Norwegian University of
Science and Technology

Polyglot Programming

A business perspective

Hans-Christian Fjeldberg

Master of Science in Computer Science

Submission date: June 2008
Supervisor: Hallvard Trætteberg, IDI
Co-supervisor: Carl Christensen, Bekk Consulting AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Polyglot programming is the activity of using several programming languages in a software system. This is commonly used in the industry, for example when embedding SQL in code, but not much research has been done on this topic. The goal of this master thesis is twofold:

1. Get an understanding of methods and techniques for polyglot programming and integration, when these are applicable, and positive and negative effects of polyglot programming.

2. Showing how polyglot programming is used in industrial projects today, and formulate guidelines for the use of polyglot programming based on these experiences.

This master thesis should give the industry in general, and Bekk Consulting AS in particular better measures for deciding whether polyglot programming is the right choice for a particular project. It should also give polyglot programming more credibility by researching scenarios that give business value, not only confirming that something is possible.

Assignment given: 15. January 2008
Supervisor: Hallvard Trætteberg, IDI

Acknowledgements

First of all would I like to thank Christian Schwarz, whose help has been essential in organising this master thesis. The supervisor for this thesis has been Hallvard Trætteberg at NTNU, and co-superviser has been Carl Christensen at BEKK. Thank you both for guiding me through this process. For excellent proof reading, I would like to thank Liv Marie Gustavson.

The case study could not have been done without some willing interview subjects. For this, I would like to thank consultants at BEKK and representatives of the customers.

Additional interviews have also been conducted to get a broader view of polyglot programming, and I would like to thank Neal Ford, Ola Bini and Jay Fields, and finally Aslak Hellesøy who put me in touch with these three.

Trondheim, June 10th, 2008

Hans-Christian Fjeldberg

Abstract

Polyglot programming is the activity of programming in more than one language within the same context. Much like humans use different languages to make expressions more effective, the goal of polyglot programming is to render simpler solutions by combining the best solutions from different programming languages.

Because the term polyglot programming has been coined only recently, no academic research has been done on polyglot programming before. This research has therefore consisted of a literature study to synthesise the opinions of those who have discussed polyglot programming, and an explorative case study to research how polyglot programming is used in a Norwegian consulting firm. The focus of the research has been business benefits and problems.

During the literature study, the expected advantages and disadvantages were found, in addition to examples were polyglot programming can give business value. A part of the literature study has also been a description of the different language paradigms, and a layered description of how they can be organised.

The case study consisted of three smaller case studies, which were analysed based on the theoretical propositions discovered in the literature study, namely the perceived advantages and disadvantages. Using pattern matching, polyglot programming was found to offer increased productivity, and in the only case where it was addressed, easier maintenance. In addition, the use of polyglot programming helped motivate and change the developers' perspective. It was also found that the frameworks written in the new languages were easy to learn because they offered a more natural solution to the problem, discrediting the perceived disadvantage of lack of knowledge. The disadvantage of tool support was, however, confirmed.

Based on the literature study and case study, guidelines for how to utilise polyglot programming are also described, and as a compromise between having free and no language choice, it is suggested that a set of languages should be used. This set should consist of different types of languages, and will give the developers more flexibility, and at the same time give management control over which languages are used.

Contents

Acknowledgements i

Abstract iii

1 Introduction 1

1.1 Research method . . . 2

1.2 Research contributions . . . 2

2 Research context 3

2.1 Research goal . . . 3

2.2 Research questions . . . 4

2.3 Research approach . . . 4

2.3.1 Literature study . . . 4

3 Polyglot programming 5

3.1 Definition . . . 5

3.2 Advantages associated with polyglot programming . . . 6

3.2.1 Productivity . . . 7

3.2.2 Maintainability . . . 8

3.3 Disadvantages associated with polyglot programming . . . 8

3.3.1 Knowledge . . . 9

3.3.2 Maintainability . . . 9

3.3.3 Developer tools support . . . 9

3.4 Summary . . . 10

4 **Technical platforms** **11**
4.1 The business platforms 11
4.1.1 The .NET platform 12
4.1.2 The Java platform 13
4.1.3 Comparison of the polyglot programming support in the .NET and the Java platforms 15
4.2 Language classification 15
4.2.1 The imperative paradigm 16
4.2.2 The object oriented paradigm 16
4.2.3 The functional paradigm 17
4.2.4 The logical paradigm 19
4.2.5 Statically or dynamically typed languages 19
4.2.6 Domain specific languages 20
4.2.7 The language layers 21
4.3 Summary 22

5 **Examples** **23**
5.1 Web development 23
5.2 Testing 24
5.3 Concurrency 26
5.4 Business rules 27
5.5 Summary 28

6 **Research process** **29**
6.1 Case study research 29
6.1.1 Data collection 30
6.1.2 Data analysis 31
6.2 Case study design 32
6.2.1 Data collection 32
6.2.2 Data analysis 33
6.3 Case study implementation 33

6.3.1 Data collection . . . 33

7 **Bekk Consulting AS - A case study** **35**

7.1 Web development . . . 35

7.1.1 Buypass . . . 36

7.1.2 Web based extranet solution . . . 37

7.2 Testing . . . 39

7.2.1 Statens vegvesen . . . 39

7.3 Summary . . . 39

8 **Discussion** **41**

8.1 Polyglot programming context of the cases . . . 41

8.2 Advantages . . . 42

8.2.1 Productivity . . . 42

8.2.2 Maintainability . . . 42

8.2.3 Motivation and change of perspective . . . 43

8.3 Disadvantages . . . 43

8.3.1 Knowledge . . . 43

8.3.2 Maintainability . . . 44

8.3.3 Tool support . . . 45

8.4 Critique of polyglot programming . . . 46

8.5 Research questions . . . 46

8.5.1 RQ1: How is polyglot programming used today? . . . 46

8.5.2 RQ2: Guidelines for using polyglot programming . . . 46

9 **Conclusion** **49**

References **51**

A **Interviews** **57**

A.1 Neal Ford . . . 58

A.2 Ola Bini . . . 61

A.3 Jay Fields 64

B Technologies used in the case studies **67**

B.1 Ruby 67

B.2 Ruby on Rails 68

B.3 RSpec 68

B.4 Watir 69

C Abbreviations **71**

List of Figures

4.1 The relationship between DLR, CLR, CTS and CLS 12

4.2 The relationship between different parts of the Java platform . . 14

4.3 The relationship between the stable, dynamic and domain layer. 22

7.1 Buypass' architecture 37

List of Tables

3.1 Levels of polyglot programming of the example architectures . 7

Chapter 1

Introduction

Throughout much of the last decade, the focus in business has been on one standard language for software development, mainly C# or Java. The rationale for this is that it makes things easier, both for the developers, management when tutoring and hiring employees and the system administrators. The rationale is not wrong, but ignores one important aspect. In enterprise development, the programming environment is partly language, but more so frameworks. Normally, frameworks for XML, SQL, web development and web services are used to make it easier for the developers. So even though all these frameworks are written in the same language, they introduce new abstractions and normally require extensive knowledge on how to configure the framework. It is therefore a good idea to look for languages that can do the same job as the framework, but better. If it is more naturally implemented in the other language, the cost of learning a new language will be equivalent to learning a new framework (Fowler, 2007a).

This approach has been coined polyglot programming. To be a polyglot is to know or use several languages, and is manifested with the introduction of English words in non-English languages, like for example Norwegian. One extreme example of polyglotism is the upper class Lebanese, who speak a mix of French, English and Arabic at the same time. Even though this is not always useful, sometimes expressions in another language are more effective than translating the expression to the native language.

Applied to software engineering, to be a polyglot is to use several programming languages when creating software. This is used extensively in practise, for example SQL and HTML embedded in PHP code, but only limited research has been conducted on this topic. Even though polyglot programming is possible, it is more important to emphasise the real business benefits of using it. Possible examples include testing, concurrency, and web development. Problems to consider include management issues like training and support for new

languages. It is also important to have a good framework for choosing the correct language to solve the problem at hand.

1.1 Research method

The chosen research methods for this research are a literature study to get an extensive overview of the field, and an explorative case study to investigate how polyglot programming is used in a Norwegian consulting firm.

The literature study will focus on polyglot programming in general but with the business perspective in mind. As an example of how polyglot programming can be achieved technically, polyglot programming using a managed runtime will also be described.

The case study will investigate how polyglot programming is used in a Norwegian consulting firm. Lessons learned from this case study will be compared to the findings from the literature study. Data collection will consist of interviews of key persons at the consulting firm. In addition, key persons within the community also be interviewed, both as part of the literature study, and as part of the discussion.

1.2 Research contributions

During the time period when this research was conducted, a lot of discussion has appeared concerning polyglot programming. The main reason for this is the interest Microsoft and Sun has shown towards other languages, improving support for all languages on their runtimes. Microsoft has officially added support for Ruby, JavaScript and F#, in addition to Python, C#, C++ and Visual Basic that are already supported. Sun is adding support for Ruby, Groovy and Scala, in addition to Python, JavaScript and Java support. However, a lot of this discussion is personal opinions based more on feelings than anything else.

Thus, this report constitutes the first ever attempt at approaching polyglot programming from an academic point of view. In addition to synthesising the opinions available, this research will give a definition of polyglot programming. Because it is a broad concept, different degrees of polyglot programming will also be presented.

Based on a case study conducted on a Norwegian consulting firm, this research will also investigate how polyglot programming is used in a business setting. This way, not only personal opinions, but also business perspectives will form the basis for evaluating polyglot programming.

Chapter 2

Research context

This chapter will describe the context within which the research will be conducted. This includes the research goal, research questions and the research approach selected.

2.1 Research goal

Based on the problem description given for this research, the research goal has been simplified to the following: *discover how polyglot programming can be and is used in businesses.*

Previously articles written on polyglot programming have focused on how polyglot programming can be achieved technically (Meyer, 2002; Bini, 2008a; Byrne, 2008; Leghari, 2008). This research will focus on the business perspective, as this has not been researched to the same extent and has the most unanswered questions. In this respect, the business perspective is more than just the technical aspects. It also involves managerial aspects like knowledge, support and decision making.

In addition to focusing on the business perspective, as an example of how polyglot programming can be achieved technically, polyglot programming using a managed runtime will also be described. The rationale for focusing on this type of polyglot programming is that it is on these managed runtimes a lot of language research is being done, and a lot of languages are being made able to run on them. Both Microsoft and Sun are officially adding support for languages including JavaScript, Ruby, Groovy, F# and Scala. The managed runtimes also offer a lot of possibilities and integration that is not possible using some of the other techniques, for example seamless inheritance between classes in different languages, and the integration between the languages are handled automatically by the runtimes, freeing the developers to focus on the

problem.

2.2 Research questions

To discover how polyglot programming can be used, a literature study will be conducted, as well as interviews of key persons. To discovered how polyglot programming is used, a case study will be conducted on a Norwegian consulting firm. The emphasis will be not so much on the technical perspective, but rather on the business perspective and the advantages and disadvantages surrounding the use of polyglot programming. The following research questions will be the basis for the research:

RQ1: *How is polyglot programming used today in businesses?*

RQ2: *Which general guidelines can be formulated based on current experiences?*

2.3 Research approach

Two research approaches are identified based on the research goal, namely literature study and case study. The literature study will consist of a description of polyglot programming, technical aspects of polyglot programming including a language classification, and examples where polyglot programming can be used. The case study design is further described in Chapter 6, but will consist of an explorative case study because little has been written about the business perspective of polyglot programming, and what little literature there is tends to be riddled with personal opinions and biased statements. The case study approach was adopted because it is appropriate when the research has a descriptive and exploratory focus (Yin, 2003).

2.3.1 Literature study

Because the information available on polyglot programming mainly consists of personal opinions, the literature study will synthesise these opinions. Based on the research goal and questions, the focus will be on four areas: (1) the definition of polyglot programming, (2) advantages and disadvantages of polyglot programming, (3) a language classification to form the basis when choosing languages, and (4) examples where polyglot programming can be applied.

The first two areas will consist of personal opinions from web pages and articles. The third area is well established, and the literature will therefore be

of the academic kind. The forth example will take examples from literature when found, but will also consist of my own contributions. When collecting literature, it will be analysed and categorised according to these focus areas.

Chapter 3

Polyglot programming

Polyglot programming is not a new idea, and two realisations have pushed the need for it. The first is the realisation that there is "No Silver Bullet"; that there is no tool that is best at solving all problems (Brooks, 1987). It is therefore important to figure out which tool is most appropriate in each case, which in software development prompts examination of languages, frameworks and development tools. The second realisation is that developers are more expensive than hardware, and programmer productivity is therefore becoming more important than runtime performance.

Polyglot programming is also known as multi language programming (Kullbach, Winter, Dahm, & Ebert, 1998). Language oriented programming (Fowler, 2005b; Dmitriev, 2004; Ward, 1994) also expresses similar ideas, but is a precise development methodology where the creation of domain specific languages are at the centre.

This chapter will describe polyglot programming and propose a simple definition, with additional information about the degree of polyglot programming. Advantages and disadvantages will also be discussed in light of current literature.

3.1 Definition

The term polyglot programming has not been defined in a research context before, and those who describe it tend to use slightly different wordings (Bini, Fields & Ford, personal interviews). The goal is the same, however, and is perhaps best defined by Watts (2008): "programming in more than one language within the same context".

This just pushes the definition onto what the context is. In a personal inter-

view, Neal Ford suggested viewing polyglot programming as the use of different languages on the same managed runtime. Using a managed runtime is definitely polyglot programming, but the definition should not put any restrictions on the architecture. Polyglot programming is mainly about choosing the best programming language for the job.

From a business perspective, the context can be seen in light of the people working on the project. The context will depend on the number of teams, and how the applications they produce are integrated. If one team uses different languages, that will constitute polyglot programming regardless of architecture. If different teams use different languages, but the integration between the applications created is tight, for example using a managed runtime, that will also constitute polyglot programming. However, an application is no longer polyglot when the different teams do not require information about what languages the others are using to be able to use their application, because then the applications could be seen as separate. An example would be a service-to-service application where the interfaces between the applications are the only requirements to use it.

Thus, the definition of polyglot programming can be expanded to *programming in more than one language within the same context, where the context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved.*

In addition to the definition of polyglot programming, a *degree* of polyglotism is suggested, making it possible to differentiate the use of it. The different levels of polyglotism are integration, organisation of code, the processes within languages run, and the data being manipulated. Integration can be either networked or non-networked, the code can be organised either in the same or different files, either the same or different processes can be used, and the languages can manipulate either the same data or the same object. Example architectures where polyglot programming can be utilised are service oriented architecture (SOA), managed runtimes, C integration and polyglots where different languages are presented in the same file. HTML in conjunction with CSS, JavaScript and a server side language (here called HTML++ for abbreviation) is an example of a polyglot. The level of polyglot programming of these architectures are represented in Table 3.1.

3.2 Advantages associated with polyglot programming

Polyglot programming promises advantages including programmer productivity and maintainability. These are only perceived advantages based on per-

Table 3.1: Levels of polyglot programming of the example architectures SOA, managed runtime, HTML++ server and client side and C integration. Integration is either networked or non-networked. The organisation of code is either in the same or different files. Either the same process or different processes are used to run the different languages, and they can access either the same data or same object.

Architecture	Integration	Organisation	Process	Data/object
SOA	Networked	Different files	Different	Same data
Managed runtime	Non-networked	Different files	Same	Same object
HTML++ server	Non-networked	Different files	Different	Same data
HTML++ client	Non-networked	Same file	Same	Same object
C integration	Non-networked	Different files	Different	Same data

sonal opinions however, and will be discussed in light of the case study in Chapter 8.

3.2.1 Productivity

Productivity, its definition and especially how it is measured, are much debated aspects of programming languages. The most used metrics are lines of code (LOC) per unit time (Delorey, Knutson, & Chun, 2007; Maxwell, Van Wassenhove, & Dutta, Oct 1996) and function points per unit time (Delorey et al., 2007; Maxwell & Forselius, 2000). An additional problem is assessing the productivity of different languages regardless of metric. Brooks (1995) states that the productivity is constant regardless of programming language, but according to Delorey et al. (2007), who provide evidence to the contrary, this assumption is based on insufficient data. Problems with measuring productivity include human factors like experience, skill and motivation and environmental factors like IDE support and library support, and it is therefore hard to generalise any findings from case studies.

Polyglot programming, however, promises increased productivity. Because no single language is best at solving all problems, combining the best solutions from different languages and integrating them may render a simpler solution to the given problem. Regarding LOC, a language better suited for a particular problem normally has a shorter solution because of the built-in primitives and idioms to aid in solving it. If it is assumed that developers produce the same amount of LOC regardless of programming language, high-level languages that require less LOC will be more productive. When interpreted languages are used in a polyglot environment, the productivity can be further enhanced because no compile cycle is required to test new features. A compile cycle in large applications normally takes several minutes. Vinoski (2008) gives an example for XML processing. Where a general purpose language normally uses an XML parser, a better solution would be to use languages that support

literal XML, for example ECMAscript for XML (E4X) and Scala, which makes it possible to write XML directly within the language's syntax. This would remove the overhead of parsing the XML file, and also provide a more natural way of interacting with the data using the familiar dot notation. In Chapter 5, more examples where polyglot programming can be productive are presented.

In addition to a shorter solution, the thought process will normally be shorter because the solution comes naturally in the appropriate language. It will also enable the developer to work on the problem and not the required plumbing. One example is Erlang's built-in primitives for message passing between processes, without worrying about race conditions and deadlocks because of Erlang's functional nature. This can be referred to as the Sapir-Whorf hypothesis, stating that thought is constrained by the language (Wexelblat, 1980; Whorf, 1941).

The business value of increased productivity is reduced cost, either because of a shorter development cycle and therefore faster time-to-market, or the possibility to create the same application using fewer developers. In a fast-moving market, the ability to deliver is paramount.

3.2.2 Maintainability

The initial development of applications is only a part of the total cost. Large important systems normally have a long life cycle, often 5-10 years. Therefore, the cost of maintaining an application will in many cases dwarf that of initial development. Because of this, productivity gains from choosing the correct language can be even more important in the maintenance phase than development phase (Vinoski, 2008).

The rationale behind this is that an application written in the languages where the solution comes naturally and can be expressed using fewer lines of code, will have fewer lines of code to maintain, as well as fewer instructions. Several studies cited in Brooks (1995) show that the effort required to develop and maintain applications rises exponentially with the numbers of instructions, and it is therefore important to have as few instructions as possible (Vinoski, 2008). This is further enhanced by the fact that research also shows that the number of faults per LOC increases with the number of LOCs in the application (Lipow, 1982).

3.3 Disadvantages associated with polyglot programming

Polyglot programming also has its disadvantages, the most notable being the knowledge required to use different languages, the maintainability of different languages and the tools support. Similarly to the advantages, these are also based on personal opinions, and will be discussed in Chapter 8.

3.3.1 Knowledge

In order to benefit from polyglot programming, it is important to have knowledge about different programming languages, and the different problem areas which best suits each language. This is a problem, because some developers do not know many languages, and will not be interested in learning a new language (Spiewak, 2008). This is further enhanced in businesses where developers have gotten used to using one language, normally Java or C#, and the infrastructure, tools and certifications are all built around this language (Bini, personal interview). The seminal work of Hunt and Thomas (1999) suggests that developers should learn at least one new language each year. In many situations this is not realistic (Duarte, 2008; Nilsson, 2008), and in many cases it will take more than one year to learn a new language (Braithwaite, 2007; Norvig, 1998).

In addition to this, the management also need increased knowledge of programming languages, especially in the hiring process, but also when selecting languages to use (Ford, personal interview).

3.3.2 Maintainability

Even though maintainability is an advantage of polyglot programming, because of the decreased code base, it does require that the developers who maintain the application know the languages used. For large applications, with a life cycle of 5-10 years, it is likely that different companies will be responsible of the administration. Every time a new language is added, the pool of developers who have enough knowledge to maintain the application will decrease (Spiewak, 2008).

3.3.3 Developer tools support

Developers using Java and .NET are used to having comprehensive IDE support, with support for version control, syntax highlighting, refactoring, debugging and much more. Creating support for a new language usually requires a lot of work, and will normally only be implemented if the language gains enough popularity and traction. If the tools do not bridge the borders between languages, different tools must be used for the different languages, increasing the overhead of using different languages (Bini, personal interview). One example is the refactoring of a dynamic language, which is, at best, very difficult. The reason is that the IDE cannot know the type of a given variable, because this is only revealed at run-time (Eyler, 2006). While developers could use the automatic refactoring tools for C# parts, they would need to manually refactor Python parts.

3.4 Summary

This chapter has defined polyglot programming as *programming in more than one language within the same context, where same context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved.* This is the first attempt at a formal definition of polyglot programming.

Within this definition, the degree of polyglot programming can vary between four levels, namely integration, organisation of code, the processes within languages run, and the data being manipulated.

Perceived advantages of polyglot programming are productivity and maintainability, and the perceived disadvantages are knowledge, maintainability and tool support. These will be further discussed in Chapter 8. Based on this introduction to polyglot programming, Chapter 4 will describe the techniques required to use polyglot programming, and Chapter 5 will give examples where using polyglot programming will benefit businesses.

Chapter 4

Technical platforms

Two aspects are essential in a polyglot environment. The first aspect is the platforms used for integration. As mentioned in Chapter 2, the technical focus will be managed runtimes because of the development being done to support polyglot programming on these. This chapter will therefore describe the two dominant business platforms, the .NET platform and the Java platform. The second aspect is the different languages available on the given platform, and a language classification is therefore given to form a basis for choosing the appropriate language for the task at hand. The language classification will also be the basis for the examples given in Chapter 5, and it is therefore important to know the advantages and disadvantages of different paradigms.

4.1 The business platforms

In the business world, two platforms dominate; the .NET and the Java platforms. The languages used on these platforms are not the business-critical aspect; it is the infrastructure that is built around the platforms that is important for the businesses. Being able to create new infrastructure without having to rewrite old legacy code is essential, and recent development on these two platforms is one of the reasons why polyglot programming has the potential for success. This section will describe the two platforms, focusing on the importance of these platforms in the business world and the techniques used to support polyglot programming, as well as a comparison on how these platforms support polyglot programming.

4.1.1 The .NET platform

When introduced, the .NET platform was a new framework for programming on the Windows platform. After decades of backward compatibility, the frameworks had to be updated, without including outdated solutions and technologies. Before .NET, two languages, Visual Basic and C++, were predominant on the Windows platform. C# was introduced together with the .NET platform, and because of these three languages, the architecture of .NET was developed to support polyglot programming from the beginning (Robinson et al., 2004). .NET has become especially popular in businesses already running on a Windows platform, and in businesses adopting a SOA architecture because of the web service support.

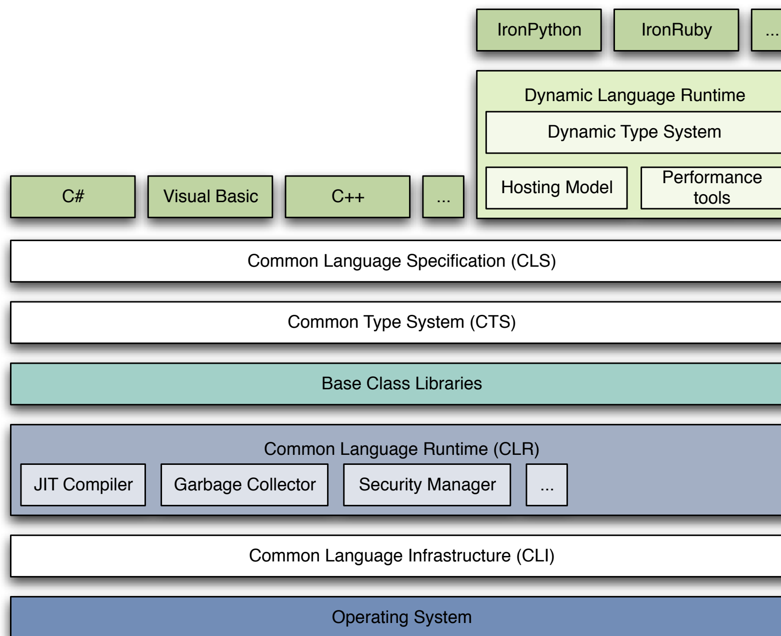

Figure 4.1: The relationship between DLR, CLR, CTS and CLS. Compiled languages integrate through CLS, while dynamic languages must additionally integrate through DLR. Inspired by (Troelsen, 2003).

The .NET platform consists of four parts; the Common Language Runtime (CLR), the Common Type System (CTS), the Common Language Specifications (CLS), and the newly created Dynamic Language Runtime (DLR) (Robinson et al., 2004; Troelsen, 2003; Hugunin, 2007). The relationship between these four parts can be seen in Figure 4.1. The primary role of the CLR is to locate, load and manage .NET types. In addition to this, the CLR offers memory management, language integration and type safety that can be shared among all languages running on it. The CLR is built to run the Common Intermediate Language (CIL), the language which all source code running on the .NET plat-

form is compiled into. The CTS fully describes all possible data types and programming constructs supported by the runtime, how they interact and the .NET metadata format. Because of the wide range of languages that can run on top of the .NET platform, a type can be either *class*, *structure*, *interface*, *enumeration* or *delegate*. For statically typed languages, the final part they need to run on the .NET platform is a compiler that conforms to the rules defined by CLS. CLS enables languages to be hosted and accessed in a uniform manner by all languages that target the .NET platform (Troelsen, 2003).

To support dynamic languages, the DLR has been created based on experiences from the IronPython project (Lam & Hugunin, 2007). The DLR adds to CLR support for dynamic languages through a shared dynamic type system, a standard hosting model, and support to make it easy to generate fast dynamic code and symbol tables. This enables dynamic languages to interact with each other, and because of the interaction with CLR, also lets statically typed and dynamically typed languages interact. Initially, DLR will support four languages, namely Python, JavaScript, Visual Basic and Ruby, but the shared features makes it easy to support new languages (Hugunin, 2007).

4.1.2 The Java platform

The Java platform first became successful on the Internet, but has since been widely adopted in business because of its high portability. The Java Virtual Machine (JVM) is built to run on almost all platforms, from cell phones to personal computers to super computers. In addition, the widespread use of Java means that libraries exist for almost any task. This has made the JVM very popular, and it is becoming more attractive to use other programming languages on the JVM.

Because JVM was created to run the Java programming languages it is designed specifically to support the restrictions and idioms of this language. JVM also supports other languages as long as they are specifically targeting the platform, and obey the same rules (Venners, 1999). Problems occur, however, when existing languages that do not follow these rules are implemented on JVM. Take C++ for example, with its multiple inheritance; this cannot be supported on the JVM without fundamental changes. Code written in dynamic languages, which might change even after it has been run, is another architectural problem that the JVM cannot currently support well. The equivalent to CIL on the Java platform is the bytecode, and is what is executed on the JVM. Because the bytecode targets the JVM and not the underlying operating system, the bytecode allows the code to be cross-platform (Venners, 1999). The Java platform can be seen in Figure 4.2.

To improve support for polyglot programming on the Java platform, JSR 223

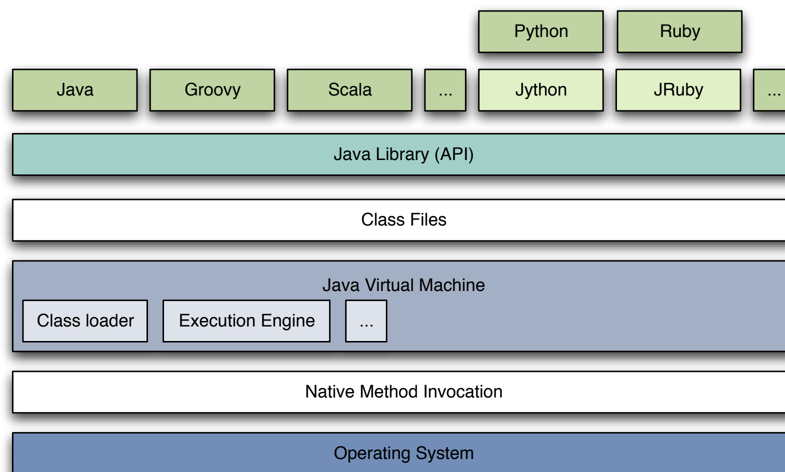

Figure 4.2: The relationship between different parts of the Java platform. Languages targeting the platform can talk directly to the Java libraries, while interpreted languages like Python and Ruby must use an interpreter written in Java to get the same access.

Scripting APIs has been added to JDK 6. These APIs allow arbitrary code to be evaluated by Java, provided there exists a scripting engine for that language. This has been possible before; one example the Bean Scripting Framework (BSF) for writing web pages in other languages than Java using Java Server Pages (JSP), but JSR 223 aims to standardise these efforts (Zukowski, 2006). In addition to this, the JSR 292 Supporting Dynamically Typed Languages expert group is working on adding a new bytecode, *invokedynamic*, which supports efficient and flexible execution of method invocations. The goal is to help improve performance for dynamic languages (Rose et al., 2008).

JSR 292 is part of the Da Vinci Machine project, with a mission of extending the JVM with first-class architectural support for languages other than Java, especially dynamic languages. The emphasis is on general purpose extensions, that will benefit all the languages being implemented on JVM. The goal is to remove some of the problems when running new languages, including limitations on calling sequences and control stack management, finite inheritance, and scaling problems when generating classes, and the new capabilities are planned for inclusion in JDK 7 (Krill, 2008; *The Da Vinci Machine Project*, 2008).

In addition to being able to evaluate code written in other languages from the Java side, interpreters that closely co-operate with the JVM have been written for many languages, for example JRuby for Ruby (*JRuby: Java powered Ruby implementation*, n.d.) and Jython for Python (*The Jython Project*, n.d.). These interpreters make it possible to use Java libraries from inside the respective languages and also to utilise the best features in any given language. These interpreters normally have support for compiling some of the code, while other

parts must be evaluated.

4.1.3 Comparison of the polyglot programming support in the .NET and the Java platforms

The difference between the two platforms is not as large as one might think. Both virtual machines are built to run an assembly-like language, and are in it self not aware of the language the source has been compiled from. Even though the .NET platform was designed to run C++, only managed C++ can coexist with C# and the other languages running on the runtime. Managed C++ removes many of the properties of the language; for example, multiple inheritance is not allowed. Because both runtimes abstract out the operating system with their CIL and bytecode language, they are both inherently cross-platform. The difference is that the JVM has been created to run on as many platforms as possible, while the CLR was only created to run on a Windows platform. To run .NET on other platforms, Novell has created Mono (*Mono project*, n.d.), which will run .NET on Linux and Mac OS X.

To support dynamic languages, however, the two platforms use different philosophies. While the DLR works above the level of the CLR without enhancing it, the Da Vinci Machine project is extending the JVM and the libraries at the same time. The end result will be the same, however, using different approaches (Rose, 2008).

4.2 Language classification

According to Pigott (2006) there exists more than 8500 programming languages. These include both English and non-English languages, ranging from assembly languages to dynamically typed interpreted languages. Very few of these languages are still in use, but inspiration and knowledge from previous languages has been vital in shaping the languages used today. In business, even fewer languages are used; newer programs mainly target the .NET and the Java platforms. The business is changing however, and support for polyglot programming may prove essential for businesses to modernise in the future, without having to rewrite legacy code.

Not all programming languages will be discussed, rather different classes of languages, and the most used languages within these classes. Strengths and weaknesses of the classes will be discussed to point out the situations in which each class will be best suited, as well as support for the .NET and the Java platforms. Following Sebesta (2008), languages will not be classified within

categories, but according to paradigms.

4.2.1 The imperative paradigm

The imperative paradigm is the oldest and most-well developed paradigm, emerging as early as in the 1940s. Even though programming languages from this paradigm are used extensively today in areas where low-level details are very important, for example operating systems and drivers, these are not used extensively in business.

The architecture of the von Neumann-Eckert model is the basis of the imperative paradigm. In this architecture, both program instructions and data values are stored in memory. Important aspects of imperative programming are assignments, variable declarations, expressions, conditional statements, loops and procedural abstraction. In imperative programming, the instructions are normally executed in the order in which they appear, all though conditional statements and loops can change the flow of execution (Tucker & Noonan, 2007).

Because the imperative paradigm offers little abstraction, this enables good performance. However, the lack of abstraction is also one of the limitations, as it becomes hard to organise and manage large applications.

Languages

Because the von Neumann-Eckert model is so fundamental to computer programming, almost all programming languages possess imperative properties, including Java and C#. Of the purely imperative programming languages still in use today, C is the most important one and is the basis for C++, C#, and Java. C is also used in all major operating systems because of its performance and possibility of communicating with hardware. It is not possible to run C on either the .NET or Java platform, but both have facilities for invoking code written in C.

4.2.2 The object oriented paradigm

Because of limitations in imperative programming, efforts such as Simula 67 tried to introduce data abstraction as well as procedural abstraction to the paradigm. The goal was to have a better encapsulation for logically related constants, variables, methods and so on. However, these efforts did not go far enough in its support, lacking automatic initialisation and finalisation, as well

as a simple mechanism for extending data abstractions. More importantly, during the 1980s, developers were realising that imperative programming was not well suited for a significant range of applications, for example graphical user interfaces and large applications in general (Tucker & Noonan, 2007; Sebesta, 2008).

The result of the evolution was object oriented programming, in which object decomposition is a central aspect instead of functional decomposition and data abstraction. These objects are normally related to human aspects, like *Person* and *Tree*, and are represented using a class concept. A class encapsulates constants, variables and functions, but also support inheritance. The specific implementation of the class concept varies from language to language. The class also includes support for visibility and information hiding in that variables and methods can be declared public or private (Tucker & Noonan, 2007; Sebesta, 2008). The advantage of higher abstraction is a natural and simple organisation of related aspects, at the cost of performance. This is especially beneficial in large applications.

Languages

Because of the popularity of object-orientation a lot of languages have been created within this paradigm. The first popular language to include object-orientation was C++, which is still used today for a lot of applications. For enterprise applications, C# and Java are being used with great success, and are the main languages for the .NET and Java platforms respectively. On the Mac OS X platform, Objective-C is the main programming language, and in a lot of banking services, COBOL is still being used. COBOL 2002 added support for object orientation. For complex calculations like weather simulation, FORTRAN is still being used, which added support for object orientation in its 2003 revision.

These are all statically typed languages, but this paradigm also has a range of dynamically typed languages, including Groovy, JavaScript, Python, Ruby, Perl and PHP, that can all run on the .NET and Java platforms. These languages are often called scripting languages to indicate that they are used for smaller applications, but can also be used in large applications. The difference between statically and dynamically typed languages is discussed in Section 4.2.5.

4.2.3 The functional paradigm

At the same time as the von Neumann-Eckert model was developed, others developed alternative computational models. One of these was the lambda calculus, and is the basis for the functional paradigm (Backus, 1978). Functional

programming adopts mathematical thinking, where everything is a function with input and result. Functions interact with each other through functional composition, conditionals and recursion (Tucker & Noonan, 2007). Functional programming offer features that help developers build elegant yet powerful and general libraries, and shift the focus from how it should be computed to what should be computed (Thompson, 1999). In recent years, interest in functional programming has increased because the introduction of multi-core processors requires more concurrency, something that functional programming can offer without side-effects and deadlocks.

For those who understand functional languages, code can become very expressive and succinct, as illustrated by the following quick sort example:

```
qsort []           = []
qsort (pivot:list) = qsort less ++ [pivot] ++ qsort more
    where less     = filter (< pivot)  list
          more     = filter (>= pivot) list
```

The first statement says that sorting an empty list results in an empty list. The second statement says that by sorting a list consisting of a pivot and a list, one will first sort all elements smaller than the pivot, then add the pivot, and finally sort all elements equal to or larger than the pivot. This code is much more expressive than either C# and Java, and do not care about how elements should be moved, only what should move.

Unlike imperative programming, functional programming does not involve the use of variables. The effect of this is that no functions can have any side effects, since there exists no states to change. This allows expressions to be evaluated in any order, and makes it easier to build concurrent and transactional programs. This is called referential transparency (Thompson, 1999; Hughes, 1989). Other important characteristics of functional programming include concurrency, lazy evaluation and higher order functions. Concurrency in functional programming comes naturally because there are no mutable objects, and therefore functions can be evaluated in any order. Lazy evaluation means that no function will be called or no value evaluated unless it is necessary and it is therefore possible to create infinite expressions, using only the values needed. Higher order functions are the basis of functional programming, and make it possible to send functions in addition to values to other functions (Hughes, 1989).

The major problem with functional programming is the steep learning curve for developers used to object oriented programming, especially the different mindset needed for functional programming. So called hybrid functional languages, for example F# and Scala, are among the solutions to this problem, and offer a mix between object oriented programming and functional programming (Werner, 2008).

Languages

Several languages have been designed for functional programming. Lisp was one of the first, and is still important in the world of artificial intelligence. Other widely used languages include Erlang and Haskell, both of which can be run on the .NET and Java platforms. More recent additions to the list include F# and Scala, specifically designed for the .NET and the Java platforms respectively.

4.2.4 The logical paradigm

Unlike the other three paradigms, in logical (declarative) programming, rather than declaring how something should be accomplished, what should be accomplished is declared. This is often given as a collection of assertions, or rules about the outcomes and the constraints. Logical programming is often called rule-based programming for this reason, and is most used within artificial intelligence and database information retrieval (Tucker & Noonan, 2007).

Two distinct properties of logical programming are nondeterminism and backtracking. Nondeterminism is the property of being able to find several solutions to a problem, and backtracking means that it is possible to reason about and reproduce decisions made (Tucker & Noonan, 2007).

The advantage of logical programming is the ability to specify what should happen, enabling the machine to decide how it should be accomplished and optimise the performance. The disadvantage is that the paradigm is limited to the field of artificial intelligence and database information retrieval, making it very specialised.

Languages

Within the field of artificial intelligence, Prolog is the most notable logical programming language, and has been used with great success within for example natural language processing (Amble, 2000). Prolog can be used both on the .NET and the Java platforms. In addition to Prolog, CLIPS has also been used with success within expert systems, and is available on the Java platform through the Jess framework (*Jess, the Rule Enginge for the Java Platform*, n.d.).

For database information retrieval, the Structured Query Language (SQL) is most used. Inspired by SQL, Microsoft has created the Language Integrated Query (LINQ) framework, which lets the developer define queries over objects in the language's syntax (*Language-Integrated Query (LINQ)*, 2008).

4.2.5 Statically or dynamically typed languages

Independently of language paradigm, a programming language can be either statically or dynamically typed. For statically typed languages, the types of all variables are declared before compile time and cannot change, while for dynamically typed languages the types are declared at run-time and can change. One exception of this rule is type casting, which is checked in run-time in many languages. Examples of statically typed languages include C, C++, C#, Java, Haskell, and dynamically typed languages include Perl, Python, Ruby, Lisp (Tucker & Noonan, 2007).

The advantage of a statically typed language is the certainty that all the types are correct, and that no other are allowed. This normally means that statically typed languages have better performance because the compiler can make optimisation based on type. However, statically typed languages often require more typing because the type of each variable must be specified.

The advantages of a dynamically typed language are that they allow new features to be tested without compilation, because they are often interpreted, and that they require less typing because the type is implicit. However, these features are added at the cost of performance.

4.2.6 Domain specific languages

Domain specific languages (DSL) are computer languages created specifically to solve a particular kind of problem. DSLs are used extensively and examples include CSS, regular expressions and Ant (Fowler, 2008). Domain specific languages are also essential in language oriented programming (Ward, 1994), as mentioned in Chapter 3. The importance of DSL is also apparent in Bini (2008b, 2008d, 2008c), where he suggests that applications will consist of three language layers, as described in Section 4.2.7.

Fowler (2008) separates domain specific languages into two categories; internal and external. An internal DSL is one that uses a general purpose language, but uses the language in a particular and limited manner. Only some aspects of the general purpose language is used, and only a limited part of the application is affected. This is sometimes called fluent interfaces (Fowler, 2005a), and has successfully been used in the Lisp and Ruby community. In the following example, an internal DSL is used in Rails to specify the relationship between the database model `Person` and various other models.

```
class Person < ActiveRecord::Base
  has_many :jobs
  has_many :companies, :through => :jobs
  has_many :studies
```

```
  has_many :schools_programmes, :through => :studies
  belongs_to :city
  belongs_to :minority
  belongs_to :role
end
```

Unlike an internal DSL, an external DSL uses a language different from the one used in the application. This can either be created using its own syntax, as is the case with CSS and regular expressions, or by using some familiar syntax, as in the case of Ant that uses XML. In most cases, when creating an external DSL, a parser must also be created to enable use of the information in the application. An example of an external DSL is the following from the RSpec story framework (*RSpec*, n.d.):

```
Story: transfer from savings to checking account
  As a savings account holder
  I want to transfer money from my savings account to my
    checking account
  So that I can get cash easily from an ATM

  Scenario: savings account has sufficient funds
    Given my savings account balance is $100
    And my checking account balance is $10
    When I transfer $20 from savings to checking
    Then my savings account balance should be $80
    And my checking account balance should be $30
```

Fields (n.d.) describes what he calls Business Natural Language (BNL), a subset of DSL but with a focus on the business rules. The goal is to create a domain vocabulary similar to the one the domain experts use in their daily life, with the goal that the experts can maintain the rules themselves. An example of a BNL is given in Section 5.4.

To integrate with general purpose languages, the DSL must either be created in the syntax of the language, as is the case of internal DSLs, or a parser that translated from the DSL to the general purpose language must be used, as is the case of external DSLs.

The advantage of creating DSLs is increased productivity because the language is designed specifically for the problem. The disadvantage is the extra effort needed to create the language, and it is important to analyse the return on investment for each problem.

4.2.7 The language layers

One of the problems when developing in a polyglot environment, is how the different languages should be organised. Bini (2008b, 2008d, 2008c) suggests

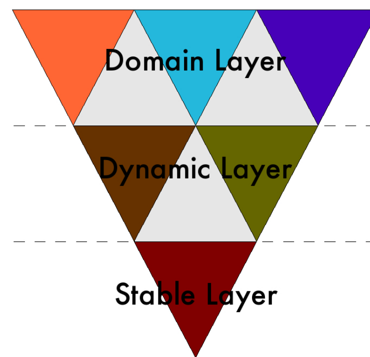

Figure 4.3: The relationship between the stable, dynamic and domain layer. The domain layer contains the domain rules, the dynamic layer consists of most the application code, and the stable layer is the thin foundation that everything else is built upon (Bini, 2008b).

that an application should consist of three language layers, namely the stable layer, the dynamic layer and the domain layer. He calls this fractal programming because of the bounded fractal representation of the languages, as shown in Figure 4.3.

Following this representation, the domain layer will consist of one or more DSLs to define the actual domain rules. The DSLs can be either internal or external, but it must be possible to change rules in production. Beneath the domain layer is the dynamic layer, whose most important property is that application code should not need to be compiled. Languages from all the paradigms can be used, as long as they are dynamic and do not require compilation. The stable layer is what everything else is built on top of, and should preferably be a thin foundation. Statically typed languages from all paradigms are suited for this layer, as performance and stability is of the utmost importance. All interfaces to external applications are also defined in this layer, as this will provide type safety and enable other clients to trust it.

4.3 Summary

This chapter has described the .NET and Java platforms, and how they support polyglot programming. A short comparison has also been made to evaluate whether there is any difference in their support for polyglot programming, and found that they use different approaches, but with the same goal.

In addition to the platforms, the different classes of programming languages have been described. First of all, the different paradigms the languages can support have been described, as well as the difference between statically and dynamically typed languages. In addition, domain specific languages have

been described, and how the different languages can be organised into language layers.

The next chapter will introduce examples of situations where it can be beneficial to use languages from the different paradigms, and how different languages can work together to create the best solution. The examples will also be related to the various language layers.

Chapter 5

Examples

Polyglot programming is not useful unless it gives business value; the fact that it is possible to integrate different languages is not enough. Without using the term polyglot programming, it has been used in games (Civilization 4 uses C++ and Python, and World of Warcraft uses C++ and Lua (Walters, 2008)), to script applications (Applescript for Mac software, Ruby in Google Sketchup (*Google SketchUp Ruby API*, n.d.)) and to mediate requests (Google uses Python to push binary data between servers (Harrison, 2006)).

This chapter will describe examples of situations where polyglot programming can give real business value. The common business value for all the examples is decreased costs of development and maintenance, as the result of the increased productivity gained by using the most efficient tool. For each example, the language paradigm best suited will be described, as well as how the example fit into the language layers presented in Section 4.2.7.

5.1 Web development

Web development is one area within software development that has always been polyglot (Ford, 2006). Best practise today is to separate the content from the presentation using cascading style sheets (CSS). More advanced web applications also include JavaScript to make the web pages more interactive, and interact with a database, using a server side language, where information is stored. This means that in most web applications, at least four different languages are already used for development, from at least two different classes. HTML and CSS are DSLs, while JavaScript and in most cases the server side language are object oriented languages. This is a non-networked type of polyglot programming where the different languages are in the same file, but because of the increasing support for polyglot programming in the .NET and Java

platforms, polyglot programming using a managed runtime is also becoming more popular for web development.

The major reason for the increased popularity of using polyglot programming for web development is the excellent frameworks created using dynamic languages, promising increased productivity and faster turnover. These frameworks normally also include support for generating JavaScript. The most important of these include Ruby on Rails (*Ruby on Rails: Web development that doesn't hurt*, n.d.) created in Ruby, Django (*Django: The Web framework for perfectionists with deadlines*, n.d.) created in Python, and Grails (*Grails: The search is over*, n.d.) created in Groovy. The importance of these frameworks can be seen in that frameworks in other languages use the same ideas, and the fact that the consulting firm ThoughtWorks uses Ruby on Rails for around 40% of their new projects in the US (Fowler, 2007b).

The separation between the stable layer and the dynamic layer as described in Section 4.2.7 fits very well for web development. Because the web interface is likely to change more often than the services it is built upon, a dynamic language is a good fit since it will offer faster turnover. Using this model, the web interface will work as a mediator between the requests and the stable layer, and the stable layer will do the heavy lifting.

The advantage of using more productive frameworks is further increased because the decrease in performance is not that important for web development. This is because the main bottleneck is the Internet connection of the users; the users will not notice it if the application uses 100ms instead of 10ms, as long as it takes 1 second to send the data anyway.

5.2 Testing

The popularity of automated testing has increased recently, especially since it is commonly a key concept in many of the new agile development methodologies, including test driven development (TDD), behaviour driven development (BDD) and extreme programming (XP) (Janzen & Saiedian, 2005; North, 2006; Beck, 1999). Automated testing also gives the developers confidence that the code is still working after changes have been made, especially in dynamic languages which lack a compile cycle. Testing can also be a first introduction to polyglot programming in many businesses; as the test code is not an integral part of the application, it will be easier to introduce new languages here. Since testing is a continuous activity that preferably should include the whole application, it will not fit into one of the three language layers, but will interact with all three.

A problem when testing complex code is that it can be time consuming and

introduces a high amount of coupling in the test process. The time it takes to run the tests is dramatically increased if the code relies on databases and web services, and small changes in the database table might break all tests. To overcome this problem, mock and stub objects are created, to mimic the behaviour of other objects. The purpose of using mock and stub objects is to test in isolation, without relying on the other objects (Thomas & Hunt, 2002). This is an area where testing using a statically language can be replaced with a dynamic language, as the following example from Ford (2008) shows:

The code to test is the interaction between an `Order` class and a `Warehouse` class. The following code is the test using JMock (*jMock - A Lightweight Mock Object Library for Java*, n.d.), a mock framework written in Java.

```
Order order = new OrderImpl(TALISKER, 50);
Mock warehouseMock = new Mock(Warehouse.class);

warehouseMock.expects(once()).method("hasInventory")
    .with(eq(TALISKER),eq(50))
    .will(returnValue(true));

warehouseMock.expects(once()).method("remove")
    .with(eq(TALISKER), eq(50))
    .after("hasInventory");

order.fill((Warehouse) warehouseMock.proxy());

warehouseMock.verify();
assertTrue(order.isFilled());
```

The following code is the same test using JRuby and the Mocha (*Mocha*, n.d.) framework.

```
order = OrderImpl.new(TALISKER, 50)
warehouse = Warehouse.new

warehouse.stubs(:hasInventory).with(TALISKER, 50).returns(true)
warehouse.stubs(:remove).with(TALISKER, 50)

order.fill(warehouse)

assert order.is_filled
```

The Ruby code is much more concise, and the intent of the code is much clearer, because the programming language provides the freedom to remove unnecessary syntax.

Because the testing code will not be run in production, the runtime performance of the language used is not important, but rather programming productivity. Using an interpreted language also enables new tests to be run without compilation.

5.3 Concurrency

With the introduction of multi-core processors the developers were faced with a new challenge, namely how to best utilise these processors using existing tools. One of the hardest aspects of imperative and object oriented programming is dealing with threads, because these introduce race conditions and the risk of deadlocks, because different threads can modify the same variables. Although concurrency has long been an issue for some developers, with multi-core entering the mainstream, concurrency has become important for all developers. It is therefore important to make it easier to manage threads and concurrency (Tucker & Noonan, 2007).

Functional programming aids the process of creating concurrent programs because there are no shared states between processes. This removes the risk of deadlocks, as no data is shared directly. This also makes it easier to add concurrency to an existing application written in a functional programming language (see Section 4.2.3 for more details on functional programming).

In Erlang, a programming language designed for concurrency by Ericsson, data sharing is achieved using message passing. This is similar to communication between people and is therefore a more natural programming idiom (Armstrong, 2007). When benchmarking Yaws (*Yaws webserver webpage*, n.d.), a web server written in Erlang, against the Apache web server, Apache dies at about 4000 parallel sessions, but Yaws is still functioning with more than 80000 parallel sessions (Ghodsi & Armstrong, 2007). This, and the fact that Ericsson has been using Erlang in the telecommunication area for several years, shows that functional programming can be used with great success for concurrent problems.

Although functional languages are good for concurrency, they are not as good when developing user interfaces, for example. By using them in a polyglot environment, the parts that need to be highly concurrent can be written in a functional language, while the rest are written in languages more suited as general purpose languages. Facebook recently used this approach when creating their new chat client, integrating an Erlang chat client with the existing infrastructure written in C++, PHP and JavaScript (Letuchy, 2008).

The business value of better concurrency is first of all about improved scalability because the overhead of spawning new processes is decreased. Because of the independency between the processes, it will also be easier to distribute load on multiple cores and machines. Because the load can be distributed on multiple cores, this also means that the hardware is better utilised.

5.4 Business rules

In most applications today, the business logic is an integrated part of the source code, making it difficult for the domain experts to verify that the business rules are implemented correctly. For the same reason, it is difficult for the domain experts to change the rules, which might happen quite often in business.

A better solution would be to implement the business rules either using a BNL, or a rules engine. As described in Section 4.2.6, a BNL is a language created specifically for describing the business rules, and is a subset of DSLs. This can be created in a vocabulary that the domain experts are already using, making it easier to verify the business rules, and enabling the experts to change the rules. Other benefits include clear communication of the intent of the application and easier communication with the domain experts because they actually understand the rules (Fowler, 2008).

An alternative to a BNL is to use a rules engine. The advantage of using a rules engine as opposed to a BNL is that the developer does not have to create a new parser for the language. The disadvantage is that the language must follow the syntax of the rules engine. Several rules engines have been created. Popular rules engine for the Java platform include Drools (*Drools*, n.d.) and Jess, for the .NET platform InRule (*InRule Technology - Business Rule Engine for .NET*, n.d.) and BizTalk Server (*Microsoft BizTalk Server*, n.d.), and the logical languages are rule engines in themselves.

When creating a BNL or using a rules engine, it is important to consider who will be authoring the rules. Unless the rules are readable and maintainable to the authors, they will require help from the developers, which goes against part of the purpose of using this technique. Another consideration is how often the rules will change. If the rules rarely change, using this technique will probably not be a good investment, but in cases where the rules are changed often, the technique will be beneficial (Fields, 2008). To illustrate this benefit the following example is described in detail in Fields (n.d.):

You have been contracted to replace a payroll calculation system for a consulting company. Each employee have a varying compensation package, the following is for employee John Jones:

```
employee John Jones
compensate $2500 for each deal closed in the past 30 days
compensate $500 for each active deal that closed more than 365 days ago
compensate 5% of gross profits if gross profits are greater than $1,000,000
compensate 3% of gross profits if gross profits are greater than $2,000,000
compensate 1% of gross profits if gross profits are greater than $3,000,000
```

Instead of bending the rules to match the syntax of a programming language, the programming language is made to understand the given rules, so the do-

main expert does not have to learn a new syntax. In this example this only involves string manipulation, but will be more advanced for other problems, so the effort of parsing the rules must be considered.

5.5 Summary

This chapter has described situations where polyglot programming will offer increased business value. In most of the examples the increased productivity is the main reason for choosing a polyglot approach.

This concludes the literature study, which will be followed by a case study, to shed some light on how polyglot programming can be used in a business environment.

Chapter 6

Research process

In Chapter 2, two research methods were identified. The first was a literature study, the second a case study. This chapter will describe case study research in general according to Yin (2003), how this method is planned used in this case study, and how it was actually used.

6.1 Case study research

Although there are many different types of case studies, a common definition is that a case study is "an empirical inquiry that investigates a contemporary phenomenon within its real-life context" (Yin, 2003, pg. 13). Case studies are especially relevant when the boundaries between phenomenon and context are not clearly defined. Because case studies are performed within real-life contexts, they are one of the preferred research methods when asking "how" and "why" something happens, without being able to control influential variables. A case study can consist of both single-case or multiple cases, and both qualitative and quantitative data can be used in the same case study (Yin, 2003).

Case studies have the advantage of researching a phenomenon in its real-life context, which might lead to findings not possible outside this context, and is especially true whenever humans are involved. Case studies also result in detailed description that can be used as basis for new research (Yin, 2003).

According to Yin (2003), because case studies are loosely defined, and because many case studies have been poorly performed, prejudice exists. One of the greatest concerns with case studies is the lack of rigour. Because the research method has been poorly defined, many researchers have allowed biased views to influence the direction of findings and conclusions. Bias will exist in any research method, but more so in case studies because bias can be introduced in all phases of the research. When preparing and choosing sources, one can for

example, choose to look at only the ones which favour some predetermined agenda. Similarly, during data collection, certain types of people may be selected for interviews, and the questions asked may be biased and cause answers to lean in a particular direction.

Another concern about case studies is that the results give little basis for scientific generalisation, because only a single case is studied. But case studies can be generalised to theoretical propositions, and should be used in combination with experiments to increase the credibility of results (Tichy, 1998). The detailed descriptions resulting from a case study can also be used as the basis for new hypothesis and further experimentation (Fitzgerald, Hartnett, & Conboy, 2006; Yin, 2003). A third concern is that case studies are time consuming and often produce documentation that is too detail-oriented to be useful. This can be a result of bad practises, and is not necessarily true for all case studies (Yin, 2003).

6.1.1 Data collection

In order to increase the rigour and the reliability of case studies, Yin (2003) proposes three principles which should be followed when collecting data. The first principle is the use of multiple sources. It is possible to base a case study solely on one of the methods described in Section 6.1.1.1, but being able to use multiple sources is one of the strengths of case studies. The use of data triangulation enables the researcher to address a broader range of issues. Triangulation can also address problems related to construct validity, which is one of the four tests used to evaluate the quality of research (Kidder & Judd, 1986), and is used to test whether the research follows correct operational measures (Yin, 2003).

The second principle is to create a case study database; to organise and document the data collection. This allows other researchers to use the data without being biased by opinions and conclusions drawn in the related research, and therefore increase the reliability of the whole case study.

The third and last principle is to maintain the chain of evidence. This allows other researchers to follow the train of thought from research questions to conclusions, and will increase the reliability of the research. Thus, making sufficient citations, and organising and documenting all data, is necessary throughout the collection of data.

6.1.1.1 Data collection methods

When collecting data several methods exist, including documentation, interview, observation and participation (Yin, 2003; Marshall & Rossman, 2006). All of these will be described in short detail.

Documentation

Documentation is most likely to be relevant for every case study, because this will give the researcher access to information already available about the topic (Marshall & Rossman, 2006). Documentation is a stable source of information that can be reviewed repeatedly, and normally has a broad coverage of the studied case, although it might be difficult to retrieve and access. When selecting documentation it is important for the researcher to stay unbiased (Yin, 2003).

Interview

Kahn and Cannell (1957) calls the interview "a conversation with a purpose", and that purpose is for the participant's perspective of the phenomenon of interest to unfold according to the participant's point of view, not the researcher's (Marshall & Rossman, 2006). To achieve this, it is important that the questions are not addressed in a way that leads the participant into answering a certain way, but that the questions are neutral (Marshall & Rossman, 2006). In a successful interview, the participant should be an informer instead of a respondent, and achieving this cooperation is essential (Yin, 2003).

Observation

The use of observation allows the researcher to cover events in real time and also cover the context within which the events happen. Results in a lab, where conditions are carefully controlled, may be very different from the ones observed in the natural work environment. The problem with observation is that it is time-consuming and may affect the behaviour of the research subjects because they are aware that a researcher is present (Yin, 2003).

Participation

Participation goes one step further than observation. Not only is the researcher observing; he or she is also actively taking part in the events studied. This

gives the same benefits as observation, but from another perspective, as the researcher is actually part of the events that occur. The problems are also the same, but because of researcher interruption, the risk of bias increases (Yin, 2003).

6.1.2 Data analysis

When analysing the collected data, the goal is to treat it fairly and produce compelling analytic conclusions, and to rule out alternative interpretations. Analysis consists of recombining the data to address the initial research questions, but is especially difficult because of the qualitative nature of the data produced by a case study (Yin, 2003).

Yin (2003) describes two general strategies for analysing data. The first strategy is to rely on the theoretical propositions that led to the case study, and shaped the data collection. This will help focus attention on certain data while ignoring other data. The second strategy is to develop a case description, suitable for descriptive case studies. As this study will be explorative, no further details will be included on this second strategy.

6.2 Case study design

Based on the literature study and Section 6.1, this section will describe how this particular case study is designed.

6.2.1 Data collection

To increase the credibility of the data collected in the case study, the three principles described in Section 6.1.1 will be followed as best as possible. The first principle, data triangulation, will be followed through the collection of data from documentation and interviews. This way it will be easier to verify that what the interview subjects answer are similar to that which is documented. To support the second principle, creating a case study database, the recount of what has happened during the case study will be presented objectively and the interview transcripts will be available in Appendix A. The third principle, maintaining a chain of evidence, will be supported by referencing to relevant parts of the literature study and case study.

6.2.1.1 Data collection methods

As mentioned, the data collection methods selected for this case study are documentation and interview. Observation and participation were not selected because of the geographical distance between the researcher and the consultant firm, and a questionnaire was not selected because it would not have given the appropriate information. For software development, possible documentation include source code, planning documents, and in the case of consulting firms, the actual offer to the customer. The source code can help in highlighting problems related to technical aspects, while planning documents and offers can highlight problems related to managerial aspects.

In the case study, interviews will be conducted on management, for example the project leader, but also the developers. This way, a more holistic view of how it was working in a polyglot environment will be discovered, from managerial to technical issues. The plan is to have open ended interviews, but the "conversation" will be steered towards the business perspective and the business benefits of using polyglot programming.

6.2.2 Data analysis

Section 6.1.2 describes two general data analysis strategies. For this case study, the first will be used, namely to rely on the theoretical propositions. These were established in the literature study. As a part of this general strategy, pattern-matching will be used, comparing the actually observed patterns with the predicted ones (Yin, 2003).

When conducting the data collection, findings will be related to what has been discovered in the literature study. To put the case into context, it will be related to the definition of polyglot programming to see what kind of technologies are used to integrate the different languages, and how the team or teams are organised. The languages involved are also of relevance.

The case will also be related to the perceived advantages found in the literature study. These were productivity and maintainability. If other advantages are discovered, these will also be described.

Lastly, the cases will be related to the perceived disadvantages. These were the difficulty of learning new languages, maintainability and tool support. If other disadvantages are discovered these will also be described.

6.3 Case study implementation

Based on Section 6.2, a case study of a Norwegian consulting firm was conducted. The case study consisted of three smaller case studies of either finished or ongoing projects. Two of these were web development projects as described in Section 5.1 and one was related to testing as described in Section 5.2.

6.3.1 Data collection

The original plan was to use both documentation and interview as data collection methods. Due to confidentiality, none of the documentation was made available.

Several employees were however interviewed. All of the interviews were open ended, but the first interview less so than the other. For the first interview, because the researcher had some knowledge about the interview subjects and the projects he had participated in, some of the questions were already created and sent to the subject. The goal was to discover the business benefits, problems and possible resistance during the project. This subject was also part of both the web development projects, and therefore shared valuable information about both, that was later used to steer the other interviews. The other interviews were more open ended, but also tried to steer the conversation in the same direction as the first. For the first web development, the interview subjects were the CTO of the customer, and a senior consultant. For the second web development the same senior consultant was interviewed, as well as a consultant.

In the testing case study, only one consultant was interviewed. Because of the low degree of polyglot programming in this case, the interview revolved more around technical aspects than managerial ones. This made this particular case study less relevant than the other two.

Chapter 7

Bekk Consulting AS - A case study

Bekk Consulting AS (BEKK) is a Norwegian consulting firm delivering advisory services, technology services and application management. BEKK is a part of ErgoGroup AS, and currently has 200 employees (*BEKK's webpage*, 2008). Even though BEKK is relatively small compared to international companies, the projects they are involved in are not; examples include ABB, Telenor, NAV, StatoilHydro.

The rationale for doing a case study of BEKK is their ambition for developing projects using new technology. Examples include their use of agile methodology and Ruby on Rails. At BEKK, polyglot programming is used in at least two areas, web development and testing, and both these areas are targeted by this case study.

This chapter will describe two web development projects using Ruby on Rails, and a project using RSpec and Watir for testing purposes. The findings are later analysed and discussed in Chapter 8 and the technologies involved are described in further detail in Appendix B.

7.1 Web development

In Norway, BEKK is one of the leading actors within web based self-service solutions and portals. Traditionally these are created using technologies like ASP.Net, Java EE and Lotus/Domino, but recently, BEKK has finished one proof of concept project, and has another one in development using Ruby on Rails (Rails). Utilising JRuby makes it possible to develop modern web pages, and at the same time utilise existing functionality written in Java. This section will present both these projects. As mentioned in Chapter 6, data collection has been achieved by interviewing key persons within BEKK and the customers. The transcripts are not included because of confidentiality.

7.1.1 Buypass

Buypass is a Norwegian company delivering identification and payment solutions for the web. Customers include NAV, Posten and Norsk Tipping (*Buypass' webpage*, 2008). Most of Buypass' solutions are written in Java, but they wanted to try out Rails because they wondered if it could be used to increase their productivity and reduce the cost of web development. They also wanted to evaluate whether it was possible to use the framework and language in their existing architecture.

BEKK was hired to create two prototypes. For the first prototype, the goal was to get to know Rails and the libraries that already existed at Buypass, and to verify the ease with which integration between these technologies could be achieved. Because these libraries were written in Java, the decision was to use JRuby and access these libraries directly.

Once it was verified that it was possible to integrate the Rails solution with the existing services, the goal of the second prototype was to verify that Rails could be used in their existing architecture. Because Buypass delivers identification and payment solutions, security is the number one priority. All the clients therefore do not have direct access to the database, but must use an existing Java server. Between the database and the Java server, firewalls are inserted for security purposes, and additional firewalls are introduced between the Java server and the clients. The business logic is also centralised in the Java servers, making it easier for all the clients to use it without duplication. This architecture is shown in Figure 7.1.

To overcome the problems with the architecture, and at the same time utilising ActiveRecord, an object-relational mapping framework offered by Ruby on Rails, a connection adapter for ActiveRecord was created. Instead of talking to a database, the adapter routed the requests to the Java server. This allowed the client to use ActiveRecord as though it was using a database, when in fact it was not.

Because ActiveRecord creates objects per database table, the Java solution was extended to include support for this. The reason was that the existing solutions used the database at a much higher granularity level, and created objects according to concept instead of tables. To automate the generation of the methods needed to support this, the SQL created from ActiveRecord was analysed.

Because none of the developers had prior experience with neither Ruby nor Rails in a professional environment, Buypass and BEKK arranged a course prior to the initiation of the project. The course was held by “Dr.” Nic Williams (*Dr Nic*, n.d.), a renowned developer in the Rails community. In addition to this course, Aslak Hellesøy, the CTO at BEKK, who has broad experience using Ruby and Rails, acted as a coach for the developers in case they needed any

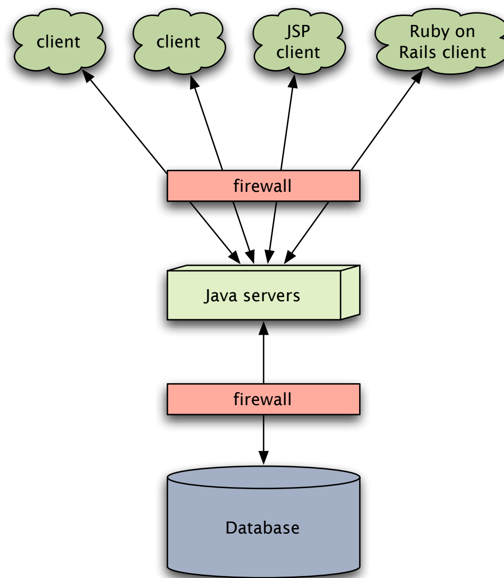

Figure 7.1: Buypass' architecture. Firewalls exist between the database, the Java servers and all the clients using the Java servers. The Ruby on Rails client access the Java server through the same firewall as the rest of the clients.

guidance.

The developers experienced it as easier to learn Rails than the equivalent Java frameworks, mainly because of the "convention over configuration" and DRY (don't repeat yourself) principles that were used when creating the framework, and because everything is inside one framework, and therefore follow the same standards.

They also experienced a productivity gain from using Rails. First of all, because it was not necessary to configure the application, developers could start developing at once. Second, no compilation meant the developers did not have to wait until compilation was finished to see the changes. However, although the language and framework was perceived as more productive, the lack of automatic refactoring and the need to write additional tests did counteract some of the productivity gain.

Buypass used continuous integration and unit tests to verify that changes did not break the application.

7.1.2 Web based extranet solution

BEKK is currently creating a web based extranet solution for one of its customers. The customer originally contacted BEKK to change both the extranet

and executive work solution because the existing solutions were created in outdated Java, and were nearly ten years old. The original idea was to use Java, but because BEKK offered to do the same project cheaper and faster using Rails, the customer was convinced to try it out, but only for the extranet solution.

Based on the Buypass project, the developers had concluded that Rails was best for database driven web applications. Thus, it was a candidate for the extranet solution, which was going to be built on top of a legacy database, consisting of hundred of database tables. Where necessary, existing Java libraries could also be used because of the ability to use JRuby.

In the beginning, the customer was sceptical to using Rails. They did not know much about the language and framework, and did not want to throw out all of their existing Java code. Because of this, a technical offer comparing frameworks, in addition to a prototype of core functionality in the executive work solution was made. However, this prototype was too simplistic, and it did not use the legacy database.

The plan was to create a new database, but since only the extranet solution was to be created, the executive work solution would still use the legacy database. Additionally, a data warehouse application also used the legacy database, and a new database would require a rewrite in that application. To follow the conventions in Rails, they decided to create MSSQL views, which translated the existing tables into a form that could be used by Rails without any configuration.

As BEKK already had the application management of the old system, some of the developers managing it became part of the Rails team. These had little experience with Rails prior to this project, and found the framework easier to learn than the equivalent in .NET or Java. In addition, one of the consultants from the Buypass project, and Aslak Hellesøy joined the team. No time was allocated to dedicated learning, but pair programming was used, and pairs were changed regularly, paring the experienced Rails developer with the inexperienced.

The possibility of using JRuby to interact with existing Java infrastructure was one of the major selling points when choosing Rails. It has been decided not to use this possibility, but the application will be deployed on JRuby and Glassfish for performance reasons. The integration with existing services are achieved using servlets.

7.2 Testing

At BEKK, testing is of the utmost importance to increase the quality of the code, but also because they like to use agile development methodologies in their projects. BEKK therefore has a group dedicated to quality and testing. The consultant interviewed is a part of this group, and an interview with him is the basis for this section. The interview transcript is not included because of confidentiality.

7.2.1 Statens vegvesen

Statens vegvesen is the Norwegian public roads administration, and is responsible for the planning, construction and operation of the national and county road networks, vehicle inspection and requirements, driver training and licensing (*Norwegian Public Roads Administration*, n.d.). BEKK was hired to replace Autosys, the old system for driver licensing- and vehicle registration (Kristiansen, 2006).

The new system, called Au2sys, is a web application written in Java. Because of the complexity of the business rules and treatments rules present in the web interface, BEKK has decided to use extensive automated web testing.

It was decided to use RSpec and Watir, as these were evaluated as the best tools available when the project started. Watir is a framework for interacting with the web browser, and RSpec a framework for testing. This architecture allowed the web testing to be independent from the actual Java code.

7.3 Summary

This chapter has described three projects where BEKK has used polyglot programming when developing software. The first two projects used the Rails web development framework in conjunction with existing Java solutions, using JRuby for integration in the first project, and servlets in the second. The third project used RSpec and Watir to enable web testing of a web application written in Java. In the next chapter, the findings from these case studies will be discussed and contrasted related to the literature study.

Chapter 8

Discussion

In this chapter, the polyglot programming context of the case studies are first discussed. Thereafter, findings in the literature study are discussed in light of the case study conducted on BEKK, as well as the interviews conducted on key persons within the community. Additional findings discovered during the case studies will also be discussed. To conclude the research, the use of case study research will be discussed, a critique of polyglot programming given and a compromise between free and no language choice given.

8.1 Polyglot programming context of the cases

The three projects studied at BEKK all use a different degree of polyglot programming. At Buypass, they use a managed runtime, allowing the languages to interact directly. At the web based extranet solution project, however, the Ruby application will be deployed on a managed runtime, but will interact with existing Java services using a networked servlet approach. This allows for lower coupling between the two languages, and technologies can change independently of each other. Yet another degree of polyglot programming is achieved at Statens Vegvesen, where the web browser becomes the integration between Ruby and Java. The web based extranet solution project and the project at Statens Vegvesen would not have been polyglot, had it not been for the fact that one team is responsible for both languages.

In all the cases studied, the languages involved are Ruby and Java, and the reason is the frameworks available in Ruby, namely Rails, RSpec and Watir in this case. Since its release, Rails has become increasingly popular, and promises increased productivity and less code. Especially for consulting firms, this is an important argument because it means that they can use fewer employees and deliver the solutions faster than they would normally do. Ruby and Rails are

also getting support from Microsoft, Sun and other commercial actors, making it a safer bet for the future. RSpec has also become popular and offer a new approach to unit testing. It is becoming the new standard for testing Ruby applications.

8.2 Advantages

During the literature study, two business benefits of polyglot programming were discovered, namely productivity and maintainability. In this section, these are related to the case studies.

8.2.1 Productivity

Polyglot programming promises increased productivity because the correct tool is used for the task at hand. In all the cases, a dynamic language was chosen because of the frameworks created in them. But using the dynamic language also offers a higher level of abstraction, and frees the developers from a lot of repetition. One example is the meta-programming support present in Ruby, which enables code to be generated in run-time.

Especially in Buypass and the web based extranet solution project did the developers notice a productivity increase. Part of this was the productivity gained of using the language and framework, but productivity was also increased because the principles of "convention over configuration" and DRY (don't repeat yourself) that is the philosophies driving the development of Rails. The lack of configuration meant that the developers could start developing at once, and because Ruby is an interpreted language, developers did not have to wait until compilation was finished to see the changes. However, although the language and framework is more productive, the lack of automatic refactoring and the need to write additional tests did counteract some of the productivity gain, further discussed in Section 8.3.3.

8.2.2 Maintainability

The advantage of using polyglot programming from a maintainability perspective is mainly the belief that choosing the correct tool will result in shorter and more natural code. In the case of Buypass, the application created was new, and they therefore did not have any equivalent code written in another language to compare it with. In the case of Statens Vegvesen, the code was only for testing, and therefore not part of the application. As was the case with

Buypass, Statens Vegvesen did not have any pre-existing equivalent code, and thus no comparison was possible.

In the case of the web based extranet solution project, the size of the current code base was one of the motivations for rewriting it. The existing solution consisted of over a hundred thousand lines of code (LOC), and the extranet solution to rewrite consisted of over 20000 LOC of application code alone, with almost no test code. Although the project was not finished when the case study ended, the new codebase written in Rails consisted of a total of around 10000 LOC, but this is including around 7000 LOC with tests.

8.2.3 Motivation and change of perspective

An additional advantage discovered during the case studies, is that the developers had fun while developing, and had to change their perspective of the existing applications. Especially developers that enjoy learning and using new tools will thrive in a polyglot environment. For many developers, being allowed to only use one language will feel like a limitation.

At Buypass, this was especially noticeable. First of all, the development methodology changed when using a dynamic language. Because they could no longer rely on the type safety present in statically typed languages, and because there is no compile cycle that run through the whole application, they used extensive unit testing to ensure type safety and that the code did not break when changes where made. Continuous integration was also used to discovered errors as early as possible. Second, because Rails follow “convention over configuration”, the interaction with the database had to be changed in order to use the framework without configuring it.

8.3 Disadvantages

During the literature study, disadvantages were also discovered, concerning knowledge, maintainability and tool support. In this section, these are related to the case studies and personal interviews conducted on key persons, and counter measures to the problems are discussed.

8.3.1 Knowledge

One of the perceived disadvantages of using polyglot programming is the increased knowledge required both from the developers and the management. It is also argued that developers are not interested in learning new languages.

From the developers' perspective, none of the case studies conducted confirm this statement. According to the developers interviewed, they found the new framework easy to learn and use, more so than it would be to learn the equivalent frameworks in Java. One of the reasons will of course be that the languages are in the same language paradigm, and the concepts are therefore the same. Using a web framework written in a functional programming language, for example Lift (*Lift, the Scala web framework*, n.d.) written in Scala, would most likely be more effort to learn.

BEKK also put in place measures to counter the knowledge gap of using a new language and framework. In the case of Buypass, they arranged a course together prior to the initiation of the project. In addition, Aslak Hellesøy acted as a coach. In the the case of the web based extranet solution project, the team consisted of consultants with varying knowledge and experience about Ruby and Rails. A new course was not held, but the developers worked in pairs, enabling the inexperienced to learn from the experienced.

From the management's perspective, the web based extranet solution project confirm that the management needs increased knowledge, and might be more reluctant to introduce new technology. Because the old system was written in Java, the original plan was to use Java for the new application as well. The thought of throwing away all the old code was especially intimidating. To convince the management, they needed more knowledge about the tools, and a prototype was created to show it was capable of solving the problem.

When it comes to knowledge, developers can be divided into two classes; those who enjoy developing software, and those who do it to bring bread to the family. Those who enjoy developing software will have no problems adopting new technologies. In a personal interview with Ola Bini, he goes so far as suggesting that enterprise should not hire developers that are not proficient in more than one language. Most businesses cannot afford this luxury; they need developers, even if they are average or worse. Jay Fields also suggests that the management should only focus on hiring the best developers possible, and let the developers choose the platform themselves. In most businesses, it is unlikely that the management will give the developers this freedom.

8.3.2 Maintainability

The problem with maintainability when introducing new languages is that the pool of developers that can maintain the application decreases.

In the case of Buypass, maintainability was not originally an issue because only prototypes were created. However, after BEKK left the project, Buypass decided to further enhance the second prototype created and use this in pro-

duction, but has not put any further thoughts to how it will be maintained. For the web based extranet solution project, BEKK currently has the responsibility for maintenance. At Statens Vegvesen, the code written in Ruby is mainly for testing purposes, and maintaining it becomes less critical.

The problem with maintainability is not much different from today's situation however, where a lot of old legacy applications in FORTRAN, PL/1, and COBOL are still being used and maintained. Developers with knowledge in these languages are getting older, and the same will eventually happen with any language. As long as the niche languages are avoided, it should not be a problem to maintain an application.

8.3.3 Tool support

Especially developers using Java and .NET are used to having comprehensive IDE support. In these languages the IDE becomes necessary to increase the productivity, and when new languages are introduced, it is important not to change the way developers work.

In the cases of Buypass and the web based extranet solution project, some of the productivity gain was counteracted because of the lack of automatic refactoring. IDEs currently also lack support for integrated debugging, interrupting the normal workflow of the developers.

One of the developers at Statens Vegvesen described the situation where he would have to manually refactor the Ruby tests, as his biggest concern if he were to use RSpec to test the existing Java code, instead of using JUnit.

Although the support for other languages are currently not good enough, a lot of research is being done to improve the support in general, specifically refactoring and debugging. NetBeans, Eclipse, Komodo and IntelliJ will all have improved support for languages running on the JVM in forthcoming versions. Visual Studio is likely to include support for additional languages as they release their dynamic language runtime.

Steele (2004) describe what he call the language-maven and tool-maven. Developers living in a language-maven use most of their time learning about languages and how to use them, while those living in a tool-maven use most of their time mastering the development tools. Polyglot programming will probably be of better use to those who thinks of the languages, and not the IDE as the tools.

8.4 Critique of polyglot programming

As seen in the case studies, the number one reason for choosing Ruby was because of the frameworks available for it. Then, what if the frameworks could just be ported to another language. In some cases, this would be possible, while in others it would not. The elegance and flexibility of a framework written in a dynamic language for example, where types do not have to be declared, are not easily ported to a static language without changing the look and feel. The concurrency support for functional programming languages are also hard to port because they rely on the interpreter or virtual machine specifically created for the problem. This is a problem with F# and Scala, that are bound to the workings of their underlying managed runtime, and can therefore not offer for example lazy evaluation.

8.5 Research questions

Based on the research results, this section will answer the research questions listed in Chapter 2.

8.5.1 RQ1: How is polyglot programming used today?

As seen in Chapter 5 and in the case studies, polyglot programming is used in a variety of problem areas. These areas include game development, scripting, web development and testing. In all of these cases, polyglot programming is used because it offers the best solution to the problem, utilising the best features from the different languages.

8.5.2 RQ2: Guidelines for using polyglot programming

To answer the second research question, this section will present guidelines for when and how to use polyglot programming, based on both the literature study and the case study. This will involve the situations where polyglot programming is beneficial, choosing how to integrate the different languages and how to choose languages.

8.5.2.1 When to use polyglot programming

Examples where polyglot programming can be used are described in Chapter 5, but whenever another language offer a solution that is radically better than

the one in the general purpose language should it be considered.

8.5.2.2 Choosing how to integrate

The focus of this research has been on managed runtimes, as these offer easy integration between languages, and have support for many languages (see Section 4.1 for more details). Regardless of this, in only one of the cases studied was this used, in the other two a more loosely coupled integration was preferred. The use of polyglot programming should preferably not involve too much change in the current environment, as this will hinder its adoption, and the integration between languages should have been solved already to minimise overhead of introducing new languages.

8.5.2.3 Choosing languages to use

Based on the literature study and case study it is evident that the language that is best suited for the problem should be chosen, either because the language has built in support to help solving it, or because a framework has been written in that language that helps solve it. This may however, introduce too many languages in the application, so a compromise between allowing all languages to be used, and to use only one language, is to follow Google's example, where a restricted set of languages are allowed. In this case, they allow JavaScript, Python, Java and C++ (Yegge, 2007). This will also help minimise the knowledge disadvantage, as both the developers and management only have to relate to a set of languages, and build support accordingly. Ideally, this should be a minimal set that achieves the goals of the business, and the set should take into account changes in the environment. A good idea would be to pick at least one language from each of the paradigms described in Section 4.2, and preferably both a statically and a dynamically typed language.

Chapter 9

Conclusion

The goal of this research has been twofold. The first goal was to describe how polyglot programming can be used, and has been achieved using a literature study. Because no formal definition has been given of polyglot programming, it has been defined in this research as *programming in more than one language within the same context, where same context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved.* Within this definition, the degree of polyglot programming can vary between four levels, namely integration, organisation of code, the processes within languages run, and the data being manipulated.

Perceived advantages of polyglot programming were productivity and maintainability, and the perceived disadvantages were knowledge, maintainability and tool support.

To aid developers in choosing which languages to use, a language classification has been given, describing the imperative, object oriented, functional and logical paradigm. The difference between statically and dynamically typed languages has also been discussed, and domain specific languages, that are specialised languages for each problem, have been described. The different languages can be organised into three language layers, namely the stable layer, the dynamic layer and domain layer.

Based on the this information, examples where polyglot programming can be used include web development, testing, concurrency and business rules. Web development fits well within the dynamic layer, and recent frameworks promise increased productivity. Testing is an activity that interacts with all language layers, and utilising a dynamic language can make tests shorter are more understandable. Concurrency is best solved using a functional programming language, but will most often be combined with other languages because it is not suitable as a general purpose language. Business rules are implemented on the domain layer, and using a business natural language will allow domain

experts to change rules without interacting with the developers. The common business value for all the examples was decreased costs of development and maintenance, as a result of increased productivity.

The second goal of this research has been to describe how polyglot programming is used, and has been achieved through an explorative case study. The case study was conducted on BEKK, a Norwegian consulting firm, and consisted of three smaller case studies, two using polyglot programming for web development, and one using it for testing. Based on these case studies, the advantages found in the literature study was confirmed. The advantage of productivity was confirmed in both the web development cases, and the advantage of maintainability was confirmed in the case of the web based extranet solution project. The other two did not consider maintenance. An additional advantage that was discovered was increased motivation and change of perspective, especially in the case of Buypass. The disadvantage of knowledge was not confirmed from the developers' point of view, but was confirmed from the management's. Neither cases considered maintainability as an issue, but the disadvantage of tool support was confirmed.

Based on the literature study and case study, guidelines for how to utilise polyglot programming has been described, and as a compromise between having free and no language choice, it is suggested that a set of languages should be used. This set should consist of different types of languages, and will give the developers more flexibility, and at the same time give management control over which languages are used.

Further work

As this has been the first academic work describing polyglot programming, it is important to do further research. A natural way to expand on this research will be to do an experiment to verify that the findings are correct. Especially the advantages and disadvantages concerning maintainability are inconclusive as this was not addressed specifically in any of the cases.

More case studies on businesses using polyglot programming should also be conducted to increase awareness and credibility around it. In many cases, polyglot programming is used without developers knowing it, and increasing their awareness can help them utilise the technique better.

References

Amble, T. (2000). BusTUC: A natural language bus route oracle. In *Proceedings of the sixth conference on applied natural language processing* (pp. 1–6). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Available from http://portal.acm.org/ft_gateway.cfm?id=974148&type=pdf&coll=GUIDE&dl=GUIDE&CFID=8421618&CFTOKEN=81844230

Armstrong, J. (2007). *Programming Erlang: Software for a concurrent world*. The Pragmatic Programmers.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Communications of the ACM*, *21*(8), 613–641. Available from http://portal.acm.org/ft_gateway.cfm?id=1283933&type=pdf&coll=Portal&dl=GUIDE&CFID=71008603&CFTOKEN=93682548

Beck, K. (1999). Embracing change with extreme programming. *Computer*, *32*(10), 70–77. Available from http://ieeexplore.ieee.org/iel5/2/17277/00796139.pdf?tp=&arnumber=796139&isnumber=17277

Bekk's webpage. (2008). Retrieved 2008-04-21, from http://www.bekk.no

Bini, O. (2008a). *Connecting languages (or polyglot programming example 1).* Retrieved 2008-05-08, from http://ola-bini.blogspot.com/2008/04/connecting-languages-or-polyglot.html

Bini, O. (2008b). *Fractal programming.* Retrieved 2008-06-07, from http://ola-bini.blogspot.com/2008/06/fractal-programming.html

Bini, O. (2008c). *Language explorations.* Retrieved 2008-05-08, from http://ola-bini.blogspot.com/2008/01/language-explorations.html

Bini, O. (2008d). *Viability of Java and the stable layer.* Retrieved 2008-05-08, from http://ola-bini.blogspot.com/2008/01/viability-of-java-and-stable-layer.html

Braithwaite, R. (2007). *The challenge of teaching yourself a programming language.* Retrieved 2008-06-02, from http://weblog.raganwald.com/2007/10/challenge-of-teaching-yourself.html

Brooks, F. P. (1987, April). No silver bullet: Essence and accidents of software

engineering. *Computer*, *20*(4), 10–19.

Brooks, F. P. (1995). Calling the shots. In *The mythical man-month: Essays on software engineering* (Anniversary ed., pp. 87–94). Addison-Wesley.

Buypass' webpage. (2008). Retrieved 2008-04-21, from http://buypass.no/

Byrne, D. (2008, January). Integrating Java and Erlang. *TheServerSide.com*. Retrieved 2008-05-12, from http://www.theserverside.com/tt/articles/article.tss?l=IntegratingJavaandErlang

The Da Vinci machine project. (2008). Retrieved 2008-05-22, from http://openjdk.java.net/projects/mlvm/

Delorey, D. P., Knutson, C. D., & Chun, S. (2007, May). Do programming languages affect productivity? A case study using data from open source projects. *Emerging Trends in FLOSS Research and Development, 2007. FLOSS '07. First International Workshop on*. Available from http://ieeexplore.ieee.org/iel5/4273068/4273069/04273079.pdf?tp=&arnumber=4273079&isnumber=4273069

Django: The web framework for perfectionists with deadlines. (n.d.). Retrieved 2008-04-30, from http://www.djangoproject.com/

Dmitriev, S. (2004, November). Language oriented programming: The next programming paradigm. *onBoard*(1). Available from http://www.onboard.jetbrains.com/is1/articles/04/10/lop/

Dr Nic. (n.d.). Retrieved 2008-06-09, from http://drnicwilliams.com/

Drools. (n.d.). Retrieved 2008-06-04, from http://www.jboss.org/drools/

Duarte, G. (2008). *Language dabbling considered wasteful.* Retrieved 2008-06-02, from http://duartes.org/gustavo/blog/post/language-dabbling-considered-wasteful

Eyler, P. (2006). *A new JRuby interview and more.* Retrieved 2008-05-20, from http://www.linuxjournal.com/node/1000103

Fields, J. (n.d.). *Business natural languages - Domain specific languages for empowering subject matter experts.* Retrieved 2008-05-22, from http://bnl.jayfields.com/

Fields, J. (2008, March). Business natural languages. In *QCon London.* Available from http://www.infoq.com/presentations/fields-business-natural-languages-ruby

Fitzgerald, B., Hartnett, G., & Conboy, K. (2006). Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*(15), 200–213.

Ford, N. (2006). *Polyglot programming.* Retrieved 2008-05-08, from http://memeagora.blogspot.com/2006/12/polyglot-programming.html

Ford, N. (2008). Polyglot programming. In *The ThoughtWorks anthology: Essays on software technology and innovation* (pp. 60–69). The Pragmatic Programmers.

Fowler, M. (2005a). *Fluent interface.* Retrieved 2008-05-22, from http://martinfowler.com/bliki/FluentInterface.html

Fowler, M. (2005b, June). *Language workbenches: The killer-app for domain specific languages?* Available from http://www.martinfowler.com/articles/languageWorkbench.html

Fowler, M. (2007a). *One language.* Retrieved 2008-01-15, from http://martinfowler.com/bliki/OneLanguage.html

Fowler, M. (2007b). *Railsconf 2007.* Retrieved 2008-04-30, from http://martinfowler.com/bliki/RailsConf2007.html

Fowler, M. (2008). *Domain specific languages.* Available from http://martinfowler.com/dslwip/ (Work in progress)

Ghodsi, A., & Armstrong, J. (2007). *Apache vs. Yaws.* Retrieved 2008-03-13, from http://www.sics.se/~joe/apachevsyaws.html

Google SketchUp Ruby API. (n.d.). Retrieved 2008-06-09, from http://code.google.com/apis/sketchup/

Grails: The search is over. (n.d.). Retrieved 2008-04-30, from http://grails.codehaus.org

Harrison, M. (2006). *Python at Google (Greg Stein - SDForum).* Retrieved 2008-06-09, from http://panela.blog-city.com/python_at_google_greg_stein__sdforum.htm

Hughes, J. (1989). Why functional programming matters. *Computer Journal*, *32*(2), 98–107. Available from http://www.math.chalmers.se/~rjmh/Papers/whyfp.html

Hugunin, J. (2007). *A dynamic language runtime (DLR).* Retrieved 2008-02-13, from http://blogs.msdn.com/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx

Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley.

InRule technology - Business rule engine for .NET. (n.d.). Retrieved 2008-06-04, from http://www.inrule.com/firstTime.aspx

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43–50. Available from http://ieeexplore.ieee.org/iel5/2/32339/01510569.pdf?tp=&arnumber=1510569&isnumber=32339

Jess, the rule enginge for the Java platform. (n.d.). Retrieved 2008-06-02, from http://www.jessrules.com/

jMock - A lightweight mock object library for Java. (n.d.). Retrieved 2008-06-04, from http://www.jmock.org/

JRuby: Java powered Ruby implementation. (n.d.). Retrieved 2008-03-13, from http://jruby.codehaus.org/

The Jython project. (n.d.). Retrieved 2008-03-13, from http://www.jython.org/Project/index.html

Kahn, R. L., & Cannell, C. F. (1957). *The dynamics of interviewing: Theory, tech-*

nique, and cases. John Wiley, New York.

Kidder, L., & Judd, C. (1986). *Research methods in social relations* (5th ed.). Holt, Rinehart and Winston, New York.

Krill, P. (2008). *Sun's Da Vinci machine broadens JVM coverage.* Retrieved 2008-05-22, from http://www.infoworld.com/article/08/01/31/davinci-machine_1.html

Kristiansen, M. (2006). Bekk vant storkontrakt med statens vegvesen. *Computerworld*. Available from http://www.idg.no/computerworld/article34935.ece

Kullbach, B., Winter, A., Dahm, P., & Ebert, J. (1998, October). Program comprehension in multi-language systems. *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, 135–143. Available from http://ieeexplore.ieee.org/iel4/5867/15624/00723183.pdf?tp=&arnumber=723183&isnumber=15624

Lam, J., & Hugunin, J. (2007). Just glue it: Dynamic languages on Silverlight. In *Mix07*. Available from http://msstudios.vo.llnwd.net/o21/mix08/07_MP4s/DEV02.mp4

Language-integrated query (LINQ). (2008). Retrieved 2008-06-03, from http://msdn.microsoft.com/hi-in/library/bb397926(en-us).aspx

Leghari, N. (2008). *Integrating .NET and Erlang using OPT.NET.* Retrieved 2008-05-12, from http://weblogs.asp.net/nleghari/archive/2008/01/08/integrating-net-and-erlang-using-otp-net.aspx

Letuchy, E. (2008). *Facebook chat.* Retrieved 2008-06-09, from http://www.facebook.com/note.php?note_id=14218138919&id=9445547199&index=1

Lift, the Scala web framework. (n.d.). Retrieved 2008-06-04, from http://liftweb.net/index.php/Main_Page

Lipow, M. (1982, July). Number of faults per line of code. *IEEE Transactions on Software Engineering*, *SE-8*(4), 437–439. Available from http://ieeexplore.ieee.org/iel5/32/35929/01702967.pdf?tp=&isnumber=&arnumber=1702967

Marshall, C., & Rossman, G. B. (2006). *Designing qualitative research* (4th ed.). SAGE Publications.

Maxwell, K., & Forselius, P. (2000, Jan/Feb). Benchmarking software development productivity. *IEEE Software*, *17*(1), 80–88.

Maxwell, K., Van Wassenhove, L., & Dutta, S. (Oct 1996). Software development productivity of European space, military, and industrial applications. *Software Engineering, IEEE Transactions on*, *22*(10), 706–718.

Meyer, B. (2002, May). Multi-language programming: how .NET does it. *Software Development*. Available from http://se.ethz.ch/~meyer/publications/softdev/multi-language.pdf

Microsoft BizTalk server. (n.d.). Retrieved 2008-06-04, from http://www.microsoft.com/biztalk/en/us/default.aspx

Mocha. (n.d.). Retrieved 2008-06-04, from http://mocha.rubyforge.org/

Mono project. (n.d.). Retrieved 2008-06-09, from http://www.mono-project.com/Main_Page

Nilsson, N. (2008). *Should you really learn another language?* Retrieved 2008-06-02, from http://www.infoq.com/news/2008/05/should-you-learn-languages

North, D. (2006). *Introducing BDD.* Retrieved 2008-06-03, from http://dannorth.net/introducing-bdd

Norvig, P. (1998). *Teach yourself programming in ten years.* Retrieved 2008-06-02, from http://norvig.com/21-days.html

Norwegian public roads administration. (n.d.). Retrieved 2008-06-04, from http://www.vegvesen.no/cs/Satellite?c=Publication&pagename=SVV%2FSVVforwardToSite&sitename=engelsk

Pigott, D. (2006). *HOPL: An interactive roster of programming languages.* Retrieved 2008-01-15, from http://hopl.murdoch.edu.au/

Robinson, S., Nagel, C., Watson, K., Glynn, J., Skinner, M., & Evjen, B. (2004). *Professional C#* (3rd ed.). Wiley Publishing.

Rose, J. (2008). *Bravo for the dynamic runtime!* Retrieved 2008-06-02, from http://blogs.sun.com/jrose/entry/bravo_for_the_dynamic_runtime

Rose, J., Coward, D., Bini, O., Cook, W. R., Pedroni, S., & Theodorou, J. (2008). *JSR 292: Supporting dynamically typed languages on the Java platform.* Retrieved 2008-03-13, from http://jcp.org/en/jsr/detail?id=292

RSpec. (n.d.). Retrieved 2008-05-22, from http://rspec.info/

Ruby on Rails: Web development that doesn't hurt. (n.d.). Retrieved 2008-04-30, from http://www.rubyonrails.org

Sebesta, R. W. (2008). *Concepts of programming languages* (8th ed.). Pearson Addison Wesley.

Spiewak, D. (2008). *The plague of polyglotism.* Retrieved 2008-05-20, from http://www.codecommit.com/blog/java/the-plague-of-polyglotism

Steele, O. (2004). *The IDE divide.* Available from http://osteele.com/archives/2004/11/IDES

Thomas, D., & Hunt, A. (2002). Mock objects. *IEEE Software*, *19*(3), 22-24. Available from http://ieeexplore.ieee.org/iel5/52/21654/01003449.pdf?tp=&arnumber=1003449&isnumber=21654

Thompson, S. (1999). *Haskell: The craft of functional programming* (2nd ed.). Addison-Wesley.

Tichy, W. T. (1998, May). Should computer scientists experiment more? *Computer*, *31*(5), 32–40. Available from http://ieeexplore.ieee.org/iel4/2/14870/00675631.pdf?tp=&isnumber=&arnumber=675631

Troelsen, A. (2003). *C# and the .NET platform* (2nd ed.). Apress.

Tucker, A. B., & Noonan, R. E. (2007). *Programming languages: Principles and paradigms* (2nd ed.). McGraw Hill.

Venners, B. (1999). *Inside the Java virtual machine* (2nd ed.). McGraw Hill.

Vinoski, S. (2008). Multilanguage programming. *IEEE Internet Computing*, *11*(3), 83–85.

Walters, C. G. (2008). *Polyglot programming.* Retrieved 2008-06-09, from http://cgwalters.livejournal.com/17292.html

Ward, M. (1994). Language oriented programming. *Software — Concepts and Tools*(15), 147–161. Available from http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf

Watts, N. (2008). *Even more than polyglot programming.* Retrieved 2008-05-11, from http://thewonggei.wordpress.com/2008/01/22/even-more-than-polyglot-programming/

Werner, B. (2008). *The rise of functional programming: F#/Scala/Haskell and the failing of Lisp.* Retrieved 2008-02-13, from http://www.brandonwerner.com/2008/01/13/the-rise-of-functional-programming-fscalahaskell-and-the-failing-of-lisp/

Wexelblat, R. L. (1980). The consequences of one's first programming language. In *Sigsmall '80: Proceedings of the 3rd acm sigsmall symposium and the first sigpc symposium on small systems* (pp. 52–55). ACM. Available from http://portal.acm.org/ft_gateway.cfm?id=802823&type=pdf&coll=GUIDE&dl=GUIDE&CFID=71501395&CFTOKEN=12997147

Whorf, B. (1941). The relation of habitual thought and behavior to language. In L. Spier (Ed.), *Language, culture, and personality, essays in memory of Edward Sapir* (pp. 75–93). Sapir Memorial Publication Fund.

Yaws webserver webpage. (n.d.). Retrieved 2008-03-13, from http://yaws.hyber.org/

Yegge, S. (2007). *Rhino on Rails.* Retrieved 2008-06-09, from http://steve-yegge.blogspot.com/2007/06/rhino-on-rails.html

Yin, R. K. (2003). *Case study research: Design and methods* (3rd ed.). SAGE Publications.

Zukowski, J. (2006). *Java 6 platform revealed.* Apress.

Appendix A

Interviews

During this study, interviews have been conducted on key persons within BEKK, Buypass and in the community.

The purpose of interviewing employees from BEKK and Buypass was to gain knowledge on how they used polyglot programming in their projects, and the purpose of interviewing persons in the community was to get a broader perspective. In this appendix only the interview transcripts of the key persons within the community can be found, the others are not included because of confidentiality. The interview subjects and their roles were:

- **Neal Ford**, meme wrangler
- **Ola Bini**, core contributor to JRuby
- **Jay Fields**, working with DSL and BNL

A.1 Neal Ford

Neal Ford is a "meme wrangler" and talks about polyglot programming on his blog, keynotes, podcasts and books.

Neal Ford was interviewed by email on May 17, 2008.

Question 1. Definition of polyglot programming

Talking to Aslak, it seems that I might have misinterpreted the meaning of polyglot programming. How would you define it?

In my definition, polyglot programming defines a new kind of software stack: utilizing languages most suited to a particular problem running on the same managed runtime. We'll build applications by composing languages instead of using a single, general purpose language (like Java), trying to enhance it with increasingly complex frameworks.

In my definition, I exclude things like web services because I use the level of integration as a measurement. Do you agree with this definition or is it polyglot as long as different languages are used?

I would agree with your separation. I define a polyglot solution as languages that run on the same runtime, not as an integration solution. That would, for example, exclude the common places where people are using multiple special purpose languages now, such as using Java, SQL, XML, and JavaScript in a single application. Polyglot solutions for me produce the same bytecode.

Question 2. How to choose what languages to use?

If corporations are to embrace polyglot programming, both developers and managers must become proficient in more than one language, or the corporation must hire developers that know the different languages.

If corporations are to embrace polyglot programming, both developers and managers must become proficient in more than one language, or the corporation must hire developers that know the different languages.

How is the corporation supposed to choose what languages to use, and what metrics do you suggest they use?

Corporations already do this to a massive degree. Look at the standard Java solution stack: Java, of course, but also Ant, Spring, Struts, Hibernate, SQL, JavaScript, etc. Each framework is its own language because of the ubiquitous use of XML. Each configuration file is it's own language, which share the same

syntax but different semantics. Each XML file has it's own grammar (expressed in DTD or Schema). Developers already know more than 10 languages. But often these languages are ills-suited to the problem to which they are applied. For example, the declarative nature of Ant greatly restricts its power, and the syntax in XML is horrid. Something like Gant (and DSL written atop Groovy) is imperative, expressive, and much more suited to the task at hand. Trying to shoe horn development into a few language means you use lots of tools not suited to their tasks.

Question 3. What do you see as polyglot programming benefits?

The benefits are more concise, expressive code, closer to the problem you try to solve. My colleague Ola Bini has published a pyramid that helps define the new software stack. In it, he has a stable layer on the bottom, written in a language that is performant and amenable to verification (either through a static type system or mathematically). On top of that is a dynamic (or series of) dynamic languages that allow developer to get work done more effectively. On top of that are domain specific languages (DSLs), getting abstractions closer to the problem domain in a way that is much more effective than with frameworks.

Regarding the benefits, do you know of any experiments or experiences that show this?

Not on a large scale yet, but I would argue that we've been doing this in a not very effective way for the last 12 years, with the advent of code reuse via frameworks in Java. Another place where this is used to great effectiveness is the Unix file system. Developers who are familiar with it routinely use sed, awk, perl, and bash to solve problems.

Question 4. What do you see as polyglot programming problems?

Any modern software problem! Let's say you have an application ostensibly written in Java, but you have one part that needs extreme multi-threading, like a scheduling algorithm. Instead of trying to write that in Java (threading in Java is difficult at best), use Jaskell or Scala, which are inherently thread-safe because they are functional. It would be cumbersome to write the entire application in Scala, but it is well suited for the parts that are ill suited to Java. Let's say this application has a Swing-based user interface. Writing that in Java is insane: static typing adds a ridiculous amount of overhead to UI code. So, write the user interface in Groovy, using its SwingBuilder DSL. The entire thing runs on the JVM, but you've composed the application from multiple languages.

Regarding the problems, do you know of any experiments or experiences that show this?

Currently, people solve this problem by using separate platforms. SQL is a classic case here: we endure the terrible impedance mis-match between sets and objects and the headaches of crossing machine boundaries to justify the higher efficiency of SQL. The new stack will avoid the impedance problems by running most of the applications on the same managed run-time.

Question 5. Where do domain specific languages belong in a polyglot programming model?

See question 3!

A.2 Ola Bini

Ola Bini is one of the core contributors to JRuby, enabling Ruby to run on the Java Virtual Machine. He is also a member of the JSR 292 expert group, assessing the possibility of adding a invoke_dynamic bytecode to the JVM which will increase performance for dynamic languages. In addition to this he writes about polyglot programming and programming languages in his blog.

Ola Bini was interviewed by email on April 17, 2008.

Polyglot programming

Question 1. Definition of polyglot programming

Talking to Aslak, it seems that I might have misinterpreted the meaning of polyglot programming. How would you define it?

In my definition, I exclude things like web services because I use the level of integration as a measurement. Do you agree with this definition or is it polyglot as long as different languages are used?

This is actually a bit hard. My general gut feeling is that we are talking about polyglot programming when you are using more than one language inside one application. That just pushes the definition onto what an application is, of course. But in some cases if you have a good restful architecture that is tight and used within one application with different languages, that would still constitute polyglot programming.

Question 2. How to choose what languages to use?

If corporations are to embrace polyglot programming, both developers and managers must become proficient in more than one language, or the corporation must hire developers that know the different languages. How is the corporation supposed to choose what languages to use, and what metrics do you suggest they use?

Metrics is always a problem. To some degree this will of course be a problem for the enterprises and corporations. Polyglot programming is generally more suited for better programmers, and larger enterprises doesn't necessarily have lots of high-end programmers. Most good developers should be able to choose the right language based on the task at hand. I do think that Google's approach is really good - they have chosen a few languages at different levels (C++, Java, JavaScript and Python), and then build their infrastructure around these languages. That means that training can be handled better, ways of figuring

out which language should be used at different places gets easier and so on.

For the other question in there, I think that no one should hire a programmer who is not proficient in more than one language.

Question 3. What do you see as polyglot programming benefits?

In my opinion, there are lots of benefits to the approach. The most obvious one might be that you get access to more libraries and frameworks (an example is using Ruby with Java. You get Rails together with Java's XML processing libraries, for example).

Languages are good at different things, and to a degree programming is always a trade-off between using what comes natural in the language or working around deficiencies in it. Design patterns in Java is a good example here. Many of them doesn't exist in languages with higher order functionality, and using them in Java helps you around some of the things that are generally painful in that language.

One of the things that I like about the approach is that I can use a dynamic language that gives me ultimate flexibility, while at the same time I can fall back on a static language in those places I need it. Since I never will know perfectly well where bottlenecks might show up, this gives me quicker turnaround and less problems when something goes wrong.

Regarding the benefits, do you know of any experiments or experiences that show this?

I don't know of any specific experiments about this approach, although I know that ThoughtWorks has used these approaches for a while - especially with JRuby.

Question 4. What do you see as polyglot programming problems?

Regarding the problems, do you know of any experiments or experiences that show this?

I think there is a certain resistance to it. Programmers have gotten used to living in one programming languages for a long time and it's hard to convince them about the benefits. Another problem is tool support. In many cases the current tools doesn't correctly cross the borders between languages, so you need to use different tools in different places of the application.

Question 5. Where do domain specific languages belong in a polyglot programming model?

I definitely believe that DSL's are absolutely crucial for polyglot programming. Programming is getting more and more complex and it's obvious that we need approaches to separate the parts that need to be more malleable from the more static parts of an application. Generally all domains have domain languages, jargon and so on, and this makes total sense to translate into a layer in the application. This gives many benefits - the most visible one might be that it's easier to make sure that business rules actually model the domain correctly. The closer the language the programmer use to the language used by business experts, the easier it will be to verify the correctness. And it will also be easier to identify and fix errors.

Java platform

Compared to the .NET platform, the Java platform was created to run Java, as opposed to C#, C++ and VB. As far as I have understood it this means that support for new languages are built differently on the two platforms. But there is a lot of new development being done on the Java platform, including JSR 223, JSR 292, and the Da Vinci Machine. It is also possible to compile code from any language into bytecode, as long as that language follows the constraints of the bytecode.

Question 6. How are these new technologies utilised and how are the constraints overcome in JRuby, including meta-programming?

Actually, the Java platform and the .NET platform is extremely similar. .NET is really a machine for running C# good, and both C++ and VB needed to be modified quite heavily to work well on the .NET platform. So .NET is really not more multi-language than Java is. And the support is actually extremely similar between the environments. They need exactly the same layers. There are some small functional differences but in the large it's extremely similar.

Currently in JRuby, we compile down to quite dynamic bytecode. Since the dynamic dispatch happens here, metaprogramming is really simple, even though we pay a performance hit on this it's not really hard to get it working. When JSR292 arrives with invoke_dynamic we will be able to utilise this for some performance benefits, but it won't really change our model that much.

A.3 Jay Fields

Jay Fields use Rails in his daily work, and talks a lot about domain specific languages and business natural languages in his blog.

Jay Fields was interviewed by email on April 16, 2008.

Polyglot programming

Question 1. Definition of polyglot programming

Talking to Aslak, it seems that I might have misinterpreted the meaning of polyglot programming. How would you define it?

Concisely, I would define Polyglot Programming as using many languages in combination to provide the simplest solution.

I think we've been doing Polyglot programming for a very long time. For example, to do my day job I need to know Ruby, SQL, Javascript, HTML and so on. Polyglot programming is not new, and neither is the recognition that it is a good thing. I first heard the idea under as "It's better to be a specializing generalist than a specialist". Unfortunately, I can't remember where that comes from, it was many years ago.

Polyglot programming also fits well with the "No Silver Bullet" and "The right tool for the job" memes. I don't think it's important because it's a new idea, but I do think it's a great way to concisely express what we've already been talking about for several years.

In my definition, I exclude things like web services because I use the level of integration as a measurement. Do you agree with this definition or is it polyglot as long as different languages are used?

For simplicity I'd probably stick with more than one language equals polyglot.

Question 2. How to choose what languages to use?

If corporations are to embrace polyglot programming, both developers and managers must become proficient in more than one language, or the corporation must hire developers that know the different languages. How is the corporation supposed to choose what languages to use, and what metrics do you suggest they use?

I don't think corporations should choose the languages. I think they're best off by hiring the best IT employees that they possibly can and then let the employees make the decision.

Of course, you'll need to include not just the developers, but also the support staff. However, assuming you hire the best for all roles they should be able to work together to find the best solution for everyone.

Question 3. What do you see as polyglot programming benefits?

Regarding the benefits, do you know of any experiments or experiences that show this?

I don't know of any experiments, but I think experience is all around us. The fact that we already use several different languages to perform different tasks proves that there are benefits. If we could do everything in one language I think we would. I don't think anyone desires to learn many different languages, I think everyone would prefer a silver bullet. However, experience has shown us that we need different tools for different tasks; therefore, every day I need to use around 6 different languages to be effective.

Question 4. What do you see as polyglot programming problems?

Regarding the problems, do you know of any experiments or experiences that show this?

Again, I don't know of any experiments, but we also have plenty of experience with problems. First of all, there are many more bad programmers than good. Good programmers use the best tool for the job and grow their experience so they know what the best tool is. Bad programmers believe that they are too busy working to become better at their job. They don't spend time learning other languages. As a result they solve problems using the one or two languages that they know. Generally good programmers have to come in and clean up those mistakes. Polyglot programming also requires constant learning. Today's languages will quickly be eclipsed by the languages of tomorrow that solve problems in superior ways. The rate at which technology is currently moving requires good programmers to learn a new language every year, and often requires you to already know several before you can be effective. It also provokes application rewrites every few years (of course, bad programmers contribute to requiring rewrites).

Domain specific languages

Question 5. Where do domain specific languages belong in the polyglot programming model?

Domain Specific Languages are already a large part of polyglot programming. For example, the configuration files in modern Java frameworks are domain specific languages that every Java developer needs to learn in order to be productive. Build systems, regex, etc are also DSLs that every good programmer knows and uses to be better at their job.

If what I call "Business Natural Languages (BNL)" gain more popularity they could slightly change what polyglot programming looks like for a developer. BNL is what I use to describe Domain Specific Languages that are written and maintained entirely by domain experts. I've done a few projects that used this model and we were very successful with it. However, introducing this type of solution means that developers need to know many different languages, but they also need to know how to design a language for their domain experts to use. It remains to be seen whether this will be a realistic mainstream solution in the future. Having done it successfully twice I find it fascinating and very beneficial; however, people smarter than me (Martin Fowler is one of them) believe that it's unrealistic that domain experts will ever take the place of programmers for designing the business rules of the system.

Appendix B

Technologies used in the case studies

In the case studies, BEKK used new frameworks that might be unknown for many. This appendix will explain these, namely Ruby, Ruby on Rails, RSpec and Watir.

B.1 Ruby

Just like the dog is a human's best friend, Matz is with Ruby trying to create a programmers best friend. Inspired by a mix of Smalltalk, Perl, Eiffel, Ada and Lisp, Ruby is trying to capture the best from all worlds. The principle behind Ruby is simply put: "Principle of least surprise", manifested in how natural it is to read Ruby code, see the code below.

```
2.times do
  articles.each do |article|
    article.read
  end
end
```

This short example shows off some of Ruby's great features. First of all, everything in Ruby is an object, so in this example, **2** is an object. It is therefore possible to call the function *times* on this object, which is an iterator in Ruby. The code inside the *do* and *end* is called a block. A block is similar to a lambda function and can be called explicitly. *Each*, the function called on *articles*, is another iterator in Ruby, which will iterate over all the articles, and for each article will call the function *read*. As can be seen from all the functions in this example, function calls can omit the braces if the function takes no parameters.

Ruby is also dynamically typed, meaning that types only exist in run-time, using duck typing (if it walks like a duck and quacks like a duck, it have to be a duck) to decide what class an object belongs to. An important feature with Ruby is that all classes are open, giving Ruby build in support for meta-programming, where code is added to classes in run-time.

B.2 Ruby on Rails

Rails is a web framework written in Ruby following the MVC (model-view-controller) pattern. In essence, Rails is a domain specific language for the web, making it easy and natural to develop web applications. Rails enhanced the Ruby language through added functionality, utilising Ruby's capabilities to extend all classes in runtime, and offer a systematic way of building web applications.

Rails principles are DRY (don't repeat yourself) and "convention over configuration". This makes programmers very productive, and means a lot of boilerplate code can be omitted. A good example of this is that the object relational mapper in Rails called ActiveRecords that use meta-programming to populate objects with the corresponding data and methods needed to read and change this data.

B.3 RSpec

RSpec is a behaviour driven development (BDD) framework written in Ruby. It provides two frameworks, namely a story framework for describing behaviour at the application level, and a spec framework for describing behaviour at the object level. The capabilities of the spec framework are similar to regular unit test frameworks, but use a different approach. The approach is to create a domain specific language for describing the expected behaviour. The following example tests that a new account has a balance of $0:

```
describe Account, " when first created" do
  before do
    @account = Account.new
  end

  it "should have a balance of $0" do
    @account.balance.should eql(Money.new(0, :dollars))
  end

  after do
    @account = nil
```

```
  end
end
```

B.4 Watir

Watir is a simple library for automating web browsers, and is optimised for simplicity and flexibility. One of the main areas of use is automatic web testing. Watir drives browsers the same way people do. It clicks links, fills in forms and presses buttons. Watir also checks results, such as whether expected text appears on the page. Watir is a Ruby library that works with Internet Explorer on Windows. The following example tests that a search in Google for "pickaxe" include "Programming Ruby":

```
require 'watir'
require 'test/unit'

class TC_article_example < Test::Unit::TestCase
  def test_search
    ie = Watir::IE.new
    ie.goto("http://www.google.com/ncr")
    ie.text_field(:name, "q").set("pickaxe")
    ie.button(:value, "Google Search").click
    assert(ie.text.include?("Programming Ruby"))
  end
end
```

Appendix C

Abbreviations

AJAX: Asynchronous Javascript And XML

API: Application Program Interface

ASP: Active Server Pages

BEKK: Bekk Consulting AS

BDD: Behaviour Driven Development

BNL: Business Natural Language

BSF: Bean Scripting Framework

CIL: Common Intermediate Language

CLR: Common Language Runtime

CLS: Common Language Spesification

CSS: Cascading Style Sheet

CTO: Chief Technology Officer

CTS: Common Type System

DLR: Dynamic Language Runtime

DRY: Don't Repeat Yourself

DSL: Domain Specific Language

DTD: Document Type Definition

E4X: ECMAScript for XML

EJB: Enterprise Java Bean

GC: Garbage Collector

GUI: Graphical User Interface

HTML: HyperText Markup Language

IDE: Integrated Developer Environment

JDK: Java Development Kit

JIT: Just In Time

JMS: Java Messaging Service

JSP: Java Server Pages

JSR: Java Specification Request

JVM: Java Virtual Machine

LOC: Lines Of Code

MRI: Matz Ruby Interpreter

MSSQL: Microsoft SQL

MVC: Model-View-Controller

.NET: Microsoft Platform

ORM: Object-Relation Mapping

RBAC: Role Based Access Control

REST: Representational State Transfer

RPC: Remote Procedure Call

SQL: Structured Query Language

XML: eXtensible Markup Language