used, because it can be realized as a mixture of a `<atomic:>` annotated original method and a `<atomicDoNotTransform>` annotation on the referenced alternative behavior. The annotation instructs the compiler to use an alternative in-worlds behavior for a given method and tells it not to perform any transformations on the method. One scenario we used the annotation in, was to fix the reflective execution of message sends via `#perform:with:` and friends.

Some programs and frameworks, first and foremost Morphic, make intensive use of reflective message sends (cf. listing 4.9). These would normally break the in-worlds execution, as they would send a normal selector from the scope of a world. To fix that behavior, the message selector passed to `#perform:with:` must be transformed to its in-worlds form before it is actually sent to the method. That was realized using an indirection with the help of `<atomicUseUntransformed: aSym>` (see listings A.3 and A.4 in Appendix A).

```
reflectiveBar

  "Is not correctly translated to the respective
  in-worlds behavior"
  ^ self perform: #bar
```

Listing 4.9: A reflective message send which breaks the in-worlds behavior

Table 4.2 summarizes the three available annotations and their impact on the in-worlds behavior of annotated methods.

Table 4.2.: The impact of annotations on method generation (and execution)

annotation	behavior variation[a]	transform[b]
`<atomicDoNotTransform>`	no	no
`<atomic: aSym>`	yes	yes
`<atomicUseUntransformed: aSym>`	yes	no

[a] behavior variation possible to derive in-worlds behavior [b] transformation done when deriving in-worlds behavior

4.3. Core Implementation

Given the customizable two-layered approach, which clearly separates in-worlds and normal behavior, it is quite straight forward to implement the core of DWorlds. State accessors such as `#instVarAt:` get annotated to be replaced with special in-worlds behavior. That in-worlds behavior delegates state access to the Worlds core classes (cf. listings 4.8 and 4.7). At the same time, in-worlds reflective state accessors and Worlds API methods are guarded using `<atomicDoNotTransform>` annotations. That makes sure that Worlds core functionality gets never executed in the scope of a world.

4.3.1. Architecture

The architecture of DWorlds is shown in figure 4.2. It closely resembles the architecture Renggli and Nierstrasz used to implement transactional memory for Smalltalk [31]. In the center of the implementation is the `DWorld`, the incarnation of a world. It manages world-local side effects and gets referenced in a process local `currentDWorld` variable when the world is currently active. Additionally it has a parent world unless it is the top-level world.

Operation wise it implements the Worlds API we described in section 3.1.2. Thereby, it offers facilities to create a new child world via `#sprout`, merge the changes into the parent world via `#commit` and evaluate code scoped to it using `#eval:`. Furthermore, it offers a class side method which can always be used to obtain the process local current world. A number of other methods manage the world-local state diff to the parent scope (either another experimental world or the top-level world). They can be employed both to query the in-worlds state of a particular object and to register custom changes.

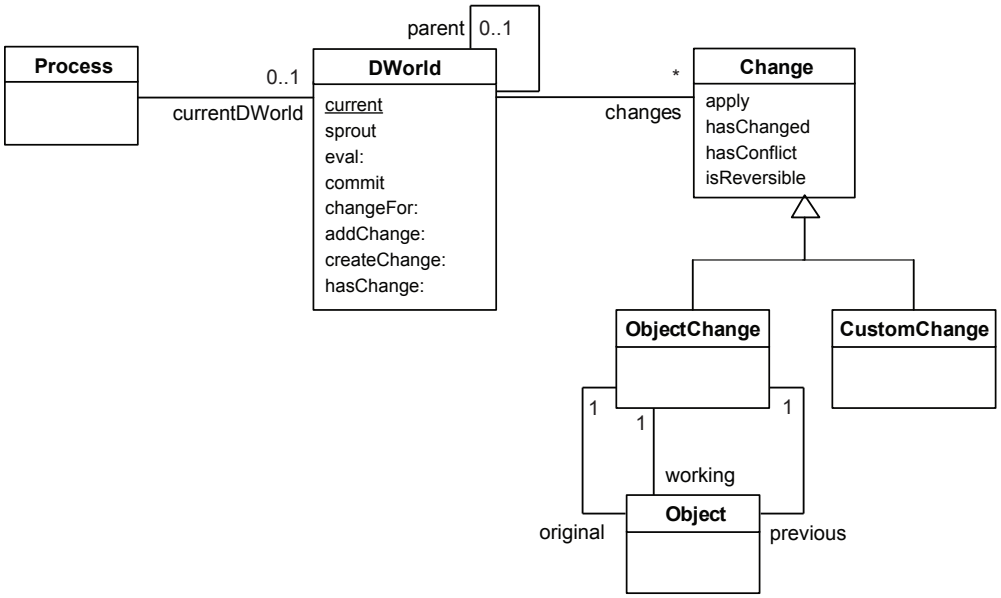

Figure 4.2.: The architecture of DWorlds visualized in a UML class diagram

4.3.2. Change Model

The change model distinguishes between two kinds of changes: Object changes and custom changes. Object changes are represented as instances of the `ObjectChange` class and incarnate world-local changes in particular Smalltalk objects. They hold references to the original state of an object, the world-local working copy of the object and the object at the time the change was created (see figure 4.2). Using these three different objects an object change implementation is able to ensure the consistency properties Worlds longs for (cf. section 3.1.2). It is worth noting that DWorlds does not perform any kind of object-local caching as the original Worlds implementation does. Instead the world-local state of an object must and is always be queried through the active world. The interaction needed to realize the in-worlds lookup of a instance variable value is shown in figure 4.3.

Custom changes allow it to capture side effects that lie outside the Smalltalk image[6]. Operations on the external file system for instance are often irreversible and can be implemented as a `CustomChange`. During the commit of a world, custom changes are handled in a different way than object changes. Rather than committing the change and causing the destructive operation, the

[6]Something, the original Worlds implementation is unable to deal with.

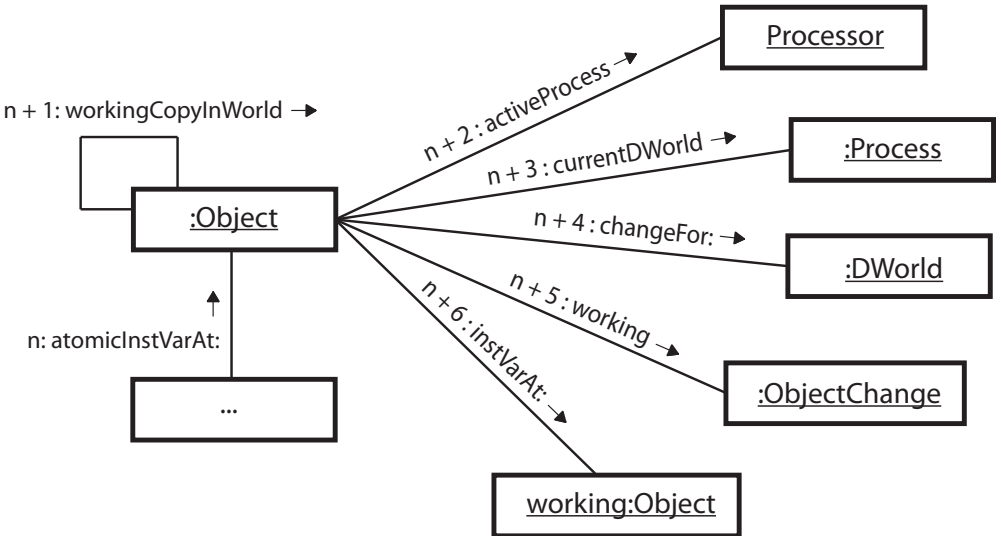

Figure 4.3.: UML collaboration diagram showing the interaction of different DWorlds components to realize the in-worlds lookup of an instance variable

change is carried over to the parent world and eventually committed when the top-most non-top-level world commits.

4.4. Spatial Scoping

The previous chapter emphasized the need to perform spatial scoping of experiments in addition to the usual execution context based scoping in `DWorld#eval:` blocks. We identified two main use cases for spatial scoping: Using Worlds in the context of Morphic and—more importantly—debugging and auxiliary tool support demand the ability to persist the world-scoping of particular objects. Only this way it was possible to safely interact with them from unscoped objects such as development tools or the morphic world. In the context of Morphic we highlighted that particular global resources exist which must be explicitly not-scoped in order to safely use them from inside a world-scoped execution.

The Worlds mechanism we presented in the previous sections is able to cut off in-worlds execution through source code annotations. Thereby it allows in-world scoping in space by explicitly unscoping particular methods and eventually classes such as the class `DWorld`. However, it fails to satisfy more special case scenarios such as explicit per-object scoping and non-scoping.

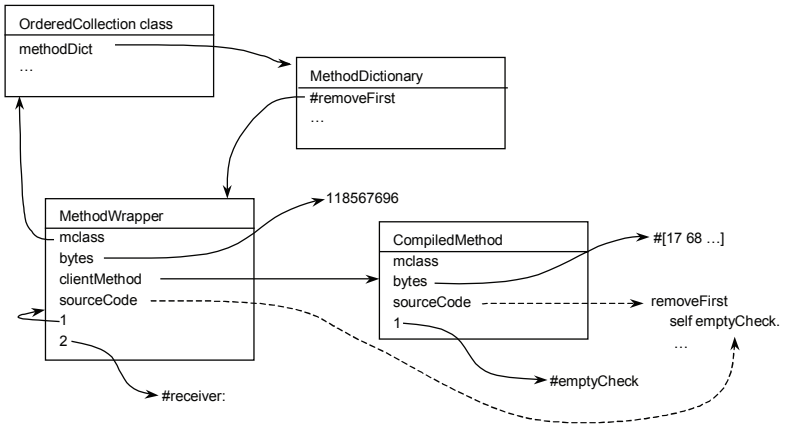

Figure 4.4.: Method wrapper installed on the `#removeFirst` method of the class `OrderedCollection`. Taken from [6].

In this section we will show how such functionality can be realized using method wrappers [6], a technique first introduced by Brant et al. Further, we present an algorithm which is employed in method wrappers to realize the safe interaction between diversly scoped objects.

4.4.1. Method Wrappers to the Rescue

Method wrappers [6] wrap the actual method in the method dictionary of a class. Thereby they allow it to execute code before, after or in place of the original method. This works because methods, classes and method dictionaries are plain objects in Squeak/Smalltalk.

When installed into the respective slot in a method dictionary a method wrapper replaces the original method. At the same time, it holds a reference to the original method, which is often called `clientMethod`. Looking up the method in the method dictionary now returns the wrapper, which can—but is not obliged to—forward the method invocation to the original method (compare to figure 4.4 visualizing the relation).

Method wrappers have seen a wide range of employment, from instrumentation of test cases [6] to the implementation of aspect-oriented [14] and context-oriented [24] language features. In our case they prove useful as they allow us to instrument objects without the need to change an object's identity or destructively modify (e.g. transform) its original behavior.

4.4.2. Reconstituting Explicit Scoping

Based on the scoping information provided by a registry, we employ method wrappers to reconstitute the scoping for explicitly scoped objects. Upon a scope change, we additionally scope method arguments and method return values explicitly to ensure clear boundaries between diversely scoped objects.

Three properties are crucial for the underlying scope reconstitution algorithm. The *contextual scope* is the process local world active for a particular execution. The *current local scope* is the scope from which an object invokes methods on itself or other objects. It is either the global scope in case a normal method is accessed or the contextual scope at the time the invocation happens in case the invocation targets an in-worlds method. Finally the *explicit scope* of an object is the world it was pinned to and in which it should be evaluated.

Based on the three properties the basic algorithm is defined as follows:

#0 Let o be the current object, `args` the method arguments, c the current local scope, *e* the explicit scope of o and m the original method.

#1 If *e* is not set or *e* equals c invoke wrapped method and return the method result (perform a quick return to skip the algorithm when no scoping is required).

#2 Explicitly scope `args` to c unless they are already explicitly scoped.

#3 Reconstitute the explicit scope *e*.

#4 Invoke m in the scope of *e*[7]. Let r be the result of the method invocation.

#5 Explicitly scope r to *e* unless it is already scoped.

#6 Reconstitute the previous scoping c.

#7 Return r.

A few things are noteworthy about the algorithm. The algorithm activates only if there is a scope mismatch between the current scope and the explicitly specified scope (step #1). Before invoking the method, it locally establishes the explicitly defined scope (step #3). Later, it reverts the scope change after the behavior was executed (step #6).

Furthermore, the algorithm fixes the scope of method arguments and the return value by explicitly scoping them to the scope they originated from

[7]That might require to switch from the in-worlds version of the method to the normal version or vice versa.

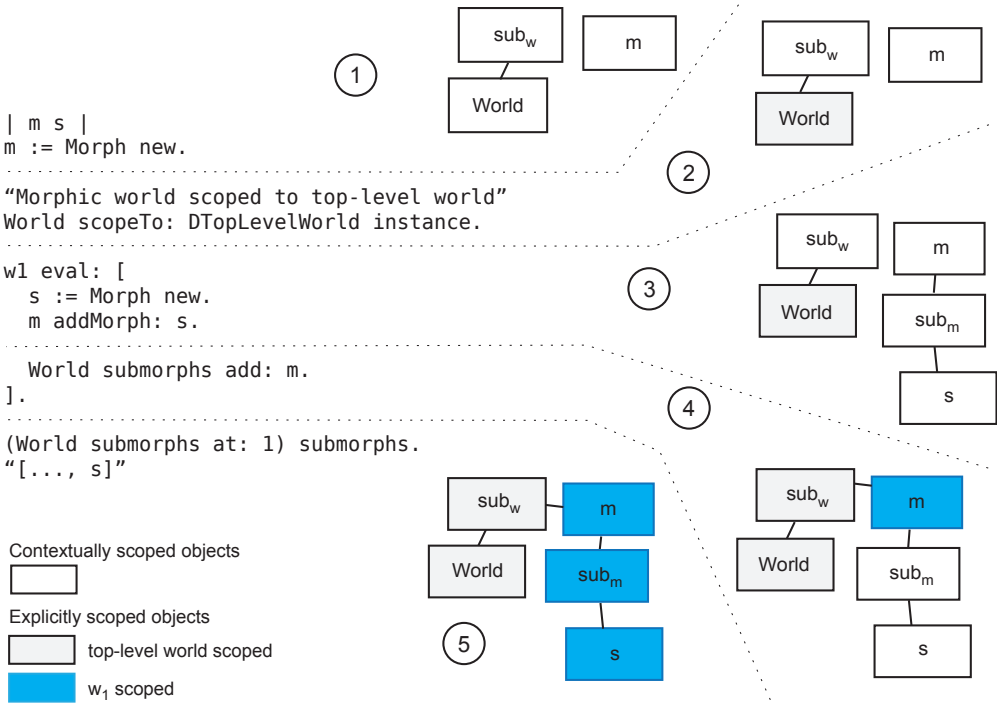

Figure 4.5.: Explicit scoping in action

(steps #2 and #5). Thereby it guarantees that scopes in object-graphs form a transitive closures with explicitly scoped elements representing the boundaries of that closure. That in term ensures, that there are no surprises when diversely scoped objects interact with each other.

Finally, the algorithm distinguishes between two kinds of objects: Explicitly scoped and contextually scoped ones. An explicitly scoped object got its scope fixed and interaction with the object will always happen in the explicitly assigned scope. Contextually scoped objects are objects which not yet got explicitly scoped. They are scoped according to the currently active world or none if no world is active. That said, local world-scoped execution is still possible and will work as in the original Worlds implementation. Given, of course, that no objects participating in a world-scoped execution are explicitly scoped in which case the execution semantics differ slightly.

Figure 4.5 shows the algorithm in action. It visualizes the different steps throughout the addition of a morph to a morphic world and highlights the relation between objects and their scoping. The initial piece of the code ①

sets up a morph `m`. Next to the morph, it depicts the other important already existing entities, `World`, the morphic world and `subw`, its list of submorphs. ② explicitly binds `World` to the root scope—the top-level world—and thereby makes it a global object. That is something which is not done, usually. Rather, a user may assume that the morphic world is already a globally scoped resource. Steps ③ and ④ execute in the scope of a world, `w1`. In step ③, a new morph `s` is created which gets added to `m`'s list of submorphs. Up to that point nothing spectacular took place. Both, the list of submorphs `subm` and `s`, are introduced contextually scoped so that the addition of `s` to `m` is not reflected outside `w1`. In step ④, however, two things essentially happen. First, the explicitly scoped `World` is accessed to return its list of submorphs, `subw`, in the scope of `w1`. As a result it is explicitly pinned to the top-level world to safely be able to work with it from within `w1`. Second, `m` is added to the list and thereby passed from `w1` to the top-level scope attached to `subw`. To safely work with `m` from within `subw` `m` gets explicitly scoped to `w1` so that it retains its special `w1` semantics. The whole operation has only one goal, which is allowing the safe bi-directional interaction between the top-level scoped `World` and the morph `m` and its submorphs. As soon as `m` exposes submorphs to `World`, they get explicitly scoped so that their `w1`-scoping is retained ⑤. Same applied when communication was initiated from the opposite direction[8].

Spatial scoping is meant to be something which is rarely used and which should—in most cases—be transparent to the user[9]. However, wherever it *is* used, the presented algorithm facilitates the safe interaction of diversely scoped objects.

4.4.3. Re-enabling Local Experiments

Unfortunately, spatial scoping using the presented algorithm jeopardizes the ability perform local in-worlds execution. The reason being, that the algorithm gives explicit scoping to a particular worlds precedence over the presence of a contextually active world. This leads to the fact that local experiments cannot safely be performed as soon as an explicitly world-bound object participates in it (listing 4.10 depicts that problem).

[8] Figure A.1 in Appendix A depicts the more complex bi-directional communication between a morph and a globally scoped morphic world. It showcases both, the interaction during the invocation of `Morph#openInWorld` and during a morphic world cycle.

[9] In fact, the way spatial scoping is used is not settled. Consequently, this thesis does not imply any particular use of it. What it does, though, is offering a mechanism which enables spatial scoping as such and an algorithm propagating that scoping to connected objects.

```
| t w2 |

"Assume t to be explicitly scoped to the experiment w1"
t := Person new.
t pet: nil.

w2 := w1 sprout.
w2 eval: [
  t pet: Pet new.
].

t pet. "a Pet, explicitly scoped to w1 rather than nil"
```

Listing 4.10: Local experiments broken through the introduction of the scope reconstitution algorithm

To re-enable the safe execution of local experiments execution context based world-scoping must have precedence over world-scoping in object space. We realize this by re-defining the explicit scope e for local experiments and retaining the contextual scope through the invocation of global resources. To start with we extend the original rescoping algorithm with an additional step before the quick return (step #1):

#0-1 Let c_{ctx} be the contextual scope.

#0-2 If e is set and e and c_{ctx} are both local experiments refine e to c_{ctx}.

Additionally we change step #3 of the algorithm to keep the contextual scoping when global resources are invoked from within an experiment:

#3 If e is an experimental scope reconstitute the scope e.

The extension of step #3 conserves the currently active scope when global objects are invoked from within experiments. This is possible because the in-worlds behavior can simply be cut off by switching the normal execution behavior. As a result we execute the normal object behavior while a world remains contextually active[10]. Whenever the normal behavior initiates the communication with a world-scoped object we can reconstitute the contextually active world rather than the world the object was explicitly scoped to. Thereby we allow the world-scoped object to participate in the currently active experiment rather than the one it was originally pinned to.

[10]That is what the Worlds core does, too

Table 4.3.: Scope variations and effective local scope transitions as performed by the final rescoping algorithm

source scopes		target scopes			effective local transition
local[a]	context[b]	explicit[c]	local[a]	context[b]	
global	global	none	global	global	global → global
w_1	w_1	none	w_1	w_1	$w_1 \rightarrow w_1$
global	w_1	w_1	w_1	w_1	global → w_1
global	w_2	w_1	w_2	w_2	global → w_2
w_1	w_1	global	global	w_1	$w_1 \rightarrow$ global
w_2	w_2	w_1	w_2	w_2	$w_2 \rightarrow w_2$

[a] local scope in which the object (caller or receiver) operates in
[b] contextual scope active (e.g. scope active in process local variable)
[c] explicit scope assigned to an object

This extension gives explicit scoping of objects two flavors: Objects explicitly scoped to the top-level world are always scoped to it. Entities explicitly scoped to a particular nested world get scoped to the world when no other world is contextually active. In the presence of a contextually active world, however, they get scoped to that world instead.

Section A.3 in appendix A shows a complete picture of the final scope reconstitution algorithm along with an implementation of the algorithm for Smalltalk (cf. listing A.5). Furthermore, table 4.3 summarizes the specialties of the algorithm regarding scoping and scope transitions. Finally, listing A.6 in appendix A showcases the usage of explicit scoping in Smalltalk code for a complex example.

⁂

In this chapter we introduced DWorlds, a general purpose Worlds mechanism. It comprised two parts: A Worlds core implementation building on the co-existence of normal and in-worlds behavior and a scope reconstitution mechanism which employs method wrappers to enable spatial scoping.

5. Evaluation and Discussion

In this chapter we discuss DWorlds and its application to debugging. Section 5.1 evaluates the applicability of DWorlds as a general purpose Worlds. Following up, section 5.2 presents and evaluates the dwdbg, a debugger extension that employs DWorlds to allow the exploration of run-time behavior in reversible experiments. Based on the findings section 5.3 depicts open topics and gives directions for future work on the topic.

5.1. DWorlds as a General Purpose Mechanism

In the previous chapter we presented DWorlds, the language level implementation of a generic Worlds for Smalltalk. In this section we evaluate the provided Worlds mechanism with regards to the requirements identified in section 3.4.

5.1.1. Generic Worlds Dispatch

The Worlds dispatch of a generic Worlds has to fulfill a number of basic requirements which we identified in section 3.4.1 as pluggability, customizability, transparency, separation of concerns and performance. In the following paragraphs we evaluate DWorlds with regards to these requirements.

Pluggability is given as applications do not usually need to be adapted to use the mechanisms. Throughout the implementation and testing of DWorlds only 56 methods in the whole Squeak/Smalltalk class system had to be changed in order to enable major scenarios such as opening an omnibrowser window inside an experimental world and interacting with it[1]. The changes were of generic nature and—because of that—no particular application-specific adaptions had to be made[2]. Furthermore, each individual change was limited

[1] We reference that case as Morphic is an interaction and collaboration intensive library. In addition the omnibrowser is regarded as one of the most complex programs in the Squeak/Smalltalk system.

[2] For example, classes belonging to the Morphic framework had not to be touched at all

to annotating the particular method and optionally re-implementing its in-worlds behavior. Additional techniques such as the on the fly generation of in-worlds methods or installation of method wrappers allow applications to use the DWorlds functionality without any kind of up-front setup.

The ability to define custom in-worlds behavior through the use of annotations gives the mechanism great customizability. For instance, it made it possible to fix reflective method sends for in-worlds behavior or adapt primitively implemented methods such as `Array#replaceFrom:to:with:startingAt:` for the use in experiments. Furthermore, it makes it easy to customize the worlds core, e.g. by implementing object-local caching of Worlds state for subsets of the class hierarchy[3].

Despite the fact that in-worlds behavior is transformed, the actual core mechanism is transparent to the user. That is possible because the transformation is done on byte-code level, rather than in source code. Methods in Smalltalk hold a mapping between source- and byte-code which is used to guide the user through a method, e.g. when debugging. The compilation of in-worlds behavior retains the mapping between the original source code and the transformed byte-codes. Thereby it facilitates stepping through in-worlds methods while showing the original source code. The scope reconstitution mechanism is supposed to be transparent to user, development tools and run-time environment.

Because the implementation cleanly separates the definition of in-worlds and normal behavior, the separation of concerns is good as much as the DWorlds core mechanism is concerned. That also holds concerning the instrumentation needed to employ the mechanism as only methods that actually use Worlds need to be touched in order to enable it. At the same time Worlds code is invoked on a per-object basis and only when an in-worlds execution is actually active.

Things look differently as far as the scope reconstitution mechanism is concerned. To allow explicit scoping wide ranges of classes and their respective methods have to be instrumented using method wrappers. As a matter of fact, the scope reconstitution mechanism can only guarantee safe experiments when all methods of explicitly scoped objects are instrumented. As method wrappers decorate methods on a *per-class* basis, the instrumentation spills over all instances of a given object independent of whether a particular instance requires it.

[3] A feature implemented in the original Worlds for Smalltalk implementation [44].

Similar to the transactional memory for Smalltalk implementation by Renggli and Nierstrasz, the Worlds core mechanism slows down the performance of state access by the factor 20[4]. At the same time, however, it does not impact the system performance when Worlds is inactive.

Again, things differ as far as the scope reconstitution mechanism is concerned. As previously mentioned, the mechanism is based on method wrappers which work on a per-class basis. Thus it is rolled out class wise for each explicitly scoped entity in order to allow safe scope reconstitution. As a result the mechanism causes a run-time overhead on instrumented classes independent of whether worlds-scoping is used or not.

A small benchmark of the method wrapper technique for Squeak 4.1 (cf. listing A.7) was conducted on a developer machine with 8 GByte RAM. The execution times were 24ms (uninstrumented) and 129ms (instrumented) which makes up for a slowdown of factor six without any of the participating objects actually being scoped. Another benchmark which used re-scoping between objects was run in 276ms (uninstrumented) and 27508ms (instrumented). That accounts for a slowdown of factor 100 when explicit scoping is used heavily (cf. listing A.8).

5.1.2. Tool Support

Given the explicit spatial scoping capabilities of DWorlds, tool support for the inspection of world-scoped objects can be easily achieved. System tools can safely inspect worlds-local objects, when these have been explicitly scoped to the currently active world. When and how the explicit scoping is performed is generally tool dependent. For use cases like inline `aParticularObject inspect` statements in the code, the integration for Squeak/Smalltalk can look like shown in listings 5.1 and 5.2, respectively.

The implementation has the drawback that the explicit scoping of an object has to be reverted in order to be able to inspect it in other worlds or the global scope. Whenever the interaction between tool and object is one-directional, proxies (object wrappers in Smalltalk [6]) can solve that problem by retaining an proxy-local world scope. However, they have the limitation that they cannot proxy the special method `class` and thus fail to proxy the object to class mapping. As a result dynamic views have to be adapted to work around that issue. Independant of the technique being used, the internally applied

[4]Refer to [31] for a detailed analysis of the performance impact

algorithm must remain as depicted in section 4.4 in order to allow for safe spacial scoping.

```
inspect: anObject
  "Open an inspector on the given object..."

  <atomic:#inspectInWorld:>

  self default ifNil:
    [^self inform: 'Cannot inspect - no ToolSet present'].
  ^self default inspect: anObject
```

Listing 5.1: `ToolSet class#inspect:` method annotated to use a special in-worlds behavior

```
inspectInWorld: anObject
  "Open an inspector on the given object..."

  <atomicDoNotTransform>

  "Scope object to current active world"
  anObject scopeTo: (DWorld current).

  ^ self inspect: anObject
```

Listing 5.2: In-worlds implementation of `ToolSet class#inspect:` which explicitly scopes a given object before it gets inspected

5.1.3. Experimenting with Morphic

One of the drivers for spatial scoping in worlds was the application of Worlds in the context of GUI applications. Persisting the world scoping only makes it possible to experiment with these applications because the world scoping would otherwise not survive the next redraw cycle.

Given the explicit scoping facilities it is possible to show morphs in the scope of a world (cf. listing 5.3). If the parent morph (in the shown example the world) is globally scoped, the added morph gets explicitly scoped to the given experiment when it is added to the world. This way it retains the scope during all interactions carried out between morph and its owner. Figure A.1 in appendix A visualizes the effect of re-scoping during various standard interactions between a experiment-bound morph and the globally scoped

morphic world. It shows the bi-directional multi-message interaction needed to add the morph to the global morphic world. That interaction—in fact the first message sent to the world—results in the explicitly scoping of the morph to the experiment. Later, communication with the morph initiated by the morphic world during a world cycle properly reconstitutes the experimental context so that safe working with morphs in experiments is made possible.

```
| morph w1 |
"The morphic world"
World

"Permanently pin it to the global scope"
World markGlobal.

"Create a new experimental world"
w1 := DWorld current sprout.

w1 eval: [
  morph := SimpleButtonMorph new.
  morph openInWorld.
].
```

Listing 5.3: Displaying a morph in the scope of a world

5.1.4. Limitations

DWorlds has a few limitations that revolve around spatial scoping and the application of method wrappers.

Method wrappers instrument methods on a per-class basis rather than on a per-object basis. Because of that, they cannot be used to instrument methods of classes used by the Worlds core implementation[5] as accessing an instrumented method from within the method wrapper causes an endless recursion. Other approaches, such as proxies or object wrappers, were proposed in literature [6] which allow to wrap behavior generically and on a per-object basis[6]. These approaches, however, fail to abide the object class contract because they can neither wrap nor intercept calls to special methods such as `class`. As a result, querying the class of a wrapped object through an object wrapper returns the wrapper class rather than the class of the wrapped object.

[5] These classes include, among others, `Array`, `Dictionary`, `OrderedCollection`, `WriteStream` and `Object`

[6] For instance using the `become:` mechanism in Smalltalk

Figure 5.1.: dwdbg in the button bar of the Squeak/Smalltalk debugger (highlighted in red)

For reasons yet to be investigated, current distributions of Squeak/Smalltalk have a number of issues with method wrappers which limit their large-scale employment. Squeak 4.1 for instance would sporadically dead lock when method wrappers are widely used. Support for method wrappers in current versions of Pharo/Smalltalk—originally a branch of Squeak—is severely broken. To exemplify that, Pharo 1.3 would neither recognize wrapped methods nor their original implementation in a number of situations, e.g. inside the execution performed by the morphic UI cycle.

In the context of the DWorlds prototype for Squeak 4.1 the problems could partially mitigated by excluding a number of methods from being instrumented. These include methods responsible for event processing on `Morph` and all methods on `Object`. However, even with these adjustments made the system would rarely deadlock or fail with primitive errors when method wrappers are widely enrolled[7].

5.2. Reversible Experiments Using DWorlds

Over a long detour via "the generic Worlds mechanism", we finally arrived back at our original topic: Debugging. In this section we take the chance and present a prototype of a Worlds-enabled debugger, the DWorlds experimentation debugger (dwdbg). It implements the experimentation model introduced in section 3.2 and thereby allows us to safely comprehend run-time behavior in reversible explorations.

5.2.1. The dwdbg Debugger Extension

The Worlds-enabled debugger is a extension of the class `Debugger`, the standard Squeak/Smalltalk debugger. It introduces a new button to access the experimental functionality (shown in figure 5.1). Upon click, the button exposes a context menu which can be used to control the current experiment or start a new one (cf. figure 5.2). Menu entries in the experiment context menu

[7] E.g. on morphic applications

Figure 5.2.: Context menus exposed by the dwdbg debugger extension: During an active experiment, no active experiment and when no world is active (from left to right)

link to the debugger which implements the respective experiment operations as instance side methods.

The extension maintains a stack of started experiments which is stored as a process local variable and kept as long as the debugged program is running. The currently running experiment is the stack top. Each experiment is an instance of `DExperiment`. `DExperiment` and its operations closely reassemble the experimentation model described in section 3.2.1. Each experiment resorts to the debugger to perform the execution context reset (the method `Debugger#restart` serves as a basis for that). Figure 5.3 summarizes the main components that make up the dwdbg.

The debugger implementation incorporates two additions to the earlier introduced experimentation model. Foremost it deals with explicitly scoped objects to let them safely participate in exploratory experiments. Whenever a new experiment is started or an existing experiment is reset, it rebases all objects explicitly scoped to the previously active world to the newly activated experimental world. Second, the debugger implementation introduces robust semantics for the scoping of explorations in local and global experiments.

5.2.2. Local and Global Explorations

As already introduced in section 2.3.2 there is the distinction between local and global explorations. Most experimental explorations are *local*, that is they are tied to the execution context they are started in. Local experiments have the sole purpose of understanding a particular method by observing the behavior triggered by executing it. Thus they are finished as soon as the actual method returns which can happen either because the user steps out of it or because the program is resumed and proceeds beyond the method boundaries. In

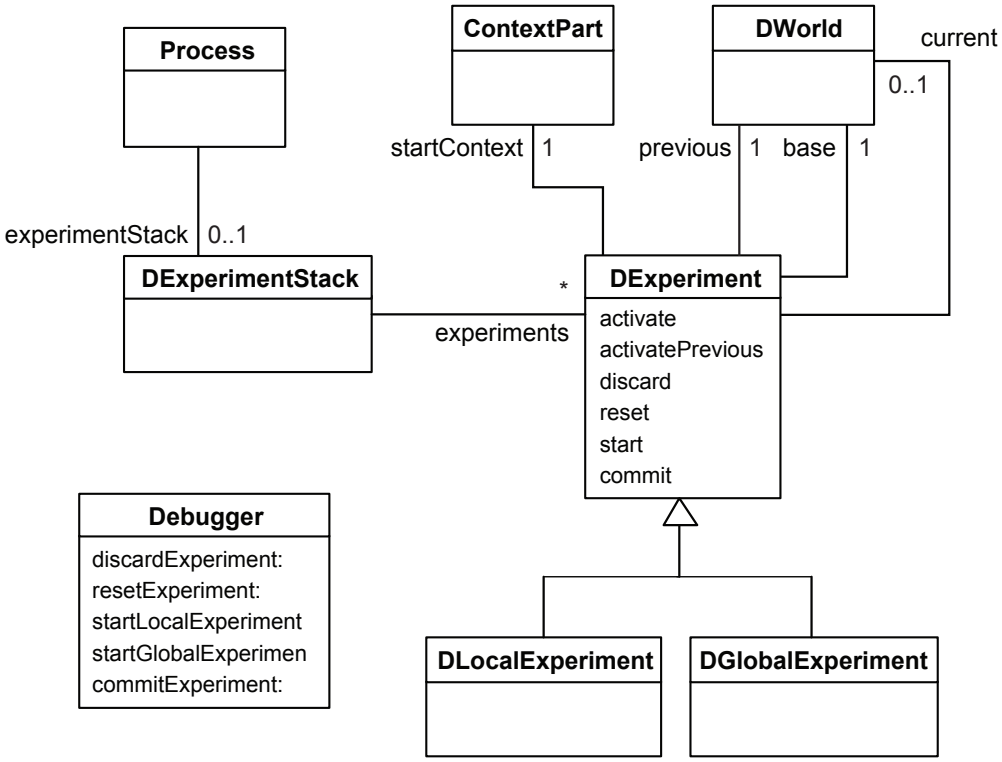

Figure 5.3.: The dwdbg implementation of the experimentation model

any case local experiments must be properly cleaned up when the execution context in which they were created is destroyed.

Some experiments are inherently not local as they are tied to and based on a particular application state only. We called these experiments *global*. Good examples of global experiments can be found in the area of GUI application comprehension. State changes in a GUI application can be triggered both by user interactions or internal stepping mechanisms. Often it is desired to observe such an application starting from a particular well-defined point in time to understand the impact of upcoming user interactions or changes during periodic stepping (cf. section 2.3.2). To be able to support global experiments, the Worlds-enabled debugger must allow it to start and reset them in a well-defined manner.

Same as local experiments, global experiments are started from within a particular execution context in a halted application. The difference, however, is that the experiment survives the destruction of its start context which makes

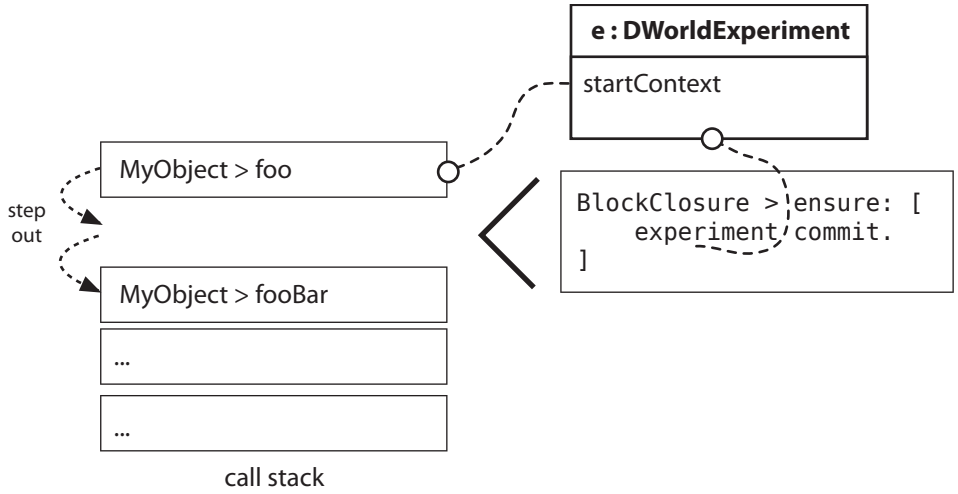

Figure 5.4.: Enabling locality of experiments

it possible to undo its side effects at any later point during the program execution.

Implementation

To implement local experiments `dwdbg` manipulates the call stack upon the start of a local experiment. It introduces an additional context around the start context of the experiment. Via that context the changes captured in a local experiment get commited as soon as the user steps out of the context the experiment is based on (see figure 5.4).

Global experiments are permitted outside the scope of their starting context. As a result, they cannot be simply reset to their starting context as it may not be on the call stack anymore. Still, a developer might want to be able to roll back the execution to a well-defined context on the call stack when resetting or discarding a global experiment. That context is often similar to the one the experiment was started in (compare figure 5.5). The `dwdbg` gives users all choices by letting them decide freely to which context on the call stack he wants to unwind a global experiment to. It supports that procedure by proposing contexts that are similar to the context in which a global experiment was created (cf. figure 5.6).

In contrast to local experiments, globally defined experiments cannot be discarded automatically. Instead the user has to manually discard or—as desired—commit them.

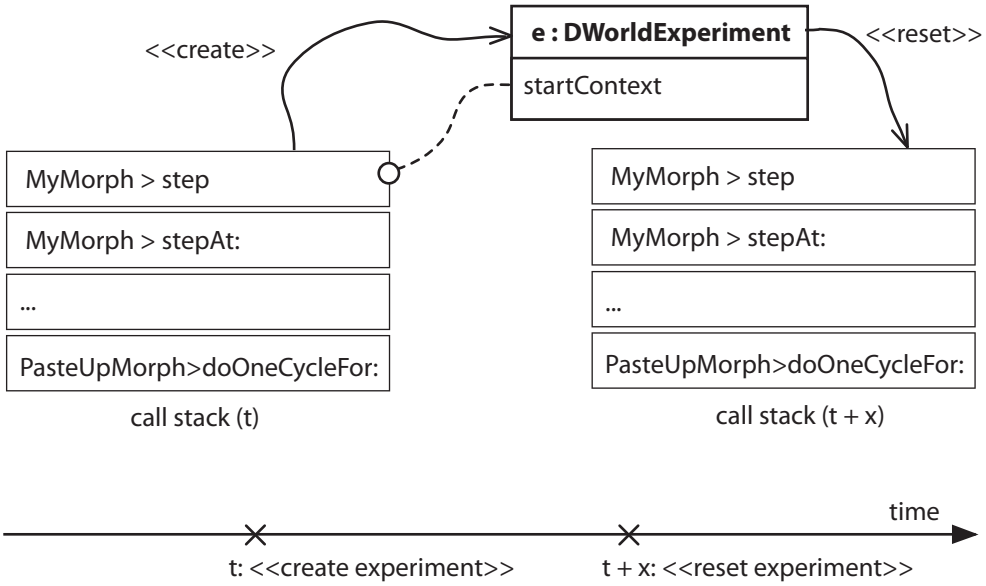

Figure 5.5.: Resetting global experiments to a different call stack

5.2.3. Discussion

The dwdbg enables quasi deterministic re-execution of program behavior inside a debugging session by making local exploratory experiments conducted during debugging reversible. Arguably that enhances run-time program comprehension as it allows a developer to safely re-examine program behavior he failed to understand in the first run.

The mechanisms underlying the dwdbg are simple. Exploratory experiments are used to denote reversible parts during a debugging session. Inside an experiment side effects are captured in special worlds and thus are easy to discard. At the same time Smalltalk's reflective features are employed to reset the call stack to the point an experiment was created. Both, resetting the program state and the execution context, gives the user the delusion of traveling back to the point the experiment was started whenever he decides to re-start a exploratory experiment.

Based on DWorlds, the generic Worlds implementation presented in chapter 4, an easily extensible Squeak/Smalltalk debugger and the reflective facilities of Smalltalk the dwdbg was simple and fast to realize. In fact the implementation of the dwdbg is rather compact as it consists of around 100

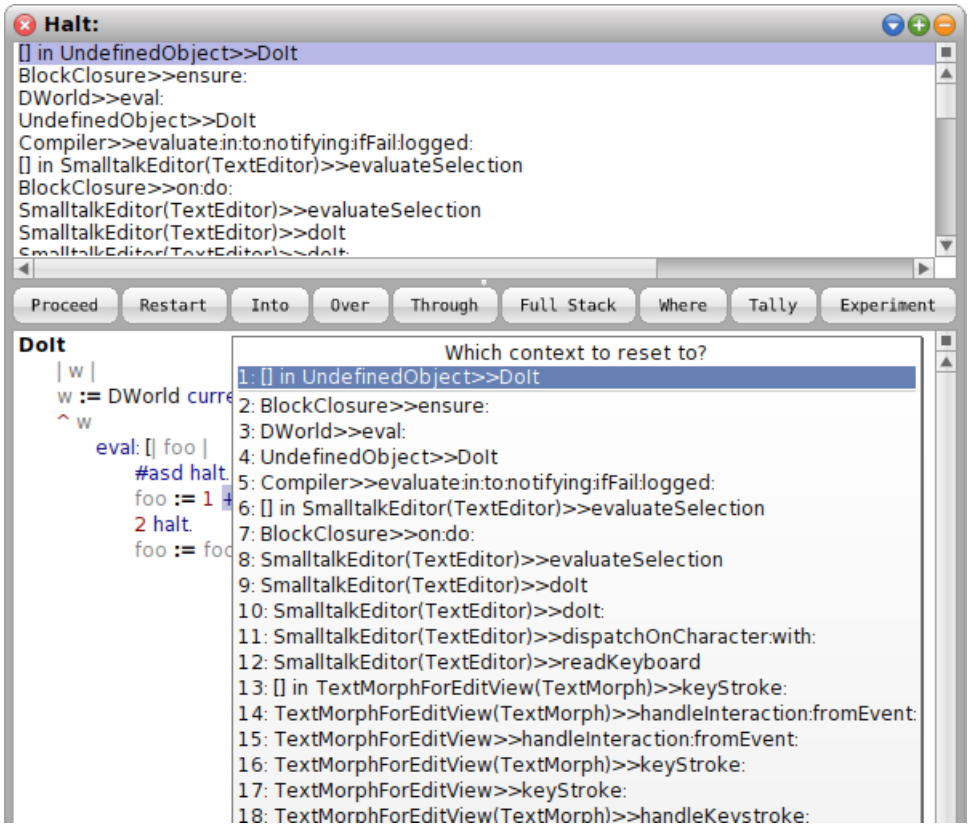

Figure 5.6.: Asking the user for assistance when reverting a global experiment

lines of code, only. These are spread over four new classes and the existing `Debugger` and `Process` implementations.

The experimentation model introduced in section 3.2.1 proved to be solid and useful, too. Despite the fact it needed some minor refinements, it basically made it straight forward to derive the dwdbg. As a result, most parts of the debugger implementation can directly be mapped to the model and vice versa.

Most notably, however, DWorlds as a general purpose mechanism to scope side effects in Smalltalk programs made it possible to implement support for the repeated examination of run-time behavior basically for free—that is, with a minimal amount of additional work needed[8].

[8]It accumulated in around one and a half hours of implementation effort, only.

5.2.4. Limitations

The dwdbg integration into the Squeak/Smalltalk debugger is in a prototypical state and as such far from feature complete. As a result developers have to work around a number of issues when using it.

First of all, the debugger currently requires that programs to be examined must be started in an experimental world in order to explore them repeatedly. One of the reasons for that is that on the fly switching from normal to the in-worlds behavior is not implemented. That, however, is doable with the help of the reflective facilities provided by Smalltalk. At the current point in time developers need to foresee if they will need dwdbg-specific features. If that is the case they have to start the program to be debugged from inside an in-worlds execution. Furthermore, the dwdbg provides only a very loose coupling between the experimentation features and the actual debugger UI. Arguably, a more sophisticated integration into the debugger could make experiments more tangible and, thus, would probably aid debugging in experiments.

5.3. Open Topics and Future Work

Given both DWorlds and its application in the dwdbg a number of open topics remain.

5.3.1. On DWorlds

Concerning DWorlds different directions exist to enhance the implementation as a general purpose Worlds. Further research efforts can investigate if parts of DWorlds can be implemented on the virtual machine level. Especially the explicit scoping mechanism would benefit from such an implementation as a VM-level implementation would allow it to keep the scope mapping closer to the actual scoped object, for instance as a tag on the object itself. That in term could enable a faster implementation of the scope reconstitution mechanism.

Additionally, more time can be spent in the area of per-object wrappers to come up with VM-supported *perfect proxies* which can wrap special methods such as `class`, too. A *perfect* proxy mechanism would allow more powerful instrumentation, probably at the expense of some VM-internal performance optimizations. It would not only benefit DWorlds but also other applications such as aspect- and context-oriented programming. To still be able to interact with perfectly proxies, mirrors [5, 23] could be a great tool to employ.

Both a VM-level rescoping mechanism and perfect proxies would allow it to implement DWorlds safely and without method wrappers and their incompatibilities with current Smalltalk distributions of Squeak and Pharo.

Last but not least time and brain power can be spent to prove or correct the scope reconstitution algorithm employed in the general purpose Worlds prototype. Up till now it has been proven useful but not yet correct.

5.3.2. On the dwdbg

Future work on the application of DWorlds for debugging can span various topics. Foremost it can focus on eliminating the previously mentioned issues and providing general enhancements to the dwdbg. In addition, it must involve actual user studies to prove or disprove the usefulness of the debugger extension empirically.

In the current version of the dwdbg Worlds is employed solely as a mean to manually make program explorations reversible. Worlds opens the field for a whole lot of other applications such as the comparison of object state before and after method execution or the parallel execution of a piece of code based on different prerequisites. Additionally it represents a efficient way to perform incremental checkpointing. That could make it possible to semi-automatically safe-point a debugging session in regular intervals or special occasions such as manually carried out step operations.

Future work should also review these applications to come up with more potential improvements for the comprehension of running programs. Finally, it should examine the application of the dwdbg in the context of multi-process applications, a interesting topic that has been willingly ignored thus far.

⁂

In this chapter we evaluated DWorlds as a general purpose Worlds mechanism and its application to debugging. In that course we introduced the dwdbg, a debugger that employs DWorlds to allow quasi deterministic re-execution of behavior inside a debugging session. We presented some of the details of the debugger implementation and showed how both local and global experiments can be conducted using the debugger. As part of the evaluation, we presented limitations of DWorlds and the dwdbg and summarized future work on the topic. In the context of DWorlds we highlighted the need to dispose method wrappers as a technique to realize spatial scoping, either by employ-

ing VM-support or perfect proxies. For dwdbg, we identified four important open topics: Improving the debugger, empirically asserting its usefulness, researching advanced applications of DWorlds in the dwdbg and investigating the debuggers application in multi-threaded programs.

6. Related Work

This chapter summarizes work adjacent to this thesis. It will present tree distinct areas: Debugging (section 6.1), encapsulating change (section 6.2) as well as perspectives and spatial scoping (section 6.3).

6.1. Debugging

Two limitations restrict the utility of most debuggers for program comprehension: High setup cost to inspect a particular part of the system as well as the inability to safely re-examine behavior. For both, related work has presented partial solutions which we go through in this section.

Rather than understanding the program as a whole, debuggers are often employed to examine certain behavioral characteristics in detail or to understand a particular code section [32, 41]. To do so, however, a user must step through the execution of a program to isolate the particular part of interest. That in term can be time consuming and complicated due to the lack of entry points into the system that can be used for debugging [41] as well as the lack of expressiveness of the user-debugger interaction [32].

To mitigate the high setup costs for stepping into a particular code section, Steinert et al. advocate the use of test-cases as entry points for debugging sessions [41]. They show how automatic analysis of a test suite can pick up tests that cover a particular method of interest. Furthermore, they present an IDE extension that runs a appropriate test and debugs right into the execution context associated with the method [41].

Ressia et al., in contrast, identify a gap between questions developers ask about a running software system [36] and the possibilities of conventional debugging tools to help answering them [32]. That lack of expressiveness leads to the fact that certain behavioral characteristics can be hard to isolate and observe in conventional debuggers.

To improve the situation they propose *object-centric debugging* as a novel approach to interact with a program at run-time [32]. In contrast to traditional debugging tools, which interrupt a program when certain positions in the

source code are reached, object-centric debuggers can interrupt a program when object interactions or state changes on objects occur. To facilitate this approach, the user sets break points on *state-related* operations (e.g. write or read of a certain instance variable) and *interactions* (e.g. message send to another object) with respect to certain objects. As objects in a running system are the subjects of breakpoints, the technique requires a program to be running and already interrupted. Thus, object-centric debugging is an extension to normal debugging approaches and could nicely be employed on top of the `dwdbg`.

Mostly driven by the need to fix hard to reproduce bugs, a number of techniques have been developed that enable deterministic re-examination of behavior executed during a debugging session. Some facilitate recording and replaying of recorded executions [16, 28, 39] on to a running program. Others rely on recording program actions only to reconstruct the executed behavior after the program died [22, 23] while a few approaches allow a developer to manually create checkpoints during a debugging session to restart the execution from there [10].

Record and replay debuggers capture instructions executed by a program or messages sent in between subsystems to allow replaying them in case of an error [28]. Srinivasan et al. for instance realize re-execution of behavior on Linux operating systems using a mixture of *snapshotting* program state and record and replay [39]. They employ the `fork` command [25] to snapshot running programs and replay recorded instructions on top snapshots to allow re-examination of behavior on the running program [39].

Back-in-time debuggers capture the full execution trace through the run of a program [22, 23]. Using that trace, they can reconstruct the program state at any point in the execution of the program. Back-in-time debuggers do not re-execute behavior of a program but can rather simulate its execution step-by-step both forwards and backwards in time. They are used *postmortem*—that is after the program has died—and thereby do not allow interaction with the debugging subject in a particular execution context of interest.

Both, record and replay and back-in-time debuggers make it possible to repeatedly examine run-time behavior, technically even in case of multi-threading. However, they have performance as well as scalability issues as significant parts of the execution trace have to be recorded at run-time [30, 32]. Some improvements have been proposed for both approaches [23, 39] which, after all, only defer the problems [32]. As a more important drawback, though, the approaches fail to support local re-examination on top of running

programs. That renders them impractical for the understanding of run-time behavior as proposed in this thesis.

Checkpoint debugging [10] is an approach that allows for the re-examination of behavior by letting users manually indicate points during a debugging session they want to return later. It is closest to the approach presented in this thesis. Conventional checkpointing uses platform specific tricks to snapshot the program state whenever a new checkpoint is created [11]. In contrast, the application of Worlds in the dwdbg safes only changes which occurred in between checkpoints. Thus it can be seen as a mean to perform incremental checkpointing, which can be more space efficient if a big amount of checkpoints is created. In contrast to conventional checkpointing, our solution does not incur additional costs for snapshotting and restoring program images. Additionally it enables the re-examination of run-time behavior in a platform independent manner—by employing the ability of DWorlds to scope side effects.

6.2. Encapsulating Change

Two different areas are concerned when talking about the encapsulation of change: Isolating structural changes and capturing side effects in running programs. Capturing structural changes such as the renaming of classes or introduction of new methods is not a primary goal of DWorlds, neither is it currently supported by the implementation. However, other approaches have specialized on the encapsulation and modeling of structural changes.

Classboxes, presented by Bergel et al., isolate structural changes done to a program. Thereby they make it possible to extend a software system inside a classbox without impacting other applications that run outside of it [1]. A classbox can be seen as a world in which the structural changes can safely be implemented without the fear to break already running programs.

Changeboxes [9] superficially resemble classboxes. In contrast to classboxes, though, they are first-class entities which encapsulate the semantics of structural change in running software systems. As a first class representation, a changebox can be used to capture refactorings. Later, it makes it possible to apply or revert these changes to support the safe evolution of in-production software systems such as web servers [9].

Both classboxes and changeboxes focus on encapsulating structural change. For this reason they have a slightly different scope than DWorlds. DWorlds—as

presented in this work—aims to control the scope of side effects in running programs. That in term is often regarded program state rather than structural changes. Worlds as a concept generalizes that to enclose structural changes, too. In that regard, Worlds it is closer to classboxes than to changeboxes as it captures and isolates change, only, rather than capturing its semantics.

The idea to reify program state goes back to Johnson and Duggan who proposed *first-class stores* for the GL programming language. The stores were meant to represent "different versions of computer memory" [18]. The authors proposed the use of first-class stores and partial continuations to capture the remainder of a debugging session in them. In that regard, their idea is close to the application of DWorlds in the dwdbg. First-class stores were meant to operate in conjunction with partial continuations in order to aid the implementation of development tools in a interactive programming environment, only. DWorlds, in contrast, is intended as a simple, general applicable mechanism to control the scope of side effects. Consequently, debugging is only one of the many areas it can be employed in.

Transactional Memory [13, 31, 35] must offer a way to capture side effects during the execution of a particular code section with the goal to commit these side effects with transactional semantics. These include atomicity, isolation and consistency. In contrast to Worlds, which offers a reification of state along with the ability to store references to different worlds inside a program, transactional memory usually does not offer any first class representation of transactional state. In fact, the concept of Worlds is more powerful and would easily allow the implementation of transactional memory semantics [44].

Renggli and Nierstrasz introduced transactional memory for Smalltalk [31]. They provided a mature implementation of it as a general purpose mechanism in the Smalltalk system. Consequently, our work inherited some of the mechanisms which backed up their implementation.

6.3. Perspectives and Spatial Scoping

Smith and Ungar introduce perspectives on objects in course of their implementation of Us [38], the predecessor of today's context-oriented programming (COP) languages. The authors explore the idea that object state and behavior depends on even these perspectives—in that regard perspectives are similar to worlds. The difference, however, is that perspectives do not form hierarchically structured scopes and thus cannot be consolidated using

a *commit* operation. The authors also discuss the question when changes in perspectives occur and ultimately utilize a "minimal motion policy" in their implementation of Us. While introduced in a slightly different context, the policy closely resembles scope changes during explicit scoping in Worlds.

The need to explicitly bind objects to a particular world, both for the sake of tool support and GUI applications, was one of the crucial insights made in this thesis. On a similar track, Lincke et al. identified the need to explicitly attach the layer activation in their COP [15] implementation for JavaScript [24]. In both, Worlds and COP, the activation of a worlds-scoping or COP layers is dependent on the perspective from within an object is perceived. That perspective us usually tied to the current execution context but must be persisted when objects are accessed from outside that context.

In contrast to COP layer activation, which is inherently per object, Worlds scoping is transitive (i.e. propagates to other objects in an object graph). That made the application of a sophisticated re-scoping algorithm in DWorlds inevitable.

⁂

In this chapter we presented work related to this thesis. Most notably, we distinguished the application of DWorlds for debugging from other approaches which facilitate the re-examination of run-time behavior. Further, we looked at research which came to related findings such as the application of first-class stores for debugging and the need for explicit scoping in COP.

In the following chapter we summarize the main findings of this thesis and conclude the topic.

7. Conclusion and Outlook

Debuggers are often used to inspect the run-time behavior of a program in order to understand the underlaying source code. They differ substantially from conventional dynamic views as they make it possible to experience a running program as it executes. Thereby they encourage experimental learning and induce a deeper kind of understanding of the inspected behavior.

This work evaluated Worlds [44] to further increase the utility of debuggers for program comprehension by allowing the safe re-examination of program behavior during debugging. The idea was that Worlds could be employed to capture side effects caused by stepping through a running program. By doing so, it would be possible to simply discard these effects and, thus, make it safe to repeatedly execute behavior inside a debugging session.

The evaluation results are two-fold: Yes, the concept of Worlds can be employed for the purpose and no, the already existing implementation of Worlds for Smalltalk [44], cannot be used to realize the idea. The reason for the latter is, that it is a special purpose Worlds that works on a small number of classes, only. To make the concept of Worlds applicable for debugging, a general purpose Worlds implementation is needed that facilitates the scoping of side effects for arbitrary objects. Such an implementation and its application in a debugger for Squeak/Smalltalk are the practical results of this work.

7.1. Summary of Contributions

In the course of this thesis three major contributions were made. First, the principal application of Worlds for debugging was documented. Second, DWorlds, a general purpose Worlds for Smalltalk, was presented. Third, a debugger was introduced which builds upon DWorlds to realize the safe re-examination of run-time behavior inside a debugging session.

To start with, chapter 3 described the application of Worlds in the context of debugging. The insight was that explorations performed during a debugging session can be made reversible by capturing and isolating the side effects

caused by them. Based on that idea, this work introduced an experimentation model which formalized the application of Worlds during exploratory experiments. Adhering to that model makes it possible to safely re-examine run-time behavior during debugging.

Chapter 4 presented DWorlds, a general applicable Worlds mechanism for Smalltalk. The implementation was needed in the context of debugging in order to scope side effects in large-scale experiments which comprise arbitrary objects. That is something the original Worlds for Smalltalk implementation is incapable to do. With DWorlds, we showcased a transparent Worlds mechanism that got implemented using the reflective features of Smalltalk alone. We highlighted three important aspects of the implementation. To begin with, we presented a generic Worlds dispatch which allows us to scope side effects in arbitrary classes. Further, we described a two-layered approach which cleanly separates normal and in-worlds behavior. That approach makes it possible to customize behavior in the scope of side effect capturing worlds. Additionally, it enabled it to safely implement the DWorlds core on top of standard Smalltalk classes. As a last important aspect of DWorlds, we depicted a mechanism for spatial scoping of world-local experiments. That mechanism enables the application of Worlds in GUI applications and ensures the safe interaction between diversly scoped objects. At the same time, it permits the usage of conventional dynamic views to display objects with their in-worlds state.

As the third overall contribution of this work, chapter 5 presented the dwdbg, a debugger prototype for Squeak/Smalltalk. The debugger employs DWorlds to capture side effects during the inspection of run-time behavior. To do so, it reifies explorations and allows a developer to indicate parts of a development session he might want to examine repeatedly. Based on DWorlds, the experimentation model and the Squeak/Smalltalk debugger, the dwdbg was simple and straight forward to implement. In its current form, it facilitates the safe re-examination of both local program execution and globally observable behavior.

7.2. Outlook

When implementing the concept of Worlds as a general purpose mechanism on Squeak/Smalltalk, a number of obstacles appeared. Some, such as the need for spatial scoping, raised implementation specific issues which could not be mitigated completely in the course of this thesis. At the same time, the

practical application of DWorlds in the dwdbg showed that the general purpose Worlds mechanism is ready to serve real world purposes and, thus, is more than a little toy project.

Open topics, both regarding DWorlds and the dwdbg, exist and have been well documented in section 5.3 of this work. For DWorlds these include VM-support and/or the invention of perfect proxies for Squeak/Smalltalk. Both allow it to dispose method wrappers and provide a more robust implementation of per object spatial scoping. For the application of DWorlds in the dwdbg topics revolve around a multitude of aspects. They include carrying out user studies to assert the debuggers' usefulness, exploiting different Worlds features to further support program comprehension and improving the graphical representation of the dwdbg to make the experiment metaphor more tangible.

It remains to be seen if future research continues the work on any of those topics or whether it employs DWorlds to evaluate new applications of the Worlds concept on Smalltalk.

Bibliography

[1] Alexandre Bergel et al. "Classboxes: A Minimal Module Model Supporting Local Rebinding". In: *JMLC*. Ed. by László Böszörményi and Peter Schojer. Vol. 2789. Lecture Notes in Computer Science. Springer, 2003, pp. 122–131.

[2] Roland Bertuli et al. *Run-Time Information Visualization for Understanding Object-Oriented Systems*. 2003.

[3] Ted J. Biggerstaff et al. "The Concept Assignment Problem in Program Understanding". In: *ICSE*. 1993, pp. 482–498.

[4] Andrew Black et al. *Squeak by Example*. http://squeakbyexample.org. Square Bracket Associates, 2007.

[5] Gilad Bracha and David Ungar. "Mirrors: design principles for meta-level facilities of object-oriented programming languages". In: *OOPSLA*. Ed. by John M. Vlissides and Douglas C. Schmidt. ACM, 2004, pp. 331–344.

[6] John Brant et al. "Wrappers to the Rescue". In: *ECOOP*. Ed. by Eric Jul. Vol. 1445. Lecture Notes in Computer Science. Springer, 1998, pp. 396–417.

[7] Thomas A. Corbi. "Program understanding: challenge for the 1990's". In: *IBM Syst. J.* 28.2 (June 1989), pp. 294–306. ISSN: 0018-8670.

[8] Bas Cornelissen. "Evaluating Dynamic Analysis Techniques for Program Comprehension". PhD thesis. Delft University of Technology, 2009.

[9] Marcus Denker et al. "Encapsulating and exploiting change with changeboxes". In: *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*. ICDL '07. New York, NY, USA: ACM, 2007, pp. 25–49.

[10] *GDB Online Documentation: 4.12 Setting a Bookmark to Return to Later*. `http://sourceware.org/gdb/onlinedocs/gdb/Checkpoint_002fRestart.html`. [Online; accesssed 2012-03-17]. 2012.

[11] *GDB Online Documentation: Algorithms*. `http://sourceware.org/gdb/onlinedocs/gdbint/Algorithms.html`. [Online; accesssed 2012-03-17]. 2012.

[12] Adele Goldberg. *SMALLTALK-80: the interactive programming environment*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[13] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *ISCA*. 1993, pp. 289–300.

[14] Robert Hirschfeld. "AspectS - Aspect-Oriented Programming with Squeak". In: *NetObjectDays*. Ed. by Mehmet Aksit et al. Vol. 2591. Lecture Notes in Computer Science. Springer, 2002, pp. 216–232. ISBN: 3-540-00737-7.

[15] Robert Hirschfeld et al. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (2008), pp. 125–151.

[16] Derek Hower et al. "Two hardware-based approaches for deterministic multiprocessor replay". In: *Commun. ACM* 52.6 (2009), pp. 93–100.

[17] Christian M. Itin. "Reasserting the Philosophy of Experiential Education as a Vehicle for Change in the 21st Century". In: *The Journal of Experiential Education* 22.2 (1999), pp. 91–98.

[18] Gregory F. Johnson and Dominic Duggan. "Stores and partial continuations as first-class objects in a language and its environment". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '88. New York, NY, USA: ACM, 1988, pp. 158–168.

[19] Andrew J. Ko et al. "Information Needs in Collocated Software Development Teams". In: *ICSE*. 2007, pp. 344–353.

[20] David A. Kolb. *Experiential Learning: Experience as the Source of Learning and Development*. New Jersey: Prentice-Hall P T R, 1984.

[21] Thomas D. LaToza et al. "Maintaining mental models: a study of developer work habits". In: *ICSE*. 2006, pp. 492–501.

[22] Bil Lewis. "Debugging Backwards in Time". In: *CoRR* cs.SE/0310016 (2003).

[23] Adrian Lienhard et al. "Practical Object-Oriented Back-in-Time Debugging". In: *Proceedings of the 22nd European conference on Object-Oriented Programming*. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 592–615.

[24] Jens Lincke et al. "An open implementation for context-oriented layer composition in ContextJS". In: *Sci. Comput. Program.* 76.12 (2011), pp. 1194–1209.

[25] *Linux Man Pages: Fork*. `http://linux.die.net/man/2/fork`. [Online; accesssed 2012-03-17]. 2012.

[26] John Maloney. "Squeak: Open Personal Computing and Multimedia". In: ed. by Mark Guzdial and Kimberly Rose. Prentice Hall, 2002. Chap. 2: An Introduction to Morphic: The Squeak User Interface Framework, pp. 39–68.

[27] Anneliese von Mayrhauser and A. Marie Vans. "Program Comprehension During Software Maintenance and Evolution". In: *IEEE Computer* 28.8 (1995), pp. 44–55.

[28] Satish Narayanasamy et al. "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging". In: *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 284–295.

[29] Eric Osman. *DDT Reference Manual*. AI memo. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1971.

[30] Michael Perscheid et al. "Immediacy through Interactivity: Online Analysis of Run-time Behavior". In: *WCRE*. 2010, pp. 77–86.

[31] Lukas Renggli and Oscar Nierstrasz. "Transactional Memory for Smalltalk". In: *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*. ICDL '07. New York, NY, USA: ACM, 2007, pp. 207–221.

[32] Jorge Ressia et al. "Object-centric debugging". In: *ICSE*. IEEE, 2012, pp. 485–495.

[33] Don Roberts et al. "A Refactoring Tool for Smalltalk". In: *TAPOS* 3.4 (1997), pp. 253–263.

[34] David Röthlisberger et al. "Exploiting Runtime Information in the IDE". In: *ICPC*. 2008, pp. 63–72.

[35] Nir Shavit and Dan Touitou. "Software Transactional Memory". In: *Distributed Computing* 10.2 (1997), pp. 99–116.

[36] Jonathan Sillito et al. "Questions programmers ask during software evolution tasks". In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 23–34.

[37] Michael P. Smith and Malcolm Munro. "Runtime Visualisation of Object Oriented Software". In: *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. VISSOFT '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 81–.

[38] Randall B. Smith and David Ungar. "A Simple and Unifying Approach to Subjective Objects". In: *TAPOS* 2.3 (1996), pp. 161–178.

[39] Sudarshan M. Srinivasan et al. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging". In: *USENIX Annual Technical Conference, General Track*. USENIX, 2004, pp. 29–44.

[40] Jamie Starke et al. "Searching and skimming: An exploratory study". In: *ICSM*. 2009, pp. 157–166.

[41] Bastian Steinert et al. "Debugging into Examples". In: *TestCom/FATES*. 2009, pp. 235–240.

[42] Masaki Suwa and Barbara Tversky. "External Representations Contribute to the Dynamic Construction of Ideas". In: *Diagrams*. 2002, pp. 341–343.

[43] Tarja Systä. "Understanding the Behavior of Java Programs". In: *WCRE*. 2000, pp. 214–223.

[44] Alessandro Warth et al. "Worlds: Controlling the Scope of Side Effects". In: *ECOOP*. 2011, pp. 179–203.

Appendix A.

Additional Code Examples and Illustrations

A.1. State Access in Smalltalk

State access in Smalltalk comprises instance variable access (cf. listing A.1) and indexed field access (cf. listing A.2).

```
Person#secondName: aString

  | nameInstVarIndex |
  nameInstVarIndex := self class instVarIndexFor: #name.

  "set name"
  name := aString.

  (name = self instVarAt: nameInstVarIndex)
    ifTrue: [ "reflective read worked" ].

  "reflective write"
  self
    instVarAt: nameInstVarIndex
    put: (aString, ' the second').

  name "... the second"
```

Listing A.1: Instance variable access in Smalltalk

```
| anArray |
anArray := Array new: 2. "array with two slots"

"indexed field write"
anArray at: 1 put: 'FOO'.

"indexed field read"
```

```
anArray at: 1 "FOO"

"#basicAt: and #basicAt:put: could be used accordingly"
```

Listing A.2: Indexed field access in Smalltalk

A.2. Fixing Reflective Message Sends in DWorlds

The problem is that invoking a reflective message send in the scope of a world would normally break the in-worlds execution chain (cf. listing 4.9). The solution is to transform all selectors passed to `#perform:with:` and friends to their in-worlds form before sending the actual `#perform:with:` method. That can be done by creating an annotation-guided indirection (see listings A.3 and A.4).

```
perform: aSymbol with: anObject
  "Send the selector, aSymbol, to the receiver [...]"

  <atomicUseUntransformed: #atomicPerform:with:>
  <primitive: 83>
  ^ self perform: aSymbol withArguments: (Array with: anObject)
```

Listing A.3: Usage of the `<atomicUseUntransformed:>` to redirect reflective message sends in the case of in-worlds method execution

```
atomicPerform: aSymbol with: anObject
  "Send the selector, aSymbol, to the receiver [...]"

  ^ self perform: (aSymbol asAtomicSelector) with: anObject
```

Listing A.4: In-worlds implementation of the `#perform:with:` method which adapts the message selector to its in-worlds form before it invokes the original `#perform:with:` method with these arguments

A.3. Final Scope Reconstitution Algorithm

Based on the findings in section 4.4.2 and section 4.4.3 the final scope reconstitution algorithm looks as follows (numbers have been rearranged from previous samples):

#0 Let o be the current object, *args* the method arguments, c the current local scope, *e* the explicit scope of o and m the original method.

#1 Let c_{ctx} be the contextual scope.

#2 If e is set and e and c_{ctx} are both local experiments refine e to c_{ctx}.

#3 If e is not set or e equals c invoke wrapped method and return the method result (perform a quick return which works in most cases).

#4 Explicitly scope $args$ to c unless they are already explicitly scoped.

#5 If e is an experimental scope reconstitute the scope e.

#6 Invoke m in the scope of e[1]. Let r be the result of the method invokation.

#7 Explicitly scope r to e unless it is already scoped.

#8 Reconstitute the previous scoping c.

#9 Return r.

A.3.1. Implementation in Smalltalk

Listing A.5 shows a optimized implementation of the algorithm in Smalltalk.

```
dispatchTo: anObject withArguments: anArrayOfObjects

  "Implementation of the scope reconstitution algorithm"

  | localScoped ctxSwitch quickReturn effectiveCtx contextCtx
   objectCtx argumentsScope returnValueScope args result |
  self wasActive: true.

  objectCtx := DWorld scopeFor: anObject.

  "Quick return for contextual scoped objects"
  objectCtx ifNil: [
    ^ self clientMethod valueWithReceiver: anObject
        arguments: anArrayOfObjects.
  ].

  quickReturn := ctxSwitch := false.

  localScoped := self isScoped.
  contextCtx := DWorld current.

  "Enable quick return for right binding _or_ other experiment"
  (objectCtx == contextCtx)
    ifTrue: [ quickReturn := true ]
```

[1] That might require to switch from the in-worlds version of the method to the normal version or vice versa.

```
  ifFalse: [
    "Object ctx and current ctx both denote experiments
     (non of them top-level world)"
    (objectCtx isTopLevel)
      ifTrue: [
        localScoped
          ifTrue: [ effectiveCtx := objectCtx ]
          ifFalse: [ quickReturn := true ]]
      ifFalse: [
        (contextCtx isTopLevel)
          ifTrue: [
            effectiveCtx := objectCtx.
            ctxSwitch := true ]
          ifFalse: [
            localScoped
              ifTrue: [ quickReturn := true ]
              ifFalse: [ effectiveCtx := contextCtx.
                 ctxSwitch := true ]]]].

quickReturn ifTrue: [
  ^ self clientMethod valueWithReceiver: anObject
       arguments: anArrayOfObjects ].

argumentsScope := localScoped
  ifTrue: [ contextCtx ]
  ifFalse: [ DTopLevelWorld instance ].

returnValueScope := objectCtx.

args := self scopeArguments: anArrayOfObjects
          to: argumentsScope.

ctxSwitch
  ifTrue: [ DWorld current: effectiveCtx ].

[
  result := anObject
    perform: (self selectorFor: (self selector)
                in: effectiveCtx)
    withArguments: args.

  result isScoped ifFalse: [
    result scopeTo: returnValueScope ].
] ensure: [ ctxSwitch ifTrue: [ DWorld current: contextCtx ]].

^ result
```

Listing A.5: Implementation of the scope reconstitution algorithm in Squeak/Smalltalk

A.4. Showcasing Complex Scoping and Re-scoping of Objects

The following listing showcases scoping in action. Furthermore, figure A.1 shows the effect of re-scoping during various standard interactions between a expriment-bound morph and the globally scoped morphic world.

```
| ctx person pet petsPet |

"Pet should always be unscoped"
pet := DTestPerson new.
pet makeGlobal.

ctx := DWorld current sprout.
ctx eval: [
  person := DTestPerson new.
  person pet: pet.
  pet name: 'Walter'.
  person name: 'Klaus'.

  self assert: (person name) equals: 'Klaus'.
  self assert: (pet name) equals: 'Walter'.
].
self assert: (person name) equals: nil.
self assert: (pet name) equals: 'Walter'.

"Person should access unscoped walter internally"
  ctx eval: [
  self assert: (person meAndMyPet) equals: 'Klaus and Walter'
].

"Pin person to ctx"
person scopeTo: ctx.

"Person should evaluate in context of current world"
self assert: (person name) equals: 'Klaus'.
self assert: (person meAndMyPet) equals: 'Klaus and Walter'.

" ------------------------------------------ "
" advanced concepts"
" ------------------------------------------ "

" We will create a scoped petsPet to our unscoped pet"
ctx eval: [
  petsPet := DTestPerson new.
  petsPet name: 'Lucy'.
].
```

```
self assert: (petsPet name) equals: nil.

ctx eval: [
   self assert: (petsPet isScoped not).

  "Scoping of petsPet is decided here:
  As petsPet is passed to pet in a scoped context,
  it will remain scoped to it"
  pet pet: petsPet.

  "Pets pet is scoped to ctx now"
  self assert: (petsPet isScoped).

  self assert: (pet pet name) equals: 'Lucy'.
].

self assert: (pet pet name) equals: 'Lucy'.
```

Listing A.6: Scoping in action

A.5. Scope Reconstitution Micro Benchmarks

Micro benchmarks used to measure the performance for the method wrapper scope reconstitution mechanism are shown in listing A.7 (using no scoping) and listing A.8 (using explicit scoping).

```
| rawTime instrumentedTime block |

block := [
  5000 timesRepeat: [
    | p c |
    p := DTestPerson new.
    c := DTestPerson new.
    p name: 'Walter'.
    p pet: c.

    p name.
    p meAndMyPet.
    c age.
    c name.
  ]].

rawTime := block timeToRun.

DWorldMethodWrapper installOn: DTestPerson.
```

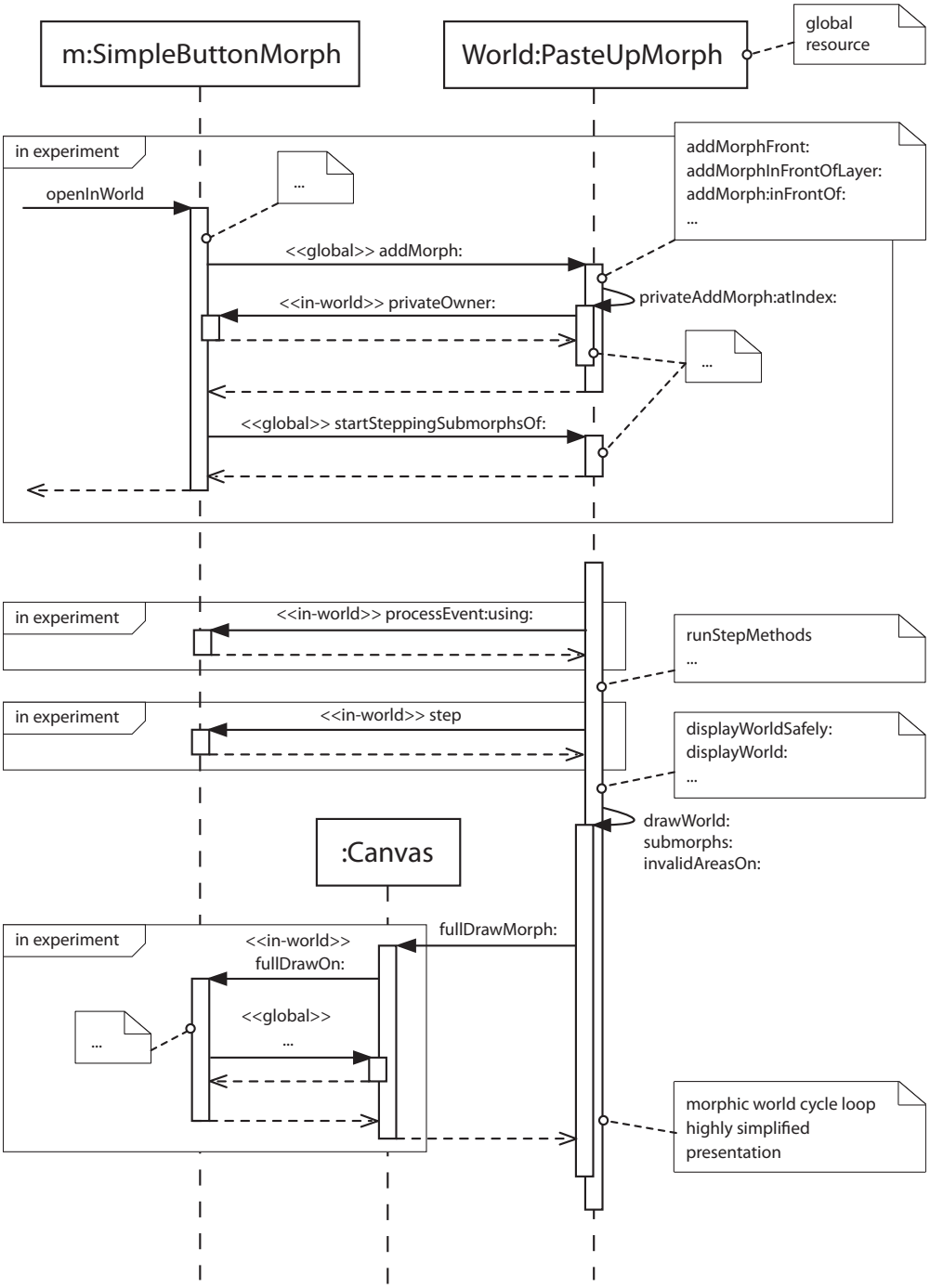

Figure A.1.: Explicit scoping in action during the communication of an in-experiment morph and the global morphic world

```
instrumentedTime := block timeToRun.

Transcript
  show: 'raw: ', (rawTime asString); cr;
  show: 'instrumented: (instrumentedTime asString); cr.
```

Listing A.7: Microbenchmark not performing any actual scoping

```
| rawTime instrumentedTime block |

block := [
  5000 timesRepeat: [
    | p c |
    p := DTestPerson new.
    c := DTestPerson new.
    p scopeTo: DWorld current sprout.
    c scopeTo: DTopLevelWorld instance.
    p name: 'Walter'.
    p pet: c.

    p name.
    p meAndMyPet.
    c age.
    c name.
  ]].

rawTime := block timeToRun.

DWorldMethodWrapper installOn: DTestPerson.

instrumentedTime := block timeToRun.

Transcript
  show: 'raw: ', (rawTime asString); cr;
  show: 'instrumented: (instrumentedTime asString); cr.
```

Listing A.8: Microbenchmark which uses the actual scoping of objects

Appendix B.

Setting up and Working with DWorlds and the dwdbg

DWorlds and the dwdbg both operate are reported to operate on Squeak/Smalltalk version 4.1. A previous version of DWorlds operated on Pharo/Smalltalk 1.3. Yet, that version had to be ported to to Squeak because support for method wrappers in Pharo is inherently broken. This section goes through the installation of DWorlds and the dwdbg on a vanilla Squeak 4.1 image[1].

```
"Install required refactoring browser and
optionally ocompletion"

(Installer ss
  project: 'MetacelloRepository')
  install: 'ConfigurationOfOCompletion';
  install: 'ConfigurationOfRefactoringBrowser'.

((Smalltalk at: #ConfigurationOfOCompletion)
  project
  version: '1.1.2')
    load.

"Fix minor bugs that occur during setup"

((Smalltalk at: #ConfigurationOfRefactoringBrowser)
  project
  version: '1.6')
    load.
```

Listing B.1: Configuring Squeak for DWorlds

To prepare the Squeak image for DWorlds, the usage of underscores as assignment operators must be forbidden. That is done using a refactoring rule and by configuring the compiler.

[1]Which can be downloaded from `http://ftp.squeak.org/4.1/`

```
| environment rule change |
"Fix underscores for all classes"
environment := BrowserEnvironment new
    forPackageNames: Smalltalk organization categories.

rule := RBUnderscoreAssignmentRule new.

SmalllintChecker runRule: rule onEnvironment: environment.

change := CompositeRefactoryChange new.
change changes: rule changes.
change execute

"Configure the compiler / parser"
Scanner prefAllowUnderscoreSelectors: true.
Scanner allowUnderscoreAsAssignment: false.
```

Listing B.2: Removing underscore assignments from a Squeak image

Now DWorlds and its direct prerequisites can be loaded into the image[2]. In the right order, those are the packages `LRSTMPrerequisites`, `LRSTM`, `MethodWrappers` and `DWorlds`. Before DWorlds and the dwdbg can actually be used, the DWorlds compiler has to be enabled explicitly (see next listing).

```
"Initially and to reload the compiler"
DWorldCompiler load.

"Unloading the compiler (in case of emergencies)"
DWorldCompiler unload.

"Uninstalling the method wrappers"
DWorldMethodWrapper nuke.
```

Listing B.3: Activating DWorlds in a Squeak/Smalltalk image

For the usage of DWorlds one can refer to the test cases or the numerous code samples shown in this thesis. The dwdbg compiler extension is enabled as soon as the `DWorlds` package is installed in the Squeak image.

[2] They are located on a montichello repository at `http://nixis.de/~nikku/uni/master/thesis/dworlds/`

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Berlin, den 30. Juli 2012

Nico Rehwaldt