

DEBUGGING DOMAIN-SPECIFIC LANGUAGES

by

Aran Donohue

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2010 by Aran Donohue

Abstract

Debugging Domain-Specific Languages

Aran Donohue

Master of Science

Graduate Department of Computer Science

University of Toronto

2010

Much research focuses on the techniques, tools, and benefits of domain-specific language creation and use. However, since software maintenance is an important part of software projects, we should consider the maintainability of domain-specific language programs. These programs differ in many ways from their general-purpose language counterparts; we should not expect results applicable to one set to generalize to the other. This thesis presents the results of a case study of the maintenance of domain-specific language programs in industry. Results show that easy maintainability is not an intrinsic property of domain-specific languages.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | DSL Fundamentals | 3 |
| 2.1 | Introduction | 3 |
| 2.2 | Example Domain-Specific Languages | 5 |
| 2.3 | Things That Are Not Domain-Specific Languages | 8 |
| 2.4 | Related Work | 11 |
| 2.5 | Position of this research | 13 |
| 3 | Research Questions and Propositions | 18 |
| 3.1 | Rationale | 18 |
| 3.2 | Research Questions | 18 |
| 3.3 | Propositions | 19 |
| 3.4 | Synthesized Propositions | 29 |
| 4 | Study Design and Execution | 32 |
| 4.1 | The Case Study Research Method | 32 |
| 4.2 | Data Collection | 33 |
| 4.3 | Execution | 33 |
| 5 | Results | 36 |
| 5.1 | Analysis | 36 |

| | |
|-------------------------------------|-----------|
| 5.2 Other Results | 40 |
| 6 Discussion and Conclusions | 42 |
| 6.1 Validity | 43 |
| 6.2 Future Research | 46 |
| Bibliography | 48 |

Chapter 1

Introduction

In their search for better ways to build software, academics and industry practitioners occasionally contemplate domain-specific languages and the related subject of language-oriented programming.

Academic research has studied the nature and benefits of domain-specific languages. Several domain-specific languages (DSLs) are described in the literature [3, 13, 19, 25, 29, 45]. Researchers have explored methodologies and tools for constructing DSLs. Numerous papers describe empirical evaluations of specific DSLs [26, 40]. A few describe techniques for automatic creation of debuggers and other tools for DSLs [12, 50, 51].

On the industry side, prominent businesses host tool development projects to simplify the creation of DSLs [7, 8, 24]. Books, keen for sales, highlight their applicability to DSLs [32]. Numerous essays cover many aspects of DSLs, from history to future, implementation techniques, benefits, and more.

That maintenance constitutes a large part of software effort is now common knowledge, but, to our knowledge, little attention has been paid to the maintainability of programs developed using DSLs. Like all deliverables, they are subject to requirements change; like all programs, they are vulnerable to defects. There is a need to construct debuggers for DSL programs, as evidenced by numerous efforts in that area [12, 20, 49].

Together this implies a gap in our understanding of the total value of domain-specific languages, as this value could be affected by unexplored costs of maintenance.

As we will see, there are claims in the literature that DSL programs are easy to use and maintain. Some say that DSLs can be so easy that even non-programmers can read and maybe write DSL programs. To our knowledge, these claims are not backed by published evidence. In this study, we look at properties of DSLs that differentiate them from general-purpose languages. We apply several theories of bugs and debugging to generate propositions about the maintainability of DSLs. From the literature, we derive two more propositions about the use of DSLs by non-programmers. We test our propositions through interviews of DSL users and we find that easy maintainability is not an intrinsic property of DSLs. Thus, this thesis is a first step toward filling the gap in our understanding of DSL value.

Chapter 2 of this thesis surveys the field of domain-specific languages and language-oriented programming. It provides a thorough standalone introduction to the state of the art in tools and knowledge, including pointers to other surveys and resources. Chapter 3 poses and justifies specific research questions relevant to the maintainability of domain-specific language programs. Chapter 4 details the design and execution of the main case study of this thesis. Chapter 5 presents the analysis of results. The results are discussed in Chapter 6 along with suggestions for future research.

Chapter 2

DSL Fundamentals

2.1 Introduction

In this chapter, we define *domain-specific language* and give examples of domain-specific languages. Along the way, we introduce some common DSL terminology and patterns as illustrated by examples. We then explicitly exclude some related ideas from consideration as DSLs. Finally we present a brief summary of important literature on the subject of domain-specific languages, including references to other literature reviews.

2.1.1 Definitions

Numerous definitions of *Domain-Specific Language* exist in the literature. Wu et al. [51] define a DSL as follows:

A domain-specific language (DSL) is a programming language with concise syntax and rich semantics designed to solve problems in a particular domain.

Van Deursen [40]:

A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain.

Bentley uses the term *little language* [3]:

... a language is any mechanism to express intent... A computer language enables a textual description of an object to be processed by a computer program. ... A little language is specialized to a particular problem domain and does not include many features found in conventional languages.

Waite [45] avoids the term DSL and instead focuses on *declarative specifications*, which he defines as descriptions

...which explicitly list properties that a solution must have.

Fowler [17]:

...a limited form of computer language designed for a specific class of problems.

Wikipedia [46]:

In software development, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

Spinellis [37]:

A DSL is a programming language tailored specifically to an application domain: rather than being for a general purpose, it captures precisely the domain's semantics.

We use the definition given by Mernik et al. [30]:

Domain-specific languages are languages tailored to a specific application domain.

Note that we use the term *Domain-Specific Language* or *DSL* to refer to a notation. The term *Domain-Specific Language program* or *DSL program* refers to a string expressed in the language, i.e., a file that could be saved on a computer.

DSLs with their own parsers are called *external* (to a host general-purpose language). We contrast external DSLs with *internal* or *embedded* DSLs, which are written in a general-purpose language's syntax and therefore avoid the need for a separate parser.

2.2 Example Domain-Specific Languages

As examples of DSLs, we present two object-relational mappers (ORMs)¹ and two software project build tools.

2.2.1 Hibernate

Hibernate is an ORM for Java supported by Red Hat, Inc. [22] Hibernate mapping files are XML files that declare how Java objects are stored in a relational database.

Hibernate is illustrative of external DSLs that use XML to simplify parsing. A Hibernate configuration is used to construct data structure.

Figure 2.1 shows how to configure the persistence of a simple Person class in Hibernate.

2.2.2 ActiveRecord

ActiveRecord is an ORM for Ruby named for the Active Record design pattern². As described in its documentation [31]:

¹An object-relational mapping allows developers to use an object model for persistent storage to a database, instead of directly using SQL or another database language [2].

²*Active Record* is the name of a design pattern, defined as “an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.” [16] [15]

```
<hibernate-mapping package="org.hibernate.tutorial.domain">
  <class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
      <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>
  </class>
</hibernate-mapping>
```

Figure 2.1: Part of a Hibernate configuration for a database of people and their ages, from [23]. The *class* tag links a Java class to a specific database table. The *id* tag specifies the primary key of the table. The three *property* tags specify properties of Java objects to be stored in equivalently-named database columns.

Active Record connects business objects and database tables to create a persistent domain model where logic and data are presented in one wrapping. It's an implementation of the object-relational mapping (ORM) pattern by the same name...Active Record's main contribution to the pattern is to relieve the original of two stunting problems: lack of associations and inheritance. By adding a simple domain language-like set of macros to describe the former and integrating the Single Table Inheritance pattern for the latter, Active Record narrows the gap of functionality between the data mapper and active record approach.

ActiveRecord is an internal DSL in Ruby. Ruby function calls are overloaded to appear as declarations of relationships. At run-time, an ActiveRecord specification generates code to be used by the rest of the application.

For example, see Figure 2.2, which models the relationships of a firm.

```
class Firm < ActiveRecord::Base
  has_many   :clients
  has_one    :account
  belongs_to :conglomerate
end
```

Figure 2.2: ActiveRecord configuration for a database of companies and their clients, from [31]

2.2.3 Ant

Ant is a Java build tool created by the Apache Foundation [14]. XML is used to declare dependencies and how to satisfy them. Although it is similar in appearance to Hibernate,

an Ant file specifies a dependency-driven computation, rather than a database configuration. Creating Ant files is much more akin to programming than creating Hibernate files, since an Ant file is used to drive an interpreter. For example, see Figure 2.3 which compiles a Java program.

2.2.4 Make

Make is a program to manage dependencies between files [13]. Files called *makefiles* specify how to derive the target program from each of its dependencies [47].

The language of permissible makefiles is a domain-specific language. The syntax is custom³. Although the goals and functionality are similar to Ant, the style is different. For example, Figure 2.4 compiles a C program.

Table 2.1: Illustrative DSLs

| Language | Domain | Syntax | Execution Paradigm |
|--------------|--------------------------|-----------------|--------------------|
| Hibernate | ORM | External/XML | Data structure |
| ActiveRecord | ORM | Internal/Ruby | Code generation |
| Ant | Dependency specification | External/XML | Interpretation |
| Make | Dependency specification | External/Custom | Interpretation |

2.3 Things That Are Not Domain-Specific Languages

The definition of *domain-specific language* is naturally fuzzy. Nevertheless, there are things that we do not consider domain-specific languages but that are occasionally mistaken for them. We detail several examples below and clarify the distinctions.

³Infamously, tab characters are significant.

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
</project>
```

Figure 2.3: Ant Example—Compiles a Java program

```
helloworld: helloworld.o
    cc -o $@ $<

helloworld.o: helloworld.c
    cc -c -o $@ $<

.PHONY: clean

clean:
    rm -f helloworld helloworld.o
```

Figure 2.4: Make Example—Compiles a C program

2.3.1 Domain-Specific Libraries

Much of software development involves the creation of reusable libraries. These libraries are used to perform related functions. While the functions provided by a library may all be in the same domain, a general-purpose language program that invokes a library is not necessarily a domain-specific language program.

If, however, there were a special notation which maps to invocations of a domain-specific library, the notation would be a domain-specific language. In fact this is a straightforward method of domain-specific language creation.

A library which overloads general-purpose language constructs to map them to domain-specific constructs is also an implementation of a domain-specific language. ActiveRecord for Ruby is an example of this approach.

2.3.2 End-User Programming Environments

Some application programs present programming environments to users who do not consider themselves programmers. We refer to these as end-user programming environment [36]. This class of program includes spreadsheets with programming facilities such as Microsoft Excel, mathematical software such as GNU Octave, and scripting environments such as Apple Automator.

The application program is an end-user programming environment, not a domain-specific language. If, however, the application program accepts a notation for programs, this notation may be a domain-specific language.

2.3.3 Domain-Specific Applications

Many applications operate on data in a specific domain, such as the Microsoft Windows Registry Editor, which operates only on Windows Registry data. The interface to that data is an application, not a language.

2.3.4 General Notations

Some notations are meta-languages or meta-notations. These include XML [5] and JSON [9]. XML defines a set of permissible notations (called *XML applications*). Each XML application may be a domain-specific language, but XML itself is not domain-specific. The same argument applies to JSON: The JSON file format is not domain-specific, but if an application defines a permissible set of JSON keys and these are domain-specific, this subset of JSON files is a domain-specific language.

2.4 Related Work

Most existing work on domain-specific languages focuses on tools and techniques for implementing DSLs. We review these and refer the reader to other surveys of domain-

specific language literature. We then position our work in the context of the existing body of knowledge and justify its importance from the perspective of DSL research.

2.4.1 Implementing domain-specific languages

There are many approaches to creating domain-specific languages. These include specifying a new XML application, creating small compilers with tools such as Lex and Yacc, leveraging commercial software such as Microsoft Access, Excel or Powerpoint, designing monadic parsers in a language such as Haskell, and embedding a DSL in the syntax of a flexible programming language [48].

Many researchers and developers have created tools to simplify the creation of domain-specific languages. For example, a recent book teaches ANTLR for the creation of domain-specific languages [32]. As another example, the Khepera research project is a source-to-source compiler construction framework that carefully tracks syntax-tree node provenance through transformations. DSLs created in Khepera allow DSL-level debugging of the end-product [12]. The Eli system contains a number of anonymous DSLs to simplify compiler construction [45]. In the commercial world, tools for DSL development include Microsoft Oslo [8], JetBrains MetaProgramming System [24] and Intentional Software's Intentional Domain Workbench [7].

Several general considerations for DSL development have been addressed in the literature, including design patterns and processes. Spinellis has documented eight DSL design patterns, five of which are creational. These patterns include extending and restricting a host language, using a source-to-source transformation to a host language, using lexical substitution, and creating a notation that is isomorphic to a host data structure [37]. Mernik's survey of DSL literature also covers some high-level patterns for DSL development [30]. Strembeck [39] has documented an experience-based systematic process for the development of DSL programs.

2.4.2 Other surveys of DSL literature

Several excellent surveys of the DSL literature have been published. Wile surveyed techniques for DSL creation [48]. Christensen gives a thorough, readable review of the DSL field [6]. Van Deursen and Visser have a comprehensive survey [41], with another version available online [42]. Mernik and Sloane have a survey that includes a review of DSL benefits, DSL literature, tools supporting DSL development and high-level patterns for DSLs [30].

2.5 Position of this research

2.5.1 The importance of domain-specific languages

In addition to tool development, the properties of DSLs have attracted significant attention as a research subject. As early as 1966 researchers wanted to tailor programming languages to specific domains [28].

Academic interest in domain-specific languages remains strong. A recent issue of *IEEE Software* was dedicated to domain-specific languages [38]. The guest editors' introduction claims that DSLs are now easier to create than ever before, fueling continued interest [38]. Some believe that language-oriented programming, a methodology centered around the creation of DSLs, will be the next major programming paradigm [10].

2.5.2 Comparing domain-specific languages to general-purpose languages

DSLs purportedly have a number of advantages over general-purpose languages (GPLs). These claims are variably supported by evidence. We discuss these claims and the general differences between DSLs and GPLs below.

A key claimed advantage for domain-specific languages is the shorter length of DSL

programs compared to equally functional general-purpose counterparts [48]. This in turn suggests that DSL programs would be faster to write, given the evidence that programmers write the code at the same speed (lines of code per unit time) regardless of language [33, 34].

DSLs are also claimed to be more expressive than GPLs, as they offer notations that are appropriate to their domains. For example, Mernik and Sloane make the following claim:

[Domain-specific languages] offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [30].

Notation is known to have a large effect on human performance [4].

DSLs are claimed to have a lower semantic distance to their tasks than do general-purpose languages [3, 30, 37, 40, 51]. That is, it is easier for a person to map the meaning of DSL programs to the intentions or goals of the program.

DSLs have more restrictive syntaxes than general-purpose languages. We say that they have a smaller syntax space: there are fewer syntactic possibilities at any given point in a DSL program [40, 41, 51]. The small syntax may still be tailored to closely match the domain, permitting good expressiveness.

DSLs often have different execution models from typical general-purpose languages [41]. For example, a DSL may declare computations to be performed but not the order in which to perform them, whereas in a typical GPL program the developers specifies the control flow explicitly.

Unfortunately, DSLs also often suffer from a lack of tool support. For example, DSLs may have poor error diagnostics and they may not have debuggers [26].

2.5.3 Existing research claims

Maintenance is a large part of software cost. We would therefore like to know how DSL systems fare in maintenance. We believe that current research does not adequately address this issue with empirical evidence.

For example, Wu and colleagues explored the development of tools to ease debugging of domain-specific languages [49, 50, 51]. This implicitly indicates a desire for better tool support in DSL program maintenance. Van Deursen and Klint provide intuitive arguments that DSLs should increase maintainability, but no evidence. They call for more collection of empirical data concerning maintenance costs in systems built using domain-specific languages [40].

Van Deursen and Visser’s survey of DSL literature mentions maintainability concerns only twice [41]. Kiebert et al.’s report on an empirical experiment with a DSL indicates some confusion about DSL program maintainability:

During the post-experiment debriefing, however, the subjects’ answers to questions about ease of error location were not consistent with this data. Subjectively, only two of the four subjects stated that it was easier to locate errors in [the DSL program]. In particular, the [DSL compiler]’s error messages were poor, which impeded location of syntactic errors. For the [general purpose language] version, the error messages were good and the use of the Ada debugger allowed tracing of problems...

...For the [DSL], a minor extension to the [language] was sufficient to encompass the out-of-scope problem. However, this extension could not be provided by the subjects themselves; it required the expertise of the OGI research team who designed and implemented the [language]...

...Extension of the [equivalent program written in a GPL] was partially successful in handling the nine problems encountered. Extensibility was gained

through the Ada programming ability of the subjects that allowed them to produce new code templates. However, in the majority of instances, extensibility problems were handled not by writing new templates, but by finding workarounds or accepting partial solutions [26].

The DSL development process described by Strembeck [39] explicitly considers maintenance of the DSL itself but not of DSL programs.

Mernik and Sloane mention the benefits of DSLs, but none of their cases cover maintenance issues:

...the benefits of DSLs have often been observed in practice and are supported by quantitative results such as those reported in Herndon and Berzins [1988]; Batory et al. [1994]; Jones [1996]; Kieburtz et al. [1996]; and Gray and Karsai [2003], but their quantitative validation in general as well as in particular cases, is hard and an important open problem [30].

One common family of claims is that DSL programs will be readable, writable and/or maintainable even by non-programmer domain experts. These claims are not, to our knowledge, supported by published evidence. Examples of these include:

...enabling of software development by users with less domain and programming expertise, or even by end-users with some domain, but virtually no programming expertise. [30]

Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs. [41]

Using a DSL, the end-user (the domain expert) can be involved in the maintenance process: he or she may be able to express the modification request in terms of the DSL, or to validate the correctness of the modifications made. [40]

For some people, the goal here is to have the domain experts write the DSL so that these parts can be modified “without programming.” I tend to be skeptical of this claim, pointing out that this was the intent behind Cobol. Instead, I think the key value is providing a *business-readable* DSL, where domain experts can read the code, understand what it means, and talk to programmers directly about necessary modifications. It’s much easier to make DSLs business readable rather than business writable, but you gain most of the benefits by enhancing communication. However, several business-writeable DSLs have been very successful, so I’m not saying that you must avoid business-writability. [18]

They promise that domain experts themselves can understand, validate, modify, test, and sometimes even develop DSL programs [39].

2.5.4 Summary

Domain-specific languages are notations tailored for narrow purposes. They afford benefits including smaller code size and richer expressiveness, but sometimes suffer from a lack of tool support. Numerous tools exist for developing domain-specific languages.

Insufficient evidence has been gathered on the maintenance of DSL programs and on non-programmer DSL use. Claims that DSL programs are easier to maintain than other programs are common, but unsubstantiated.

Chapter 3

Research Questions and Propositions

3.1 Rationale

We present our research questions and then our propositions. Based on our review of the DSL literature, we have identified numerous ways DSL programs differ from general-purpose language programs¹. To generate propositions, we apply six theories from the debugging literature (which span theories of debugging, program comprehension, and programming errors) to these DSL characteristics. Finally, we generate two more propositions about non-programmer DSL use; these come from our review of the DSL literature.

3.2 Research Questions

Our first research question (RQ1) is: *How and why is DSL program maintenance different from GPL program maintenance?* By *maintenance*, we mean the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [1].

We will see that program comprehension is the crux of debugging. So it is natural to

¹Briefly: Shorter, more expressive, of simple syntax, with semantics closer to their domain, and lacking tool support. See Chapter 2.

wonder (RQ2), *how do programmers learn how DSLs work and how to use them?*

Having designed a study for these two questions, we can also ask a third question (RQ3): *How do non-programmers use DSLs?*

3.3 Propositions

To generate our predictions about the maintenance of DSL programs, we extend six existing theories that were developed for general-purpose language programs. We also generate two propositions from our review of the DSL literature. To extend each of the theories, we apply several claims about the ways DSL programs differ from general-purpose programs. Note that all of the claims are statements of general properties that permit exceptions. Note also that each extension requires an intuitive “theory evaluation” step. In some cases we assume our extensions to be self-evident. Where they are not we make our reasoning explicit.

The claimed DSL differences, from the DSL literature, are:

1. Lower semantic distance to task [3, 30, 37, 40, 51].
2. Different execution model [41].
3. Shorter programs [48].
4. Appropriate domain-specific notations [30].
5. Smaller syntax space [40, 41, 51].
6. Lack of tooling:
 - (a) Poor error diagnostics [26].
 - (b) Lack of debugger [26].

In the following sections, we will explain each theory and show how we generate propositions about DSLs from the theory. The theories vary in predictive power, but there are no significant disagreements in our predictions.

3.3.1 Ko and Myers's Theory of Bug Creation

Ko and Myers claim there are four aspects to software errors [27]. The *surface qualities* of an error denote notational issues. The *cognitive causes* of errors include forgetting, misunderstanding, or not knowing key information. Errors occur during *programming activities*, such as specification and algorithm design. More specifically, there is a specific *type of action* leading to the error: creating code, reusing code, modifying specifications or code, designing and so on.

Programming errors can reside in one of four layers and sometimes lie undetected until they interact with other errors. The four layers are *specifications*, *programmer*, *programming system*, and *program*. For example, specification-layer errors include ambiguous, incomplete or incorrect specifications. An incomplete specification might not be noticed until a user perceives a failure indicating a software error.

Each layer has its own defences against error introduction. Programmers defend against errors using their knowledge, expertise and careful attention. The programming system defends against errors using compiler error messages, static checkers, syntax highlighting, assertions and other tools.

An error at the programmer layer is called a cognitive breakdown. A cognitive breakdown has four parts: Some *kind of breakdown* occurs during some *action*, in turn performed through some *interface* on some *information*. Chains of cognitive breakdowns lead to software errors. Let us elaborate on the dimensions of a breakdown:

Kinds of Breakdown: In general, there are three types of activity, each of which has its own kind of breakdown.

1. **Skill-based activity:** Routine actions that do not require much attention, e.g.

opening a file. Skill-based activities fail due to inattention breakdowns, e.g. when a strong, wrong habit kicks in, or when a routine action is interrupted but not resumed. In some cases, overattention can also cause failures.

2. **Rule-based activity:** Actions that depend on learned expertise. Rule-based activities fail because of broken rules or wrong rule choice. An example rule would be *In C, to map an action on an array, use a for loop.*
3. **Knowledge-based activity:** Conscious, deliberate problem-solving. Knowledge-based activities fail due to cognitive limitations and biases inherent in cognition, e.g. satisficing incorrectly or a combination of confirmation bias and overconfidence.

Actions: There are six actions in programming: Design, creation, reuse, modification, understanding and exploration. These are carried out at various times in the programming process.

Interfaces: Examples of interfaces include documentation, diagrams, editors, debuggers and computer outputs.

Kinds of Information: Examples of information include specifications, code and program behaviour.

In addition to this categorization on four dimensions, breakdowns can also be separated by general category of task, i.e. *specification* of design and requirements, *implementation* and manipulation of code, and *runtime* activities such as testing and debugging.

Given this theory, we generate the following predictions about DSL programs:

1. Programs with a lower semantic distance to their goals are less likely to cause breakdowns during rule-based activities. Rules map human intentions to notation modification actions to perform—if the notation is closer to the intention, then fewer rule invocations are necessary to express an intention. If there are fewer rule invocations, there are fewer opportunities for breakdowns.

2. Programs with different execution models interact with programmer layer defences to increase errors. Many programmers have extensive training in traditional programming language execution models. New execution models may reduce the effectiveness of programmer expertise and knowledge as defences against errors, at least until the programmer becomes an expert in the new execution model.
3. Shorter programs decrease errors, as there is less to design, create, reuse, modify, understand and explore. In addition, programmer attention is concentrated on fewer potential error sites.
4. Appropriate domain-specific notations reduce errors because they increase the strength of programmer knowledge and expertise defences. They also shift activity from more difficult knowledge-based activity to rule-based activity.
5. It is unclear what the theory predicts about a smaller syntax space.
6. A lack of tooling results in more errors because it represents the elimination of programming system defences. Programming systems are collections of tools, many specifically designed to defend against errors.

3.3.2 Vessey's Theory of Debugging

According to Vessey [43], debugging is fundamentally a program comprehension task. Thus we predict, for each DSL difference, whether it ought to aid or weaken comprehension.

1. Lower semantic distance to task \rightarrow Aid comprehension.
2. Different execution model \rightarrow Either effect: An unfamiliar or complicated model could weaken comprehension, but a simple model might be very easy to understand.
3. Shorter programs \rightarrow Aid comprehension.

4. Appropriate domain-specific notations → Aid comprehension.
5. Smaller syntax space → Aid comprehension.
6. Lack of tooling → Weaken comprehension where tools could have otherwise aided it.

3.3.3 Structural Learning Theory

Hale and Haworth present a mental model for debugging/maintenance based on Structural Learning Theory [21]. The model integrates a declarative component describing the structure of knowledge and a procedural component describing a debugging process.

The procedural component describes debugging as a search problem. The problem is that the actual output differs from the desired output. The goal of the search is to modify the program such that the actual output matches the desired.

The process is based on recursive rule application. The programmer finds a rule such that the goal is a possible output of the rule and the current situation is in the domain of the rule. If the programmer is able to find a rule, he or she applies the rule and checks if the goal is satisfied. If, however, no rule is available, the programmer switches goals such that solving the new goal will change the situation.

Rules include several debugging strategies:

1. Direct diagnosis. Direct diagnosis is the final rule in a debugging episode.
2. Control flow: Follow the control flow of the program beginning to end
3. Data flow: Work backward from output
4. Slicing: Identify all statements that influence a suspicious variable

From this description we generate the following predictions:

1. When a DSL program has a lower semantic distance to task, we expect it to be more clear when a program's nature differs from the desired. Thus, we expect debugging to often only require one application of the direct diagnosis rule.
2. DSLs with different execution models push traditional debugging strategies to their limits, perhaps even forcing programmers to invent new rules.
3. Hale and Haworth explicitly predict smaller program length not to have a significant effect on debugging performance.
4. It is not clear what effect domain-specific notations have within the Hale and Haworth model.
5. It is not clear what effect smaller syntax spaces have within the Hale and Haworth model.
6. A lack of tooling will hinder rule application for some debugging strategies. If direct diagnosis is the dominant rule as predicted earlier we expect a lack of tooling not to have a significant effect.

3.3.4 Eisenstadt's Survey

Eisenstadt asked a number of developers about their debugging experiences and classified the responses [11]. Eisenstadt's survey covered GPL programs. For our purposes, we treat his conclusions as a theory of bugs. That is, we view each of his results as a prediction or explanation about GPL bugs, and we filter it through our list of general DSL properties.

His survey resulted in three categories of observations: Reasons why bugs are difficult to find, methods for finding bugs, and root causes of bugs. We relate each of these to stated DSL program differences:

Reasons why bugs are difficult to find:

1. Large temporal or spatial gap between bug cause and bug symptom → If DSL programs are shorter, we expect spatial gaps to be correspondingly smaller and, therefore, large gaps be less likely. Thus we predict shorter programs will make bugs easier to find.
2. Bugs which, due to their nature, neutralize debugging tools → If breaking tools makes bugs harder to find, then lacking debugging tools in the first place is a problem.
3. Programmer has faulty perception, e.g. misleading syntax → Faulty programmer perceptions are less likely with appropriate domain-specific notations and a smaller syntax space.
4. Programmer has broken semantic model, i.e. holds an incorrect belief about the meaning or function of some programming construct → Broken semantic models are more likely with new execution models.
5. Excessively ugly code → Excessively ugly code is less likely with appropriate domain-specific notations and a smaller syntax space.

Methods for finding bugs:

1. Inserting print statements or other debugging code → Impossible in DSLs with no print statements or non-procedural execution models.
2. Step through in a debugger → (Sometimes) impossible due to lack of tooling.
3. Comparing good and bad core dumps → Impossible due to lack of tooling.
4. Debugger breakpoints → Impossible due to lack of tooling.
5. Use of specialist tools → Impossible due to lack of tooling.

6. Hand simulation of code or other thinking procedures → Sometimes difficult due to different execution models.
7. Recognition of bug symptoms from memory → Equally easy in GPL and DSL programs.

Root causes of bugs:

1. Memory overwrites → Less likely due to narrow functionality that is not likely to contain dangerous memory operations.
2. Vendor hardware/software faults → Equally likely in GPL and DSL programs.
3. Incorrect logic → Equally likely in GPL and DSL programs, except in DSLs that do not specify logic.
4. Incorrect initializations → Less likely due to smaller syntax space or narrow functionality that does not require initialization.
5. Syntax problem → Less likely due to smaller syntax space.

3.3.5 Integrated Comprehension Model

Von Mayhauser and Vans's Integrated Comprehension Model explains program comprehension (and therefore debugging according to theories that equate the activities) as a task of building up understanding of a program at several layers at once [44]. The *top down model* is a mental model at the domain or application level. The *program model* is a low-level bottom-up model of the program, including control flow information. In between these two layers lies a *situation model*, which is a domain-independent layer of abstractions, such as algorithms. Finally, a knowledge base holds information about the overall comprehension task as well as facts as they are learned. To learn about the

program, a programmer repeatedly forms hypotheses about the program at the various levels and tests them.

Based on this theory, we view program comprehension as a process of mentally bridging the gap between the program text and the domain-level semantics the program is meant to carry.

1. Lower semantic distance to task \rightarrow When the notation has a low semantic distance to task, the mental model's layers grow closer or merge, suggesting easier program comprehension.
2. Different execution model \rightarrow No prediction.
3. Shorter programs \rightarrow No prediction.
4. Appropriate domain-specific notations \rightarrow Easier comprehension, due to merged layers of the model.
5. Smaller syntax space \rightarrow Facilitate comprehension at the program model level.
6. Lack of tooling \rightarrow No prediction.

3.3.6 Human Information Processing

The Human Information Processing model is a model of how cognition works. Ramanujan et al. applied the model to software maintenance. Information travels back and forth from our extremely short-term sensory buffers, to short-term memory, to buffer memory, to long-term memory. Short-term memory has a small capacity. Information moves from short-term memory to long-term memory by repetition. The buffer is essentially the focus of our attention, including the active subset of long-term memory.

In the Ramanujan et al. model, maintenance tasks are subdivided into program comprehension, program modification and program composition. During program comprehension, the programmer uses syntactic and semantic information from long-term

memory to create a semantic understanding of the program. This understanding is constructed in the buffer. Program modification is the manipulation and correction of the buffer structure based on discrepancies from output. Program composition is slightly more complex: A problem is analyzed in the buffer into a “given state” and “desired state.” To achieve the desired state, knowledge is first transferred from long-term memory into the buffer. Then the programmer conceives the problem solution as a plan for the program. Finally the programmer repeats stepwise refinement on this plan [35].

With this in mind, we predict:

1. Lower semantic distance to task → Simplification of program comprehension because it should be easier to construct a semantic understanding of the program in the buffer. Simplification of program modification because the required manipulations and corrections of the buffer structure should be smaller.
2. Different execution model → No prediction.
3. Shorter programs → Simplification of program comprehension because there is less information to load into the buffer. We also predict more rapid program composition due to smaller plan size.
4. Appropriate domain-specific notations → Facilitate retrieval of semantic knowledge from long-term memory.
5. Smaller syntax space → No prediction. Even though the theory has a component for syntactic knowledge in long-term memory, it is not clear how the various kinds of maintenance task would be affected by a small syntax space.
6. Lack of tooling → No prediction. The theory does not cover the role of tools in cognition.

3.4 Synthesized Propositions

Having generated predictions from these existing theories, we are in a position to review their similarities and differences. There is substantial overlap in our predictions, though each theory leads us to different explanations for the same prediction. In general, our extensions of the theories agree: The maintenance of DSL programs should benefit from a lower semantic distance to tasks, shorter programs, appropriate notations and a smaller syntax space. The maintenance of DSL programs should suffer from different execution models and a lack of tooling. One contradictory prediction comes from our extension of Hale and Haworth, explicitly predicting that program length and lack of tooling should not affect maintenance effort.

All six of these DSL properties are relevant to our first research question on the differences between GPL and DSL maintenance. Of the six, three—low semantic distance, different execution models, and domain-specific notations—are directly related to programmer cognition and therefore to our second research question.

This brings us to our third research question on non-programmer DSL use. From our review of DSL literature, described in Chapter 2, we generate two more propositions: First, that non-programmer domain experts can read and validate DSL programs, and second, that non-programmer domain experts can create and write DSL programs.

For an overview, Table 3.1 links our propositions to our research questions. Table 3.2 maps distinctive DSL characteristics to the theories along with our predictions.

Table 3.1: Propositions and Research Questions

| Proposition | Research Question |
|---|-------------------|
| 1. Lower semantic distance is helpful for maintenance | 1, 2 |
| 2. Different execution models are harmful for maintenance | 1, 2 |
| 3. Short programs are helpful for maintenance | 1 |
| 4. Domain-specific notations are helpful for maintenance | 1, 2 |
| 5. Small syntax spaces are helpful for maintenance | 1 |
| 6. A lack of tools is harmful for maintenance | 1 |
| 7. Non-programmers read and validate DSL programs | 3 |
| 8. Non-programmers create and write DSL programs | 3 |

Table 3.2: Theories and Predictions of DSL maintainability.

| DSL Property | Ko | Vessey | Hale | Eisenstadt | Von Mayrhauser | Ramanujan |
|--------------------------------------|----|--------|------|------------|----------------|-----------|
| Lower semantic distance to task | + | + | + | + | + | + |
| Different execution model | - | - | - | - | - | - |
| Shorter programs | + | + | * | + | + | + |
| Appropriate domain-specific notation | + | + | + | + | + | + |
| Smaller syntax space | + | + | + | + | + | + |
| Lack of tooling | - | - | * | - | - | - |

Legend

| | |
|-------|------------------------------------|
| Blank | No prediction |
| + | Prediction of advantage for DSL |
| - | Prediction of disadvantage for DSL |
| * | Explicit null prediction |

Chapter 4

Study Design and Execution

4.1 The Case Study Research Method

We are interested in understanding the “how” and “why” of maintainability differences between DSL and GPL programs (if any). For this reason, we chose to use a case study as described in [52].

We do not believe in the near-term possibility of a representative sampling of DSLs, DSL programs, software projects, software developers or other important variables of interest. Given this perspective, we believe that claims about DSLs as a population are very difficult to evaluate in general.

4.1.1 Units of Analysis

Our unit of analysis is the software developer. Our data sources are software developers. We are interested in developers’ experiences with DSLs and in their cognition and understanding of DSLs.

We could choose to study at some other unit of analysis, such as the DSL, team, or project/product/repository. We chose the software developer as our unit of analysis because it made our study execution possible given our limited resources.

4.2 Data Collection

Case studies have six common sources of evidence: Documents, archival records, interviews, direct observation, participant observation and physical artifacts [52]. For this study, we relied exclusively on interviews.

4.2.1 Interview Methods

Focused interviews were the primary data collection technique of this study. A focused interview is a short interview that mostly follows a set of questions from the case-study protocol [52].

Interviews were summarized and analyzed for statements relevant to the study propositions. Table 4.1 maps propositions to the relevant interview questions.

Interview subjects were software professionals and two non-programmers who use DSLs. To be a subject of this study, the candidate had to maintain or have maintained at least one DSL program. Furthermore, the existence, function and maintenance history of this program or these programs could not be confidential information.

No personal information was collected about interview subjects. Interview subjects were recruited via personal networks and participation request letters. For the detailed information collected in our interviews, see Table 4.1, which links interview questions to propositions and research questions.

4.3 Execution

We conducted ten interviews¹ over three months. One interview was with two coworkers, so there are eleven subjects. Two interviews were conducted in person, one over an internet video connection, and the rest over audio-only connections.

¹The first interview was conducted with an early prototype of the interview script.

Table 4.1: Interview Questions, Propositions, and Research Questions

| Interview Questions | Propositions | Research Questions |
|---|--------------|--------------------|
| 1. Do you have any questions before we begin? | N/A | N/A |
| 2. What DSLs do you use or have you used? | | |
| 3. How did you learn the DSL? | 2, 4, 5 | 1, 2 |
| 4. Was it hard to learn? Why or why not? | | |
| 5. How well do you understand how your DSL works under the hood? Would it be easy for you to reimplement the DSL? | 1, 2, 4, 5 | 2 |
| 6. If this DSL is targeted at non-programmers, what's the involvement of non-programmers? | 7, 8 | 3 |
| 7. What general-purpose languages do you use or have you used? | N/A | N/A |
| 8. How did you learn the language? | 2, 4, 5 | 1 |
| 9. Was it hard to learn? Why or why not? | | |
| 10. Can you tell me about a time changing requirements required a change to a DSL program? | 1–6 | 1, 2 |
| 11. How did you narrow down the location to make the change? | 3, 6 | 1 |
| 12. What tools did you use? | 6 | 1 |
| 13. Can you tell me about a time you had to had to debug a DSL program? | 1–6 | 1, 2 |
| 14. What kind of bug was it? | | |
| 15. How did you narrow down the bug location? | 3, 6 | 1 |
| 16. What tools did you use? | 6 | 1 |
| 17. Can you tell me about a time changing requirements required a change to a general-purpose language program? | 1–6 | 1, 2 |
| 18. How did you narrow down the location to make the change? | 3, 6 | 1 |
| 19. What tools did you use? | 6 | 1 |
| 20. Can you tell me about a time you had to had to debug a general-purpose language program? | 1–6 | 1, 2 |
| 21. What kind of bug was it? | | |
| 22. How did you narrow down the bug location? | 3, 6 | 1 |
| 23. What tools did you use? | 6 | 1 |
| 24. How does the difficulty of the DSL changes you've needed to make compare with the difficulty of other changes? What makes the difference? | 1–6 | 1, 2 |
| 25. How does the difficulty of the DSL bugs you've encountered compare with the difficulty of other bugs? What makes the difference? | | |
| 26. Have you ever done the same thing in both a DSL and GPL? Tell me about it. | | |
| 27. Have you ever inherited someone else's DSL program? Tell me about it. | | |
| 28. Do you have any questions for me? | N/A | N/A |

We recorded each interview, transcribed the recordings, and anonymized and summarized the transcripts. The subjects were given their transcriptions and summaries for validation. All but two validated² their data. The summaries are the data used for the analysis. They include basic demographic information and are publicly available as a case study database.

Nine subjects were professional programmers. The remaining two considered themselves non-programmers, though they both had dabbled in programming. Both were interviewed with a focus on a music creation DSL they use; questions relevant only to programmers were omitted.

The programmers discussed many different DSLs³. Several subjects had created DSLs or maintained DSL implementations.

In the following statements of basic demographic information, general statements (e.g., “all”) omit subjects who did not release the relevant datum. For example, when we say, “The subjects were between the ages of twenty-five and forty-two,” we are making no statement about the subjects who did not disclose their age, and who may be younger than twenty-five or older than forty-two. The subjects were located in Canada, the United States, the United Kingdom, South Korea, and Greece. All of the subjects were male. Non-programmers aside, six years was the least programming experience among the subjects, and the most experienced programmer had thirty-eight years of programming experience. The formal education levels of the subjects ranged from an incomplete Bachelor’s degree to multiple degrees (including some not related to software) or advanced degrees in computer science.

²During the validation process, the subjects corrected a small number of minor errors. No major changes were made.

³DSLs discussed included S/SL, Ant, JSP, ActiveRecord, Liquid, regular expressions, procmailrc, Make, CSound, Typed Scheme, Snooze Query Language, SchemeUnit, Fit, and many anonymous DSLs. We refer the reader to a web search engine for details about these and many other DSLs that are not documented in the literature.

Chapter 5

Results

5.1 Analysis

First, we analyzed our data by proposition, searching the interview summaries for relevant statements by the interviewees. Second, we searched for interesting results which were not directly relevant to our propositions. Below, we summarize each set of statements from the perspective of our propositions. In each case, specific interviews¹ are adduced to justify our result.

5.1.1 Proposition 1: Semantic distance

The semantic distance of a DSL is a characterization of how the DSLs notation maps to its users' intentions. Thus we analyzed our data looking for statements linking changes in intention to changes in DSL program.

DSLs have a lower semantic distance and this is a benefit of DSLs (Interview 2, Question 10). This is “by definition.” That is, DSLs are constructed specifically to reduce semantic distance and used specifically because they reduce semantic distance, thereby facilitating changes to meet new requirements (I2 and I3, Q10; I6, Q11 and

¹Our citations of interviews are numbered by their chronological labels in the case study database.

Q24).

One interview suggested that changes to programs are made smaller but can require more thought (I1, Q10).

In short, low semantic distance is a feature of DSLs by design and it simplifies maintenance.

5.1.2 Proposition 2: Execution models

Some DSLs do not have typical sequential-program semantics. Thus we analyzed our data looking for statements about DSLs with special execution models and the effects.

Sometimes the execution model is simple and this makes a DSL easy to learn (I4, Q9). In some cases, there is more to think about and the programmer is more likely to make a mistake (I5, Q24).

DSL programs are sometimes written specifically for different semantics (I5, Q24).

DSL program bugs can be more difficult to debug because it may be more difficult for a programmer to mentally simulate the execution of a DSL program (I5, Q24).

In short, complex or unfamiliar execution models are error-prone and can make debugging and correcting DSL programs more difficult.

5.1.3 Proposition 3: Program length

Programmers generally found it easy to locate places to make changes in DSL programs (I1–I3, I5–I10, Q11 and Q15).

It is possible that this is because DSL programs are short. However, it is also possible that this is due to low semantic distance. I2 Q11 supports the latter explanation.

We did not find evidence to support our proposition about the length of DSL programs or the effect of DSL program length.

5.1.4 Proposition 4: Domain-specific notation

Some domains have natural notations which can be exploited by a DSL. Thus we analyzed our data looking for statements about DSLs having natural expressiveness in their domains and the effects of this.

Some DSLs try to emulate natural language. These can be difficult to use (I2, Q24).

A better notation can make a program easier to write and maintain (I3, Q26). A notation can be easier to learn if it is similar to an already-familiar notation (I4, Q9). A good notation can reduce the amount of thinking required to solve a problem (I4, Q5).

In short, notations expressive in a domain have numerous benefits, including simplifying maintenance.

Note the close relationship to the semantic distance proposition in Section 5.1.1. The closer a notation to its domain, the lower the semantic distance of DSL programs to their intentions.

5.1.5 Proposition 5: Syntax size

Domain-specific languages generally have small syntaxes compared to general-purpose programming languages. Thus we analyzed our data looking for statements about the effects of DSLs being restricted in syntax.

Small DSL syntax keeps learning, development and maintenance simple (I4, Q9). In one case, the small syntax and clarity of purpose of a DSL resulted in significantly more use than an alternative DSL (I10, Q3).

Some DSLs have large syntaxes. In these cases, learning the DSL can be very time-consuming and frustrating (I7 and I8, Q3 and Q4).

In short, small syntax has an impact on how easy it is to learn and use a DSL, but we only found weak evidence to explain how it would help maintenance.

5.1.6 Proposition 6: Tool support

Domain-specific languages do not always come with the same tools many programmers expect from general-purpose languages. In particular, domain-specific languages often do not have a debugger (I1 Q15, I4 Q13).

The implementation of an embedded DSL can interfere with otherwise-applicable debugging tools, such as the debugger or stack trace (Interview 6, Question 13). This is reminiscent of Eisenstadt’s observation that bugs can be especially pernicious when they interfere, by nature, with debugging tools [11].

Of course, general purpose languages sometimes lack tools too, causing frustration when debugging (I6 Q20; I9 Q22). Also, some general-purpose language tools can be applied to DSL debugging problems (I3 Q23; I9 Q22).

In short, tool support is important for any language, DSL or not. DSL maintenance suffers without debuggers, although this is mitigated by the occasional possibility of using general-purpose language tools on DSL problems. Overall, we found support for our proposition that a lack of tool support makes maintenance of DSL programs more difficult.

5.1.7 Propositions 7 and 8: Non-programmers and DSLs

We have direct evidence (I7 and I8) and second-hand accounts (I2, I3, I4, I9 and I10) of non-programmer domain experts successfully reading, validating, writing and maintaining DSL programs.

However, there is some evidence that this cannot be guaranteed. In particular,

1. One subject criticized end-user DSL maintainability claims, explaining that DSL programming requires a systematic, reductionist thought process that does not interest most business users (I2).
2. Two subjects had worked closely with non-programmer DSL users. Many of them

will only make small tweaks to programmer-authored DSL programs, or work by copy-and-pasting sample code (I4).

3. One subject had been involved in a DSL system intended for non-programmer use, but that failed to get non-programmers to use it (I9).
4. One subject estimated that in their experience, 1/3 of non-programmer customers will refuse all DSL involvement, 1/3 will read or validate DSL programs and 1/4 to 1/3 will write DSL programs (I10).

One subject said, “I think to be honest with you, the vast majority of DSLs that say they’re targeted at non-programmers are lying through their teeth.” (I2)

Our propositions that non-programmers read, validate, create and write DSL programs were supported. This result must be qualified, however, with the evidence that it is difficult to directly involve non-programmer domain experts in the maintenance of DSL programs.

5.2 Other Results

Our interviews uncovered several results that were not the main focus of our study.

5.2.1 Refactoring to DSLs: A DSL creational pattern

Some DSLs are created in a gradual process of refactoring. In this pattern of DSL creation, features are added to the DSL over the lifetime of a project, and general-purpose language code is converted into DSL code taking advantage of the new features (I2 and I5). To our knowledge this pattern has not been discussed in the DSL literature.

Programmers will also go the other way, removing DSL code in favour of general-purpose language code (I2 and I6, Q26).

5.2.2 DSLs as strong abstractions

DSLs can be viewed not just as languages or tools, but as a strongly enforced way of implementing new abstractions (I2; I4 Q26; I10 Q24). In this perspective, a DSL is an alternative to abstractions (e.g. functions, objects, modules, etc.) provided by a programming language.

5.2.3 DSL inevitability

Sometimes programmers don't believe they have a choice in using a DSL. They may believe that a project would be impossible to complete on time without a DSL, or that a DSL is the only way to solve an important problem (I1 end; I4, Q6).

5.2.4 The difficulty of making DSLs

DSL design and implementation is viewed as difficult because it is seen as language design or the design of a reusable API (I4 Q11; I5; I9). One subject said, "...by definition you're designing your own language. I guess we're perhaps people who normally wouldn't be doing that. But now we are." (I4)

5.2.5 Bugs in two places

DSLs introduce another layer into a program. This can cause difficulties in debugging, because it is difficult to traverse the extra layer (I1, Q25; I3, Q24 and 25, I4 Q25), and because there are now more places to look for a problem: In the usage of the DSL, in the DSL implementation, or in the program (I4, Q25; I6, Q15).

On the difficulty of debugging DSL programs, one subject said, "It's like, twice as hard." (I4)

This problem does not occur when the DSL is the definitive place for all of a certain kind of functionality (I9).

Chapter 6

Discussion and Conclusions

We studied the maintainability of domain-specific language programs, including how it differs from GPL maintenance, how programmers learn how DSLs work and how to use them, and how non-programmer domain experts use DSLs. Through a review of the DSL literature, we identified six properties that are claimed to distinguish DSLs from general-purpose languages: Lower semantic distance to task, different execution models, shorter programs, appropriate domain-specific notations, smaller syntax space, and a lack of tooling. In order to develop propositions for study, we then analyzed six theories of general-purpose language maintenance, including theories of error introduction, debugging, and program comprehension. Our first contribution is an application of these six theories to the six DSL properties to generate a theory of DSL program maintenance. From this new theory, we generated six propositions about DSL maintenance. From our review of the DSL literature, we generated two more propositions about how non-programmer domain experts use DSLs.

We conducted ten interviews with DSL users to validate our properties and test our theory. Our second contribution is several results supporting our propositions:

1. DSLs are used specifically to lower the semantic distance to their domain. By definition, this simplifies changes to meet new requirements.

2. On one hand, some DSLs have simple execution models, which make it easier for a maintainer to understand a program. On the other hand, some DSLs have complex execution models, which cause problems by making comprehension and changes difficult.
3. DSLs with appropriate domain-specific notations make maintenance easier by taking advantage of familiarity with other notations and reducing the amount of thought needed to make changes.
4. The ecosystem of a DSL matters: A lack of tools can frustrate programmers and make bugs hard to find.
5. Non-programmer domain experts are able to read, validate, write, create and maintain DSL programs in some circumstances, but it would be risky to rely on it.

We were not able to support our proposition that the short length of DSL programs simplifies maintenance. We only found weak support for our proposition that small syntaxes facilitate DSL maintenance, though we did find that small syntaxes made DSLs easier to learn and use.

We had several additional findings, of which one is directly relevant to our research questions though not to our propositions: The presence of a DSL creates a new source of complexity in a system, and means more places must be searched for bugs. We will discuss our other unanticipated results in Section 6.2 on future research. Overall, we conclude that better maintainability is not an intrinsic property of DSLs.

6.1 Validity

We briefly document our steps towards a valid, reliable study. Our results are only analytically generalizable and not statistically generalizable. Our goal is not to demonstrate

“how much” of an effect there is for anything—instead we hope to illuminate why there might be and how it works.

6.1.1 Construct Validity

To our knowledge there is no commonly-accepted system of measuring maintainability as it pertains to a programming language. In the absence of such a construct we believe our line of direct questioning and observation is as good as any operational measure for the concept being studied.

Our data collection spans multiple languages, projects, developers, programs and businesses. In this way, we mitigate some of the risk that our results are an artifact of a narrow context.

We have established a chain of evidence that links our data collection to our study propositions, our propositions to our research questions, our research questions to real-world problems, and our propositions to existing theories.

Our informants reviewed their interview transcripts and summaries and had the opportunity to review drafts of this report.

6.1.2 External Validity

We do not claim that our results apply to all software developers, projects, or domain-specific languages. Instead we aim to demonstrate the theoretical and actual existence of important maintainability considerations in the design and choice of domain-specific languages.

We believe that a literal replication using our same procedures will produce similar results. Furthermore, the theories elaborated in Chapter 3 should allow theoretical replication, where further research using different data sources or a different unit of analysis should yield predictably different results.

6.1.3 Reliability

We deliberately facilitate the reproducibility of both data collection and analysis. Our interview guide is available. Furthermore, to permit a reproduction of our analysis, we publish a case study database. It contains the same summary data we used in our own analysis.

6.1.4 Threats to Validity

Some study subjects were recruited from the author’s extended network. It is possible that this network misses critical cases. This is not a traditional selection bias problem, as we do not rely on statistical generalization to populations. We address this by selecting subjects that cover a wide variety of cases. These include famous DSL experts and professional developers in small and large companies.

Given that our propositions were developed in advance, it is possible that our questions biased the responses. Furthermore, it is possible that interviewee actions did not match their words. For questions that relate to factual propositions, we focused on factual past events. From our questions that have non-factual answers, we drew conclusions only about our subjects’ opinions or cognitive processes.

It is likely that our subjects were more interested and more experienced in DSLs than average. We maintain that these issues neither take away from our logical chain of reasoning nor from the insights gleaned.

Small syntaxes, domain-specific notations, and alternative execution models are not intrinsic to DSLs. They may be common, but we could find counterexamples for each. CSound is a DSL with a large syntax; S/SL’s notation is convenient but not indicative of its domain; Liquid’s execution model is similar to normal programming languages. We could argue whether a lack of tools should qualify as a property of the DSL—in reality, it is like a figure-ground relationship. People frequently implement DSLs without

accompanying debugging tools, and this is a property of the engineering context. The best we can say is that these characteristics are common in DSLs, and now we understand them better.

It is impossible to generalize any research on domain-specific languages as much as we would like. Domain-specific languages are so numerous and vary so widely that any controlled experiment will miss important properties of the DSLs left unstudied. As for the study of DSLs in practice, we cannot claim that any result found is an intrinsic property of the domain-specific language approach. It is plausible, for example, that DSLs are only ever used where requirements are unusually precise or where the domain is easily modelled.

6.2 Future Research

We documented both successful and unsuccessful cases of DSL use by non-programmers. We do not understand the factors that lead to success in one instance and failure in another. What are the relevant properties of the context, project, language or programmers? Is there a way to tell in advance if a project has components suitable for maintenance by non-programmer domain experts?

Programmers liken DSLs to a strongly enforced abstraction in a program. Some DSL creators indicated a need for education and guidance in solving the difficult problems of designing such abstractions and languages. Further research is warranted in design principles for abstractions and little languages, and how to communicate these principles to DSL implementors.

Programmers don't always consciously choose a DSL over an alternative—sometimes a DSL seems like the only way to get something done. This view conflicts with DSL literature that implicitly assumes DSLs are an alternative to other approaches. How do developers decide whether to use a DSL and what DSL to use?

A newly discovered pattern of DSL creation teaches us that the making of a DSL can be gradual, as a general-purpose language program is refactored to use the DSL more and more. We know very little about this pattern. When is it appropriate? When is it used? Should we teach it?

We described theories and evidence about why semantic distance and usability of syntax matters in DSLs. However, we did not pursue this subject at a lower level. What are the relevant properties of a syntax? What are the characteristics of semantic distance? For example, Rake, Make and Ant do largely the same thing, but Ant was created in response to a perceived weakness in Make, and Rake in response to both. What is it about these that causes the subjective feelings of superiority?

Bibliography

- [1] IEEE Std. 610.12. *Standard Glossary of Software Engineering Terminology*. IEEE Computer Society Press, Los Alamitos, CA, 1990.
- [2] D. Barry and T. Stanienda. Solving the java object storage problem. *Computer*, 31(11):33–40, 1998.
- [3] J.L. Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.
- [4] A.F. Blackwell. SWYN: A Visual Representation for Regular Expressions. In Henry Lieberman, editor, *Your Wish is My Command: Programming by Example*, pages 245–270. Morgan Kaufmann, 2001.
- [5] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 6, 2000.
- [6] N.H. Christensen. *Domain-specific languages in software development—and the relation to partial evaluation*. PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, 2003.
- [7] Intentional Software Corporation. Intentional software. <http://intentsoft.com/>. [Online; accessed 03-Sep-2009].
- [8] Microsoft Corporation. Microsoft code name "oslo" modeling technologies. <http://msdn.microsoft.com/en-us/library/cc709420.aspx>. [Online; accessed 03-Sept-2009].

- [9] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.
- [10] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 2004.
- [11] M. Eisenstadt. Tales of debugging from the front lines. In *Empirical Studies of Programmers: Fifth Workshop: Papers Presented at the Fifth Workshop on Empirical Studies of Programmers*, page 86. Ablex Pub, 1993.
- [12] R.E. Faith, L.S. Nyland, and J.F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proceedings USENIX Conference on Domain-Specific Languages*, 1997.
- [13] S.I. Feldman. Make-a program for maintaining computer programs. *Software: Practice and experience*, 9(4), 1979.
- [14] The Apache Foundation. Apache ant. <http://ant.apache.org/>. [Online; accessed 25-Aug-2009].
- [15] M. Fowler. P of eaa: Active record. <http://www.martinfowler.com/eaCatalog/activeRecord.html>, 2002. [Online; accessed 24-Jul-2009].
- [16] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
- [17] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? <http://martinfowler.com/articles/languageWorkbench.html>, 2005. [Online; accessed 11-May-2009].
- [18] M. Fowler. A pedagogical framework for domain-specific languages. *IEEE Software*, 26(4):13–14, 2009.

- [19] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.
- [20] John Graham-Cummings. Gnu make debugger. <http://gmd.sourceforge.net/>. [Online; accessed 01-Jan-2010].
- [21] D.P. Hale and D.A. Haworth. Towards a model of programmers' cognitive processes in software maintenance: a structural learning theory approach for debugging. *Journal of Software Maintenance: Research and Practice*, 3(2), 1991.
- [22] Red Hat Inc. Hibernate. <https://www.hibernate.org/>, 2009. [Online; accessed 25-Aug-2009].
- [23] Red Hat Inc. Hibernate core tutorial, chapter 1. <http://docs.jboss.org/hibernate/stable/core/reference/en/html/tutorial.html>, 2009. [Online; accessed 25-Aug-2009].
- [24] JetBrains. JetBrains :: Meta programming system – language oriented programming environment and dsl creation tool. <http://www.jetbrains.com/mps/index.html>. [Online; accessed 03-Sep-2009].
- [25] S.C. Johnson. *YACC-yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975.
- [26] R.B. Kieburtz, L. McKinney, J.M. Bell, J. Hook, A. Kotov, J. Lewis, D.P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society Washington, DC, USA, 1996.

- [27] A.J. Ko and B.A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1-2):41–84, 2005.
- [28] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966.
- [29] M.E. Lesk and E. Schmidt. *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1986.
- [30] M. Mernik and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.
- [31] Ruby on Rails Documentation. Active record - object-relation mapping put on rails. <http://api.rubyonrails.org/files/vendor/rails/activerecord/README.html>. [Online; accessed 24-Jul-2009].
- [32] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf Raleigh, NC, 2007.
- [33] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Computer*, 33(10):23–29, 2000.
- [34] L. Prechelt. Are scripting languages any good? A validation of perl, python, rexx, and tcl against C, C++, and Java. *Advances in Computers*, 57:207–271, 2003.
- [35] S. Ramanujan, R.W. Scamell, and J.R. Shah. An experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort. *The Journal of Systems & Software*, 54(2):137–157, 2000.
- [36] J.R. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *Proceedings of the 27th international conference on Software engineering*, pages 352–361. ACM New York, NY, USA, 2005.

- [37] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems & Software*, 56(1):91–99, 2001.
- [38] J. Sprinkle, M. Mernik, J.P. Tolvanen, and D. Spinellis. Guest editors’ introduction: What kinds of nails need a domain-specific hammer? *IEEE Software*, 26(4):15–18, 2009.
- [39] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software Practice and Experience*, 39(15):1253–1292, 2009.
- [40] A. Van Deursen and P. Klint. *Little Languages: Little Maintenance?* Centrum Voor Wiskunde En Informatica, 1997.
- [41] A. Van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
- [42] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. <http://homepages.cwi.nl/~arie/papers/dslbib/>. [Online; accessed 08-Sep-2009].
- [43] I. Vessey. Toward a theory of computer program bugs: an empirical test. *International journal of man-machine studies*, 30(1):23–46, 1989.
- [44] A. von Mayrhauser and A.M. Vans. Program understanding behavior during debugging of large scale software. In *Empirical studies of programmers: Seventh Workshop: Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, pages 157–179. ACM New York, NY, USA, 1997.
- [45] W.M. Waite and C. Boulder. Beyond LEX and YACC: How to generate the whole compiler. *University of Colorado, Technical Report, Boulder, Colorado*, 1993.

- [46] Wikipedia. Domain-specific language — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=289253538, 2009. [Online; accessed 11-May-2009].
- [47] Wikipedia. Make (software) — wikipedia, the free encyclopedia. [http://en.wikipedia.org/w/index.php?title=Make_\(software\)&oldid=310008074](http://en.wikipedia.org/w/index.php?title=Make_(software)&oldid=310008074), 2009. [Online; accessed 25-Aug-2009].
- [48] D.S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–288, 2001.
- [49] H. Wu, J. Gray, and M. Mernik. Debugging domain-specific languages in eclipse. *OOPSLA Eclipse Technology Exchange Poster Session*, 2004.
- [50] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10), 2008.
- [51] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374. ACM New York, NY, USA, 2005.
- [52] R.K. Yin. *Case study research: Design and methods*. Sage Publications, Inc, 2008.