TAMPERE UNIVERSITY OF TECHNOLOGY

JYRI-MATTI LÄHTEENMÄKI

Using Scala to Boost Web Development

Master of Science Thesis

Examiner: Professor Ilkka Haikala
Examiner and topic approved by
the Council of the Faculty of Computing
and Electrical Engineering
on 8 April 2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LÄHTEENMÄKI, JYRI-MATTI: Web-sovelluskehityksen tehostaminen Scala-ohjelmointikielellä
Diplomityö, 59 sivua, 1 liitesivu
Toukokuu 2010
Pääaine: Ohjelmistotiede
Tarkastaja: professori Ilkka Haikala
Avainsanat: Scala, ohjelmointikieli, Web-sovelluskehitys, Java EE, MVC, testaus

Web-sovellusten kehittämiseen on tarjolla useita vaihtoehtoisia teknologioita. Loppukäyttäjälle valittu toteutusteknologia ei välttämättä ole näkyvissä, mutta asiakkaalle se näkyy projektista aiheutuvina kustannuksina. Lisäksi teknologiavalinnoilla on vaikutusta lopputuloksen laadulle ja projektin kehittäjien työn mielekkyydelle. Pääsääntöisesti vaihtoehdot perustuvat dynaamisiin ohjelmointikieliin, ja staattisesti tyypitetty kieli toisikin mielenkiintoisen vaihtoehdon. Tässä työssä tarkastellaan Scala-ohjelmointikielen soveltuvuutta perinteiseen Web-kehittämiseen ja arvioidaan sen tuomia etuja ja haittoja.

Scala-ohjelmointikieli on tavukoodiyhteensopiva yleisesti käytetyn Java-kielen kanssa, mutta sen funktionaalisen paradigman hyödyntäminen ja mahdollisesti hyvinkin erilainen syntaksi tekevät siitä hankalahkon opittavan Java-ohjelmoijalle. Tämän vuoksi työssä esitellään Scala-ohjelmointikielen perusteet sekä työn kannalta oleellisimmat tekniikat. Työ pyrkii tuomaan esiin eroja ja yhtäläisyyksiä Java-kieleen nähden ja luo silmäyksen tämänhetkiseen työkalutukeen.

Työssä toteutetaan Scala-kielellä perinteisiä Web-kehityksessä vastaantulevia asioita ja arvioidaan niiden etuja ja haittoja vaihtoehtoisiin tapoihin nähden. Ohjelmakoodiesimerkit tarjoavat samalla konkreettisen näkymän asiaan. Joitakin mielenkiintoisia mahdollisuuksia havaittiin, joilla voi olla merkittäväkin positiivinen vaikutus ohjelmistokehitykseen.

Työn yhteydessä toteutettu yksinkertainen esimerkkiprojekti tarkastelee työssä tehtyjä havaintoja käytännössä. Tällä pyritään tuomaan esille asioita, jotka tulisivat vastaan tosielämän projekteissakin. Projekti toteutettiin sekä Javalla että Scalalla, mikä osoittaa että Scala-kieltä voidaan käyttää saumattomasti osassa projektia ilman että se liikaa häiritsee muita osia. Esimerkkiprojektille ajettu yksinkertainen suorituskykytesti osoittaa, ettei Scala-toteutuksella ole suorituskykyongelmia Java-toteutukseen nähden.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
LÄHTEENMÄKI, JYRI-MATTI: Using Scala to Boost Web Development
Master of Science Thesis, 59 pages, 1 Appendix page
May 2010
Major: Software Science
Examiner: Professor Ilkka Haikala
Keywords: Scala, programming language, Web application development, Java EE, MVC, testing

There exists a number of alternative technologies for Web application development. The final product as seen by the end user might not reflect the underlying technologies, but for the customer it is visible in project costs. The chosen technologies have an impact on the overall quality of the product and they also play an important role in making the development pleasant. Most of the available technologies are based on dynamic programming languages and thus a statically typed language would introduce an interesting alternative. This thesis examines the suitability of the Scala programming language for Web development and evaluates its advantages and disadvantages.

Scala programming language is bytecode compatible with the well-known Java language but its utilization of the functional paradigm and possibly quite differing syntax make it somewhat hard to learn for an average Java programmer. Therefore the basics and the most relevant techniques of the Scala language are introduced. The similarities as well as differences to Java language are brought up, and the available tool support for Scala is examined and briefly compared to the excellence of Java tools.

This thesis implements some common functionality related to web development in Scala. The discovered advantages and disadvantages are evaluated against alternative methods. Program code listings offer a concrete view to the subject. Some interesting possibilities were discovered that could have a significant positive impact on development.

A simple example project investigates the observations in practice. This strives to bring out issues that could also be present in real-life projects. The project was implemented in both Java and Scala which shows that Scala can be used in some parts of a project without major interference to other parts. A simple performance test run against the example project shows that Scala does not suffer from performance issues comparing to Java.

PREFACE

My years in Tampere University of Technology were some of the best I have yet experienced. It is always a kind of a pity to leave something remarkable behind. On the other hand, life has so much to offer, and there is so much that I have yet to learn. There is probably no better way to make this transition than by producing something concrete of these years, in the form of a truly interesting subject and black leather covers.

This thesis work was made while working at Solita Oy. Big thanks go to Kimmo Kiviluoma for finding me an interesting subject, Antti Tirilä for directing and helping with the work and Olli Tuomi for technical opinions. Thanks to all the employees for providing me with a relaxing and inspiring working environment. My warmest thanks also to examiner Ilkka Haikala for finding the time for this thesis during these overly busy months.

This work could not have been finished without the continuing support from my wife Laura. Our children, Joona and Juuso, could not yet properly articulate to actually help me, but they showed some great patience and understanding during these months when I needed my quiet time.

Thank you all.

April 1, 2010

Jyri-Matti Lähteenmäki
jyri-matti@iki.fi

CONTENTS

1. Introduction . . . 1
2. Developing for the Web . . . 3
  2.1 History of Web application development . . . 3
  2.2 Alternatives to Java and Java EE . . . 4
    2.2.1 Java7 . . . 4
    2.2.2 Ruby . . . 5
    2.2.3 Groovy . . . 5
    2.2.4 Scala . . . 6
3. Introduction to Scala . . . 7
  3.1 Principles of Scala . . . 7
  3.2 Scala basics . . . 8
  3.3 More advanced Scala . . . 12
  3.4 Scala pitfalls . . . 14
  3.5 Scala vs Java . . . 16
  3.6 Tools . . . 18
4. Utilizing Scala for some common use cases . . . 20
  4.1 Domain Specific Languages . . . 20
    4.1.1 Scala for DSLs . . . 20
    4.1.2 Hibernate Criteria API . . . 21
    4.1.3 JSON . . . 22
    4.1.4 Wiki parser . . . 23
  4.2 Testing . . . 25
    4.2.1 ScalaTest and ScalaCheck . . . 25
    4.2.2 Testing the view . . . 26
5. Web project with Scala using Spring Framework and Hibernate . . . 29
  5.1 Project structure . . . 29
  5.2 Execution flow, decorators and mapping . . . 31
  5.3 Form objects, binding and validation . . . 33
  5.4 Model, Data Access Objects and Services . . . 36
    5.4.1 Model . . . 36
    5.4.2 Data access objects . . . 39
    5.4.3 Services . . . 40
  5.5 View . . . 40
  5.6 Performance tests . . . 45
6. Discussion of the results . . . 47
  6.1 Scala as the programming language . . . 47
  6.2 Web development: Scala versus traditional methods . . . 48

6.2.1 Amount of code 48
6.2.2 Code quality 48
6.2.3 Testing 49
6.2.4 Production rate 49
6.3 Example project: Differences to corresponding Java implementation . . 50
6.3.1 Amount of code 50
6.3.2 Production rate 51
6.3.3 Code quality 51
6.4 Performance 52
6.5 Tool support 53
7. Conclusion 54
References 56
Appendix A: Results of the performance tests 60

ABBREVIATIONS, TERMS AND DEFINITIONS

advice In Aspect Oriented Programming (AOP), the way to specify additional code to run at a certain location. The code can be specified to run either before, after or around the location.

annotation The way to add metadata to different program structures in Java.

Ajax Asynchronous JavaScript and XML. A common term for various web technologies that can be used to implement a web application that communicates with a server in the background.

agile In software development, a group of methodologies where the focus is on iterative progress and evolution of both requirements and solutions during the project.

AOP Aspect Oriented Programming. A Programming paradigm where some functionality not directly related to business logic is extracted from the program and inserted declaratively to appropriate locations.

API Application Programming Interface. A set of classes that defines the interface between a component implementing a feature and a component using the feature.

BDD Behavior-driven development. A development style where the program behavior is first described as text or text-like program code and then verified with tests that the implementation matches the specified behavior.

call-by-name A non-strict evaluation strategy where the method argument is evaluated only and every time when needed.

client-side validation Validation of form input occurring in the browser without server interaction.

closure A first class function that has the context of its declaration site bound to it.

contravariance A type rule obeys contravariance when it accepts a super type in addition to the specific type itself.

Convention over Configuration An approach of removing excessive configuration by using sensible default values.

covariance A type rule obeys covariance when it accepts a sub type in addition to the specific type itself.

DAO Data Access Object. An instance of a class responsible for accessing the actual underlying data store, like a database or the file system.

Decorator A design pattern where the functionality or output of a component is modified at runtime by wrapping it with another component with a similar interface.

DSL Domain Specific Language. A programming language targeted at a specific problem domain. Contrary to a general purpose language like Java or Scala.

DTO Data Transfer Object. An object intended only for transferring data between different parts of the program and thus should not contain any actual program logic.

dynamic language A programming language that is designed to perform some operation at runtime which might in other languages be done at compile time or not at all. The definition is somewhat abstract and a specific programming language cannot be unambiguously declared as dynamic or not.

EL Expression Language. An addition to the JavaServer Pages technology that allows easy access to some Java features that previously required adding Java code to JSP pages.

FSC The Fast Scala Compiler. A Scala compiler that uses a background daemon to make repetitive compilation faster.

Hibernate An open-source Java persistence framework for relational databases.

HTTP Hypertext Transfer Protocol. A stateless, application-level protocol designed to transfer hypertext content in the Internet but used also for many other tasks.

IDE Integrated Development Environment. An application that contains many common tools and features needed for software development.

immutable A component, for example an object, whose internal state cannot be changed after creation.

implicits A term used to refer to implicit conversions and implicit parameters.

invariant A type rule is invariant, or nonvariant, when it accepts only the specific type itself, not a sub type or a super type.

Java EE Java Platform, Enterprise Edition. The standard Java platform (Java SE) coupled with some libraries and running in an application server.

JSTL JavaServer Pages Standard Tag Library. A library with JSP tags for some common functionality.

JVM Java Virtual Machine. An abstract computing machine that executes Java bytecode and confirms to the Sun JVM specification.

lexer A program that is responsible for lexical analysis.

lexical analyzer See *lexer*.

LOC Lines of code. A common and simple measure of program length. Can be calculated in different ways, but this thesis measures logical lines excluding any whitespace and comments.

metaprogramming Writing a program that modifies itself or another program.

mixin A component for multiple inheritance, that can be *mixed in* to a class so that the class inherits the methods and other structure without actually being inherited from the mixin class.

MVC An architectural pattern where the business logic, user input handling and presentation are separated from each other. Often used in web applications.

partially applied function An expression where only some or none of a function's required arguments are passed in, resulting in a function which can be called with only the missing parameters.

POJO Plain Old Java Object. An ordinary Java object, often a simple class having only ordinary getters and setters for properties.

scaladoc Scala documentation system, similar to Javadoc.

scriptlet A piece of Java code embedded in a JavaServer page.

side effect A method is said to have a side effect if it either modifies the state of any component or interacts with anything outside itself.

signature The footprint of a method at binary level, which contains all information that is used to identify a method. It might consist of for example the name of the method an the number, order and types of its parameters.

statically typed A programming language is said to be statically typed when type checking occurs at compile time.

TDD Test Driven Development. A development style where test cases are thought and implemented before the actual implementation of the code being tested.

tuple Typed, fixed-size, ordered list of elements in which the elements can have different types.

type erasure The process where all information about generic type parameters and arguments is removed at compile time.

type inference The process of automatically deducing some type information from the program code. Lets the programmer leave away most of the type annotations while still maintaining type safety.

uniform access principle A principle which states that there should be no difference in syntax of handling methods, properties or fields. In other words, the caller of a class attribute should not have to know whether it is implemented as a method, property or a field.

WWW World Wide Web. The big, interlinked collection of different kinds of resources contained in the Internet.

XHTML XML variant of the HyperText Markup Language.

XML Extensible Markup Language. A simple, structured, textual and standardized data format which is in wide use.

XPath XML Path Language. A query language for selecting different parts from XML documents.

1. INTRODUCTION

The web is constantly increasing its significance in the lives of ordinary people. Services, shops and sources of information are moving to the web to gain advantage of its evident benefits of availability and reachability. People are getting accustomed to finding everything they need from the Internet and it is already generally considered abnormal if something is not attainable via the web.

Demand creates supply. Excess demand of web applications creates an attractive environment for both large and small companies of this field, as well as other groups and even individuals. The quality of a product is only as good as are its producers and thus the increased number of people working with web applications raises the question whether the quality can always be maintained.

The number of tools available to create web applications has been increasing. New tools are striving to ease the development of both small and enterprise-scale applications, as well as to increase quality by providing ready-made components and abstractions. PHP, Java and Microsoft's technologies have been the majority of the actual programming languages on which the tools are based on. Recently some new candidates have emerged amongst dynamic languages. The loss of static type safety and thus some of the help from the compiler further increases concerns for maintaining quality.

This thesis investigates whether Scala, a new statically typed language for the Java Virtual Machine (JVM), could be taken advantage of in web application development to increase productivity and quality. Scala tries to bring together the benefits of both functional and object-oriented paradigms in a statically typed and Java-interoperable language. In addition to its syntactical virtues, it also brings in some language features that enable the developers to re-think some concepts and abstractions in a way that was previously only feasible with dynamic languages. This thesis is based on Scala release version 2.7.7.

Chapter 2 gives a brief introduction to the history of web application development as well as the current state. Some noteworthy alternatives to Scala are briefly introduced after which Chapter 3 gives the reader the basic knowledge of the Scala programming language. Comparison to Java is made where appropriate to help the reader to understand the concepts. Some of the more advanced features considered most relevant to this thesis work are introduced and finally some available

development tools are briefly analyzed.

A more practical approach is taken in the following chapters. Chapter 4 gives examples of using Scala in some common use cases. It also inspects what Scala could bring to application testing. In Chapter 5 a simple example project is analyzed and the usage of Scala in the different parts of the project is introduced. The project is implemented in both Java and Scala for comparison and to investigate potential problems in a project containing both languages.

In Chapter 5 also a coarse performance test is described. It inspects how the Scala implementation performs against the Java implementation, as well as against a mixed set containing components implemented in both languages. Finally Chapter 6 summarizes the results and gives some analysis of what could be gained by using Scala.

2. DEVELOPING FOR THE WEB

The web has been around for a long time. During this time it has seen the rise of various different technologies to provide content for the viewers. These have evolved to serve more and more users as the web has gained more popularity. In this chapter the history of the web is briefly described and a small set from the abundance of technologies is introduced.

2.1 History of Web application development

When the *World Wide Web* (WWW) was proposed and the first browser implemented in 1989-1990, it contained nothing but static, hand-made *HTML (HyperText Markup Language*) pages. The amount of data was small and it was mostly static by nature, so this satisfied the needs at the time.

In 1993 the *Common Gateway Interface (CGI)* [1] standard was introduced, which was a common way for different web servers to generate dynamic data. New languages and tools targeted for the web started to emerge, for example *PHP* [2] in 1995. These provided easier ways to include dynamic content on web pages, and the nature of the web began to shift towards dynamic content.

In 1995 Netscape Communications released an initial implementation of a new scripting language designed for browsers. *JavaScript* as well as its standardized version called *ECMAScript* and Microsoft's dialect called *JScript* began the era of client-side scripting which made web ever more dynamic. Later the term *Ajax (Asynchronous JavaScript And XML)* emerged to represent a variety of technologies used to asynchronically fetch data from the server and present it in the browser.

Ever since the birth of the World Wide Web its use has been constantly increasing, as both companies and ordinary people have found more uses for it in everyday life. The server-side applications have grown bigger and more complex and created a completely new business field. The server-side technologies are constantly evolving and new tools are still emerging. They are striving to make the process of building either small or large web applications easy.

The enterprise-scale requirements have caused a number of big frameworks to appear, which combine the tools needed for the most common concepts, like security and data persistence, in the same package. These include *Java Enterprise Edition (Java EE)* [3], Apple *Web Objects* [4], *Ruby on Rails* [5], *Python Django* [6] and

Microsoft's ASP-technologies [7]. In respect to different programming languages, Java has maybe the most tools available, as the concepts of Java EE can easily be utilized with various alternative technologies such as using *Java Data Objects* [8] or *Hibernate* [9] instead of *Enterprise JavaBeans* [10] for the data persistence, or using *FreeMarker* [11], *Apache Velocity* [12] or *XSLT* [13] instead of *JavaServer Pages* [14] for drawing the view.

Many of the frameworks follow the *Model-View-Controller (MVC)* pattern where the domain model, user interface and business logic are separated to their own components. So does for example *Spring Framework* [15] which allows the use of almost any Java-technology for the model or for the view.

At the time of writing there exists a number of alternative technologies to develop both small and large applications. The requirements for web applications have grown quite versatile. The available technologies and programming languages are diverse, partly due to different dynamic languages recently gaining momentum. Some popular dynamic languages, like *Python* [16] and *Ruby* [17], can also be run in a Java Virtual Machine thus gaining access to all the existing Java libraries and components.

Java EE and Java as the programming language are widely used for enterprise level applications. This is due to good library and company support, even though Java might not be the best language for complex applications. Its shortcomings include verbose syntax and some language design choices (see for example *A critique of Java* [18]).

The trend in Java EE was first towards using *XML (eXtensible Markup Language)* for configuration and the use of dynamic template languages for the view. Later Java annotations have gained popularity in replacing XML configuration. Java EE is still in use but Java has been losing ground.

2.2 Alternatives to Java and Java EE

There are many programming languages with varying amounts of supportive libraries. There are even many running on top of the JVM. It is up to the developer's preference and the project's needs, which technologies bring in the biggest advantages. Even on the JVM platform and in the context of web development there are some alternatives to choose from. This section only lists a few that were considered the most relevant to this thesis work.

2.2.1 Java7

At the time of writing, Java7 is still under development and it is still not certain which features will be included and which left out. According to a preview article

by Alex Miller [19], Java7 would be including for example annotations on types and some dynamic language supporting features, but no closures of any kind.

Annotations on types would add some static type safety but at the cost of more verbosity in the language. Dynamic language support is an interesting feature because it would increase the performance of dynamic languages running on the JVM. Closures would be a welcomed addition since it would get rid of much of the verbosity in Java.

Despite the compromises made when choosing the features of Java7, it will still in time replace the current Java platform and its new features will probably keep the language in wide use. Unfortunately it might take years to get the new Java version ready for production use.

2.2.2 Ruby

Ruby is a dynamic language that strives for simplicity and productivity. It tries to be a natural mix of functional and imperative paradigms with an intuitive syntax. It allows extensions to the language with powerful *metaprogramming* (writing or modifying program code at runtime) while at the same time trying to be as concise as possible.

JRuby [20] is an implementation of Ruby on the JVM. This allows Ruby programs to be run over any JVM and also provides full interoperability with the existing Java libraries.

Ruby on Rails is a web application framework for the Ruby language. It strives for quick and simple development with less configuration (*Convention over Configuration*) and providing many common features out-of-the-box. Rails can also be run over JRuby, which brings advantages of the Java platform to the Rails development.

2.2.3 Groovy

Groovy [21] is a new dynamic language for the JVM. Being dynamic it does not perform as well as a statically typed language, but the JVM support for dynamic languages with Java7 will address this. One of the main advantages of Groovy is its syntax with its resemblance to Java, which lowers the learning curve for Java developers and allows exiting Java code to be converted to Groovy with only small changes.

Groovy is a superset of Java and adds additional features to the standard Java libraries. It also has native support for XML as well as many existing modules for various kinds of tasks. The number of libraries and modules has been increasing at a respectable phase.

Grails [22] is another project that extends Groovy to web application development. It builds upon existing technologies like Spring Framework and Hibernate leveraging the features of Groovy to create an agile web development environment. Also having principles like Convention over Configuration, Grails strives for a somewhat similar experience as Ruby on Rails introduced in Subsection 2.2.2.

2.2.4 Scala

Scala is what this thesis work is about. It is an object oriented language having many features of the functional paradigm. It is designed from the start to be scalable and fully interoperable with Java. Being statically typed yet close to dynamic languages in syntax, it could be described as a statically typed dynamic language. XML integration makes it an interesting option for web environment. Chapter 3 gives an introduction to Scala.

3. INTRODUCTION TO SCALA

Although this thesis work is not to analyze Scala as a programming language, some knowledge of the basic concepts and possibilities of Scala is needed to understand where and how it might come useful. The ability to read Scala syntax is an essential requirement to understand all the code examples. This chapter gives a short introduction to Scala as a programming language. It is mainly based on *Programming in Scala* [23].

3.1 Principles of Scala

In 2001 Martin Odersky and his group started designing a new programming language called Scala. The idea was to create a language that is object oriented yet functional, simple yet scalable, syntactically rich yet statically typed, and interoperable with Java. The first version was released in 2003 and the second followed in 2006.

Scala narrows the gap between static and dynamic languages by having many language features that are mostly known from dynamic languages. With its structural types it even has a form of *Duck-typing*, where the semantics of an object are determined based on its methods and properties instead of class inheritance or interface implementation.

Functional paradigm proposes every language construct to evaluate to a value of some kind, which tends to shorten expressions since often multiple expressions can be combined. Ideally there are also no *side effects*, that is, changes to the state of the world when executing a function. Scala provides many *immutable* constructs and encourages their use. Another feature known from the functional paradigm are higher-order functions, which means that functions can be used as values, for example passing them as parameters for other functions.

Scalability is one of the corner stones of Scala, object orientation being only a part of it. Scala takes object orientation further than for example Java. Every value including function values is an object and every operation is a method call. In addition, inheritance and *mixins* provide scalability at the architecture level, while operator overloading and implicit type conversions introduce new possibilities at class level.

Scala's syntax is less verbose than Java's. This is achieved mainly by type inference which is used to guess most of the type information. This allows the programmer to leave out unnecessary type declarations, although they can still be included when the additional detail is useful.

During the research preceding Scala it was realized, that a language will not be widely accepted unless it has a fairly versatile standard library [24]. Being fully interoperable with Java, Scala has direct access to all libraries written in Java. The standard JDK classes are also extended by using implicit type conversions.

One major feature of Scala is its direct compile time support for XML handling and XPath-like querying. XML has adopted an important position in various fields of software technology. This provides interesting possibilities in the perspective of web development, since the web is widely based on the XML technology family.

3.2 Scala basics

Program 3.1 shows an example demonstrating some basic Scala features. The syntax closely resembles Java, the main difference being the lack of redundant type declarations. Type declarations, when present, are also written backwards compared to Java, first the variable name and then the type after a colon. Semicolons at the end of lines are not needed although they may be added if preferred. In general, Scala is less verbose than Java. There is no need to write more than is actually needed to deduce the intentions of the programmer.

Scala provides a more powerful import mechanism than Java does. Importing multiple classes from a single package can be done in a single line. All imported classes can be renamed to avoid collisions or just to make them shorter. Imports can be used inside code blocks to reduce their visibility to the containing block.

Visibility of a declaration is by default public. It can be restricted in more fine-grained ways than in Java, for example the visibility of a class or a member can be limited only to a certain package.

Scala also supports Java-like annotations. For example by declaring the annotation called *BeanProperty* to a member variable (that is, a property) the compiler will generate JavaBean-style getters and setters for the property in addition to the ones Scala uses internally.

Calling methods that take no arguments can be done without the empty parentheses after the method name. Syntax for accessing a member property or a value is similar, so this way Scala confirms to the *uniform access principle* which states that accessing different kinds of members should look similar. This leaves the class implementor the possibility to later change for example a value definition to a method without the users of the class needing to change anything.

```
1  package net.lahteenmaki.thesis
2
3  import java.lang.{String => S, StringBuilder => JavaSB}
4
5  object HelloApp {
6    def main(args: Array[S]): Unit = {
7      val hello = new Hello
8      hello.prefix = "### "
9      val response = hello ! args
10     println("Greeted " + response._1 + " persons")
11   }
12 }
13
14 class Hello extends HelloBase with HelloWorldGreeting {
15   var prefix = ""
16   override def construct(targets: Array[S]) = prefix + super.construct(targets)
17 }
18
19 abstract class HelloBase {
20   val defaultGreeting: S
21
22   def !(targets: Array[S]): (Int, S) = {
23     val greeting = construct(targets)
24     println(greeting)
25     (targets.size, greeting)
26   }
27
28   def construct(targets: Array[S]): S = {
29     targets match {
30       case Array() => defaultGreeting
31       case Array(first) => "Hello " + first + "!"
32       case arr => "Hello " + arr.mkString("", " and ", "!")
33     }
34   }
35 }
36
37 trait HelloWorldGreeting {
38   val defaultGreeting = "Hello World!"
39   def defaultGreetingLength = defaultGreeting.length
40 }
```

Program 3.1: *Simple example program*

In the example Program 3.1, line 5 starts with the keyword OBJECT. This is a definition of a singleton class. Scala does not have the concept of static methods or variables, instead, every class may have a similarly named object as a companion which holds the global members. This singleton object is executable since it defines a method MAIN taking an array of strings as an argument and resulting in a UNIT, which corresponds to VOID in Java.

At line 7 there is a value HELLO. A VAL can not be reassigned, so it corresponds to the FINAL keyword in Java. Scala prefers the usage of values over variables as part of its functional paradigm.

Line 15 defines a variable with the keyword VAR. Member variables in Scala are considered properties, meaning that getter and setter methods are automatically added to the class by the compiler. Their behavior can be overridden when needed. At line 8 a string is assigned to the aforementioned variable. The setter method is automatically invoked instead of a direct assignment.

At line 22 there is a definition of a method named !. This can be done since the exclamation mark is a legal identifier in Scala. Methods like + and - can also be defined and so gain similar functionality to operator overloading in other languages. The method's return type on line 22 is defined to be a tuple of an integer and a string. Tuple is a typed, fixed-size, ordered list of elements in which the elements can have different types.

HelloWorldGreeting is a trait which corresponds roughly to a Java interface with optional method implementations. However, usage of traits is not limited to simple interface implementation. A class can extend multiple traits as mixins. Each trait may have abstract members and implement or override each others members. Class HelloBase has a value declaration with no actual value, which makes it abstract. It is extended by the class Hello with the trait HelloWorldGreeting. At first this might seem like another abstract class, but since the trait defines the abstract value, the resulting class is actually instantiable. Also vals, vars and types can be abstract.

Traits are a form of multiple inheritance, but not exactly the same as traditional multiple inheritance. Traits avoid the problem of multiple parents by ordering all the ancestors to a linear chain based on deterministic rules. Methods can override each others behavior and calling *super* has well-defined semantics. This brings many architectural possibilities but comes at a cost of complexity and class behavior that depends on the order of classes in the hierarchy.

Lines 29 to 33 show one of the most important features, pattern matching. Instead of a complicated set of instanceofs and ifs in Java, one can simply list different patterns. The patterns can be rather complex and the result comparing to Java is a huge reduction in lines of code. In this example the first case matches an empty array, the second matches an array of length 1 and the last one matches anything the first two did not match binding the value to a variable named arr.

The match-case construct in Scala roughly corresponds to a switch-case in Java, in that the structure is similar and the first match is selected. The alternatives, however, can be complex patterns like lists, tuples and types in addition to integers or enumeration values. Constructor patterns allow matching to any custom types. Variable names in the patterns, like first in line 31, are bound to the corresponding values and can be used in the expression to evaluate. Nested patterns and guards are also supported, see for example *Programming in Scala* [23, Chapter 15] for a more thorough explanation.

XML can be directly written in Scala. Curly braces can be used to escape out from the XML mode back to the Scala mode. For example a script defined in Program 3.2 would output:

```
<employees>
  <employee department="marketing">
    <name>jack</name>
  </employee>
  <employee department="development">
    <name>jones</name>
  </employee>
</employees>
```

```
case class Department(name: String)
val marketing = Department("marketing")
val development = Department("development")

def employee(name: String, department: Department) = {
  <employee department={department.name}>
    <name>{name}</name>
  </employee>
}

val someXML = {
  <employees>
    {employee("jack", marketing)}
    {employee("jones", development)}
  </employees>
}
println(someXML.toString)
```

Program 3.2: *Example of Scala XML syntax*

When the XML is printed out as a string, all escaping and encoding are handled appropriately. Everything is statically typed so trying to produce XML which is not well-formed will result in a compilation error. More information and examples can be obtained from the Scala XML web site [25].

Scala replaces null pointers with a type called OPTION, which can be either a NONE or a SOME. It is recommended that NULL should not be used to describe a missing optional value. Instead, the type should be an OPTION[T] where T is the type of the actual value. In the case of a method that might not return an actual value:

```
def resolveColorIfAny: Option[Color]
```

the caller of the method would be forced to handle both cases:

```
resolveColorIfAny match {
  case Some(x) => println("Color is: " + x)
  case None => // do nothing
}
```

Alternatively, the GET-method can be used to get the actual value.

The aforementioned syntax is somewhat verbose. Scala OPTION can also be seen as a collection containing either the single value or nothing. This allows the previous example to be rewritten in any of the following ways:

```
resolveColorIfAny.foreach( x => println ("Color is: " + x) )
resolveColorIfAny.map( "Color is: " + _ ) foreach println
resolveColorIfAny.flatMap( x => Some("Color is: " + x) ) foreach println
```

3.3 More advanced Scala

The previous section was an introduction to Scala. This section demonstrates some of the more advanced features. *Programming in Scala* [23] or the tutorials found in the Scala web site [26] are recommended reading for those seeking more knowledge.

Generics became a part of Java in version 5.0. They allow more restrictive typing within classes to increase static type safety, which results in less error-prone code. The generics in Scala differ from Java in that Java arrays are *covariant* whereas in Scala they are *nonvariant*. Other types are by default nonvariant in both. Covariance is that if type DOG is a subtype of ANIMAL, then REPRODUCTIVE[DOG] is a subtype of REPRODUCTIVE[ANIMAL] and for example a REPRODUCTIVE[DOG] can be assigned to a variable typed REPRODUCTIVE[ANIMAL]. Being able to assign REPRODUCTIVE[ANIMAL] to REPRODUCTIVE[DOG] would indicate *contravariance*. For *nonvariant* types neither is allowed. Another difference between the languages is that Scala uses definition-site variance whereas Java has use-site variance. See for example *Scala Overview* [27] for more detailed comparison.

A *partially applied function* is an expression where only some or none of a functions required arguments are given. The result is a function, which requires only the missing arguments to be passed in. For example in the following example:

```
1 def print3(first: String, second: String, third: String) = {
2   println(first + second + third)
3 }
4 val sayHelloTo = print3("Hello ", _: String, "!")
5 sayHelloTo("John")
```

the function PRINT3 is a function which prints out its three arguments, and SAYHELLOTO is a partially applied function which has fixed the first and the last argument. Only the missing one has to be provided when the function is called.

As already demonstrated in the previous section, complex classes can be composed by extending a class and mixing in traits. Calling SUPER.FOO() in a class normally executes a method from the first base class where it is defined. This also applies when the method is defined in a trait. Traits can extend other traits and mix in others and therefore SUPER.FOO() is a valid call in a trait.

In a complex composition it is not self-evident which method definition is executed if a method with an equal signature is defined in several of the traits. The order of execution is determined by a process called *class linearization* where all the classes from the inheritance hierarchy are ordered based on deterministic rules. For a class hierarchy like:

```
1 class B {}
2 trait T1 {}
3 trait T3 {}
4 trait T2 extends T3 {}
5 class C extends B with T1 with T2 {}
```

the linearization would be C -> T2 -> T3 -> T1 -> B. *Scala language specification* [28] gives a more detailed explanation of the linearization process.

This composition provides the programmer with some of the features of *Aspect Oriented Programming (AOP)*. Aspect oriented programming is a paradigm where some functionality is extracted from the program and inserted declaratively to appropriate places. An added behavior like this is known as an *advice*. Article *Scalable Component Abstractions* [29] provides an example of adding some logging as an *after* advice. As stated in [29], Scala does not cover all the possibilities of aspect oriented programming but gives a statically typed alternative to some common use cases.

In the normal case, when calling a method, the expressions passed in as parameters are first evaluated and then the method is invoked with the results of the evaluations. This is known as *strict evaluation* in programming language theory. Sometimes it would be preferable for an expression to be evaluated only in certain situations, for example a logging method would evaluate the message to log only if logging is actually enabled. In Java this requires passing in objects implementing a known interface in which the expression is evaluated. This is cumbersome to the user. Scala allows a method to define a parameter to be passed in as a *call-by-name*-parameter, in which case the expression for the parameter is evaluated when it is actually needed, if ever. For example to pass parameter MESSAGE by name so that it is evaluated only when logging is enabled, the argument type is to be prefixed with => symbol:

```
1 def log(level: Level, message: => String) = {
2   if (isEnabledForLevel(level)) {
3     actualLogger.log(level, message)
4   }
5 }
6
7 log(Level.DEBUG, "Data: " + largeDataObject.contentsAsXML)
```

Implicit parameters are method parameters that the user can, but does not have to, provide manually. When the compiler sees a method call that lacks a parameter that has been marked as implicit, it searches for a suitable implicit object. If either none or more than one suitable candidates are found, the compilation fails with an error. Otherwise the implicit object is used as the missing parameter. Suitable candidates are those marked with *implicit* keyword and that are within a well-defined scope. Scala language specification [28] gives a more detailed description.

Another powerful tool are *implicit conversions*. If the compiler sees a type error it looks for a conversion from the actual type to a suitable one. This applies for two kinds of uses which are demonstrated in Program 3.3. Both lines 10 and 11 are legal. In the former the compiler sees an implicit method to convert the incorrect type CAT to a correct DOG. In the latter the compiler knows that CAT has no method named BARK but there is an implicit conversion to a DOG that has. This

feature makes it possible to extend existing classes with new features, and in fact this is how Scala adds new methods to the existing Java API classes.

```
1  class Cat {}
2  class Dog {
3    def bark = "Wuf!"
4  }
5
6  implicit def Cat2Dog(c: Cat) = new Dog
7
8  def bark(dog: Dog) = dog.bark
9
10 bark(new Cat)
11 (new Cat).bark
```

Program 3.3: *Implicit conversion example*

Together with powerful operator overloading, implicit conversions allow construction of statically typed *domain specific languages (DSLs)*. DSLs are programming languages that are targeted at a specific problem domain. They are contrary to general purpose languages like Java or Scala. See Section 4.1 for an example. Implicit conversions together with implicit parameters are known as *implicits*.

When a structured data in textual format is to be handled programmatically, a parser is needed. Sometimes no existing parser for the structure is on hand. There are many tools to help in building a parser, but often their learning curve is too steep or integration to the project is difficult. Scala comes with a framework for building combinatorial parsers easily. The syntax resembles BNF closely enough and the framework has enough functionality to build simple parsers quickly. However, there is some new syntax to be learned and parsing in general can be considered an advanced topic. *Programming in Scala* [23] gives a good tutorial for the subject.

3.4 Scala pitfalls

There are some issues in Scala that may cause problems in certain situations. One issue stems from type inference:

```
1  def whoops() {
2    "This is lost"
3  }
4  def correct() = {
5    "This is correctly returned"
6  }
```

The first method does not return a STRING even though it would seem so, instead its return type is UNIT. This is because the compiler treats method definitions with no equals sign to return a UNIT. The second method is a correct example.

Another issue is related to Scala and Java interoperability. In Scala many symbols can be used in identifiers that are illegal in Java and even JVM. These symbols have to be encoded and thus for example method

```
def +() {...}
```

becomes in bytecode an encoded equivalent, which is seen in Java as

```
public void $plus() {...}
```

This does not prevent calling the method from Java side, but it does make it cumbersome. Also Scala's heavy reliance on first-class functions and other Scala-specific features makes interoperating from Java-side difficult.

The Scala compiler imports the PREDEF-object to every file. This among other things imports some implicit conversions like converting a STRING to a RICHSTRING. These may sometimes restrict the developer's possibilities. RICHSTRING for example implements ORDERED-trait with a smaller-than operator. This prevents an implicit conversion of a string to another object with a smaller-than operator because the compiler then sees two different implicit conversions from a string to a type which has that operator:

```
class MyPropertyThing {
  def <(someValue: Int):Boolean = {...}
}
implicit def string2MyPropertyThing(str: String) = new MyPropertyThing(str)
val hasLowValue = "foo" < 17     // not working!
```

A workaround for this is *unimporting* the corresponding implicit conversion from the PREDEF-object:

```
import Predef.{stringWrapper => _, _}
```

Scala's XML syntax sometimes lets content to get lost without a warning:

```
scala> val xml = {
     | {"hello"}
     | <span class="world">world</span>
     | }
xml: scala.xml.Elem = <span class="world">world</span>
```

This is easy to fix, but the problem is that bugs like this are not easy to see in the program code.

Java Enterprise Edition provides an annotation to define some fields of an object to be dependency-injected resources:

```
@Resource
private PersonDAO personDAO;
```

The field needs to be an assignable variable so that it can be assigned at runtime. Most of the examples in Program 3.4 fail. Examples 1 and 3 fail because the fields are values, that is constants which compile to have a FINAL modifier and thus cannot be assigned. Examples 2, 4 and 5 fail because Scala treats variables as properties and automatically creates getter and setter methods for them. The compiler adds the annotations to all the generated methods, but the @RESOURCE annotation is only allowed on a field. Even the PRIVATE modifier on example 5 makes no difference. Example 6 works because PRIVATE[THIS] modifier means that the variable is *object-*

local denying access even from other instances of the same class, therefore making any getter and setter methods unnecessary. Example 7 works because the annotation is on a method.

```
1  class Service(@Resource val dao1: PersonDAO,        // fails
2                @Resource var dao2: PersonDAO) {      // fails
3    @Resource
4    val dao3: PersonDAO = null                        // fails
5
6    @Resource
7    var dao4: PersonDAO = null                        // fails
8
9    @Resource
10   private var dao5: PersonDAO = null                // fails
11
12
13   @Resource
14   private[this] var dao6: PersonDAO = null          // works!
15
16   private[this] var dao7: PersonDAO = null
17   @Resource
18   def setDao7(dao7: PersonDAO) = {                  // works!
19     this.dao7 = dao7
20   }
21 }
```

Program 3.4: *Using Java EE dependency injection in Scala*

Moreover, if an injected object is only used from within an anonymous inner class, the compiled code for the variable becomes PUBLIC STATIC FINAL and thus cannot be assigned. This does seem more like a compiler issue than a language issue. One workaround is to avoid field and setter based dependency-injection whenever possible and inject constructor arguments instead.

The Scala compiler's behavior with annotations on properties is also a problem when creating Scala-based model objects:

```
1 class Employee {
2   @BeanProperty
3   @org.hibernate.validator.NotEmpty
4   var name: String = _
5 }
```

NOTEMPTY annotation is added also to the Scala-based property accessor methods, as well as the Java-based getter and setter. The annotation in this example is used by *Hibernate Validator* [30] and the implementation expects the annotation to be present only on fields or getter methods.

3.5 Scala vs Java

Scala runs on the standard Java platform and interoperates seamlessly with all Java libraries. Scala programs compile to JVM bytecode and their runtime performance should be similar to equivalent Java programs. Scala code can call Java methods, access Java fields, inherit from Java classes, and implement Java interfaces. [23]

Scala's classes corresponding to Java's primitive types are actually classes, but are implemented as Java primitives when possible due to performance issues. Scala extends many Java types with additional methods, for example JAVA.LANG.STRING where Scala has added methods like TOINT.

Compiled Scala code is not much different from compiled Java code. Therefore a complete conversion to Scala is not necessary at any point as Scala components may be added to existing Java programs gradually. Or, a new project can be made using Scala where appropriate and Java where it suits better.

Although the resulting bytecode is quite similar, Scala's syntax can be quite different. Some things differing from Java are the placement of type annotations and the lack of static fields and methods. One of the most visible differences stems from Scala's type inference, which lets the developer leave out most of the type annotations.

Scala is an object-oriented language with heavy reliance to the functional paradigm. Working with immutable classes might be difficult at first for a developer who is used to stateful programming. The Scala standard library includes many classes that are biased toward the functional paradigm, including immutable collections.

The most important differences emerge when Scala's additional language features are taken into use. Traits are like Java interfaces, but they can have method implementations and fields. Mixing in traits is something that is not available in Java. The same applies to features like pattern matching, implicits, operator overloading and XML syntax. Scala also provides more powerful type parameterization, import mechanism and method visibility definitions.

Some of the features are actually available in Java but the awkward syntax restricts their usefulness. Examples of these are lazy values, call-by-name parameters, first-class functions and partially applied functions.

It could be said that Scala's differences compared to Java are two-fold: First of all, it adds syntactic sugar to make many features practical to use. Second, it adds some features that are completely missing from Java.

Programming concurrent applications is known to be difficult. Java tried to address this issue by specifying a common memory model [31, Chapter 17] so that all Java programs could be run on any hardware no matter how different they are or how many processors they have. At the same time Java decided to allow as many compiler optimizations as possible to increase performance.

The end result is programs, that sometimes behave unexpectedly when concurrency issues are not taken properly care of. Java version 5 made some improvements to the API, but the subject is still difficult. Scala addresses this issue with an *actor* [32] based library that implements *Erlang*-based [33] actor model. Actors share no memory and only communicate with message passing, which makes concurrent pro-

gramming safer. Message passing might constitute a performance penalty comparing to the Java approach.

3.6 Tools

The Scala distribution comes with two compilers: *FSC — The Fast Scala Compiler* and *Scalac — The Scala Compiler*. It is also bundled with an interpreter for experimenting with the language.

Scalac is similar to the standard Java compiler *javac* in that it performs the basic compiling with similar options. FSC is a compilation daemon for Scala. It uses the same compiler instance to compile on subsequent runs as long as the classpath remains the same. This results in a faster compilation because the Java Runtime does not have to be started each time.

The Scala distribution also includes some *Apache Ant* [34] tasks. These are named *scalac*, *fsc* and *scaladoc*, which compile the project using either scalac or FSC compiler and generate *Scaladoc* documentation for the project. Experimenting with the Ant tasks was not in the scope of this thesis work. The Scala Ant tasks web site says that the compilation tasks have some issues with nested classes and classes named differently from the source file:

> Ant uses only the names of the source and class files to find the classes that need a rebuild. It will not scan the source and therefore will have no knowledge about nested classes, classes that are named different from the source file, and so on.
>
> *Scala Ant tasks web site [35]*

The *Maven plugin for Scala* [36] integrates seamlessly to *Maven* [37] lifecycle and understands projects with both Scala and Java source files. It also provides an easy way to run a Scala class or start the Scala interpreter with the project classpath. Another feature is a continuous compilation mode, which starts the FSC daemon to compile files whenever they change.

Java developers are used to getting significant help from the development environment. Scala, being targeted to the existing Java community, is probably expected to offer similar development experience. At the time of writing there is a Scala plugin for three major *integrated development environments* (*IDE*s): *Eclipse* [38], *NetBeans* [39] and *IntelliJ IDEA* [40].

The Scala IDE for Eclipse has the basic functionality like integration with the Eclipse build process, integration with some Eclipse views, and support for projects with both Scala and Java code. Unfortunately the plugin is not stable enough for production use. Contextual help does not work all the time and basic text editing at times fails with an exception causing the editor to switch to a read-only mode.

The plugin either requires excessive amounts of memory or leaks it, which might explain why the plugin seems to break the whole environment.

The Scala plugin for NetBeans 6.7 has the basic functionality, providing the developer help with renaming, pair matching and some code completion. Unfortunately there are some issues, like "go to declaration" or contextual help not always working. Also the whole IDE sometimes seems to become unstable when programming in Scala.

The Scala Plugin for IntelliJ IDEA feels the most advanced. It already has features like code completion, code formatting, XML support, auto-import functionality and even some simple refactoring. The biggest disadvantage is IDEA being a commercial product and hence not a viable option for every Java developer, although the free community edition can be used to develop in Scala. Also the editor does not seem to complain about all the compilation errors.

All the aforementioned IDE plugins are more or less incomplete. They are all still relatively young and a more comprehensive analysis could be carried out after a few years.

4. UTILIZING SCALA FOR SOME COMMON USE CASES

The multitude of features in Scala open various possibilities to improve and re-think existing programming concepts. This chapter covers two of them which are both recognized as important aspects in contemporary software development. First the ability to create *Domain Specific Languages* is investigated after which some possibilities to improve application testing are described.

4.1 Domain Specific Languages

Domain specific languages (*DSL*s) play an important role in contemporary software development. No single language can provide the simplest and most specific tools for all possible problems. A developer has to either settle with a language not perfect for the problem, or use a completely different language specific for the problem domain.

A language suitable for the problem domain results in a cleaner and less error-prone implementation. On the other hand, a different language makes the project less consistent and also requires additional know-how from the developers. If this language is not binary compatible with the main language it also requires an interaction layer between the components which is an additional source for potential errors.

4.1.1 Scala for DSLs

Scala has a flexible syntax as it does not require semicolons at line ends, dots in certain object traversals, parentheses after parameter-less method calls or curly braces when not needed. It also allows to leave out additional type annotations, replace argument names with an underscore or even leave out the whole arguments when passing a function with matching parameter types as a method parameter. If we define a function:

```
1 def isPositive(number: Int) = number > 0
```

then all the following are equal:

```
1 someIntSeq.filter( (x: Int) => isPositive(x) )
2 someIntSeq.filter( x => isPositive(x) )
3 someIntSeq.filter( isPositive(_) )
4 someIntSeq.filter( isPositive )
```

```
5 someIntSeq filter( isPositive )
6 someIntSeq filter isPositive
```

Most importantly, Scala has implicit arguments and implicit conversions. These together with flexible syntax allow creation of almost arbitrary domain-specific languages which are pure Scala and thus interoperate flawlessly with other components written in Scala. Everything is also statically typed. There exists several alternative tools for creating DSLs but one might argue that Scala's static type checking and close integration to other program code are big benefits.

4.1.2 Hibernate Criteria API

The criteria API is a part of *Hibernate* [9] — an open-source Java persistence framework for relational databases — and allows the construction of queries dynamically. It is powerful and statically typed but some might argue that its syntax is overly verbose.

Scala with its implicits and flexible syntax can be used to modify existing APIs. The end result is a new syntax with all the features of the Criteria API still available as before and everything statically typed. One possibility is to use an SQL-like syntax demonstrated in Program 4.5. This creates an equal query to the one in traditional syntax presented in Program 4.6.

```
1  package net.lahteenmaki.thesis
2
3  import net.lahteenmaki.scalax.data.hibernate.Criteria._
4  import net.lahteenmaki.thesis.model.Employee
5
6  import Predef.{stringWrapper => _, _}
7
8  class ExampleCriteria {
9    val detachedCriteria = SELECT DISTINCT (
10       COUNT(*) AS "employees", GROUP("lastName"), GROUP("firstName"),
11       GROUP("skillLevel"), GROUP("dep.name")
12     ) FROM (
13       classOf[Employee] AS "emp", "department" AS "dep"
14     ) WHERE (
15       ("lastName" LIKE "Jac%") AND
16       ("firstName" IN ("Joshua", "Michael", "Janet") ) AND
17       ("email" IS NOT NULL) AND
18       ( ("skillLevel" < 2) OR ("skillLevel" >= 4) )
19     ) ORDER BY (
20       "lastName" ASC, "firstName" ASC
21     )
22 }
```

Program 4.5: *Example usage of Hibernate Criteria API with SQL-like syntax*

It is a matter of taste whether the new syntax is better than the original one, and implementing the conversion is always additional work. Still, often the simplified syntax in logically important parts of the program is worth the additional work.

```
package net.lahteenmaki.thesis;

import net.lahteenmaki.thesis.model.Employee;
import org.hibernate.criterion.*;
import static org.hibernate.criterion.Projections.*;
import static org.hibernate.criterion.Restrictions.*;

public class ExampleCriteriaJava {
  DetachedCriteria detached = DetachedCriteria
    .forClass(Employee.class, "emp")
    .createAlias("department", "dep")
    .setProjection(distinct(projectionList()
      .add(alias(rowCount(), "employees"))
      .add(groupProperty("lastName"))
      .add(groupProperty("firstName"))
      .add(groupProperty("skillLevel"))
      .add(groupProperty("dep.name"))

      )
    )
    .add(like("lastName", "Jac%"))
    .add(in("firstName", new String[]{"Joshua", "Michael", "Janet"}))
    .add(isNotNull("email"))
    .add(disjunction()
      .add(lt("skillLevel", 2))
      .add(ge("skillLevel", 4))
    )
    .addOrder(Order.asc("lastName"))
    .addOrder(Order.asc("firstName"))
    ;
}
```

Program 4.6: *Equal criteria query to Program 4.5 with traditional syntax*

Criteria API and the syntax presented here are merely an example. The point is that any existing API can be transformed to another syntax to make it more accessible.

4.1.3 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format which is easy for humans to read and write and easy for machines to parse and generate. [41]

JSON documents can be generated from a suitable programmatic data structure to avoid syntax errors or escaping issues. However, sometimes the data is so problem-specific or dynamic in nature that creating a suitable data structure results in too much of additional complexity. Sometimes the amount of data is so small that it is simpler to just write it out as a string.

JSON can be written in Scala when implicits and Scala's syntax are utilized in a suitable way. Escaping is done automatically by the supporting classes and the compiler informs about errors in syntax. The whole implementation consists of only a few simple classes and the result is that the developer can write in almost pure JSON syntax, as demonstrated in Program 4.7.

```
1  package net.lahteenmaki.thesis
2
3  import net.lahteenmaki.thesis.model._
4  import net.lahteenmaki.scalax.web.json.JSON._
5
6  class ExampleJSON {
7    val allEmployees = Seq[Employee]()
8    val allDepartments = Seq[Department]()
9
10   val employees = "employees" := allEmployees.map { emp =>
11         "employee" := {
12           "id" := emp.getId +
13           "firstName" := emp.getFirstName +
14           "lastName" := emp.getLastName +
15           "department" := emp.getDepartment.getName
16         }
17   }
18
19   val departments = "departments" := allDepartments.map { dep =>
20         "department" := (
21           "id" := dep.getId,
22           "name" :=  dep.getName,
23           "amountOfEmployees" := dep.getEmployees.size
24         )
25   }
26 }
```

Program 4.7: *Example of writing JSON in Scala*

4.1.4 Wiki parser

As mentioned in Section 3.3, Scala comes with an API for creating combinatorial parsers. These are not meant to completely replace any existing parser generator tools but instead provide a simple alternative when performance is not the most critical issue. As the authors of *Programming in Scala* say:

> One downside of combinator parsers is that they are not very efficient, at least not when compared with parsers generated from special purpose tools such as Yacc or Bison.
>
> *Programming in Scala [23, Section 31.11]*

The API utilizes heavily Scala's flexible syntax. It demonstrates how one can build APIs that seem part of the language specification although they actually are not.

For demonstrative purposes, in this thesis work a parser for a simple wiki text format was created. The parser understands four different classes of commands and produces XHTML output utilizing Scala's XML support to provide static type safety internally. The parser implementation is quite simple and it can easily be extended with new commands or new output formats.

Here the implementation is briefly described with code examples. The reader is expected to have some basic knowledge of parsing and its related terminology.

One of the main parts of the parser is the *lexical analyzer*, or *lexer*. Its responsibility is to convert a sequence of characters to a sequence of *tokens* which are the entities that the actual parsing handles. The lexer consists of an ordered set of rules that map characters to tokens:

```
class WikiLexical extends Lexical with WikiTokens {
  def token: Parser[Token] = {
    (
      EofCh              ^^^ EOF
    | '\n'               ^^^ EOL
    | keyword            ^^ { case x => Keyword(x)}
    | modifier           ^^ { case x => Modifier(x)}
    | '\\' ~ modifier    ^^ { case first ~ rest => Text(first + rest) }
    ...
```

Tokens are defined as simple parsers themselves:

```
trait WikiTokens extends Tokens {
  case class Delimiter(chars: String) extends Token {
    override def toString = "'" + chars + "'"
  }

  case class Text(chars: String) extends Token {
    override def toString = "text: " + chars
  }
  ...
```

```
trait WikiTokenParsers extends TokenParsers {
  type Tokens <: WikiTokens
  type ResultType
  import lexical._
  implicit def delim(char: Char): Parser[ResultType] = {
    accept(Delimiter(char.toString)) ^^ (_.chars)
  }
  def text: Parser[ResultType] = {
    elem("text", _.isInstanceOf[Text]) ^^ (_.chars)
  }
  ...
```

Parsing rules describe the actual structure for a legal input:

```
abstract class WikiParser extends WikiTokenParsers
  ...
  def lineCommand = oneOf(lineParsers, {
    c: LineParser[ResultType] => {
      acceptLineStart ~ c.comName ~> lineCom <~ (eol|eof|pgSplit) ^^ {
        case line => c.parse(line).get
      }
    }
  })
  def paragraph: Parser[ResultType] =
    rep1(kwCom|textCom|text|special|space|(eol <~ not(lineCom))) ^^ {
      case content if (content.length > 0) => makeParagraph(content)
      case content => content
    }
  ...
```

Understanding the combinator parser API takes some time, but afterwards simple parsers can be built quickly. The syntax might seem difficult at first, though it should

look familiar to one with some knowledge in parsing.

4.2 Testing

Testing is an important part of any software development process. There are some useful tools [42] for static code and bytecode analysis that have proven to reduce the amount of errors in software products. Using a programming language with an advanced static type system can reduce the amount of type errors to a bare minimum.

Dynamic languages are becoming more and more popular. They have no static typing at all, and so limit the possibilities of static analysis tools. Yet it seems that production rate is no worse than with statically typed languages. These languages rely on testing to find the bugs and *Test Driven Development (TDD)* [43] is an important concept.

Advanced tools and programming languages cannot completely eliminate the need for testing. On the other hand, a language like Scala can provide tools to ease and improve testing.

4.2.1 ScalaTest and ScalaCheck

ScalaTest [44] is a framework designed for any kind of testing in a clean and intuitive syntax. It uses traits and implicits to create an environment where the developer can produce test cases that are easy to create and to follow. Program 4.8 shows a simple test which is a *JUnit* [45] test complemented with ScalaTest *Matchers*.

```
1  package net.lahteenmaki.thesis
2  import org.scalatest.junit.JUnit3Suite
3  import org.scalatest.matchers.ShouldMatchers
4
5  class Hello {
6    def talk = "Hello!"
7  }
8
9  class HelloTest extends JUnit3Suite with ShouldMatchers {
10   def testSaysHello() = (new Hello).talk should be ("Hello!")
11 }
```

Program 4.8: *Simple JUnit-like testing*

ScalaTest comes bundled with traits for *Behavior-driven development (BDD)*. This style focuses on writing specifications instead of procedural test cases. Program 4.9 shows an example of different specification-style test cases. Another bundled-in trait is for *features*, which is targeted for functional, integration or acceptance testing [44]. Here the features to test are written descriptively under scenarios with preconditions, after which the actual tests follow. The framework is claimed to be easily extensible for custom needs.

```
package net.lahteenmaki.thesis
import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.scalatest.{Spec, WordSpec, FlatSpec}
import org.scalatest.matchers.ShouldMatchers

class Goodbye {
  def talk = "Goodbye!"
}

@RunWith(classOf[JUnitRunner])
class GoodbyeSpecTest extends Spec with ShouldMatchers {
  describe("A Goodbye") {
    it("should say 'Goodbye!'") {
      (new Goodbye).talk should be ("Goodbye!")
    }
  }
}

@RunWith(classOf[JUnitRunner])
class GoodbyeFlatSpecTest extends FlatSpec with ShouldMatchers {
  "A Goodbye (when called)" should "say 'Goodbye!'" in {
    (new Goodbye).talk should be ("Goodbye!")
  }
}

@RunWith(classOf[JUnitRunner])
class GoodbyeWordSpecTest extends WordSpec with ShouldMatchers {
  "A Goodbye" when {
    "called" should {
      "say 'Goodbye!'" in {
        (new Goodbye).talk should be ("Goodbye!")
      }
    }
  }
}
```

Program 4.9: *Tests based on behaviour-driven development*

ScalaTest integrates well with existing tools like JUnit, *TestNG* [46], Ant and Maven. This makes it straightforward to use ScalaTest in an existing build environment perhaps complementing other tools. Since the syntax is Scala, it also integrates with Java enabling testing of existing Java classes.

ScalaCheck [47] is an extension of the *Haskell* [48] *QuickCheck* [49] library for Scala. It generates unit tests automatically based on the test specification. The test specifications consist of testable properties and generators for custom or built-in types.

ScalaCheck comes with generators for common built-in types but others can be created when needed. Program 4.10 shows an artificial example using a custom integer generator and property labeling. The example integrates ScalaCheck with ScalaTest framework and is also directly runnable with JUnit and Maven.

4.2.2 Testing the view

In web applications the view has traditionally been the hardest part to test. One reason is its complexity: a view might contain lots of information which is gathered

```
package net.lahteenmaki.thesis
import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.scalatest.FunSuite
import org.scalatest.prop.Checkers
import org.scalacheck._
import org.scalacheck.Prop._

@RunWith(classOf[JUnitRunner])
class ScalaCheckTest extends Properties("Tests") with FunSuite with Checkers {
  implicit val evenIntegersGen = Arbitrary.arbitrary[Int] suchThat (_ % 2 == 0)

  test("even integers") { check {
    forAll(evenIntegersGen, evenIntegersGen) { (i1: Int, i2: Int) => {
      ((i1 * i2) % 2 == 0) :| "multiplication is even" &&
      ((i1 + i2) % 2 == 0) :| "sum is even"
    }}
  }}
}
```

Program 4.10: *A simple test written with ScalaCheck*

from various parts of the application. Another reason is its dynamic nature: a view is often directly printed to the client without generating any resulting structure that could be analyzed. The most fundamental reason is the availability of tools used, and the possibilities they provide, which in Java EE boils down to how easy JSPs are to test.

Complexity can be reduced by splitting the view to multiple smaller parts, which all handle their simple task and are hence easier to test. With JSPs this would mean moving functionality to other JSPs, tags or tag files. With Scala this means creating regular methods which produce their corresponding part of the output.

JSPs do not produce any result structure but write straight to the output. Testing the result would mean reading in the resulting string or a byte stream and performing tests on it. Scala has language support for XML and can produce document trees and perform queries to them with a simple syntax. In Java the resulting stream can be read in as an XML document and queries performed to it with regular JDK tools. The syntax, though, is awkward and verbose and there is no easy way to test smaller units than the whole document.

Traditional JSP tags can be tested with traditional methods because they are regular Java code. Testing JSPs and custom tag files is difficult because they are meant to be compiled into runnable code only in the application server. In Scala the corresponding structures are regular methods, so testing is simply checking the return value of the method with different parameters.

One might argue that testing the view is a waste of time because all the logic should be in some other place anyway. On the other hand, the view forms the content that the users are actually looking at, and so bugs in the view are easily visible. Due to language support for XML and easy XPath-like querying, Scala

makes it rather easy to create basic tests for the view layer. Program 4.11 shows a simple test class for testing a view which renders a list of departments in the system.

```
@ContextConfiguration{val locations=Array("/WEB-INF/applicationContext.xml")}
class ListDepartmentsTest extends FunSuite with Checkers {

  @Test
  def test = check { forAll { (deps: Array[Department]) => {
    val result = new ListDepartments with ViewBaseMock {
      val departments = deps
    }.page

    val trElements = result \\ "table" \ "tr"
    val hrefs = trElements.drop(1) \\ "td" \\ "a" \\ "@href"

    def params(uri: String) = {
      uri.dropWhile( _ != '?').drop(1).mkString.split("&")
    }

    all(
      "Amount of <tr> must equal number of departments + header" |:
        trElements.size == deps.size + 1,

      "All departments must have links" |:
        hrefs.size == deps.size,

      "Each href must have the correct request parameter 'id'" |: {
        hrefs.map( href => {
          params(href.first.text).filter(_.startsWith("id=")).first
        }).sameElements (deps.map( "id=" + _.getId))
      }
    )
  }}}
}
```

Program 4.11: *Basic tests for department listing view*

5. WEB PROJECT WITH SCALA USING SPRING FRAMEWORK AND HIBERNATE

Often ideas that seem good in theory run into unforeseen problems in practice. A reference implementation is an essential part of a new concept. This chapter introduces a simple web application which was produced as a part of this thesis work to experiment with Scala's possibilities in a real-world situation. The application was implemented in both Scala and Java to allow a direct comparison. It has a small domain model and only simple business logic to add, remove and list departments and employees, which limits its ability to demonstrate some cases where Scala might prove useful. However, many important concepts are still covered.

5.1 Project structure

Scala is not yet widely used and might thus cause unpredictable problems. Interoperability with Java can be leveraged to reduce risks by using Scala in only some parts of the project. A simple web application is often split into four layers: domain model, data access, business logic and web layer for binding the other layers to the web environment.

Maven is a tool for managing projects and their dependencies. This example project uses Maven to handle the build process and to switch between different implementations. The example project in fact consists of twelve different projects (hereafter modules) illustrated in Figure 5.1. The figure also shows the relations and the implementation language for each module. Plain yellow boxes represent modules. Those on the green background are implemented in Scala, while those on the red background are in Java.

Dao and *service* are interface modules that contain the interfaces for the DAO and service implementations. Modules whose names end with *Scala* are the Scala implementations and those ending in *Java* contain the same functionality implemented in Java.

Web contains web application specific components implemented on top of Spring Framework and also the view layer — in this case XHTML — implemented using Scala. There is also a part of the view implemented with JSPs to compare the implementations and their performance.

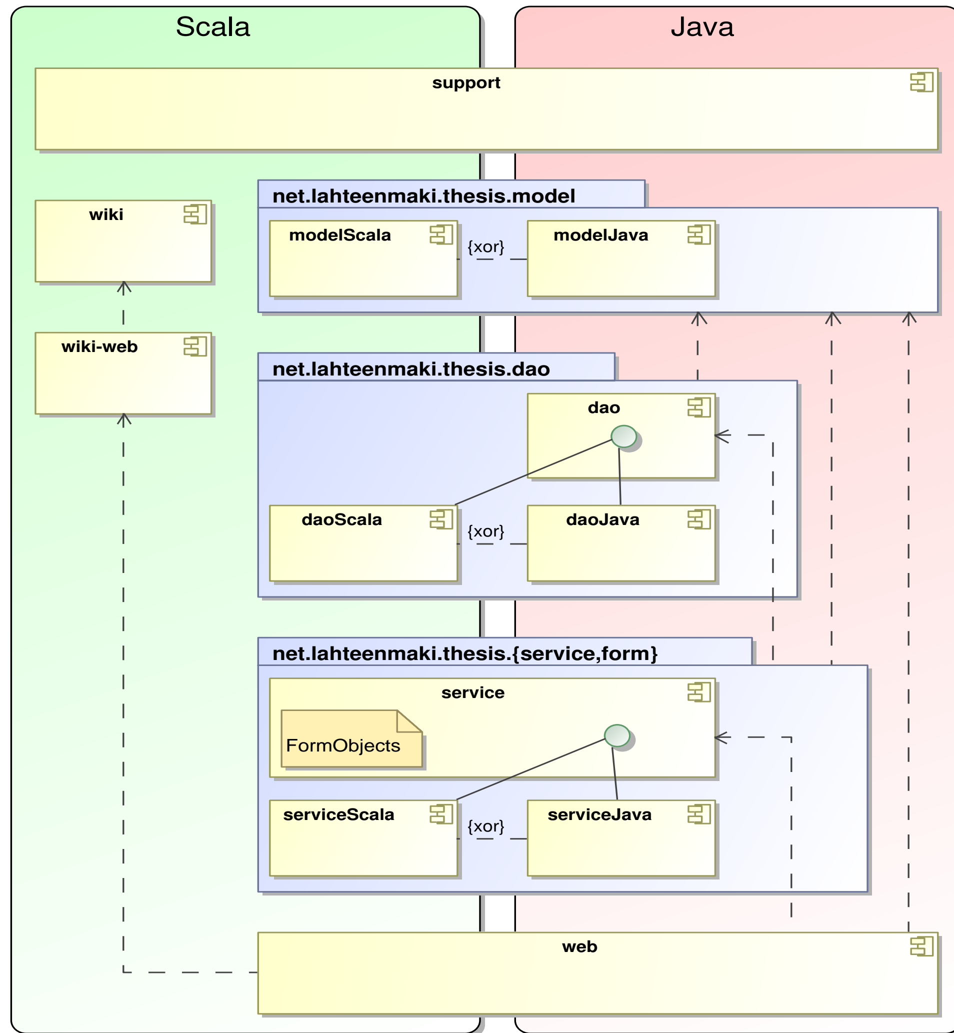

Figure 5.1: *Project structure*

Wiki consists of a small parser for a wiki language implemented with Scala's combinatorial parsers. *Wiki-web* has most of the classes needed to build a web application using the wiki parser, like model and service classes. *Support* is the main part of this work and contains all the functionality needed by the other modules, mostly implemented in Scala.

Maven can be utilized to choose which implementations to use when building the project. This way it can be shown that there is no need to implement the whole project in Scala. Instead, for example the view and DAO layers could be implemented in Scala while the model, service and web application specific parts are in Java. A part of the view can even be implemented with JSPs while the rest is in Scala.

There are, however, some issues when interoperating between Scala and Java. The interfaces have to be made Java-compatible if they are to be used from Java. This might also restrict or complicate the interface implementations.

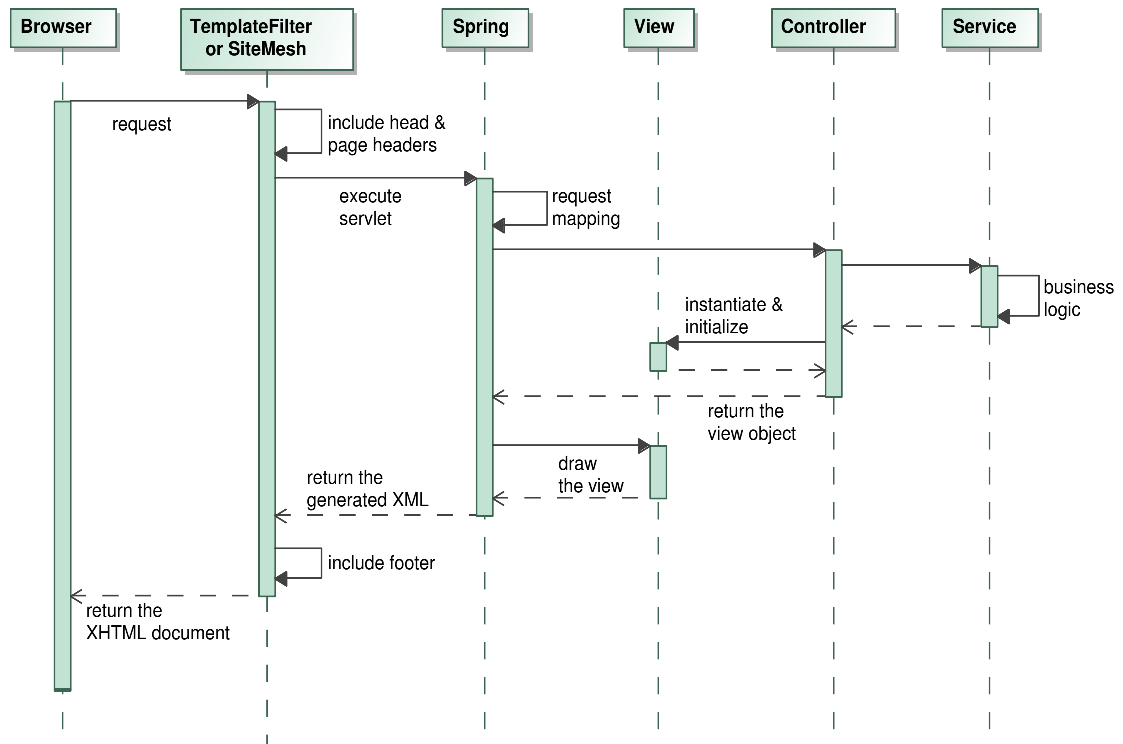

Figure 5.2: *Execution flow of the application*

5.2 Execution flow, decorators and mapping

Figure 5.2 shows the execution flow of the application, which roughly corresponds to a basic web application. An incoming HTTP (HyperText Transfer Protocol) request goes through some servlet filters and the decorator filter after which it is passed to Spring Framework. Spring maps the request to a controller binding and validating request parameters when appropriate. The controller calls appropriate service methods which interact with the database, after which the control is returned to the controller which prepares and returns an appropriate view to Spring. Spring then uses the view to produce the resulting XHTML page and returns it to the client through the decorator.

A decorator is something that can be wrapped around a regular view. A common use case is to include a header, a footer and a menu to every page of an application. Using decorators, the references to these common elements can be excluded from the pages themselves. SiteMesh [50] is one such decoration framework. The basic usage is to create the decorator page with the common elements for example with JSP and mark the place where the actual content is to be included. A roughly corresponding component was made in this thesis utilizing Scala's XML syntax so that the decorator page can be directly written in Scala. Therefore the decorator is not restricted to whatever SiteMesh or some other decoration framework provides.

There is also static typing in courtesy of the Scala compiler.

Spring provides multiple ways for request mapping including XML based and Java annotation based mapping. Doing the mapping in a pure Java class would put all the mappings to a single place and provide some static typing, but the syntax is awkward. Scala's implicit conversions and operator overloading can be used to make the syntax cleaner.

Django is a web framework written in Python. One of its main features is request mapping based on regular expressions. It also supports capturing groups as arguments for the target method. The mappings can be reverse matched so that URLs can be inserted to a page by simply referring a mapping by name. This way the URLs do not have to be written in various parts of the program and are easier to modify when needed.

Program 5.12 shows a simple mapping class which relies on exact matching of the request URL. Mappings are listed one by one in the class body. The URL and the target controller are separated by the arrow operator. This example compiles because the base class defines an implicit conversion from a string to an instance of a mapping class, which defines the arrow operator. The URL-string is implicitly converted to a mapping object, which is then initialized with the target controller. This occurs only when the base class is instantiated, so the request time overhead is minimal. The last line defines the concrete mapping object that contains all the different mappings from the previous traits.

```
1  package net.lahteenmaki.thesis
2
3  import net.lahteenmaki.scalax.web.spring.configuration._
4  import net.lahteenmaki.scalax.web.spring.controller.ParameterizableViewController
5  import net.lahteenmaki.thesis.controller.{JSONDataController => JSON}
6  import net.lahteenmaki.thesis.controller.department._
7  import net.lahteenmaki.thesis.view.Frontpage
8
9  trait SimpleMappings extends SimpleHandlerMapping {
10   val I = "/index.html" -> new ParameterizableViewController(new Frontpage)
11
12   "/deps/list.html"           -> classOf[ListDepartmentsController]
13   "/ajax/departments.json"    -> JSON.allDepartments _
14   "/ajax/mainDepartment.json" -> (JSON.department(1, _, _))
15 }
16
17 trait RegexMappings extends RegexHandlerMapping {
18   val S = "^/deps/show[.]html.*[?&]id=\\d+" -> classOf[ShowDepartmentController]

19   "^/ajax/dep[.]json.*[?]name=([^&]+)" -> (JSON.departmentByName($(1), _, _))
20   "^/ajax/dep[.]json.*[?]id=(\\d+)" -> (JSON.department(new $[Int](1){}, _, _))
21 }
22
23 object Mapping extends HandlerMappingBase with SimpleMappings with RegexMappings
```

Program 5.12: *Scala-based request mapping*

The example project is based on Spring, so the mapping object can register Spring controllers as targets. It is also possible to utilize Scala's partially applied functions

to register arbitrary methods as targets, as long as the method takes the request and response objects as parameters. Line 13 shows such a mapping. On line 14, the first parameter of a three-parameter method is provided as a constant.

RegexMappings trait in Program 5.12 contains mappings based on regular expressions. A method can be used here as a target object in the same way as on line 14. In addition, groups from the regular expression can be captured and used as method arguments, as demonstrated on lines 19 and 20. The former shows how a group can be simply referenced by syntax $(x) where x is the group index. This is actually a method call which returns a String.

Spring has extensive support for property editors defined in the *Java Beans* standard [51]. In this implementation the registered property editors can be used to automatically convert the group value to a suitable type. Line 20 shows how the group value is converted to an integer. The syntax is a bit awkward here, new keyword and the trailing curly braces are needed because an anonymous inner class has to be used to retain the type information at runtime.

The reverse mapping present in Django can be done by assigning a mapping to a value, and simply referencing that value instead of the corresponding URL. In case of SimpleMapping on line 10 the resolved URL is exactly the mapping URL. In the case of RegexMapping on line 18 the URL is resolved based on predefined rules. In this case groups are replaced with arguments given when referencing the mapping.

In addition to providing the capability for a Django-like mapping, everything is statically typed. Mapping is of course not restricted to the examples given here. It is fairly easy to create a mapping class for example for Ant-like syntax.

5.3 Form objects, binding and validation

Modifying stored data is a common use case for a web application. This can be achieved with HTML forms, which themselves provide no binding or validation logic. Web frameworks have developed ready-made components to simplify reading the form data from request parameters, performing validation and binding the data to some program components, for example Java Beans.

The objects responsible for handling the data in the forms are here referred to as *form objects*. The existing domain model classes can in general not be used for forms, since the form data does not usually have one-to-one correspondence to the model properties. A common way is to create a *Data Transfer Object (DTO)* which is a *Plain Old Java Object (POJO)* containing only the properties needed for the form in question. This DTO is then manually initialized from the corresponding model properties, manually validated after binding the request parameters and then the values are manually copied to the corresponding model objects.

Advantages of this approach include simplicity and clear separation of concerns. A disadvantage is the missing relation between the property in the form object and the corresponding property in a model object, which often correspond one-to-one even though the whole classes do not. Thus any information provided in the model object for its properties is not available for validation and no automatic binding in either direction can be performed.

The relation can be made by replacing the property in the form object with an object encapsulating the required information. This way the framework can for example perform automatic validation based on type information and annotations in the model object. At the same time rules for *client-side validation* can be generated.

Unfortunately, the syntax of Java makes all this somewhat awkward to use. In Scala due to type inference, the type declarations of the form object properties can be left out and implicits can be used to convert the form object properties to actual values transparently. Call-by-name parameters provide a clean way to initialize the form object from a model object. The result is a clean and simple form object definition with minimal information duplication.

Program 5.13 gives an example of a form object used to edit an existing department in the system. Line 19 gives the most simple property definition which has no relation to any model object and whose actual type is given explicitly as a type argument. The curly braces following each field definition force anonymous inner classes to be created for each field. This ensures that the type information is available at runtime since otherwise it would not due to type erasure.

```
1  package net.lahteenmaki.thesis
2
3  import net.lahteenmaki.thesis.model._
4  import net.lahteenmaki.scalax.web.form._
5
6  import org.hibernate.validator.{NotNull, Length}
7
8  class EditDepartmentObj extends FormObj {
9    val d = new Department
10   implicit val D = classOf[Department]
11
12   val id = new Field(d.getId) {}
13   val status = new Field("open") {}
14   val name = new Field(d.getName, "name") {}
15
16
17   @NotNull
18   @Length {val min=3, val max=10}
19   val captchaAnswer = new Field[String] {}
20 }
```

Program 5.13: *Scala-based form object definition*

Line 12 shows a field definition with initialization of the field value. The constructor argument is a by-name parameter so the method call to *d.getId* is not made unless the creator of the form object specifies the fields to be initialized. The actual

value is whatever the method call returns, and it can be specified by overriding the D-object which acts as a container for initial values:

```
1 new EditDepartmentObj {
2     val d = defaultValuedDepartment
3     override val initializeFields = true
4 }
```

The actual type of the field is inferred from the return type of the method. Line 13 is otherwise similar, but the initial value is given as a constant.

Using metadata information from the corresponding property in a model object requires the use of reflection. Line 14 gives an example where the name of the corresponding property is given as a second constructor argument. This form also takes the model object class as an implicit third argument. Line 10 is there to provide this implicit and is in this example otherwise not needed.

Program 5.13 only demonstrates the use of a single model object in a form, but nothing prevents using more than one. Form object classes can also be extended and combined.

The framework classes validate the form data against type information and *Hibernate Validator* [30] rules defined in the model class. They also generate client side validation rules that can be automatically used with *JQuery [52] Validation plugin* [53] accompanied with *JQuery Metadata plugin* [54]. The field on line 19 of Program 5.13 also has some Hibernate Validator annotations. The framework also validates the data against rules defined in the form objects.

Hibernate Validator is not a requirement but it provides some ready-made validation rules with restrictions for the database schema. It is also extensible with custom rules. JQuery Validation plugin is also an arbitrary choice. Both components are used as Spring Beans and are thus easy to turn off. Similar components can of course be created for custom annotations or some other client-side validation tool. The validation could have been implemented in Java and using Scala did not bring in any noteworthy advantages.

For binding request parameters to form object fields a Spring-based helper trait was made. This trait overrides Spring FormController's createBinder and processFormSubmission methods to create a custom binder and to populate the model object with possible error data. Spring property editor support is used to look for registered transforms from string parameters to arbitrary field types. Binding can be enabled by mixing in the trait to the controllers in question:

```
1 class AddDepartmentCtrl extends SimpleFormController with FormObjBinding
```

Binding requires that parameter names and their correspondence to form object fields are known, so some helper methods were also made for the view to ensure that the programmer needs not know about the internals when building the form view. The end result is that almost all the possibly complicated inner workings are

completely hidden from the programmer.

For client-side validation a mapper from Hibernate Validator rules to JQuery Validation plugin rules was made, and a light framework to create more mappers was implemented. As JQuery validation rules are in JSON, Scala's ability to write statically typed JSON was utilized:

```
val rules: JSONObject = hibernateValidatorAnnotation match {
  case x: Length => "rangelength" := Seq(x.min, x.max)
  case x: Max => "max" := x.value
  case x: NotEmpty => ("required" := true, "minlength" := 1)
  case x: Future => "date" := true
  case x: Range => "range" := Seq(x.min, x.max)
  case x: Email => "email" := true
  case x: CreditCardNumber => "creditcard" := true
}
```

5.4 Model, Data Access Objects and Services

The model forms the basis of an application since it describes the real-world problem domain. Services might be considered the most important components as they contain the business rules. Data access objects abstract out the actual implementation of data acquisition and persistence.

Together these form the *engine* of the application, which is not web-related in any way. In a web application these are often programmatically simple and might therefore not gain much advantage from an advanced programming language.

5.4.1 Model

Program 5.14 shows a simple model object implemented in Java describing an employee. The example utilizes *Java Persistence API (JPA)* [55] annotations for persistence and Hibernate Validator annotations for strengthened typing.

DEPARTMENT and HOMEADDRESS are references to other model objects. From the typing including the validator annotations it can be deduced that DEPARTMENT and NAME are required fields that must never be NULL while EMAIL and HOMEADDRESS are optional.

The class is rather clean except for the required getters and setters which fortunately can be generated. Optional fields can be a problem when this model object is used. As Java has no built-in mechanism to describe fields that might be null the users of this class are expected to know and check for possible nulls before dereferencing objects:

```
String emailUserName = null;
if (employee.getEmail() != null) {
        emailUserName = employee.getEmail().split("@")[0];
}
```

```
package net.lahteenmaki.thesis.model;
/* imports */

@Entity
public class Employee {
  @Id
  @GeneratedValue
  private int id;

  @ManyToOne
  @NotNull
  private Department department;

  @NotEmpty
  private String name;

  private String email;

  @OneToOne
  @Valid
  private Address homeAddress;

  /* basic bean getters and setters for all properties */
```

Program 5.14: *Model object for an employee implemented in Java*

This is quite awkward and it has been proposed to Java7 [19] that the same could be achieved with the following syntax:

```
String emailUserName = employee.getEmail()?.split("@")[0];
```

Program 5.15 gives a similar model object definition in Scala. Unfortunately there are no annotations included as the current version of the Scala compiler adds annotations declared on properties also to the generated accessor methods which breaks both JPA and Hibernate Validator functionality. This behavior may change in a future version, but in the meantime the persistence mapping has to be given in an external XML file and Hibernate Validator cannot be used.

The Scala version is shorter (no getter or setters) and it even specifies at language level that EMAIL and HOMEADDRESS are optional properties. No null checking is necessary since nullable values are replaced with Scala OPTIONS.

```
package net.lahteenmaki.thesis.model
/* imports */

class Employee {
  var id: Int = _

  var department: Department = _

  var name: String = _

  var email: Option[String] = None

  var homeAddress: Option[Address] = None
```

Program 5.15: *Model object for an employee implemented in Scala*

For this to work with Hibernate a custom accessor for the Option properties has to be declared in the Hibernate mappings:

```
...
[
<!ENTITY option "net.lahteenmaki.scalax.data.hibernate.VarPropertyAccessor">
]>
...
<many-to-one name="homeAddress" access="&option;" fetch="join">
    <column name="homeAddress_id" />
</many-to-one>
...
```

VarPropertyAccessor is similar to Hibernate's built-in direct field accessor, but it unwraps the values from the Option type. Similar accessor can be made for property based access when needed.

To make Program 5.15 compatible to Java, getters and setters have to be added. Java-compatible version is shown in Program 5.16. Scala @BeanProperty annotation is used to tell the compiler to generate regular getters and setters in addition to Scala property accessor methods.

```
package net.lahteenmaki.thesis.model
/* imports */

class Employee {
  import OptionConversions._

  @BeanProperty
  var id: Int = _

  @BeanProperty
  var department: Department = _

  @BeanProperty
  var name: String = _

  var email: Option[String] = None
  @NotEmpty
  def getEmail: String = email
  def setEmail(x: String) = email = x

  var homeAddress: Option[Address] = None
  @Valid
  def getHomeAddress: Address = homeAddress
  def setHomeAddress(x: Address) = homeAddress = x
```

Program 5.16: *Model object for an employee implemented in Scala, Java compatible*

Option types have to be handled differently, however. The getters and setters are written by hand to wrap and unwrap the actual value to and from the Option type. Cleaner alternative would be to create a Scala compiler plugin and an annotation similar to @BeanProperty which generate getters and setters for the Option's content type. The current approach, however, has the benefit that Hibernate Validator annotations can be used on the handwritten getters.

OptionConversions object contains implicit conversion methods to and from the Option type to make this class a bit cleaner.

5.4.2 Data access objects

Data access objects mainly interact with Hibernate. The interfaces would be best to use Scala Option type instead of nullable return values, although the examples here do not use it to maintain Java compatibility.

```
1  public Municipality[] getAllMunicipalities() {
2    List<?> results = manager().createQuery("select from municipality")
3                               .getResultList();
4    return results.toArray(new Municipality[results.size()]);
5  }
6
7  public Country[] getAllCountries() {
8    Session session = (Session)manager().getDelegate();
9    List<?> results = session.createCriteria(Country.class).list();
10   return results.toArray(new Country[results.size()]);
11 }
12
13 public Municipality findMunicipality(String name, Country country) {
14   return (Municipality)manager().createQuery("select from municipality"
15          + " where name=:name and country=:country")
16     .setParameter("name", name).setParameter("country", country)
17     .getSingleResult();
18 }
19
20 public Country findCountry(String name) {
21   Session session = (Session)manager().getDelegate();
22   return (Country)session.createCriteria(Country.class)
23     .add(eq("name", name)).uniqueResult();
24 }
```

Program 5.17: *Java implementation of a data access object providing location data*

Program 5.17 shows a basic Java implementation for a DAO responsible for location data. Methods getAllMunicipalities and findMunicipality use standard JPA queries whereas methods getAllCountries and findCountry use Hibernate Criteria API.

Program 5.18 shows the same parts implemented in Scala. This is similar to the Java implementation except that now Hibernate Criteria queries are replaced with SQL-like queries introduced in Section 4.1.2. The Scala implementation has somewhat less unnecessary code although there is still some due to maintained Java compatibility. For example, the method findCountry returning an Option would look like:

```
1 def findCountry(name: String) = (
2   SELECT (*) FROM (classOf[Country]) WHERE ("name" EQ name)
3 ) result
```

```
def getAllMunicipalities = {
  manager.createQuery("select from municipality").getResultList
         .asInstanceOf[java.util.List[Municipality]].toSeq.toArray
}

def getAllCountries = (SELECT (*) FROM (classOf[Country])) results

def findMunicipality(name: String, country: Country) = {
  manager.createQuery("select from municipality where"
                      + " name=:name and country=:country")
    .setParameter("name", name)
    .setParameter("country", country)
    .getSingleResult.asInstanceOf[Municipality]
}

def findCountry(name: String) = {
  (SELECT (*) FROM (classOf[Country]) WHERE ("name" EQ name))
    .result.getOrElse(null)
}
```

Program 5.18: *Scala implementation of a data access object providing location data*

5.4.3 Services

Service implementations in the example project do not differ much. The Scala implementation is a few lines shorter, but essentially the content is the same. In an all-Scala project the services could use Scala's Option type for null safety.

The services contain possibly the most important code, the business rules which have to be correct. It is essential that there is as little additional code as possible around the actual logic, and Scala can help here with its syntax.

5.5 View

An average user does not see or understand how an application works behind the view and one might argue that the view represents almost the whole application to him. A bug in the view seems therefore like a bug in the whole application even though it would actually be related only to how the data is being displayed. Hence the view is the part that forms the feeling of quality over the whole application. A simple table displayed slightly distorted might create a feeling that the whole application is not working correctly.

Building XML documents with string concatenation is error-prone and is likely to cause trouble with character escaping or producing an otherwise well-formed document. There are also security issues like JavaScript injection. This applies not only to XML documents but to basically any structured data.

In web applications the view is most often an HTML document, nowadays often in its XML variant, XHTML. The standard way to build (X)HTML documents in a Java EE environment is to use JSP technology where HTML markup is supple-

mented with ready-made or self-made tags and plain Java code. What actually happens is that an XML document is created by concatenating strings.

Therefore, building views with JSP technology is inherently bad and should not be used if there are any requirements for the correctness of the view. As long as the view is a structured document (like an XML or XHTML document) it should be created by serializing some kind of a structural data object with ready-made tools which ensure well-formedness as well as other correctness. JSP and other similar technologies may be safely used only if such a tool is utilized that evaluates all programmer output and checks for any error conditions and warns the programmer of possible flaws.

Scala with its XML syntax provides a safer way to create documents. As described in Section 3.2, XML markup can be written next to regular Scala code escaping back to Scala where necessary. Methods can take and return XML fragments as any other objects. The compiler generates a structured tree behind the scenes which can be serialized to a string. This ensures that escaping and encoding is done correctly and the resulting XML document is always well-formed.

Program 5.19 shows a traditional JSP that prints all the departments in a table. The file is mostly HTML with some JSP-tags and EL-expressions. The tag libraries used are introduced in the beginning after the page header.

Printable messages are read from resource files with *JSTL (JavaServer Pages Standard Tag Library)* FMT:MESSAGE tag which can handle parameters. Looping is done with C:FOREACH tag from the core library, which provides looping status in a separate variable. URLs are printed with C:URL tag and text escaping is done with C:OUT tag from the core library.

Types and even the existence of parameters, variables and attributes in any context cannot be known by the development environment causing the possible errors to only be seen on runtime. If the file is kept close enough to well-formed XML the development environment can warn about missing or incorrect HTML but otherwise the developer is on her own. Since the (X)HTML is effectively generated with string concatenation, escaping has to be done manually by the developer, which leaves open the possibility for JavaScript injection and similar flaws.

Program 5.20 shows the same functionality implemented in Scala. The file is a regular, compilable, statically typed Scala file. The package and import declarations as well as the class declaration are hidden in HTML comments to make the file look as much HTML as possible if viewed in a browser or a HTML editor.

The class is mixed with different helper modules which roughly correspond to JSP tag libraries and help with common tasks such as URL generation and localization. The class is marked abstract because its DEPARTMENTS value is abstract. This ensures that the value is actually provided when the view is instantiated. The type

```
<%@ page session="false" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>
<%@ taglib prefix="requestMapping" uri="/WEB-INF/requestMapping.tld"%>

<%@ page import="net.lahteenmaki.thesis.configuration.JSPMappings" %>

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><fmt:message key="department.list" /></title>
  </head>
  <body>
    <h2>
      <fmt:message key="department.all">
        <fmt:param value="${fn:length(departments)}" />
      </fmt:message>
    </h2>
    <table>
      <tr>
        <th><fmt:message key="department.id" /></th>
        <th><fmt:message key="department.name" /></th>
      </tr>
      <c:forEach var="dep" items="${departments}" varStatus="status">
        <tr class="${status.index % 2 == 0 ? 'even' : 'odd'}">
          <td>
            <c:set var="url">
              <requestMapping:url mapping="<%=JSPMappings.DEPARTMENTS_SHOW()%>"
                                  params="${dep.id}" />
            </c:set>
            <a href="${url}">${dep.id}</a>
          </td>
          <td>
            <c:out value="${dep.name}" />
          </td>
        </tr>
      </c:forEach>
    </table>
  </body>
</html>
```

Program 5.19: *JSP page to list departments*

of the value is also statically visible.

The resulting (X)HTML is the return value of an otherwise abstract method PAGE. The type of the value is SCALA.XML.NODESEQ so the result can be processed directly as XML before sending it to the browser. The following expression:

```
{?.department.all(departments.size)}
```

is part of the localization module. Scala's flexible syntax allows some of the application's messages to be put inside regular Scala classes in a clean way. This provides a statically typed alternative to resource files with arbitrary formatting possibilities. The line is equal to the following JSP fragment except for being statically typed:

```
<fmt:message var="msg" key="department.all">
  <fmt:param value="${fn:length(departments)}" />
</fmt:message>
<c:out value="${msg}" />
```

Most of the content is just regular HTML. The interesting part is the iteration

```
1  //<!--
2  package net.lahteenmaki.thesis.view.department
3
4  import net.lahteenmaki.thesis.configuration.Mappings._
5  import net.lahteenmaki.thesis.model.Department
6  import net.lahteenmaki.thesis.resource.Messages
7  import net.lahteenmaki.scalax.web.spring.view.ViewBase
8  import net.lahteenmaki.scalax.web.view.module.{Url, Flow, Localization}
9  import scala.xml.NodeSeq
10
11 abstract class ListDepartments extends ViewBase with Url
12                                              with Localization[Messages] {
13  val departments: Seq[Department]
14
15  def page = {
16 //-->
17
18   <html xmlns="http://www.w3.org/1999/xhtml">
19     <head>
20       <title>{?.department.list}</title>
21     </head>
22     <body>
23       <h2>{?.department.all(departments.size)}</h2>
24       <table>
25         <tr>
26             <th>{?.id}</th>
27           <th>{?.department.name}</th>
28         </tr>
29         {
30           departments.toArray[Department].zipWithIndex.map{ case (dep, index) =>
31             <tr class={if (index % 2 == 1) "even" else "odd"}>
32               <td>
33                 <a href={url(DEPARTMENTS_SHOW, dep.getId)}>{dep.getId}</a>
34               </td>
35               <td>
36                 {dep.getName}
37               </td>
38             </tr>
39           }
40         }
41       </table>
42     </body>
43   </html>
44  }
45 }
```

Program 5.20: *Scala page to list departments*

of departments, which is done by mapping the collection of departments to a set of table rows. Index of iteration is needed to make odd rows different from even ones, so the collection is first complemented with the indexes of each element.

The URL to show a department is constructed from the definitions in request mappings, providing the parameters to replace the groups in the regular expression. See Section 5.2 for a description of the mappings. Here the Scala implementation is quite similar to the JSP one, except again for static type safety.

Program 5.21 shows an example of a page to add another department to the system. There is a FORM module mixed in to provide common functionality like client-side validation. Functionality to display form errors is extracted to a utility trait as a method called ERRORMESSAGEBLOCK. The utility trait is mixed in with

```
abstract class AddDepartment extends ViewBase with Url
        with Localization[Any] with Form with FormUtils {

  val department: AddDepartmentObj
  def page = {
//-->

<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>{"department.add"?}</title></head>
  <body>
    <h2>{"department.add"?(new java.util.Date())}</h2>
    <form accept-charset="UTF-8" method="post" action="?">
      {errorMessageBlock(department)}
      <fieldset>
        <legend>{"department.form_basic"?}</legend>
        {inputField(department.name, "department.name"?)}
        {inputField(department.captchaAnswer, "department.captchaAnswer"?)}
        <input type="submit" value={"department.form_button_add"?} />
      </fieldset>
    </form>
  </body>
</html>
//<!--
}
def inputField(field: Field, label: String) = {
  implicit val value = field
  <label for={id}>{label}</label>
  <input type="text" id={id} name={name}
        class={errorClass("error") + validation} value={value} />
  &+ (if (errors.isEmpty)
    xml.NodeSeq.Empty
  else
    <label for={id} class="error">{errors}</label>
  )
}
```

Program 5.21: *Scala page to add a department*

the view class getting access to the same context as the main page.

The creation of regular text input fields is also extracted to a method, which this time is included in the same file. This demonstrates how common functionality is easily extracted from programs without the complexity of tag files. The approach is straightforward and keeps the extracted piece of code close to its actual context.

Binding the form values to the fields is here done with the help of the FORM module which contains methods like ID() and NAME() which take a FIELD object as a parameter. The parameter is defined to be implicit so the field object does not have to be passed in manually. This lessens the verbosity and is roughly equal to wrapping the form definition to <FORM:FORM>... </FORM:FORM> in a Spring form.

This example also demonstrates the use of another syntax for printing out messages:

```
{"department.add"?(new java.util.Date())}
```

This is another example of how Scala can be used to hide additional verbosity from program code. The syntax presented is another way for reading messages from

the application resources, and it is much less verbose than FMT:MESSAGE tag from JSTL:

```
<fmt:message var="msg" key="department.add">
        <fmt:param value="<%= new java.util.Date() %>" />
</fmt:message>
<c:out value="${msg}" />
```

5.6 Performance tests

Performance is essential when building enterprise-scale applications. Scala compiles to JVM bytecode and aims for scalability among other things, but its performance should not be taken for granted. Detailed performance comparison to Java or other JVM or non-JVM languages is out of the scope of this thesis. Instead, a simple test was made to ensure that Scala is indeed on par with Java.

The performance was evaluated using *Apache JMeter* [56] to generate requests to the web application of the example project. The tests were performed using a *Jetty HTTP server* [57] in development mode within the Maven project, so the resulting values should not be considered to reflect the actual performance in a production environment. The web application builds upon Spring Framework, *HyperSQL DataBase* [58] and Hibernate. Table 5.1 lists versions of the application dependencies used for testing.

The performance tests were run against a full Java/JSP implementation, a full Scala implementation and some mixtures. Each test was run with 2, 50 and 200 virtual users sending requests to a simple JSP view with SiteMesh and a functionally similar Scala view. A single test ended when 5000 samples were collected. This amount was chosen to give average response times enough time to stabilize.

Before the tests all additional software was shut down to let the tests have as much of the CPU power as possible. Each time the application server was started with initial 500 megabyte heap (JVM parameter -Xms512m) to ensure that no additional time is spent increasing the JVM heap size.

Table 5.1: *Versions of the application dependencies used for the performance tests*

JMeter	2.3.4
Jetty HTTP server	6.1.18
Scala	2.7.7
Spring Framework	2.5.6
HyperSQL DataBase	1.8.0.10
Hibernate	3.3.2 GA
Apache JSTL implementation	1.1.2
SiteMesh	2.4.2

After the application server start but before the actual test runs, both test pages were initialized with single requests. This was to exclude for example JSP compilation time from the actual results.

The actual results are tabulated in Appendix A and evaluated in Section 6.4.

6. DISCUSSION OF THE RESULTS

Scala programming language has the potential to improve development in various kinds of programming areas. Web applications form a small but important group of software. This chapter summarizes the results of this thesis. This includes the comparison of Scala to alternative methods in general, as well as the comparison of Scala and Java implementations of the example project. Tool support and performance aspects are also briefly discussed.

6.1 Scala as the programming language

Scala has many syntactic advantages over Java, which help the developer to express herself in a clear and intuitive way. This has the potential of reducing programming errors by making the code easier to write and understand. The number of lines of code can be somewhat reduced due to the increased expressiveness and the reduction in supportive code like getters and setters. Stronger typing, utilization of the functional paradigm and clean syntax should reduce some programming errors like null pointer or class-cast exceptions.

However, as Java was designed to be a programming language for an average programmer [59, Section 1.2.1], Scala requires somewhat more effort and dedication. Expressive languages can easily be used to write unreadable code and thus the programmer is required to have the skill to see the correct balance between expressiveness and verbosity. Reduction in the needed supportive code effectively means that more is happening behind the scenes, which has to be known and understood by the programmer. For the very least, moving from Java to mostly functional paradigm of Scala is a big step for an average Java programmer.

For a capable programmer, Scala provides some facilities to re-think the existing structures and design principles. These include the concepts of mixins and implicits. Some features that in Java require the use of supportive libraries, frameworks or technologies — for example aspect oriented programming (AOP) or dependency injection — can be implemented in straightforward Scala. This provides static type safety and removes the need to learn new tools or syntax.

With implicits, a capable programmer can create custom frameworks or domain specific languages for example to hide implementation details and bring out the business logic. Also existing languages — for example JSON or XML — can be

used within Scala code with static type safety, which has the potential to greatly reduce errors previously only visible in runtime.

Creating application architecture or new language features with mixins and implicits is an advanced topic and requires a lot of dedication from the programmer. However, when for example the implementation for statically typed JSON has been created by a single, skilled programmer, it can easily be utilized by everyone else in everyday programming.

6.2 Web development: Scala versus traditional methods

As seen in Section 5.1, taking advantage of Scala in contemporary web development does not require a complete transition to Scala. Instead, the project can choose to implement some parts in Scala while other parts remain in Java. A concrete example might be in a Spring and Hibernate based application to implement the view layer in Scala. This would bring the advantages of static typing and guaranteed XML-well-formedness while not affecting other components of the project.

6.2.1 Amount of code

Scala is a less verbose and more expressive language than Java and many other languages. Therefore it is natural that it requires fewer lines of code to create the same functionality. Higher level functions provide a more convenient syntax for anonymous inner classes and structures familiar from functional languages remove the excess use of loop structures that is easily seen in most procedural languages.

Most reduction in code, however, can only be seen when Scala is applied in a more advanced level. By re-thinking the whole architecture with the help of higher level functions, traits, implicits and so on, one could achieve a considerably smaller codebase. A simple web project cannot demonstrate these ideas, at least when the traditional Spring and Hibernate frameworks are used as the main architecture.

6.2.2 Code quality

Scala can be used to implement somewhat more reliable programs compared to Java. Immutability and stateless programming remove many common places where a Java program might have errors. The functional programming style removes many loops, which tend to be error prone parts in Java programs.

Also being less verbose and more expressive than Java, Scala has less boiler-plate and otherwise irrelevant code, which makes it easier for the programmer to focus on the relevant parts. The actual business logic is also more visible in the program code.

For many of the cases where the Java community has used dynamic languages, like JSP for the view, Scala provides a statically typed alternative. This brings the advantage of catching some errors in compile-time and lowers the bar for refactoring. Refactoring for example JSP pages is troublesome and error-prone whereas in Scala it is a natural part of development.

Inability to use Hibernate Validator or any similar declarative way to strengthen typing is a big drawback. Some of the guarantees about model object correctness are lost both in application level and in database level. Also the automated server-side and client-side validations based on those declarations are lost. The annotation handling of the Scala compiler is hopefully changed in a future version.

Code quality is not simply an inverse of the amount of bugs. Architectural choices and other design decisions are an important factor whose benefits arise when there comes a need to change or extend something, or to introduce new developers to the project.

6.2.3 Testing

Scala's syntactic flexibility allows creation of more intuitive testing frameworks than currently exist for Java. Examples include ScalaTest and ScalaCheck which are briefly covered in Section 4.2. These have the capability to improve the quality of the tests and to increase test coverage. They are quite mature but still require some work in interoperability with other tools like Eclipse.

Generating unit tests is not possible for all test cases but sometimes it is a useful approach. Best results are obtained by integrating the different methods and using the most suitable for each test case.

As seen in Section 4.2.2, Scala can make it feasible to test something that has not traditionally been tested in Java. Although this example of testing the resulting XHTML is somewhat unique, other similar cases might well exist.

6.2.4 Production rate

Production rate is hard to measure, since it is not just the amount of lines of code produced in a given unit of time. Design choices and abstractions play an important role in the long run.

The fact that with Scala the developer can most of the time just say what she wants, while in Java she is often forced to write loops and temporary variables, should already make development with Scala faster. This requires that the developers acquire enough knowledge of Scala and that the IDE support described in Section 3.6 improves notably.

Static nature of Scala brings advantages in the view layer over dynamic JSP with EL. Advanced tool support can detect many errors in JSP pages and EL syntax but it does not compare to full static type safety. This makes basic view development somewhat faster in Scala.

6.3 Example project: Differences to corresponding Java implementation

The most differences in a traditional web application seem to be in the view layer. Model objects, data access objects and service implementations might benefit a little from Scala for example in null safety and more user-friendly syntax. These are of course benefits wherever Scala is used.

6.3.1 Amount of code

The measure used here is *Lines of code (LOC)*. Whitespace and comments are excluded and the numbers are trying to reflect actual logical lines. The measure can easily be affected by code formatting, but here the projects under comparison were all written in as natural style as possible. The numbers should be taken as coarse estimates.

Comparing the model projects does not bring out the advantages or disadvantages of the programming language in use. The classes mainly declaratively state the logical model and possibly persistence mapping. There is only a little functionality included. Lines of code are 112 in Java without persistence or validation annotations, against 78 in Scala.

Many parts are annotation driven, which only differs slightly in syntax. Unfortunately, the current Scala version has some issues with annotations and Scala properties which prevents using annotations to declare persistence mapping and validation.

The DAO layer acts mainly as a thin interface between the business logic and the persistence engine, so the implementation is pretty straightforward regardless of the programming language used. Real projects having real functionality in this layer might reveal more differences. 81 lines of code in Java is not much different from 41 lines in Scala.

Same kind of similarity continues on the service layer. Simple business logic is simple to implement in any programming language and a different syntax does not make it much shorter. Again, a more complex project would show more differences. 103 lines of code in Java is not much more than 83 in Scala.

The biggest difference can be seen in the view layer. The JSP-file and the corresponding Scala file do not differ much in size or in structure. Both are basically

the resulting HTML with some additional content in-between. In JSP-pages this additional content consists of tags, *scriptlets* and EL-expressions whereas in Scala it is normal language constructs.

The Scala files might grow somewhat larger due to additional methods that contain some of the logic. Adding up all the files together, the total lines of code in Scala is much less than in JSP and Java. This is because JSP tags, definition files and backing Java beans and classes take up more lines than the actual page content and program logic. If the comparison was made with an older version of JSP and without EL-expressions, the difference would be even bigger.

In general, larger and more realistic projects might reveal more differences. On the other hand, lines of code might not even be the best measure between different programming languages since it is affected by differences in syntax.

6.3.2 Production rate

Scala is a full-featured programming language whereas JSP and EL are quite limited in functionality. Scala can be extended with regular classes and methods satisfying all the requirements. XML support and clean syntax provide a simple tool that may be easy enough to learn for a user interface designer. On the other hand, all the features that Scala provides are directly in use when needed. Creating a simple method for a task is a lot faster than creating a tag or a JSP function for it.

On the JSP side, all the programming logic has to be wrapped in tags and JSP functions unless the JSP files are allowed to contain inline Java code. Tags and functions have no support for some common programming language concepts like overloading or polymorphism, which in the end can cause code duplication in the form of multiple tags or functions performing the same operations. This also restricts the usage of certain patterns for the view layer, possibly resulting in low-quality code without useful abstractions.

6.3.3 Code quality

While this example project shows some improvements in development time quality, it cannot reflect the quality of a real project. It was clearly visible in the view layer that there was often nothing to fix after the Scala compiler was satisfied with the project. When programming the JSP pages, however, a lot of iteration was needed due to the dynamic nature. When using JSPs, it has to be made sure that all execution paths are executed to find all the errors.

6.4 Performance

According to *Programmin in Scala*, the resulting programs should perform close to equivalent Java programs, although references to actual benchmarks are not given:

> Scala programs compile to JVM bytecodes. Their run-time performance is usually on par with Java programs.
>
> *Programming in Scala [23, Section 1.3]*

Scala uses Java's primitive types, arrays and native arithmetic when possible [23, Section 1.3] to gain better performance. Resulting bytecode should hence be close to that compiled from Java.

Excessive use of traits might show in performance due to how some Java runtimes implement certain language features [23, Section 12.7]. Scala makes it easy to extract functionality from classes to traits and a developer has to be somewhat careful when to use traits and when not, although in regular code the difference should not be noticeable. On the other hand, traits have many benefits over not using them.

The performance of the example web application was roughly analyzed by running some JMeter tests against different configurations, as described in Section 5.6. The purpose of the tests was not to measure accurate performance but to compare results between a traditional Java/JSP implementation and a Scala implementation.

The tests were run in a regular laptop computer. All the requests received a successful response. Minimum, maximum and average response times for each test as well as other data are listed in Appendix A. As can be seen from the results, using Scala should not cause a decrease in performance. This applies to a simple application and should not be taken for granted for larger amounts of data or algorithmic processing.

Based on the results one could make a conclusion that implementing any layer — that is, model, DAO or service — in Java instead of Scala would actually decrease performance. The average response times are, however, close to each other and due to the coarse nature of the tests the differences can be explained by coincidence.

More interesting comparison can be made between the web layers containing the mappings, controllers and views. The average and median values show the Scala implementation to perform consistently better. Again, the tests were not appropriate for exact conclusions, but the trend is noticeable.

Initialization rows show the requests that were performed for each test set before the actual tests. Comparing the values for JSP and Scala implementation shows how the Scala implementation is clearly faster to respond to the first request. This was to be expected since the Scala classes were already compiled to bytecode before the first execution.

6.5 Tool support

The biggest disadvantage of Scala in this example project was the tool support. As described in section 3.6 the IDE support is at the time of writing almost unacceptable and the difference to the tools available for Java development is notable. Losing all refactoring capability as well as some IDE help in automatic imports and context-sensitive help will definitely lower productivity.

On the other hand, the biggest advantages of Scala were seen in the view layer, where the tool support is weak also for other technologies. IDEs provide some contextual help and validation for JSPs and EL, but it is more or less on par with what the Scala compiler provides.

The compilers and the Maven plugin seem stable enough for production use, and the continuous compilation mode that the Maven plugin provides makes a usable development environment by itself.

7. CONCLUSION

There exists multiple programming languages and tools for constructing both small and enterprise scale web applications. Recent candidates strive for rapid development with less complexity, but tend to be dynamic in nature. Dynamic languages seem to answer the needs of many developers by providing a nicer syntax and less restrictions compared to static alternatives.

Scala has a flexible syntax and few restrictions and has many of the strengths of dynamic languages while still providing static type safety. Compiling to JVM bytecode and having full Java interoperability provides a lot of libraries ready at use. Some language features of Scala should improve the code quality, and testing frameworks described in Section 4.2 make application testing more intuitive compared to what similar tools with the syntax of Java can offer.

XML handling is both easy and safe with Scala. Web applications can use this to build statically typed XHTML or other XML formats. The ability to create custom domain specific languages, as described in Section 4.1, makes printing other formats like JSON both easy and type safe. It should be noted that the examples investigated in this thesis do not cover all the cases where easy and type safe DSLs could be taken advantage of.

Performance tests described in Section 5.6 as well as references like Twitter [60] from the Scala web site [26] show that Scala can performance-wise be used to build enterprise scale web applications. The example project described in Chapter 5 shows that Scala can be utilized in all the components of a web application, or mixed with Java in some combination.

When moving from Java to Scala development, a programmer has to learn a new language with new syntax in addition to all the new features available for use. The functional paradigm itself may already be too big a step for an average programmer used to procedural and stateful programming. It should be realized that both time and practice is required before the developers can effectively use Scala. Advanced features like those listed in Section 3.3 might be best kept as a responsibility of more experienced programmers.

Reduced amount of supportive code as well as lines of code in total should make the code more understandable, less error prone and more maintainable. More important factor, however, is often production rate. Scala is somewhat faster to use

for an experienced developer, comparing to Java, due to the use of the functional paradigm and concise syntax. Proper tools are still needed to gain proper context sensitive help and refactoring capabilities. Compared to Java, Scala is lacking a lot in the quality and state of available tools and is therefore not applicable as a replacement for Java when production rate is a key factor.

This thesis only demonstrates some possibilities in a small web application. Further work with a larger and more complex application is needed to properly evaluate Scala's advantages over Java or alternatives. Also performance needs more thorough evaluation when larger amounts of data are handled or some algorithmic handling is performed.

While the focus of this thesis was in comparing Scala to Java, it might be more beneficial to compare it against some more recent alternatives like those described in Section 2.2. Such a comparison might be difficult to perform unless simply comparing the quality and other aspects of already finished projects implemented in the technologies in question.

REFERENCES

[1] CGI. http://www.w3.org/CGI/. Visited 2010-01-11.

[2] PHP. http://www.php.net/. Visited 2010-01-11.

[3] Java Enterprise Edition. http://java.sun.com/javaee/. Visited 2010-01-11.

[4] Web Objects. http://www.apple.com/ca/webobjects/. Visited 2010-01-11.

[5] Ruby on Rails. http://rubyonrails.org/. Visited 2010-01-11.

[6] Django. http://www.djangoproject.com/. Visited 2010-01-11.

[7] Microsoft Active Server Pages. http://www.asp.net/. Visited 2010-01-11.

[8] Java Data Objects. http://java.sun.com/jdo/. Visited 2010-01-11.

[9] Hibernate. http://www.hibernate.org. Visited 2010-01-11.

[10] Enterprise JavaBeans. http://java.sun.com/products/ejb/. Visited 2010-01-11.

[11] FreeMarker. http://freemarker.org/. Visited 2010-01-11.

[12] Apache Velocity. http://velocity.apache.org/. Visited 2010-01-11.

[13] XSL Transformations Specification, Version 1.0. http://www.w3.org/TR/xslt. Visited 2010-01-11.

[14] JSR-000152 JavaServer Pages 2.0 Specification. http://jcp.org/aboutJava/communityprocess/final/jsr152/. Visited 2010-01-11.

[15] Spring Framework. http://www.springsource.org/. Visited 2010-01-11.

[16] Python Programming Language. http://www.python.org/. Visited 2010-01-11.

[17] Ruby. http://www.ruby-lang.org/. Visited 2010-01-11.

[18] H. Thimbleby. A critique of Java. *Software-Practice and Experience*, 29(5):457–478, 1999.

[19] A. Miller. Java SE 7 Preview. http://puredanger.com/techfiles/090204/Java7Preview.pdf, 02 2009.

[20] JRuby. http://www.jruby.org/. Visited 2010-01-11.

[21] Groovy. http://groovy.codehaus.org/. Visited 2010-01-11.

[22] Grails. http://www.grails.org/. Visited 2010-01-11.

[23] M. Odersky, L. Spoon, and B.V. Venners. *Programming in Scala*. Artima Inc, 2008.

[24] M. Odersky. The Scala language Website - Scala's Prehistory. http://www.scala-lang.org/node/239, 08 2008. Visited 2008-11-02.

[25] B. Emir. Scala.xml Draft book. http://burak.emir.googlepages.com/scalaxbook.docbk.html. Visited 2009-12-01.

[26] Scala language web site. http://www.scala-lang.org. Visited 2010-01-11.

[27] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language, 2nd edition. *LAMP-EPFL*, 2006.

[28] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. Scala language specification, Version 2.7. *Programming Methods Laboratory, EPFL, Switzerland*, 2008.

[29] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 57. ACM, 2005.

[30] Hibernate Validator. http://validator.hibernate.org/. Visited 2010-01-11.

[31] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[32] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[33] Erlang programming language. http://www.erlang.org/. Visited 2010-01-11.

[34] Apache Ant. http://ant.apache.org. Visited 2010-01-11.

[35] Scala Ant tasks. http://www.scala-lang.org/node/98. Visited 2009-07-28.

[36] Maven2 plugin for Scala. http://scala-tools.org/mvnsites/maven-scala-plugin/. Visited 2010-01-11.

[37] Maven. http://maven.apache.org/. Visited 2010-01-11.

[38] Eclipse. http://www.eclipse.org/. Visited 2010-01-11.

[39] NetBeans. http://www.netbeans.org/. Visited 2010-01-11.

[40] IDEA. http://www.jetbrains.com/idea/. Visited 2010-01-11.

[41] JSON. http://www.json.org/. Visited 2010-01-11.

[42] N. Rutar, C.B. Almazan, and J.S. Foster. A comparison of bug finding tools for Java. In *Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE 04)*, pages 245–256, 2004.

[43] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[44] ScalaTest. http://www.scalatest.org/. Visited 2010-01-11.

[45] JUnit. http://www.junit.org/. Visited 2010-01-11.

[46] TestNG. http://testng.org/. Visited 2010-01-11.

[47] ScalaCheck. http://code.google.com/p/scalacheck/. Visited 2010-01-11.

[48] Haskell. http://www.haskell.org/. Visited 2010-01-11.

[49] QuickCheck. http://www.cs.chalmers.se/~rjmh/QuickCheck/. Visited 2010-01-11.

[50] SiteMesh. http://www.opensymphony.com/sitemesh/. Visited 2010-01-11.

[51] JavaBeans. http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp. Visited 2010-01-11.

[52] JQuery. http://jquery.com/. Visited 2010-01-11.

[53] JQuery Validation plugin. http://plugins.jquery.com/project/validate. Visited 2010-01-11.

[54] JQuery Metadata plugin. http://plugins.jquery.com/project/metadata. Visited 2010-01-11.

[55] Java Persistence API. http://java.sun.com/javaee/technologies/persistence.jsp. Visited 2010-01-11.

[56] Apache jmeter. http://jakarta.apache.org/jmeter/. Visited 2010-01-11.

[57] Jetty HTTP Server. http://www.mortbay.org/jetty/. Visited 2010-01-11.

[58] HyperSQL DataBase. http://www.hsqldb.org/. Visited 2010-01-11.

[59] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. *Sun Microsystems: Mountain View, CA*, 1995.

[60] Twitter. http://twitter.com/. Visited 2010-01-11.

APPENDIX A: RESULTS OF THE PERFORMANCE TESTS

Table A.1: *Results of the performance tests. Response times in milliseconds*

test description	samples	response time average	response time median	response time 90% line	minimum response time	maximum response time
Initializer: Scala	5	314	289	359	281	359
Initializer: JSP	5	1638	1646	1656	1615	1656
scalaModel/scalaDAO/scalaService:						
2 users: Scala	5000	4	3	7	1	88
2 users: JSP	5000	4	4	6	2	39
50 users: Scala	5000	107	107	120	3	165
50 users: JSP	5000	128	128	142	11	319
200 users: Scala	5000	490	500	530	3	1618
200 users: JSP	5000	538	568	597	15	2830
javaModel/scalaDAO/scalaService:						
2 users: Scala	5000	6	3	15	1	129
2 users: JSP	5000	7	7	10	3	65
50 users: Scala	5000	157	161	179	2	998
50 users: JSP	5000	192	192	205	20	453
200 users: Scala	5000	675	688	726	8	2235
200 users: JSP	5000	787	798	819	73	1910
scalaModel/javaDAO/scalaService:						
2 users: Scala	5000	6	3	16	2	159
2 users: JSP	5000	7	7	10	3	61
50 users: Scala	5000	153	154	168	2	1278
50 users: JSP	5000	183	184	198	8	403
200 users: Scala	5000	542	129	1483	2	13307
200 users: JSP	5000	637	423	1426	5	10329
scalaModel/scalaDAO/javaService:						
2 users: Scala	5000	6	3	17	2	194
2 users: JSP	5000	7	7	10	3	59
50 users: Scala	5000	160	163	183	2	916
50 users: JSP	5000	189	189	208	22	340
200 users: Scala	5000	550	109	1638	3	12045
200 users: JSP	5000	694	632	1056	4	8712
javaModel/javaDAO/javaService:						
2 users: Scala	5000	7	3	18	2	211
2 users: JSP	5000	7	7	10	3	68
50 users: Scala	5000	165	166	180	3	774
50 users: JSP	5000	184	185	203	19	474
200 users: Scala	5000	597	312	1311	3	10557
200 users: JSP	5000	660	504	1176	4	10590