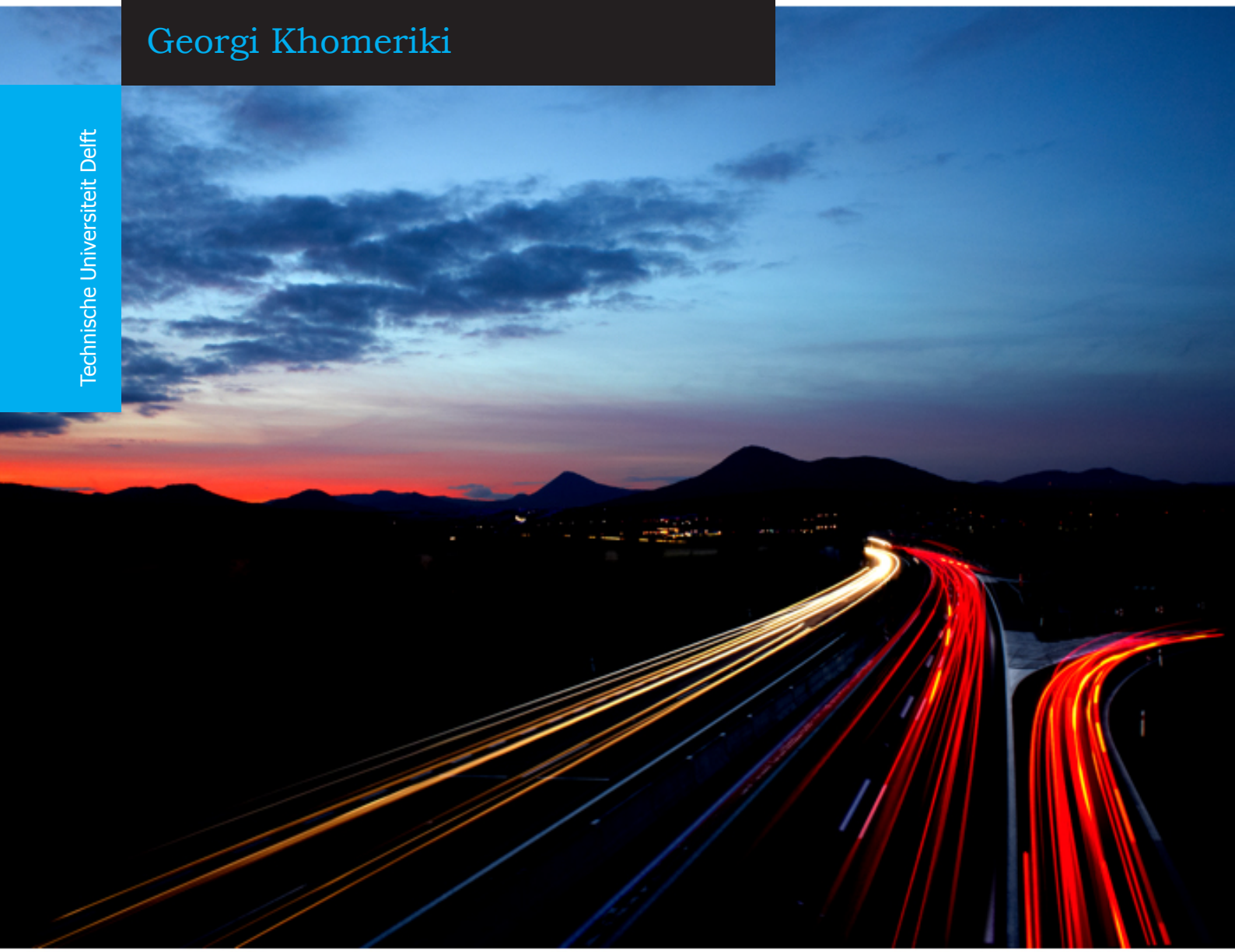
Scriptable Debugging
A Reactive Approach

Georgi Khomeriki

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Scriptable Debugging

A Reactive Approach

by

Georgi Khomeriki

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Friday June 15, 2018 at 04:00 PM.

Supervisor:	Prof. dr. H.J.M. Meijer	
Thesis committee:	Dr. G. Gousios,	TU Delft
	Dr. S. Erdweg,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Abstract

Debuggers are crucial tools for developers to support the process of developing software systems as they provide direct insights into the execution of their code. As software development in the industry is moving towards technology stacks that operate on increasingly higher levels of abstraction, debugging tools have not evolved as quickly. This creates an abstraction gap between the concrete debugging needs of developers and the support that debugging tools offer. To reduce this gap, we propose a generic framework for reactive scriptable debugging. We propose a design and reference implementation of such a scriptable debugging system. Using this framework we aim to liberate developers from rigid debugging tools and give the power of debugging back to the developers.

Georgi Khomeriki

Acknowledgments

I would like to thank professor Erik Meijer for his amazing support throughout this wonderful journey. Thanks to your supervision I have learned and achieved so much more than I could've hoped for. I will never forget the first thing that you told me when we met: "Think like a fundamentalist, Code like a hacker". Although I have a lot to improve on that front, I will always be driven by that motto.

I would also like to thank my family (Lali, Gela, Almina, Maria, Micha, Natasha, Philip, Lucas, and the rest of the gang) for all the patience and support they have provided me. Thank you for allowing me to pursue my dreams.

Many thanks go out to Eddy and Mircea for making the whole university experience so much more enjoyable. Keep doing it live guys!

Last but certainly not least, I want to thank my wife Keti. Thank you for all your love, support and for following me so far away from home. None of this would have been possible without you.

Georgi

Contents

1

Prelude

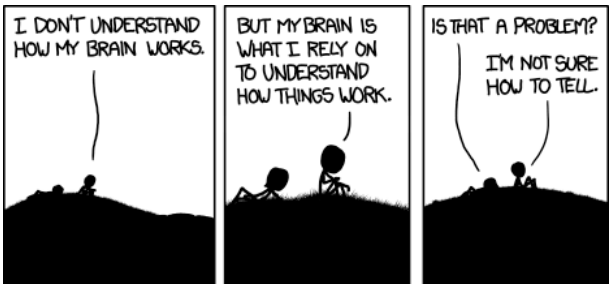


Figure 1.1: Debugger (https://xkcd.com/1163/)

Debugging is a complex activity because there is often a large amount of knowledge about a program that is not explicitly represented in its execution. Debuggers are crucial tools for developers to support the process of developing software systems as they give them direct insights into the execution of their code. Nevertheless, traditional debuggers rely on old concepts to explore and trace the execution stack and system state. These tools often do not help developers to reason about the application and domain specific on a higher level of abstraction. As software development in the industry is moving towards technology stacks that operate on increasingly higher levels of abstraction, debugging tools have been left behind. This creates an abstraction gap between the concrete debugging needs of developers and the support that debugging tools offer. To reduce this gap, we propose a generic framework for reactive scriptable debugging.

When using currently popular tools and techniques, debugging is a laborious, manual activity that often involves the repetition of primitive operations. Our proposed framework aims to enable developers to describe these repetitious operations as automated debugging scripts. We strive to show the merits of pursuing a workflow in which debuggers are exposed as programmable, i.e. scriptable, entities.

1.1. Problem description

The adoption of non-imperative programming paradigms has accelerated in recent times. Languages, frameworks and libraries that rely on a declarative programming style have emerged and are being adopted on a large scale. The declarative programming style is seeing adoption due to its advantages in terms of higher-level abstractions, clear semantics, programmer productivity, meta-programming, asynchrony, parallelism, etc. [1]. Although the way that developers are writing code is changing rapidly, the accompanying debugging tools are not evolving as quickly. Many debugging tools still rely primarily on setting breakpoints and manual stack and heap inspections by the developer. Declarative code, by definition, does not execute in a strict sequential manner. Hard-to-track bugs can emerge when you can't guarantee sequential execution [2]. This situation is made even more complicated when code is relying primarily on asynchronous and concurrent execution. Such programs are notoriously complicated to debug, because debugging tools regard the stack trace to be the most important source

of information for developers when trying to understand the execution of their code. We argue that a debugging workflow that revolves around halting the execution of a program and manually looking at the heap and stack at a single point in time is a very inefficient endeavor.

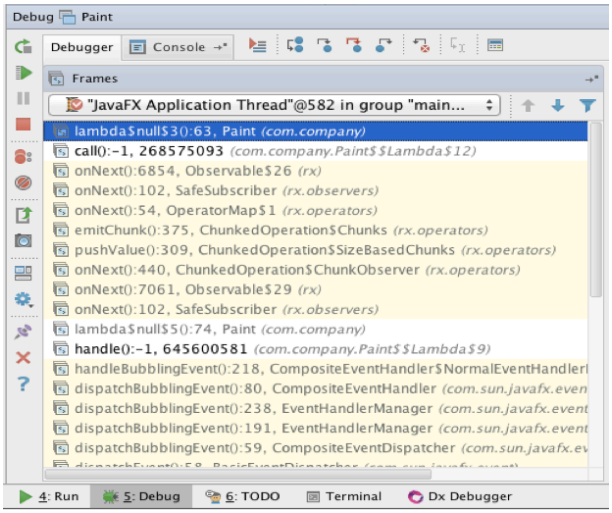


Figure 1.2: Asynchronous programs result in worthless stack traces; example based on Rx.

1.2. Research questions

In this thesis, we aim to address the following research questions:

- What is a feasible architecture for a scriptable debugging system that operates on the Java Virtual Machine?
- How can we implement the scriptable debugging system such that all relevant operational events are exposed to developers?
- Is a reactive approach useful for a scriptable debugging system?
- What are the pros and cons of different implementation techniques?
- Is it feasible to implement real-time pattern matching on top of the scriptable debugging system?
- Explore whether a graphical IDE plugin can be implemented for the scriptable debugging system.
- Explore the practicality of scriptable debugging by demonstrating the implementation of useful custom debugging tools.

1.3. Proposed approach/solution

Many developers (read: true hackers [3]) will agree that whenever a certain task becomes too laborious and complicated the first solution to consider should involve automation. Debugging has fallen into this category, it indeed has become too laborious and complicated, but for unknown reasons we have not embraced automation for this purpose yet. In this thesis, we will propose a way to automate the daunting task of debugging modern complex systems via reactive scriptable debugging. We aim to design and implement a reference implementation of such a scriptable debugging system. Additionally, our intent is to demonstrate its usefulness through concrete examples of custom automated debugging tools. In short, our purpose is to liberate developers from rigid debugging tools and give the power of debugging back to the developers.

2

Scriptable debugging

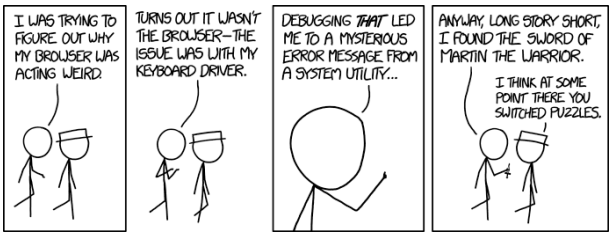


Figure 2.1: Debugging (https://xkcd.com/1722/)

In this chapter, we will provide the reader with a brief introduction to reactive programming. We refer the reader to the many available online resources if further insight is required [4]. Furthermore, we will provide the reader with a summary of the current state of scriptable debugging in literature as well as the actual usages and implementations that are available.

2.1. Why reactive debugging

We made the conscious decision to focus on *reactive* scriptable debugging when starting this thesis. We define reactive according to the definition as implemented by the Reactive Extensions (Rx) libraries. In this understanding of ”reactiveness” the emphasis is placed on higher-order, non-blocking, asynchronous event streams. Our vision of scriptable debugging is that debugger scripts simply subscribe to one or more debugging event streams that are exposed by the debuggee. Whenever an event is emitted (i.e. pushed) by any of these streams the debugger will react accordingly. Such an event-push based architecture allows the debugger to be idle when nothing relevant is happening and only react when anything of interest happens within the debuggee. As an added benefit the Rx implementation has a very large library of higher-order methods that allow many useful operations [5] to be composed in an asynchronous manner with little boilerplate on the user’s side. It is important that users can create complex debugger scripts with minimal effort and boilerplate involved, Rx allows us to achieve this.

2.1.1. Reactive programming

In this section, we will briefly introduce the concept and principle of reactive programming. Although there is no formal definition of reactive programming in general and many different opinions exist regarding what it really is, we don’t concern us with the general concept. We simply rely on the exact definition of reactive programming as originally introduced by the Reactive Extensions library initially implemented for the .NET Common Language Runtime called *Rx.NET*. This library was later ported to the Java platform as an open-source community offering called RxJava. Internally there exists a proprietary commercial/enterprise version of RxJava which was used during the execution of this thesis. We did not use the community edition because its design had been altered by the community in

a way that we did not find suitable for the implementation of a reactive scriptable debugger. Especially the particular addition and implementation of backpressure did not suite our needs. The refactoring of the open-source RxJava made it essentially unusable for true push-based data sources because it forces backpressure to be implemented. In our case this implies that the debugger must be able to apply backpressure to the event streams exposed by the debuggee. As such the debuggee's operational speed will be tightly coupled to the speed at which the debugger is able to process the events. We did not want to introduce such coupling and rather implement our system leveraging true push-based semantics. However, throughout the process of this thesis RxJava has evolved and now facilitates such push-based semantics as well. This makes it possible to refactor our system to leverage the open-source RxJava. We will leave this exercise to the interested reader.

	Single Item	Multiple Items
Synchronous	T getData()	Iterable<T> getData()
Asynchronous	Future<T> getData()	Observable<T> getData()

Figure 2.2: The four computational effects.

Figure 2.2 shows the four computational effects that developers encounter when creating systems. When developers need to perform a synchronous (i.e. blocking) action/computation they simply resort to a function/method call that returns a single value. When developers need to retrieve multiple items often they resort to a type that implement (or is similar to) the *Iterable* interface. Although the *Iterable* represents a collection of values that can be retrieved lazily, every retrieval of an entity in the collection is synchronous. The synchronous method call with a single value or an *Iterable* collection of values are said to represent the "pull model". This is in contrast to the asynchronous (i.e. non-blocking) variant on the bottom row of the table. When developers require an asynchronous computation to be performed that results in a single value they usually resort to a type or mechanism semantically equivalent to a *Future* (in Java called *CompletableFuture* [6]). This type essentially represents a callback that is invocated asynchronously whenever some computed result becomes available. The role of Rx comes into play when developers want to go beyond returning a single asynchronous value but rather want to expose a stream of multiple asynchronous values and represent these as a collection type. For this purpose, Rx implements the *Observable* [7] type that represents a stream of asynchronous events. This type is also said to represent the "push model" of computation. The *Observable* type can alternatively be described as an asynchronous collection that promotes event streams as higher-order entities (i.e. regular types rather than special constructions). Since it is represented as a collection, *Observable* offers many operations that developers are accustomed to on synchronous collections as well. For example, *Observable* streams can be transformed, filtered and composed with other streams. In addition, operations for failure handling and concurrency/parallelism are offered.

In summary, the *Observable* type allows us to represent events as asynchronous collections and easily manipulate these for our purposes. As debugging can be seen as the act of observing events in a running system, this notion of reactive programming will be very useful for our purpose of scriptable debugging. We intend to demonstrate its usefulness in this thesis.

2.2. Current state of scriptable debugging

In current software development practices, it is not so common for developers to utilize a scriptable debugging system during their debugging efforts. There aren't many tools available that allow bugs to be analyzed through scripting. Nevertheless, it seems to be quite common for developers to write automated tests when finding or resolving a bug to guard the system against future regressions [8]. It's surprising that "scripting" automated tests is a commonly accepted way to avoid bugs but the same technique is not as commonly used (nor is it commonly available) to analyze bugs in running systems. However, throughout the years there have been a decent amount of interesting publications which touch upon the topic of scriptable debugging. In this section, we will provide a chronological overview of the current state of the literature that cover this topic.

2.2.1. Related work

Dispel [9] proposed by Johnson (1981), is a run-time debugging language that allows the specification of conditional breakpoints via augmented regular expressions that are applied to event traces. Developers specify callbacks that are invoked whenever a regular expression matches. Although regular expressions are useful for scriptable debugging purposes, we intended to support a richer pattern matching system based on reactive parser combinators. These combinators allow for efficient matching of LL(1) grammars on the asynchronous debugging event streams.

The Parasight debugger [10], proposed by Aral and Gertner (1988), allows developers to instrument a debuggee program with C code at light-weight instrumentation points that are dynamically inserted and deleted. Debugging events are exposed via a regular callback based interface. The core of our debugger aims to implement similar capabilities for the JVM but based on a reactive interface so that composition of debugging event streams is facilitated in a higher-order manner.

The Dalek [11] scripted debugger by Olsson, Crawford and Ho (1990) is an extension of GDB [12]. Dalek is implemented as a separate language that, besides regular language features, provides an event based programming interface. This interface allows the registration of callback operations associated with user-specified breakpoints in a program's execution. Dalek does not utilize first-class events nor composable streams. This makes it relatively difficult for developers to express complex compositions of debugging event patterns. Moreover, as Dalek is based on actual breakpoints, it halts the execution of the debuggee upon each breakpoint and invokes the debugger in a blocking manner. This approach is often not desirable when debugging systems that are "live".

NeD [13] is an extensible debugging server that is programmable via a variation of the tcl [14] language, namely NeDtcl. The language provides 30 debugging specific functions. The intent is to run the debugging server within the debuggee. Developers can then remotely execute NeDtcl programs in the debuggee. The language is cumbersome for developers to write because it was intended as a means of communication between programs rather than between a human and a program.

DUEL [15], as described by Golan and Hanson (1992), is a high-level language that allows source-level debugging of C programs. It is implemented as an extension to GDB by adding one new command that evaluates DUEL expressions. Duel expressions are a superset of C's and include a way to query data structures using a comprehension style [16]. DUEL does not address a way to control the debuggee.

Acid [17] is a low-level dynamic (i.e. typeless) debugging language (similar to C) proposed by Winterbottom (1994). It mainly focuses on representing program state and data rather than expressing complex computations. The state and resources, such as memory, registers, variables, type information and source code are first-class citizens in the language. In addition, the language allows developers to programmatically place breakpoints and manage step commands.

Lencevicius, Hölzle and Singh (1997) [18] describe an implementation of a dynamic query-based debugger for the JVM. They propose to combine conditional breakpoints with a dynamic query-based debugger. This would allow developers to check inter-object constraints and invariants. The implementation uses a bytecode instrumentation technique which we employ as well for one version of our proposed framework. In contrast, we aim to provide developers with the same querying abilities based on high-order event streams so that we don't need to extend the language.

The Coca [19] debugger by Ducassé (1999) allows conditional breakpoints to be defined and queried. The query language, trace, exposes both control flow as well as program state (i.e. data). The querying mechanism does not require any storage; analysis is done on-the-fly in runtime memory. The execution of the queries take place in isolation, this makes it non-trivial to combine the query results (e.g. that happen over time) and build up some larger view of the system. The queries also execute synchronously; we will focus on composable asynchronous execution.

Auguston, Jeffery and Underwood (2002) [20] describe a framework for automatic debugging in which declarative specifications of debugging events (represented as event traces) are translated into execution monitors. The presented approach is to integrate event trace computations into a monitoring architecture based on a managed environment (such as the JVM or the .NET CLR). The framework allows the specification of event grammars that would scope the types of events the debugger can detect. We intend to allow the same capabilities but without introducing a separate language to describe the grammars and debuggers.

Marceau, Cooper, Krishnamurthi and Reiss (2004) [21] present a dataflow language for scriptable debugging based on FrTime [22] which in turn is based on Scheme [23]. The debugger communicates with a Java Virtual Machine to pause and resume execution, query the values of variables, and

dynamically change the debugging scripts. The implementation is based on the Functional Reactive Programming (FRP) paradigm [24]. FRP is not to be confused with the Reactive Programming (RP) approach that we employ. In short, the main difference is that RP deals with discrete event streams whereas FRP models streams using a continuous representation. The major drawbacks of this implementation are the performance of the system and the semantic differences between Java and the debugger language. For example, challenges arise with regards to datatype representations and invocation semantics. We build our debugging system based on the JVM itself so that such mismatches do not occur.

DTrace [25] is a dynamic tracing framework created by Sun Microsystems (2005). It allows users to analyze running programs in a scriptable manner. DTrace is scripted using the D programming language [26]. These scripts can access the running state of the whole system in a fine-grained manner. Although DTrace is an excellent troubleshooting tool it has certain downsides for our purposes. For example, it is very tightly coupled to the underlying operating system. As such, it is not officially supported for all operating systems. Because it provides a very low-level view of the running system, the view is not uniform across different operating systems. In our debugging tool we want to maintain the abstraction of the JVM over the underlying operating system so that debugging scripts don't have to deal with operating system specific details at runtime.

In his dissertation, Al-Sharif (2009) [27] introduces an extensible debugging architecture. The event-based debugging framework named AlamoDE provides a high-level abstraction mechanism that allows a variety of debugging tools—including source-level debuggers and custom-defined debugging tools to be created by developers. In addition, an agent-oriented debugging extension architecture named IDEA is proposed. IDEA is an extension of AlamoDE which facilitates a pluggable extension mechanism for debugging tools to execute and operate together simultaneously on the same debuggee program.

The AspectD system, proposed by Menarini, Yan and Griswold (2010) [28] aims to utilize Aspect-oriented techniques to provide a means of adding debugging code at certain pointcuts in the debuggee program. A specialized language (Java Pointcut Language) is presented that allows developers to specify the aspects for a debugging session. The aspect-oriented system was implemented for the JVM based on the Java Debugger Interface (JDI). We have used JDI in our first attempt to implement a reactive scriptable debugger and encountered the same problems as the authors of AspectD describe, mainly with regards to JDI's performance impact on the debuggee. For this reason, we decided to implement a second version based on direct bytecode instrumentation. This approach allows for better performance and avoids the need to introduce a separate language.

Expositor by Khoo, Foster and Hicks (2013) [29] is a scriptable, time-travel debugger that utilizes composable higher-order traces. For performance and composability these traces are represented by lazy data-structures. This system is also inspired by the FRP programming paradigm. It extends GDB's Python environment and uses GDB's time-travel backend UndoDB [30] to store snapshots of the program execution. As modern (distributed) systems can generate a very large amount of information with regards to their execution state across multiple program instance, we don't aim to implement a time-travel feature in this thesis. We rather focus on real-time debugging based on lazy event streams so that the technique could be adopted in live systems that run in large distributed clusters.

The Reactive programs DeBuGger (RDBG) by Jahier [31] proposes an extensible debugger based on a Read-Eval-Print Loop (REPL). The system is aimed at debugging synchronous and reactive programs, i.e. programs that continuously react to events triggered by their environment. A REPL based approach to debugging can be very useful for local debugging and could perhaps be extended to be executed remotely. The debugging system that we propose can be used from a REPL environment as well. However, a REPL based approach will likely not suffice when the aim is to debug large-scale distributed systems that run on many nodes.

3

Java Platform Debugger Architecture

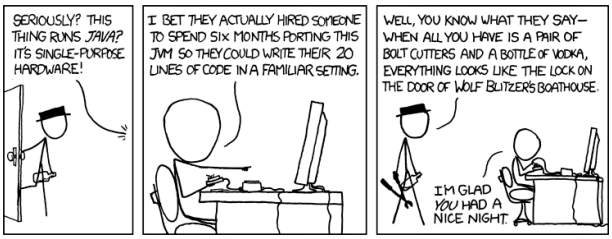


Figure 3.1: Golden Hammer (https://xkcd.com/801/)

In this chapter, we want to elaborate on the facilities that the JVM and JDK offer to developers for writing remote debugger applications. Many other platforms/languages have similar interfaces; for practical reasons we will focus on the JVM. The Java Platform Debugger Architecture (JDPA) [32] is a multi-tiered debugging architecture which allows developers to create custom tools for debugging purposes.

The JDPA consists of three layers:

- **JVM TI** - `Java VM Tool Interface`
  Defines the debugging services a VM provides.
- **JDWP** - `Java Debug Wire Protocol`
  Defines the communication between debuggee and debugger processes.
- **JDI** - `Java Debug Interface`
  Defines a high-level Java language interface which tool developers can easily use to write remote debugger applications.

Each of the above-mentioned layers are exposed as interfaces that any implementation of the Java Virtual Machine should provide an implementation for. For our purposes to define and implement a scriptable debugging system for the JVM, JDPA seems like an obvious starting point. The JVM TI and JDI layers define interesting capabilities that we can leverage to expose an even higher layer of abstraction on top of the JDPA. We will not focus on JDWP as the communication protocol between debugger and debuggee is not of particular interest to us.

3.1. JVM TI

The Java VM Tool Interface [33] is a native interface implemented by the JVM. It is a definition of the services any JVM implementation must provide for debugging. The JVM TI defines capabilities that allow tools to request information (e.g. current stack frame), actions (e.g. set a breakpoint), and notification (e.g. when a breakpoint has been hit). Debugger tools can make use of other sources

of information as well (e.g. through the Java Native Interface - JNI), but JVM TI is the source of all debugger specific information.
Specifying the VM Interface allows any VM implementer to plug easily into the debugging architecture. It also allows alternate communication channel implementations. VM implementations which do not adhere to this interface can still provide access via the Java Debug Wire Protocol (JDWP).
A concrete implementation of the JVM TI interface manifests itself as a JVM TI *agent*. When the agent is loaded the JVM can expose certain *capabilities* to the agent. These capabilities allow the agent to hook into several events that occur at runtime in the JVM. For a comprehensive list of JVM capabilities we refer to the JVM TI documentation [34].

3.2. JDI

The Java Debug Interface [35] is the highest layer of the JDPA. It provides an interface that allows external tools to access a running virtual machine's state. For example, tools can programmatically introspect which classes are loaded, the instances on the stack and heap, the memory, threads, etc. In addition to introspecting, JDI also provides explicit control over a virtual machine's execution. For example, suspend and resume threads, set breakpoints and watchpoints, notifications of method invocations, exceptions, etc.
From the scriptable debugging perspective JDI exposes a set of events that are particularly interesting. The event based interface will allow us to create a higher-level abstraction layer that exposes the JDI events as *Observable* streams. JDI exposes the following events:

- `AccessWatchpointEvent` - field access
- `BreakpointEvent` - breakpoint
- `ClassPrepareEvent` - class loading
- `ClassUnloadEvent` - class unloading
- `ExceptionEvent` - exception (caught and uncaught)
- `MethodEntryEvent` - method invocation
- `MethodExitEvent` - method return
- `ModificationWatchpointEvent` - field modification
- `MonitorContendedEnteredEvent` - a thread is entering a monitor after waiting for it to be released by another thread
- `MonitorContendedEnterEvent` - a thread is attempting to enter a monitor that is already acquired by another thread
- `MonitorWaitedEvent` - a thread has finished waiting on a monitor object
- `MonitorWaitEvent` - a thread is about to wait on a monitor object
- `StepEvent` - step completion
- `ThreadDeathEvent` - thread completion
- `ThreadStartEvent` - a new running thread
- `VMDeathEvent` - target VM termination
- `VMDisconnectEvent` - disconnection from target VM
- `VMStartEvent` - initialization of target VM

For a more comprehensive explanation regarding the semantics of the JDI events we refer to the JDI javadoc documentation [36]. To facilitate a comprehensive scriptable debugging system we want to expose all the abovementioned events to the users in a reactive manner. Unfortunately, enabling these capabilities does not come without penalties and limitations.

3.3. Proposed Architecture

To leverage the *JDPA* for our purpose of implementing a reactive scriptable debugging system we propose the following extension to the debugging architecture. Based on the *JDI* we plan to implement a layer that manages the JVM debugging events and exposes them as reactive streams. A reactive debugging script can then compose, transform, filter and eventually subscribe to these event streams. In essence, the Dx layer doesn't do much more than wrap the boilerplate code that is needed to gain access to the debugging events and expose them through a convenient reactive API. Figure 3.2 depicts the proposed architecture on a conceptual level.

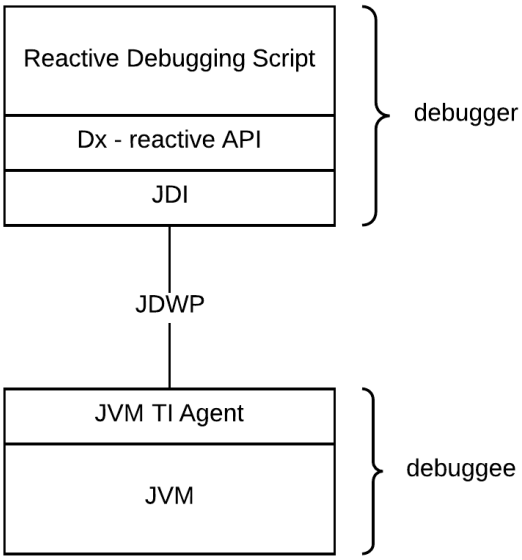


Figure 3.2: Architecture for a Reactive Scriptable Debugging System

3.4. Limitations

The events that the JDI exposes offer a wide range of applications and possibilities for scriptable debugging scripts. In this section, we will cover the limitations of using JDI for our purposes.

3.4.1. Performance impact

It is no secret that running the JVM together with the debugging agent slows down the performance. One of the goals of our scriptable debugging system is to facilitate live debugging in running systems. Therefore, it is important to understand exactly what the performance impact is and what the exact cause is of the impact. To this end we used the DaCapo benchmark suite [37]. This benchmark suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory and CPU loads.
Our approach to gaining insight into the performance impact was to first benchmark based on running the JVM without any agent. Then we would run the same benchmark using the default JVMTI agent provided as part of OpenJDK8 [38]. To accurately pinpoint the actual source of impact we modified and compiled the JVMTI agent so that only one of the following capabilities was enabled at a time:

- `breakpoints`
- `exceptions`
- `frame pop`
- `garbage collection`

- `local variables`
- `method entry`
- `method exit`
- `method order`
- `monitor`
- `single steps`
- `suspend`
- `tag objects`

For each of the ”single capability enabled” agents we would run the full benchmark as well. We ran all agent/application combinations 200 times to make sure that incidental circumstances would not have a large effect on the outcome. Since these benchmarks generated too many graphs we present the results for two of the DaCapo testing applications here. Namely the lusearch (lucene search on a large textual dataset) and jython (interpreting a pybench Python benchmark) applications.
In figures 3.3 and 3.4 we can see that the impact of enabling most of the capabilities has a relatively small impact on the performance. Unfortunately, there are certain capabilities that do have a substantial performance impact. These capabilities are: *frame pop*, *exception* and *method exit*. These capabilities are required to be able to hook in to the more insightful debugging events on the JDI level. This result also explains why runtime profiling tools can get away with minimal performance impact whereas debuggers generally cannot, they simply don’t require the high impact capabilities to be enabled.

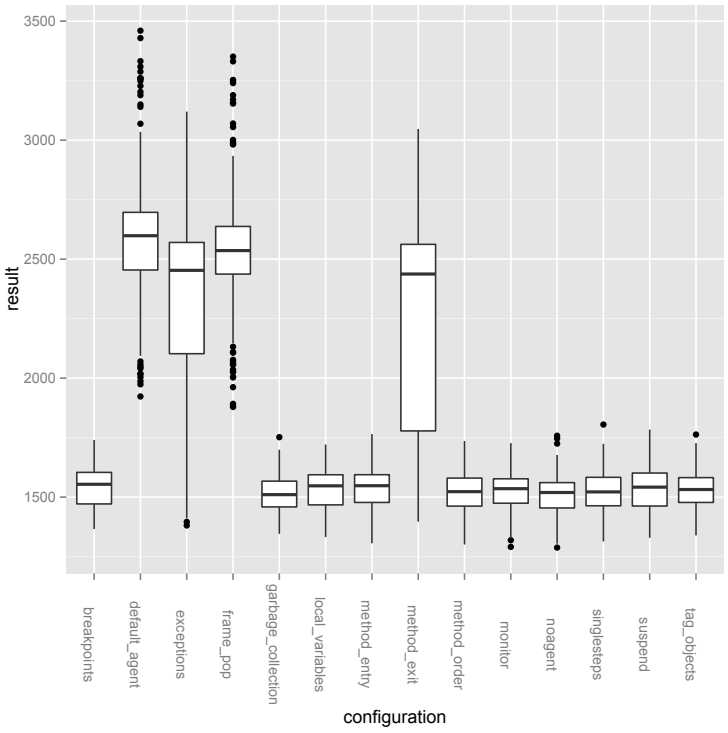


Figure 3.3: Benchmark results for lucene search application ran with several JVMTI agent configurations

To further test and verify our findings we performed additional benchmarks in which we compared three configurations: running without any agent (i.e. no capabilities enabled), running with the default agent (all capabilities enabled), and running a custom agent that was compiled to only have the capabilities enabled that show low performance impact in figures 3.3 and 3.4.

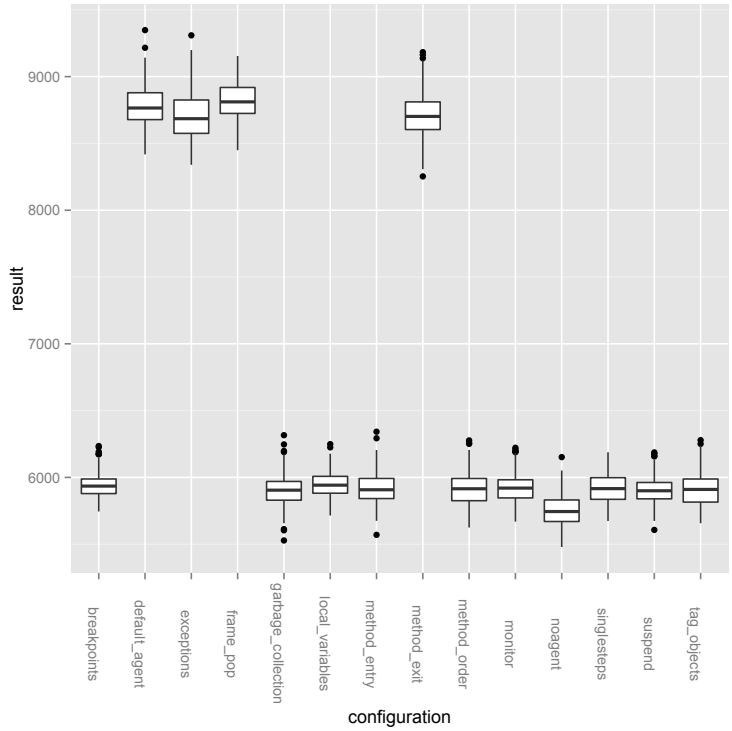


Figure 3.4: Benchmark results for jython application ran with several JVMTI agent configurations

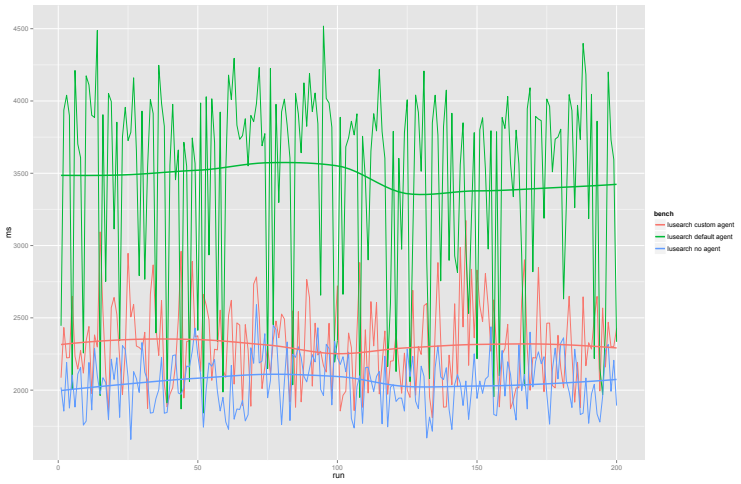


Figure 3.5: Benchmark results for lucene search executed with no capabilities, all capabilities, and only low impact capabilities enabled

Figure 3.5 shows that enabling all capabilities indeed has a relatively much stronger impact on the performance compared to only enabling the capabilities that we identified as having a low impact in our initial benchmarks.

3.4.2. Event volatility

The volatile nature of the events that JDI exposes could pose another potential issue for reactive scriptable debugging. As the JDI events do not reference persistent data structures, consumption of the debugging information is time sensitive. More concretely, the information tied to a particular JDI event becomes obsolete when the JVM is not paused and moves on to the next event. Therefore, we can only safely read the event information when the JVM is paused, otherwise we are at risk of

reading stale data or even triggering exceptions. In regular breakpoint/stepping debugging workflows and tools this is not a big limitation. But for reactive scriptable debugging this poses a considerable disadvantage as we want to handle the events in an asynchronous/non-blocking manner. A potential solution to this problem would be to copy the event related information into a persistent data structure. This would cause a large memory overhead when implemented in a generic manner. For this reason, we will leave it to the user of our reactive scriptable debugging system to filter and store information in a persistent manner.

4

Bytecode instrumentation

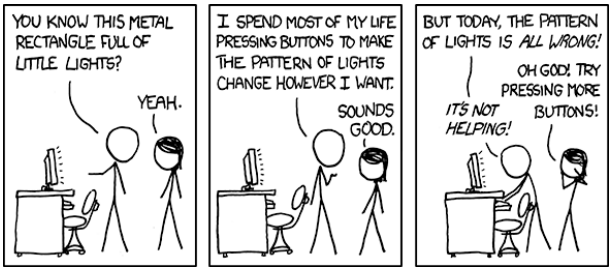


Figure 4.1: Computer Problems (https://xkcd.com/722/)

Due to our findings with regards to the performance impact of using the JPDA for our reactive debugging system, we decided to look for alternatives. One potential alternative that we came up with involves bytecode instrumentation and manipulation [39]. Instead of relying on the JVM to run in debug mode and attaching the JVMTI/JDI agent, we can instrument the debuggee with additional functionality at runtime. This additional functionality would take care of exposing the same events and information to the debugger application as JDI would. In essence our reactive scriptable debugger scripts would run as the Java agent itself. There would be a performance impact at startup time for the instrumentation to take place. But at runtime the performance impact could be on a by-need basis. This would be a great improvement over the constant impact that enabling the JVM TI capabilities has on the runtime.

4.1. Byte Buddy

Byte Buddy [40] is a code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application and without the help of a compiler. Other than the code generation utilities that ship with the Java Class Library, Byte Buddy allows the creation of arbitrary classes and is not limited to implementing interfaces for the creation of runtime proxies [1]. Furthermore, Byte Buddy offers a convenient API for changing classes either manually, using a Java agent or during a build.

Byte Buddy is written on top of ASM [42], a mature and well-tested library for reading and writing compiled Java classes. Because ASM is a low-level API that is quite error-prone to write code in, we decided to stick with the higher-level and type-safe implementation that Byte Buddy offers.

4.2. Proposed Architecture

The intent with bytecode instrumentation through Byte Buddy is to offer an agent that can be attached to a debuggee JVM. The agent will contain the debugger script that will introspect the debuggee and

[1] We did consider the possibility of leveraging dynamic proxies [41] for our scriptable debugger. The limitation of mandatory interfaces made us decide to not pursue this route.

perform the actual debugging session on it. There are two ways in which the debugger agents can be employed. When the debugging session is executed locally then the agent can simply contain a particular debugging script within the agent directly. Alternatively, if the intent is to debug a running system without the need to restart it, then a generic debugging agent can be attached which exposes the asynchronous event streams through a remote protocol like gRPC [43]. Various debugger scripts could then hook onto the remotely running agent and subscribe to the exposed event streams.

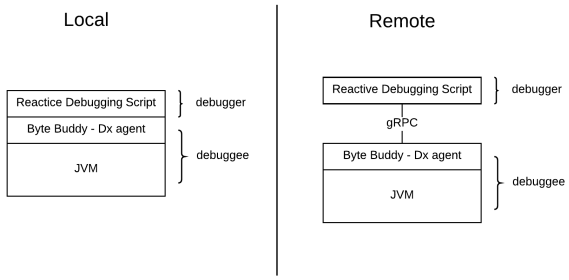


Figure 4.2: Architecture for a Reactive Scriptable Debugging System based on Byte Buddy

The following snippet illustrates how Byte Buddy can be used to set up an agent that intercepts for example all method entry, exit and exception events (as exposed by JDI as well). The agent is created through a builder that injects itself eagerly to any type that is not filtered out (using the *isIgnored* predicate). For each remaining method, a method interceptor is registered. This interceptor essentially acts as an aspect [44] wrapping all method calls with additional logic before and after the method invocation. In our case the interceptor simply does three things. It emits a method entry before the invocation. Wraps the invocation in a try/catch block and emits an exception event when a failure occurs (and re-throws the exception to let the debuggee handle it as intended). Finally, it emits a method exit event after the invocation. All these three events are emitted on separate event streams.

```
new AgentBuilder.Default()
  .with(AgentBuilder.InitializationStrategy.SelfInjection.EAGER)
  .type(t -> !isIgnored(t.getActualName()))
  .transform((builder, typeDescription, classLoader) ->
    builder.method(ElementMatchers.any())
          .intercept(MethodDelegation.to(MethodAccessInterceptor.class)))
  .installOn(instrumentation);
```

The actual event streams for method entry, method exit and exception events are generated in the *MethodAccessInterceptor*. This interceptor can be dynamically configured to be enabled or disabled. As such, when it is disabled there is minimal overhead that the Dx agent adds to the debuggee. The overhead is mainly in the invocation of the *intercept* method. Although there's still overhead, this overhead is much smaller compared to the impact that the JDI agent introduces as discussed in the previous chapter. We have performed various benchmarks (again leveraging the DaCapo suite) using this agent. Figure 4.3 shows one of those benchmarks. When we configure the agent to emit events for all method invocations the overhead is still considerable during the debugging session. However, when we only emit events for certain parts of the debuggee code that we are interested in, the performance impact drops significantly. This confirms our main intended benefit of being able to filter which code to instrument on a more granular level. Additionally, this approach allows us to run the agent without fearing a major performance impact when we're not using the Dx debugger at all. Of course, there is additional overhead when the debugger script actually subscribes and the events are indeed propagated. Performance impact during the debugging session is an acceptable situation, as long as the impact is not present before and after the debugging session. A simple implementation for the interceptor is shown in the following code snippet.

```
public class MethodAccessInterceptor {

  // the three asynchronous event streams for the different events
  private static Subject<MethodEntry> methodEntries = new Subject<>();
  private static Subject<MethodExit> methodExits = new Subject<>();
  private static Subject<Throwable> exceptions = new Subject<>();

  // the actual interceptor
  @RuntimeType
  public static Object intercept(@Origin Method method,
                                 @SuperCall Callable<?> zuper,
                                 @AllArguments Object[] args) throws Exception {
    if (isDisabled(method)) {
      return zuper.call();
    }
    final StackTraceElement[] stackTrace =
          cleanStackTrace(method, Thread.currentThread().getStackTrace());
    methodEntries.onNext(new MethodEntry(method, args, stackTrace));
    try {
      final Object result = zuper.call();
      methodExits.onNext(new MethodExit(method, result, stackTrace));
      return result;
    } catch (final Throwable t) {
      exceptions.onNext(t);
      throw t;
    }
  }

  ...

  public static Observable<MethodEntry> getMethodEntries() {
    return methodEntries.asObservable();
  }

  public static Observable<MethodExit> getMethodExits() {
    return methodExits.asObservable();
  }

  public static Observable<Throwable> getExceptions() {
    return exceptions.asObservable();
  }

}
```

The interceptor essentially hooks into all method invocations in the JVM for which the classes are instrumented. Dx is able to filter the instrumentation of these classes as well as the invocation of the interceptor based on a by-need basis. The interceptor receives as input a reference to the methods metadata, the *Callable* representing the virtual method to invoke the execution, and the arguments that were supplied. This information is exactly what we need for our debugging events as well. In the interceptor, we invoke the *Callable* in an aspect-oriented manner, i.e. we surround the invocation by additional logic to emit events for method entries, exits and exceptions. One additional thing we need to account for is that if Dx is configured to expose the callstacks (this is optional as additional performance and memory impact is involved) we need to ”clean up” the stack trace because the interceptor itself will appear throughout the stack trace. We want to represent the stack trace as if no interceptor has been executing in the system, therefore we need to filter the data structure before sending it to the debugger.

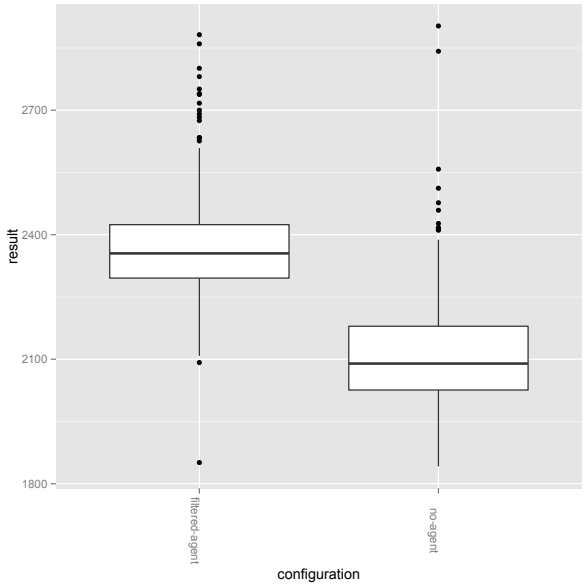


Figure 4.3: This graphs depicts the runtime comparison between running the Lucene Search benchmark without any agent and with a filtered Dx agent that is configured to introspect the query manager section of the application.

4.3. Limitations

There are certain limitations and complications involved when going down the path of implementing Dx based on bytecode instrumentation. One of the limitations is that the actual bytecode transformations are only allowed to happen once during the class load time. After this time, no re-transformations are allowed. This puts a certain limitation on the dynamic re-configurability of the Dx agent. Without this limitation, the agent would be much more flexible in terms of changing behavior without the need to restart the JVM. More importantly, if dynamic re-transformations would be possible, the Dx could be implemented in a way that would allow zero performance overhead when the debugger is not actively used. The bytecode transformation could take place only when the debugger subscribes to the agent's event streams. Unfortunately, this seems to be impossible to achieve due to the mentioned limitation.

As mentioned earlier, when utilizing the method interceptors in an aspect-oriented style all the stack traces contain the interceptor calls itself for each instrumented method. If the debugger has the need to inspect the callstacks the Dx agent will have to perform a stack "clean up" by filtering out all the stack entries that originate from the agent itself. This means that there is a performance overhead that grows with every additional instrumented method. Although in many cases this can be limited, it is something to keep in mind when trying to eliminate overhead. More importantly, the debugger agent adding frames on the stack could be problematic for particular systems and bugs. Essentially, the stack is artificially enlarged by the usage of the Dx agent. It is imaginable that certain (likely rare) bugs could be affected with such a change in behavior in the JVM internals.

Another limitation to point out is that the actual implementation of the bytecode instrumentation is very specific to a particular version of the JDK and the JVM. As new versions of the Java ecosystem are published this can have a big impact on the Dx implementation. The Byte Buddy library already tries to abstract over the details of the bytecode so there is some level of indirection offered. Nevertheless, even Byte Buddy breaks when the class formats or the runtime execution changes significantly. For example, we used Java 8 during this thesis but with the release of Java 9 the Byte Buddy library version that we used is completely broken. The main reason for breaking is the introduction of the module system, a.k.a. project Jigsaw [45]. As Byte Buddy is an active project, it is already updated to support Java 9 directly, in the interest of time we have remained on Java 8. Another thing to keep in mind is that with these JVM changes the API and semantics of the Byte Buddy implementation also change. This means that the Dx implementation will need to adopt the new implementation eventually as well.

Finally, to reiterate, our current implementation has (for our purposes) acceptable overhead when the agent is used but the debugger is not running. This does not mean that the overhead is acceptable for large scale systems running in production. The performance overhead still needs to be investigated

further, and our hope is that someone will continue the work to implement an actual zero overhead implementation of a scriptable debugging system.

5

Reactive scriptable debugging

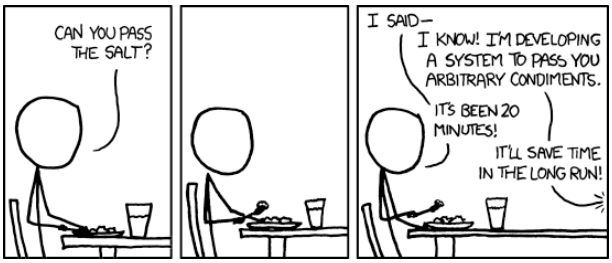


Figure 5.1: The General Problem (https://xkcd.com/974/)

In this chapter, we intend to present the two implementations that we created over the course of this thesis project. The first implementation based on JDI and the second implementation based on bytecode instrumentation using Byte Buddy as explained in previous chapters. We will show how the Dx scriptable debugging system can be used by showing practical examples of custom tools that can be implemented. Although the two Dx implementations differ in the way they are initialized and setup, the actual API that exposes the debugging event streams to the debugger scripts is very similar. Therefore, we will demonstrate examples arbitrarily using either implementation to save the reader from repetition. We will also present an additional library (RxParsec) that we decided to implement for the purpose of real-time pattern-matching on asynchronous event streams. We deemed such a library to be very useful to express certain debugging scenarios in a concise manner. Finally, we will shortly elaborate on the graphical IDE plugin that we developed which integrates the Dx system with the Intellij IDEA [46] development environment. We developed this IDE plugin to explore the possibilities of making scriptable debugging more accessible and easier for developers by hiding many of the boilerplate code involved with setting up the debugging event streams. For this purpose, the plugin employs code generation techniques to alleviate the developer as much as possible.

5.1. Dx event streams

The API of Dx is built upon the Reactive Extensions library (Rx). The intent of the API is to expose the debugging event streams as first-class entities. This means that the debugging events are represented using the highly composable, asynchronous data-structure called the *Observable*. To this end, the Dx API exposes methods that have a return type that is conceptually based on the following generic stream type: *Observable<? extends DebuggingEvent>*. In practice, the generic type parameter of the stream will be specific to the particular events being exposed, e.g. *MethodEntry*, *MethodExit*, *Exception*, etc. Such a representation allows a debugger script to subscribe to the events streams and react with certain actions whenever an event is emitted. The following snippet shows how to subscribe to the streams:

```
debuggingEventStream.subscribe(
  event -> processEvent(event), // onNext callback
  error -> processError(error), // onError callback
  () -> processCompletion() // onCompleted callback
);
```

The above snippet shows that the debugger can subscribe to the event streams by registering callbacks for the three different events that can occur in an *Observable* (event, error and stream completion). The debugger can choose to subscribe to a stream using any permutation of the three callbacks based on its use cases. Of course, this is the simplest example that would allow a debugger script to tap into the debugging events. One of the great benefits that the usage of Rx offers us is the comprehensive collection of composable operators that are defined for *Observable* streams. Using these operators, debugger scripts are able to filter, transform, combine, and in general orchestrate complex expressions based on the event streams that Dx exposes. In the remainder of this chapter we will present examples that demonstrate these capabilities.

5.2. Basic examples (CLI)

In this section, we'll show how to initialize Dx scripts and we'll also show how certain basic debuggers can be scripted. As our system natively supports Java as well as Scala we use the languages interchangeably.

5.2.1. Running Dx scripts

To setup a debugger script that hooks into a given debuggee via DxBB a developer needs to do two things. The first step is to create a debugger that implements the *Debugger* interface. This interface requires a single method *debug()* to be implemented. In the implementation of the *debug()* the developer has access to the various event streams that Dx exposes. The streams can be transformed, filtered, combined and eventually subscribed to. Once this class has been implemented all that remains is to instantiate the debugger and invoke the *setup()* method from within the debuggee. With this approach the developer doesn't need to create an actual standalone agent, the agent will be created and loaded at runtime through the *setup()* invocation. If altering the debuggee's code is not an option, e.g. in a deployed system, the developer can also prepackage the debugger agent and attach it to the JVM at startup time. The following example code illustrates a minimal setup that simply subscribes to the event streams and prints out every emission of an event.

```
public class TestDebuggeeApp {
   public static void main(String[] args) {
      new TestDebugger().setup();
      // code executed after this point will be visible to the debugger
   }
}

// A simple debugger that prints to the console upon certain events
class TestDebugger implements Debugger {
  @Override
  public void debug() {
    this.methodEntries()
      .subscribe(method -> System.out.println("[Dx] method entry: " + method));

    this.methodExits()
      .subscribe(method -> System.out.println("[Dx] method exit: " + method));

    this.exceptions()
      .subscribe(t -> System.out.println("[Dx] exception: " + t.getMessage()));
  }
}
```

5.2.2. Scripted breakpoints and state inspection

The following example shows how simple it is to create a script that sets a breakpoint in a program and then automatically invokes step requests to the JVM. Upon each step, some state, in this case a variable *i*, is inspected and printed to the debugger console.

```
// set a breakpoint
val breaks = DxScala.setBreakpoint(events, className, line)

// step through the rest of the method
val steps = breaks.flatMap(brk => {
  events.createStepRequest(brk.thread(), StepRequest.STEP_LINE,
      StepRequest.STEP_OVER)
    .filter(e => e.className == className)
    .toObservable
})

// print value of i as we step through the code
steps.subscribe(step =>
  println(s"> ${step.location.lineNumber}: i = ${step.getValue("i")}"))
```

5.2.3. Scripted state manipulation

Often debugging leads developers to ask "what if" questions regarding the state of their programs. For those cases, it is useful to allow developers to manipulate the state of the debuggee from the debugger scripts. The following snippet shows a simple example in which some particular state of a simple program is selectively mutated based on its value during execution.

```
DxScala.setBreakpoint(events, className, line).subscribe(e => {
  val x = DxScala.getValue(e, "state.x").asInstanceOf[IntegerValue]
  // print state
  println("Dx> state.x: " + x)
  // check for a certain condition
  if (x.value() % 13 == 0) {
    println("Dx> Found invalid state bug!! Mutating state to fix...")
    // manipulate state
    DxScala.setValue(e, "state.x", vm.mirrorOf(x.value() + 1))
  }
})
```

5.3. Scripted debugging tools

In this section, we want to demonstrate some of the possible applications of reactive scriptable debugging. In particular, we focus on the ability of developers to quickly create custom debugging tools that are reusable. Creating custom tools allows a great deal of efficiency and agility for the developers as these tools can be created to automate many debugging and monitoring/profiling tasks. We will try to demonstrate (albeit subjectively) the relative ease with which Dx and the exposed debugging event streams allow such tools to be created.

5.3.1. Example: Instance graps (DxJDI)

The following example creates a real-time visualization of the number of instances per loaded class at runtime. An animated histogram shows the number of instances in the debuggee program. The debugger script subscribes to the method entry event stream and samples it periodically. Upon each sampled method entry event the script retrieves the set of all loaded classes in the virtual machine. This set of classes is zipped together with the set of respective number of instances per class. For visual convenience, all classes with less than a configurable number of instances are filtered out. The debugger script switches to the UI thread and updates the data of the histogram. Figure 5.2 shows the result of this debugger script.

```
events.createMethodEntryRequest().toObservable
  .sample(Duration(millisPerFrame, TimeUnit.MILLISECONDS))
  .map(e => {
    val classes = virtualMachine.allClasses()
    val counts = virtualMachine.instanceCounts(classes)
    counts.zip(classes).filter(x => x._1 > instanceThreshold)
  })
  .observeOn(JavaFxScheduler())
  .subscribe(xs => {
    series.getData.clear()
    xs.foreach(x => series.getData.add(new XYChart.Data[Number, String](x._1,
        x._2.name())))
  }, _ => Unit)
```

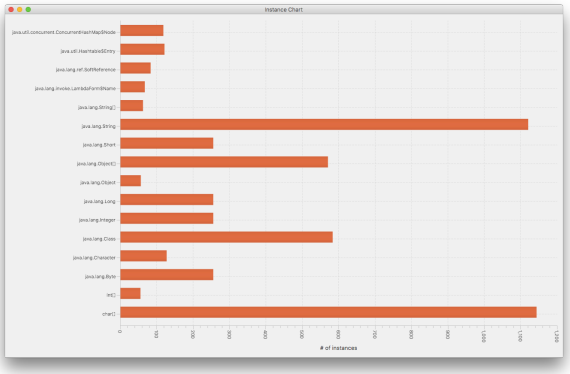


Figure 5.2: Visualizing the number of instances per loaded class in real-time

5.3.2. Example: Resource usage (DxJDI)

A common issue that developers face is memory or resource leakage. Resource leaks are often caused due to unexpected execution paths in the code that lead to lingering references to certain assets. For example, opening a file but not closing the handle to the file in all possible execution branches of the program. The following example shows a debugger script that allows a developer to easily visualize how many file handles a program holds at any point in time. The visualization allows developers to easily discover whether there is a leak in their program.

The example starts by retrieving the stream of call preparation events for a *File* class, it then places a breakpoint in the constructor to create a stream that emits an event every time a *File* entity is instantiated. To track every time a file handle is closed, a method exit stream is created for the *Close* method. These two event streams combined allow a graphical depiction of the number of open file handles at any moment in time as depicted in Figure 5.3.

```
// Get stream of class prepare events for the File class
val file: Observable[ClassPrepareEvent] = events.createClassPrepareRequest()
  .filter(e => e.className == "examples.resources.File")
  .toObservable.take(1).share()

// Get stream of all calls to File("name") by putting a breakpoint inside the
    constructor.
val alloc: Observable[Open] = file.flatMap(classPrepare => {
  events.createBreakpointRequest(classPrepare.referenceType().location(line))
    .toObservable
    .map(breakPoint => {
      val frame = breakPoint.thread().frame(0)
        Open(frame.thisObject().uniqueID(),
            frame.visibleVariables.map(frame.getValue).head)
```

```
    })
}).share()

// Get stream of all calls to Close() by intercepting MethodExit out of Close.
val dealloc = events.createMethodExitRequest()
  .filter(exit => exit.className == "examples.resources.File")
  .toObservable
  .filter(exit => exit.method().name() == "Close")
  .map(exit => {
    val frame = exit.thread().frame(0)
    Close(frame.thisObject().uniqueID())
  }).share()
```

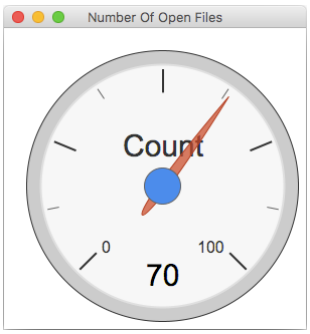


Figure 5.3: Visualizing the number of open File handles, to inspect for resource leaks

5.3.3. Example: Tracking method calls (DxBB)

The following example shows how to track all calls and the given parameters to a given method. In this particular example we hook into two different implementations of a Fibonacci number computing method. One implementation uses simple recursion whereas the other, although also recursive, is more optimized with a memoization strategy. We want to compare their behavior by tracking how often the method is invoked with each parameter. For both the implementations we create two event streams that emit upon each method entry event in the respective Fibonacci implementations. The example then subscribes to these streams and visualizes the method calls in a histogram as depicted in Figure 5.4. A simple change in the debugger script would allow for alternative visualizations, for example the real-time updated pie chart as shown in Figure 5.5.

```
final Observable<MethodEntry> fibCalls =
   methodEntries().filter(methodEntry ->
       methodEntry.getMethod().getName().equals("fib"));

slowFibCalls = fibCalls
  .filter(methodEntry ->
     methodEntry.getMethod().getDeclaringClass().getCanonicalName()
    .equals(SlowFibonacci.class.getCanonicalName()))
  .map(methodEntry -> Integer.class.cast(methodEntry.getArgs()[0]));

fastFibCalls = fibCalls
  .filter(methodEntry ->
     methodEntry.getMethod().getDeclaringClass().getCanonicalName()
    .equals(FastFibonacci.class.getCanonicalName()))
  .map(methodEntry -> Integer.class.cast(methodEntry.getArgs()[0]));

slowFibCalls.observeOn(JavaFxScheduler.get()).subscribe(i -> {
  final XYChart.Data<String, Number> dataPoint = (XYChart.Data<String, Number>)
     slow.getData().get(i);
  final int n = dataPoint.getYValue().intValue();
```

```
  dataPoint.setYValue(n+1);
});

fastFibCalls.observeOn(JavaFxScheduler.get()).subscribe(i -> {
  final XYChart.Data<String, Number> dataPoint = (XYChart.Data<String, Number>)
      fast.getData().get(i);
  final int n = dataPoint.getYValue().intValue();
  dataPoint.setYValue(n+1);
});
```

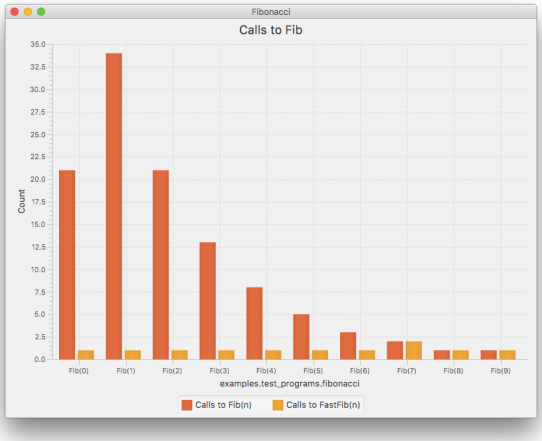


Figure 5.4: Visualizing the number of method calls per input parameter in a real-time histogram

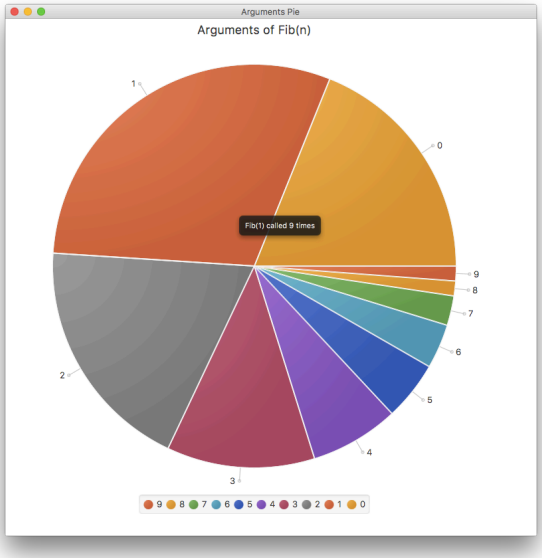


Figure 5.5: Visualizing the number of instances per loaded class in a real-time pie chart

5.3.4. Example: Flame graps (DxBB)

Flame graphs allow the visualization of the most frequent code-paths of a debuggee to be identified quickly and accurately. [47] With Dx we provide an example debugger script that allows a real-time flame graph visualization of a running debuggee. The x-axis shows the stack profile population, and the y-axis shows stack depth. Each rectangle represents a stack frame. The wider a frame is, the more

often it was present in the stacks. The top edge shows what is on-CPU, and beneath it is its ancestry. The colors are not significant, picked randomly to differentiate frames. An example of the visualization is depicted in Figure 5.6.

The debugger script that creates the real-time flame graphs is essentially governed by a single reactive expression as depicted below. The expression retrieves the stream of all method entry events and maps those events to their respective stack traces. Via the *scanLeft* operator it accumulates all stacks to a single data structure called a *CallStack*. This accumulated call stack contains all the information required to render the flame graph. The resulting stream is sampled at the rate with which the graphical interface will be refreshing, as we only need the real-time accumulated *CallStack* per rendered frame. Then a function is mapped to the stream that actually renders the flames and returns a set of coordinates per stack entry so that tooltips can be added. This is done by combining the resulting streams with a stream of mouse move events and subscribing to it with a function that renders a tooltip with additional metadata per stack entry.

```
final Debugger debugger = Debugger.simple();
debugger.setup();

debugger.methodEntries()
  .map(this::getStackEntries)
  .scanLeft(new CallStack(), CallStack::update)
  .sample(millisPerFrame, TimeUnit.MILLISECONDS)
  .observeOn(JavaFxScheduler.get())
  .map(cs -> createFlames(CallStack.root, 0, sceneHeight - blockHeight,
      sceneWidth, gc))
  .withLatestFrom(mouseMoves(canvas), Pair::of)
  .subscribe(pair -> handleTooltip(pair.fst, pair.snd, tooltip, root));
```

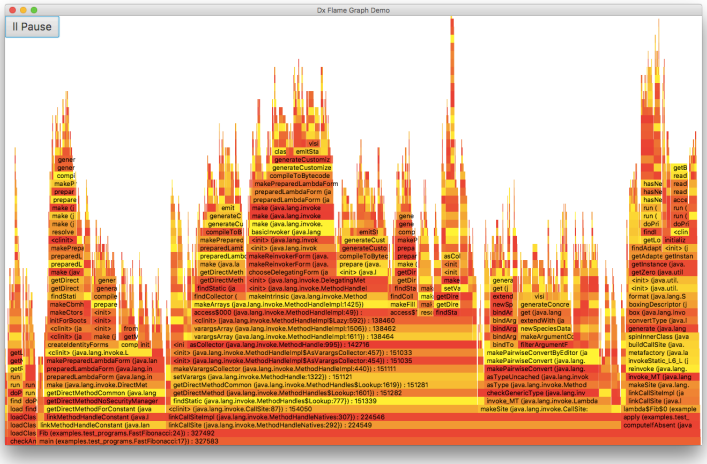


Figure 5.6: Real-time flame graph visualization

5.4. Pattern matching

Exposing debugging information as asynchronous streams using Rx allows us to e.g. filter, transform and combine these streams. What Rx does not directly implement is a way for the user to filter streams based on certain patterns. For example, if the user of our library would like to observe a specific chain of events in the debuggee it would be non-trivial to implement with Rx alone. For our scriptable reactive debugging system, we want to allow the user to leverage pattern matching in the deubgger scripts. Our system should fire an event when a specified pattern is matched on the debugging event streams. Since Rx does not offer such functionality on the Observable type, we have implemented it ourselves.

5.4.1. RxParsec

Our implementation of a pattern matching system for asynchronous systems is based on the concept of parser combinators. We have essentially ported and adjusted the implementation of the parsec library [48]. Parsec implements a parser combinator library for synchronous data structures. The reason for picking parsec is that it allows for a space and time efficient implementation. Additionally, parsec implements almost all of its combinators without the need for a "look-ahead", this makes it particularly useful for adoption with asynchronous data structures. Efficient parsers can be written in a concise manner that will not cause severe performance and memory overhead when limited to parsers for LL(1) grammars. This means that the users of our library will be able to specify any pattern that can be expressed by such grammars without worrying too much about the added overhead.

We have implemented the same set of combinators as offered by the parsec API. We had to implement the internals in a different way than what is described in the original paper in order to adapt the system to asynchronous streams. It is worth noting that our RxParsec library can be used for pattern matching on any Rx stream; the library is generic and thus not limited to debugging purposes.

Due to the compositional nature of parser combinators, basic parsers can be employed to build more complex parsers which can be used to build parsers that are even more complex. This will allow our users to define a sophisticated set of parsers for common patterns that can be combined and reused.

5.4.2. Example: Debugging with patterns

The following example shows how to leverage pattern matching on *Observable* streams to detect certain execution patterns. In this example, we want to detect whether some part of our system attempts to write to a *File* that has already been closed earlier. In other words, we wants to detect invalid file writes. Without a pattern matching system, we would need to keep track of some global mutable state in the debugger script to be able to detect such occurrences. Using RxParsec together with Dx allows us to write the following declarative debugger script that has no global mutable state at all. The example also shows how complex parsers can be created by combining simple parsers. First, a parser is declared that matches on any invocation of the *File.close* method. Secondly, we create a parser that matches on any invocation of the *Files.write* method for some given *File* instance. Finally, we combine these two simple parsers to a parser that can detect whether the debuggee attempts to write to a file that was closed earlier. We use a monadic *for* comprehension for our expression. We first match on a file closed event, which returns to us the particular file in question. Finally, we use the built-in RxParsec combinators *endBy* and *many* (discussed in the following sections) to match whether at some point a write event occurs to the file that was closed.

```
// a parser that matches all invocations of the File.close method
def fileClosedEvent: Parser[DxEvent, File] =
  satisfy(dxEvent => dxEvent.clazz == File.class && dxEvent.method.name == "close")

// a parser that matches all invocation of the Files.write method for a given File
def fileWriteEvent(file: File): Parser[DxEvent, File] =
  satisfy(dxEvent =>
            dxEvent.clazz == Files.class
        && dxEvent.method.name == "write"
        && DxScala.getValue(dxEvent, "path").toFile == file)

// combine the above parsers to match for invalid writes
def invalidWrites: Parser[DxEvent, File] =
  for {
    file <- fileClosedEvent
    _ <- endBy(fileWriteEvent(file), many(anyEvent))
  } yield file

// execute the parser on the stream of method entry events
invalidWrites.run(dx.methodEntries).subscribe(file =>
    println("Invalid write: " + file))
```

This is just a simple example of the usage of pattern matching in debugging scripts. RxParsec has many more combinators built-in that would allow developers to perform complex pattern matching for

the detection of event sequences that otherwise would be very hard to perform.

5.4.3. RxParsec implementation

As stated before, our implementation of parser combinators that work with Rx is strongly based on the implementation of parsec [48]. Parsec is a "monadic parser combinator library". It is designed to be efficient in terms of space and time usage. In case of a parse error it is able to report both the position of the error as well as all grammar productions that would have been legal at that point in the stream. The compositional nature of its combinators allows complex parsers to be created from simpler ones. In fact, most of the default combinators that parsec provides are compositions of other simpler combinators. The three main combinators that "bootstrap" the rest of the Parsec implementation are the monadic bind **(»=)** (a.k.a. flatMap), a method to perform a predicate check **satisfy** and a conditional combinator **(<|>)** (i.e. OR).

```
// a stream parser with Observable<S> as input and Observable<R> as output
public class Parser<S,R> {

  // a function 'Observable<Optional<S> -> Observable<Parse<S,R>>' representing
      the actual parser
  private Func1<Observable<Optional<S>>, Observable<Parse<S,R>>> p;

  // execute the parser on a given input Observable<S> yielding an Observable<R>
  public Observable<R> parse(Observable<S> os) { ... }

  // satisfy combinator
  public static <S> Parser<S,S> satisfy(Func1<S,Boolean> pred) { ... }

  // monadic return
  public <T> Parser<S,T> to(T t) { ... }

  // monadic bind
  public <T> Parser<S,T> flatMap(Func1<R,Parser<S,T>> f) { ... }

}
```

The following abstract data type represents the **Parse** class that is used to propagate events and manage state in the reactive parser combinators. This type is exclusively used internally to maintain the state of the composed parser. Instances of this type are therefore never exposed to the users of the library. The user is only exposed to the conceptual parser which is represented as a function that parses and transforms a source stream (**Observable<S>**) into a result stream (**Observable<R>**). In fact, only the core combinators mentioned in the snippet above directly refer to this type all other combinators are composed based on them and thus needn't refer to the **Parse** type directly. This allows easy refactoring of the internals of the libraries as the internal representation can be changed easily while only impacting a handful of operators.

```
// the abstract data type
public abstract class Parse<S,R> { ... }

// type constructor for a parsing result
class Res<S,R> extends Parse<S,R> {
  private R r;
  ...
}

// type constructor for a source element
class Src<S,R> extends Parse<S,R> {
  private Optional<S> s;
  ...
}
```

```
// internal flag for the completion of a successful parse
class Win<S,R> extends Parse<S,R> { ... }

// internal flag for a failed parse, carries a failure mesage
class Fail<S,R> extends Parse<S,R> {
  private String msg;
  ...
}
```

The following snippet shows some of the derived combinators implemented in RxParsec. Notice that they do not refer to any implementation specific construct, only the core combinators. The implementations of these derived combinators are for the most part a direct port of their parsec counter-parts, simply translating the logic from Haskell to Java.

```
public static <S> Parser<S,S> oneOf(List<S> xs) {
  return satisfy(xs::contains);
}

public static <S,R> Parser<S,List<R>> many(Func0<Parser<S,R>> p) {
  return many1(p).or(() -> unit(new LinkedList<R>()));
}

public static <S,R> Parser<S,List<R>> many1(Func0<Parser<S,R>> p) {
  return p.call().flatMap(x -> many(p).flatMap(xs -> unit(x,xs)));
}

public static <S,R,O,C> Parser<S,R> between(Func0<Parser<S,O>> open,
    Func0<Parser<S,C>> close, Func0<Parser<S,R>> p) {
  return open.call().andThen(p).flatMap(xs -> close.call().andThen(() ->
      unit(xs)));
}

public static <S,R,Sep> Parser<S,List<R>> sepBy1(Func0<Parser<S,R>> p,
    Func0<Parser<S,Sep>> sep) {
  return p.call().flatMap(x -> many(() -> sep.call().andThen(p)).flatMap(xs ->
      unit(x,xs)));
}

public static <S,R,Sep> Parser<S,List<R>> endBy(Func0<Parser<S,R>> p,
    Func0<Parser<S,Sep>> sep) {
  return many(() -> p.call().flatMap(x -> sep.call().andThen(() -> unit(x))));
}
```

With this implementations of parser combinators for Rx Observables we can support pattern matching on event streams. This will be very useful for our scriptable debugging purposes as the act of debugging often boils down to searching for patterns in the execution of a program. Developers can now formulate the patterns in a script and automate the search instead of looking for them manually.

5.5. IDE plugin

As we were developing examples for Dx we noticed that many common patterns revealed themselves. Many debugger scripts contained similar pieces of code to setup the debugging event streams. In order to avoid boilerplate and needless repetition we decided to experiment with an IDE plugin that would allow developers to easily generate the setup code for their debugging scripts. To this end, we created a plugin for the Intellij IDEA development environment. The plugin allows the user to visually select certain types of events in the code, the plugin would generate the necessary debugging code that is needed to setup the event streams. The IDE allows developers to easily generate the code to create event streams for setting breakpoints, method entry/exit events, field notifications, class load/unload events, exceptions, threads and monitors. Such a plugin lowers the barrier to entrance to scriptable

debugging and allows developers to focus on their bugs rather than the tools. The Dx IDE plugin is depicted in Figure 5.7.

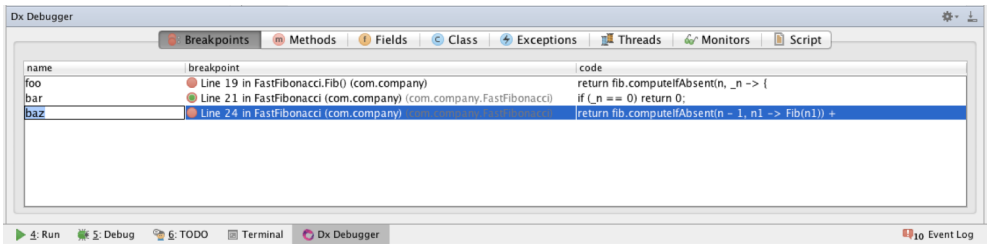


Figure 5.7: Dx Intellij IDEA plugin

5.5.1. Code generation

The main part of the Dx IDE plugin is the code generation module. It essentially offers a UI component in the IDE that allows users to start leveraging scriptable debugging sessions quickly. The user can create, modify and delete the following event streams relating to the following entities through the IDE:

- breakpoints
- methods
- fields
- classes
- exceptions
- threads
- monitors

The plugin allows users to select the event streams that he/she would want to expose in their scripted debugging session. When the user makes a selection the plugin automatically generates source files that expose the particular streams of interest. These source files contain all the ”boilerplate” that goes with setting up the event streams. The user can simply access these streams directly by means of a simple function call. In addition, the user is also able to easily access the generated streams from a REPL (e.g. the Groovy or Scala REPL as supported by Intellij IDEA). The REPL allows a more interactive debugging session to be easily scripted for quick verification and simple debugging. For more complicated debugging sessions, it would be more practical to use a separate script. The following snippet shows an example of the code that the plugin generates when the user sets a method entry and a method exit breakpoint. The name of the method is *fib*, the plugin generated two methods: *fibEntry* and *fibExit*. The implementations of these methods simply hide all the boilerplate that is required to set up the event streams for these events. The user can now simply call these methods from a debugging script or a REPL session and directly get access to the streams emitting *MethodEntryEvent* and *MethodExitEvent* instances.

```
public class DxFibonacci {
   private final com.sun.jdi.VirtualMachine vm;
   private final Events events;

   public DxFibonacci() {
      vm = com.applied_duality.dx.VirtualMachine.launch(”com.company.Fibonacci”,
            System.getProperty(”user.dir”) + ”/out/production/DxPluginTest:” +
                System.getProperty(”user.dir”) + ”/lib/Dx-Scala.jar”);
      events = VirtualMachineExtensions.events(vm);
   }
```

```
    public void start() {
        events.connect();
        vm.resume();
    }

    public Observable<? extends MethodEntryEvent> fibEntry() {
        MethodEntryRequest methodEntries = events.createMethodEntryRequest();
        methodEntries.addClassFilter("com.company.Fibonacci");
        return EventRequestExtensions.toObservable(methodEntries).asJavaObservable()
                .filter(entry -> entry.method().name().equals("Fib"));
    }

    public Observable<? extends MethodExitEvent> fibExit() {
        MethodExitRequest methodExits = events.createMethodExitRequest();
        methodExits.addClassFilter("com.company.Fibonacci");
        return EventRequestExtensions.toObservable(methodExits).asJavaObservable()
                .filter(exit -> exit.method().name().equals("Fib"));
    }
}
```

When the developer changes the original code in the debuggee it is possible that the previously generated debugging classes are no longer valid. Ideally, the Dx plugin would automatically update the previously generated debugging code to reflect the latest state of the application. Since this is quite a time-consuming feature to implement we chose to leave this capability for future work. In the plugin's current state the user can simply trigger a full regeneration to execute on demand.

6

Postlude

Figure 6.1: Thesis Defense (https://xkcd.com/1403/)

In this chapter, prior to concluding this thesis we will present some of our ideas and considerations for future work.

6.1. Future work: Debugging production

In this thesis we have discussed the possibilities and merits of scriptable debugging from a local debugging perspective. We have presented several practical examples of scripts and tools that can be created in order to gain insights into complex systems. The scriptable debugging approach that was presented can be employed in a local setting as well as in a remote setting. Nothing is stopping us from hooking the scriptable debugger into a running system (that is running with either the JDI agent or the generated ByteBuddy agent) in order to do ”live” debugging. In this section we want to explore the possibilities of debugging systems that are running in ”production” [49]. To this end we will present reference debugging architectures that could be employed to debug the most common software architectures employed in the industry nowadays. Namely, monolithic architectures and cloud based microservices architectures [50].

6.1.1. Debugging monoliths

Nowadays many monolithic software architectures (especially the ones operating at large scale) are often refactored in favor of microservices architectures [51]. Nevertheless, monoliths are still quite common (especially in legacy applications). Therefore, many developers still find themselves developing and debugging systems that have this architectural style. For such monolithic systems, we will propose the following approach to leverage our scriptable debugging system. In a monolithic architecture, the actual system is a single cohesive executable application that is deployed on a cluster

comprising of one or more nodes. Every node in the cluster is running identical software. Often the incoming requests and interactions to these nodes are distributes across the cluster via a load balancer. This means that although bugs can be explored completely by looking at a single node, at runtime it could be difficult to identify on which exact node bugs are occurring. This is especially the case when these bugs do not result in explicit failures to be exposed to the ”outside world” (for example, invalid internal state transitions). Therefore, to debug such systems we recommend the approach as depicted in Figure 6.2.

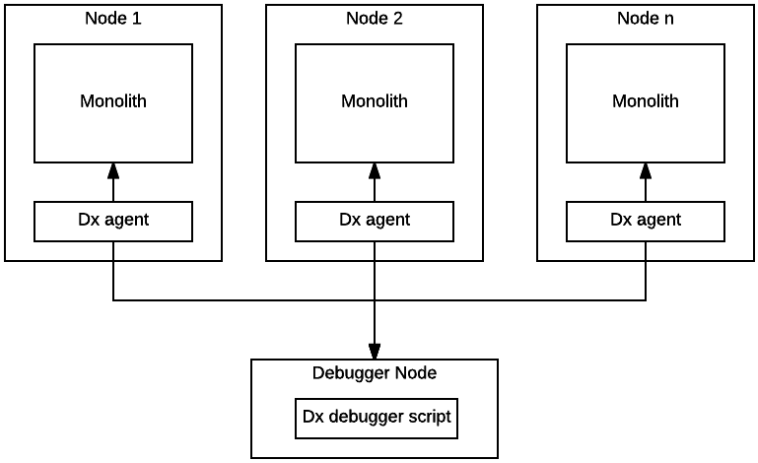


Figure 6.2: Reference architecture for debugging monoliths

Every node in the deployed cluster is running with the Dx agent loaded into the running JVM. The Dx agent is configured such that when the debugger script is not subscribed to the debugging event streams the event interceptor does nothing (i.e. no-op). This is important to allow the system to not suffer from performance impact when the debugger is not being used. Unfortunately, in our current implementation there is still some performance impact on the running system. In the Dx implementation based on JDI this impact is unacceptably high as explained in chapter 3. The implementation based on bytecode instrumentation does allow us to have much less performance impact because we can simply no-op when no one is subscribed to the event streams. Unfortunately, there still is some overhead due the invocation (i.e. method dispatch) of the event interceptors. We will leave further improvement of Dx to allow for zero performance overhead for future work.

The actual debugger would be running on a separate node that will be able to subscribe to the debugging event streams of one or more nodes in the cluster. Any processing, computations, visualizing, etc. that this node needs to do for the analysis of the sought-after bug will not impact the running system. The debugger node can utilize the debugging event streams to perform any custom debugging task. For example, it could be scanning for certain ”unexpected” patterns using the RxParsec pattern matching library and trigger some report to be generated when the patterns occur.

6.1.2. Debugging the cloud

Systems that are running in the cloud are often employing the microservices approach to software architecture. Whereas monolithic systems represent a single centralized artefact that contains all components of the software, microservices allow a more distributed approach to systems development. Often a single system is divided into multiple smaller services that communicate with each other via a certain remote protocol. Many such services are event-driven and rely heavily on concurrency and asynchrony in their implementations. This type of architecture allows for a much more resilient and scalable system, but does come at a cost of a much more difficult debugging cycle.

Bugs in a microservices architecture are also much harder to trace. An inconsistent state of a particular service might reveal itself as a bug in a completely different service that potentially doesn’t even have a direct dependency on the service that contains the true source of the bug. The transitive impact of bugs in distributed systems make it much harder to locate the actual source of misery. This is not made easier by the fact that, in contrast to monoliths, microservices have completely different codebases for each service. To maintain some level of observability across the complete architecture

implementers often use a technique called distributed tracing (a.k.a. transaction marking or end-to-end tracing) [52]. This technique is employed to be able to identify a single request on an end-user facing system and trace its complete distributed execution across all the microservices involved in serving that particular request. As one can imagine, in large scale systems such tracing can generate a lot of data. Manually looking at the data of such distributed trace logs seems highly inefficient. To alleviate this situation, we propose the employment of the Dx system as depicted in Figure 6.3.

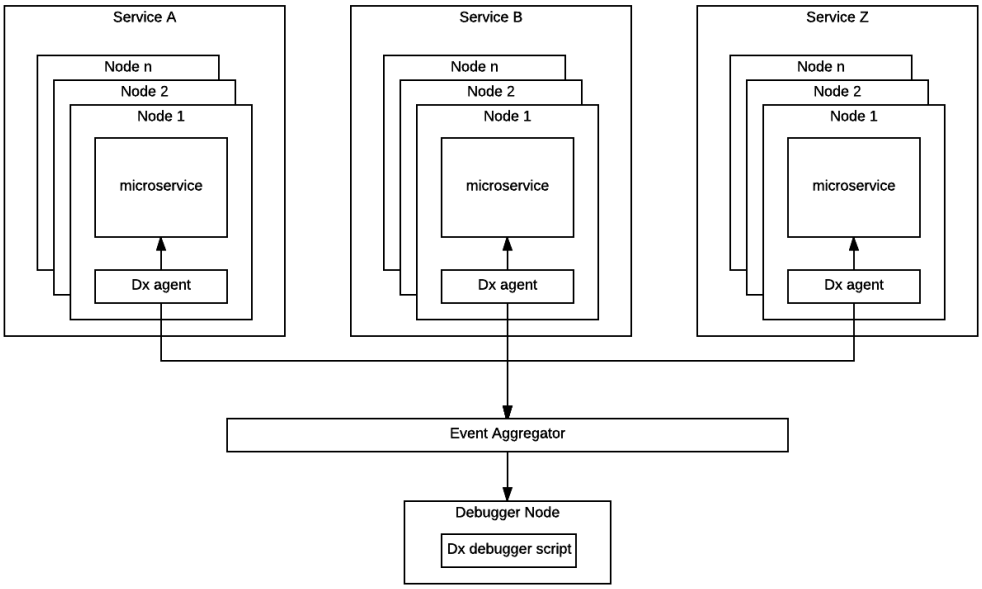


Figure 6.3: Reference architecture for debugging microservices clouds

In this setup, the complete architecture is comprised of multiple decoupled services. Every service is deployed on several nodes; these nodes can be running in one or more clouds that are located in one or more data centers. Thus, not only is there a high level of distribution between the several services but a single service internally is also deployed in a distributed manner. To deal with this level of distribution and loose coupling an event aggregator is recommended. The event aggregator is the single system where all observed debugging events of the complete architecture come together. Such a level of indirection is especially useful when many microservices are deployed across several clouds, datacenters, and perhaps even cloud providers (e.g. AWS, Azure, etc.). There are many systems that have proven themselves to be efficient event streaming systems in high scale systems. Examples, of such systems include Apache Kafka, ZeroMQ, ActiveMQ, RabbitMQ, Splunk, ElasticSearch, etc. For our conceptual purposes, it doesn't matter which of these systems is utilized, therefore we won't make a concrete recommendation. Similar to the setup for debugging monoliths, in the microservices setup the debugger script is also running on a separate node. For very complex/intensive debugging a cluster of debugging nodes could be utilized as well. Though the main difference in this case is that the debugger doesn't directly connect to the debugger agents but it rather subscribes to the event aggregator. The actual Dx agents are setup to be aware of the whereabouts of the event aggregator's nodes. The Dx agents push debugging events into the event aggregator based on dynamic cloud configuration. This dynamic configuration can be used to select at runtime which events from which nodes will be pushed. From the perspective of the debugger script there is still a single stream of all events in the complete system. Of course, in this case it is crucial that all events contain some metadata with regards to its origins (i.e. node hostname, cloud, datacenter, service name, etc.). It is therefore equally as important that this metadata is exposed by all services in a uniform manner. This should not be complicated to achieve as all events are generated by the same Dx agent.

6.2. Future work: Further recommendations

The work of developing a fully battle-proven scriptable debugging system is far from done. In this thesis, we have discussed two implementation techniques, one based on the JDPA and one based on bytecode instrumentation.

For the former we recommend that further exploration is done to achieve a debugging agent (based on the JVMTI) interface that does not cause a substantial performance impact as discussed in this thesis. Especially in the case that the debugger is not being used the agent should not cause a slowdown of the system. For the bytecode instrumentation, it would be interesting to dive further into the possibilities and alternative approaches, perhaps based on other implementations than Byte Buddy. In our implementation based on Byte Buddy, the more information a debugger script needs the higher the performance impact becomes. Optimizing the generated agents to have the smallest footprint possible would have great benefits for the purpose of debugging live systems.

Furthermore, it would be useful to explore what additional styles of debugging can be facilitated through the use scriptable debugging systems. Our implementation of RxParsec to allow pattern matching on asynchronous event streams is a first step in the direction of expanding the possibilities. Those that are interested are advised to look at its implementation and perhaps derive a more efficient or a more feature-rich implementation.

We have only scratched the surface when it comes to IDE integration. Although we have demonstrated that it is indeed possible to integrate scriptable debugging into the graphical IDE and aid developers through code generation, it seems like there is a lot of interesting innovation possible in this space. A survey could also be conducted to assess the usefulness of such integrations.

Performing a case-study and analysis of the proposed approaches to debug monoliths and microservices architectures would certainly yield interesting results. There is a lot of improvement possible in the proposed approaches and putting the system to work in real-life systems would allow for further insights into the pros and cons.

Finally, it would be interesting to investigate the application of reactive scriptable debugging in the context of systems that are part of the ”Internet of Things”. Distributed systems that are running on small devices embedded in many everyday entities seems to call for a novel approach to debugging as well.

6.3. Conclusion

In this thesis we have performed a deep dive into the design, implementation and analysis of reactive scriptable debugging. We have presented two different ways of implementing our reactive scriptable debugging system (Dx).

In our research we set out to understand the feasablility of a reactive scriptable debugging system from an architectural and implementation-oriented perspective. We looked at leveraging the Java Platform Debugging Architecture (JPDA) to achieve our goals. Although we were able to show that implementing the Dx API is indeed possible on top of the Java Debugger Interface (JDI) we had to conclude that it is not a good fit to debug systems that are running live in ”production”. The performance impact that is caused by using JDI is the main reason that lead us to conclude this.

As an alternative, we decided to implement Dx based on bytecode instrumentation. With this approach, we showed that it is possible to implement a scriptable debugging system that does not mandate the high performance impact, especially when the debugger is not connected to the system. Whereas using JDI impacts the runtime even when the debugging session is not active, our alternative approach allows us to avoid this. Nevertheless, there is a lot of room for improvement to minimize the impact during the actual debugging sessions as well.

In our research questions we also phrased the intent to explore the practicality of scriptable debugging by demonstrating the implementation of useful custom debugging tools. To this end, we have demonstrated various practical reactive scriptable debugging utilities and tools implemented using Dx. While creating many of these examples we encountered noticeably many repetitive patterns in the debugging scripts.

To alleviate developers from writing such ”boilerplate” and to explore the possibilities of integrating Dx into the developer environment (IDE) we experimented with the implementation of a Dx IDE plugin for Intelij IDEA. The plugin allows developers to graphically interact with Dx and use it to generate debugging code that can be customized and extended.

Another question we set out to answer in this thesis was whether it is feasible to implement real-time pattern matching on top of asynchronous debugging event streams. During the process of writing examples, we noticed that although the provided Rx operators for *Observable* streams allowed us to do most of the scripting there were certain more complex patterns that we were unable to react to

without the debugger scripts becoming very complicated. To this end, we implemented an additional library that complements Rx and allows reactive pattern matching by specifying the patterns in the form of LL(1) grammars. The idea of this pattern matching library is an adaptation of the Parsec library originally implemented for Haskell. We adopted the ideas of Parsec, which operates on pull-based data structures, to make it work for push-based data structures like the Rx *Observable*.

Finally, we discussed some of our ideas for future work. We have mostly focussed on the utilization of Dx for the purpose of debugging live production systems.

Bibliography

[1] J. W. Lloyd, *Practical advtanages of declarative programming.* in *GULP-PRODE (1)* (1994) pp. 18–30.

[2] M. Donat, *Debugging in an asynchronous world,* Queue **1**, 50 (2003).

[3] *How to become a hacker,* http://www.catb.org/esr/faqs/hacker-howto.html.

[4] *Reactivex,* http://reactivex.io/ ().

[5] *Reactivex - operators,* http://reactivex.io/documentation/operators.html ().

[6] *Completablefuture (java platform se 8),* https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html.

[7] *Observable (rxjava javadoc 2.1.14),* http://reactivex.io/RxJava/javadoc/io/reactivex/Observable.html.

[8] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing* (John Wiley & Sons, 2011).

[9] M. S. Johnson, *Dispel: A run-time debugging language,* Computer languages **6**, 79 (1981).

[10] Z. Aral and I. Gertner, *High-level debugging in parasight,* in *ACM SIGPLAN Notices*, Vol. 24 (ACM, 1988) pp. 151–162.

[11] R. A. Olsson, R. H. Crawford, and W. W. Ho, *Dalek: A gnu, improved programmable debugger.* in *USENIX Summer*, Vol. 90 (1990) pp. 221–231.

[12] *Gdb: The gnu project debugger,* https://www.gnu.org/software/gdb/.

[13] P. Maybee, *Ned: The network extensible debugger.* in *USENIX Summer* (1992).

[14] J. K. Ousterhout *et al.*, *Tcl: An embeddable command language* (University of California, Berkeley, Computer Science Division, 1989).

[15] M. Golan and D. R. Hanson, *Duel: a very high-level debugging language,* in *USENIX Winter*, Vol. 107 (1993) p. 118.

[16] D. A. Turner, *Recursion equations as a programming language,* in *A List of Successes That Can Change the World* (Springer, 1982) pp. 459–478.

[17] P. Winterbottom, *Acid: A debugger built from a language.* in *USENIX Winter* (1994) pp. 211–222.

[18] R. Lencevicius, U. Hölzle, and A. K. Singh, *Query-based debugging of object-oriented programs,* in *ACM SIGPLAN Notices*, Vol. 32 (ACM, 1997) pp. 304–317.

[19] M. Ducassé, *Coca: An automated debugger for c,* in *Proceedings of the 21st international conference on Software engineering* (ACM, 1999) pp. 504–513.

[20] M. Auguston, C. Jeffery, and S. Underwood, *A framework for automatic debugging,* in *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on* (IEEE, 2002) pp. 217–222.

[21] G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss, *A dataflow language for scriptable debugging,* in *Proceedings of the 19th IEEE international conference on Automated software engineering* (IEEE Computer Society, 2004) pp. 218–227.

[22] *Frtime: A language for reactive programs,* http://docs.racket-lang.org/frtime/.

[23] G. L. Steele Jr and G. J. Sussman, *The Revised Report on SCHEME: A Dialect of LISP.*, Tech. Rep. (MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1978).

[24] C. Elliott and P. Hudak, *Functional reactive animation,* in *ACM SIGPLAN Notices*, Vol. 32 (ACM, 1997) pp. 263–273.

[25] *dtrace.org - about dtrace,* http://dtrace.org/blogs/about/.

[26] *Home - d programming language,* https://dlang.org/.

[27] Z. A. Al-Sharif, *An Extensible Debugging Architecture Based on a Hybrid Debugging Framework*, Ph.D. thesis, University of Idaho (2009).

[28] M. Menarini, Y. Yan, and W. G. Griswold, *Aspectd: Enhancing a standard debugger with aspects,* (2010).

[29] Y. P. Khoo, J. S. Foster, and M. Hicks, *Expositor: scriptable time-travel debugging with first-class traces,* in *Proceedings of the 2013 International Conference on Software Engineering* (IEEE Press, 2013) pp. 352–361.

[30] *Undo products,* https://undo.io/products/undodb/.

[31] E. Jahier, *Rdbg: a reactive programs extensible debugger,* in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems* (ACM, 2016) pp. 116–125.

[32] *Java platform debugger architecture (jpda),* https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html ().

[33] *Java virtual machine tool interface (jvm ti),* https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html ().

[34] *Jvm(tm) tool interface 1.2.3,* https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[35] *Overview (java debug interface),* https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/index.html.

[36] *Event (java debug interface),* https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/package-frame.html.

[37] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, *The dacapo benchmarks: Java benchmarking development and analysis,* SIGPLAN Not. **41**, 169 (2006).

[38] *Jdk 8,* http://openjdk.java.net/projects/jdk8/.

[39] S. Liang and G. Bracha, *Dynamic class loading in the java virtual machine,* Acm sigplan notices **33**, 36 (1998).

[40] R. Winterhalter, *Byte buddy - runtime code generation for the java virtual machine,* http://bytebuddy.net/.

[41] *Dynamic proxy classes,* https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html.

[42] *Asm - home page,* http://asm.ow2.org/.

[43] *grpc / grpc.io,* https://grpc.io/.

[44] T. Elrad, R. E. Filman, and A. Bader, *Aspect-oriented programming: Introduction,* Commun. ACM **44**, 29 (2001).

[45] *Project jigsaw,* http://openjdk.java.net/projects/jigsaw/.

[46] *Intellij idea: The java ide for professional developers by jetbrains,* https://www.jetbrains.com/idea/.

[47] B. Gregg, *The flame graph,* Commun. ACM **59**, 48 (2016).

[48] D. Leijen and E. Meijer, *Parsec: Direct style monadic parser combinators for the real world,* (2002).

[49] *history - why do we call it "production"? - software engineering stack exchange,* https://softwareengineering.stackexchange.com/questions/68136/why-do-we-call-it-production.

[50] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,* in *Computing Colombian Conference (10CCC), 2015 10th* (IEEE, 2015) pp. 583–590.

[51] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, *Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages,* in *Proceedings of the XP2017 Scientific Workshops*, XP '17 (ACM, New York, NY, USA, 2017) pp. 23:1–23:5.

[52] J. Mace, *End-to-End Tracing: Adoption and Use Cases*, Survey (Brown University, 2017).