Comparison of different programming languages and environments

(Motivation) We're still unsure about what technologies we should use. The greatest common denominator is: All of us know how to write code in Java. However: All of us want to learn something new, and since no one of us has ever built a large-scale web-application we'd need to learn the ropes either way.

(What is *this*) Because of this I compiled a list portraying some progamming languages and paradigms suitable for developing such an application. Since it is possible that we'll have to implement a rather complex algorithm we're interested in a rather speedy language. Personally I'd also like to code in a language that eases development.

(What I'll show you) The structure of the document is the following: First there is a brief introduction to different execution models of programming languages (compiled, interpreted) and what that means for us. I would then try to give an overview of different paradigms in programming as well as a a short remark about web-frameworks. I'll try to keep it short. Last but not least I'll introduce some programming languages.

Execution model

Compiled vs. Interpreted

(Compiled) C, C++, Objective-C and Assembler are examples for compiled languages. It's simple: Code is compiled into a machine-readable format and instructions are executed by the processor directly. Compiled code has two major downsides: First, your applications have to be compiled separately for every target plattform. If your application uses external libraries you'll also have to care about linking in these, binary compatibility may also be an issue. The other one is they greatly prolong development since things like automatic garbage collection are mostly not available. But: compiled languages are the fastest solutions and produce most efficient code (if coded correctly).

(Interpreted) PHP, Perl, Python and even Java are examples for interpreted languages. They run inside a programm which hosts your application. Since code is not executed directly interpreted languages offer a great deal of security - a defective application won't crash the whole system, webserver or whatever it is executed in. They also greatly speed up development since more flexible concepts can be implemented than in most compiled languages. However, flexibility and security comes at a price: PHP for example is roughly twenty times slower than compiled C. To execute an application written in an interpreted language one needs the appropriate software for doing so. Whilst easy portable on the one hand, software that needs to be interpreted will only run on a platform featuring the appropriate interpreter.

It is worth noting that it is of course possible to write an interpreter for an otherweise compiled language, gaining some of the benefits of interpreted languages. It is also remarkable that most interpreted languages today are in fact compiled into an intermediate format, like Java Bytecode or the .NET CIL *(Common Intermediate Language)*. One should also remember that there is often more than one implementation of a single language. People tend to mix up Java and the Java Virtual Machine, or Perl and Parrot, or — one of the most prominent examples — Python and its various implementations, like CPython (A compiling interpreter written in C), Jython (Python that runs in the Java Virtual Machine) or PyPy (An interpreter itself written in Python).

It is also possible to compile an otherwise interpreted language. In fact, this is what facebook does with HipHop for PHP, which compiles PHP into C++. GHC *(Glasgow Haskell Compiler)* uses a similar approach, it compiles Haskell into C and compiles that into object-code (that is, a machine-executable program).

A short word on typing disciplines

The overall notion one will get when comparing some of these languages is that the more interpreted a language is, the more static is it's type-system. One can say in general that a more dynamic type-system allows for more flexible development. However, excessive use of dynamic typing leads to confusing code, and since in a dynamic type system the type is determined at runtime it will also be slower than a static type system that knows at compile-time what types are in use. Though being a static and strong type system, Haskell implements another very interesting concept. Here the type is inferred statically from the context — a method very similar to duck typing in Ruby or Python, where the type of an object is determined from the operations applies on it.

For the sake of completeness: There are also weak type-systems like those used in C or Assembler. These languages allow data — whatever it is — to be interpreted as what you want it to be. Though very flexible it is also said to be error-prone, since it's

the programmers responsibility to interpret the data correctly.

Programming paradigms

Today there is a multitude of programming paradigms, that is, the principle way we tackle problems. The term *"paradigm"* itself has many facettes: I'll focus on the general approach a programming language offers to solve a problem, in contrast to paradigm as **the way data is processed** (parallel, sequential), **the way functions are applied** (concatenative for example, like in Postscript or Enchilada) or anything else.

Object oriented programming

Buzz-Words: Java, Smalltalk, information hiding, encapsulation, API *(Application Programming Interface)*, imperative,

Probably the most prominent modern programming paradigm. Object oriented progamming allows for abstraction of real objects in the world into abstract classes and instances in your language, it prinipially is a way for structuring a program. Control-flow itself is imperative.

Functional programming

Buzz-Words: haskell, lisp, scheme, lambda, declarative programming, closures, higher order functions

This is another approach to programming than imperative/object-oriented programming. The key-concept is the evaluation of functions in a mathematical sense. Languages supporting functional programming differ significantly in detail. There is for example Haskell, a purely functional programming language, that does not know variables in the same sense like an imperative programmer would suspect; on the other hand there are languages like Python, which basically are imperative programming languages featuring functional elements.

(A first remark) Personally I think that object-oriented programming is both a very intuitive and workable approach for writing large-scale applications that deal with significant amounts of data, since OO allows for modelling and structuring real objects easily. However, functional elements of a language come in handy quite often, since they concentrate on solving mathematical problems, I'd not want to miss them.

Aspect-oriented programming

Buzz-Words: aspect, joinpoint, cutpoint, AspectJ, debugging, profiling, assert

Best-known representative today is AspectJ, a Superset of the Java-Language from which one can conclude that Aspect-oriented programming somehow is related to object-oriented programming. Indeed, it is, but it takes the concept a bit further. The basic idea is, to hook in code anywhere without modifying the source files. To achieve this one defines a **jointpoint**, say,

before: myfunction

. With this jointpoint code can be associated, and that way the code is like inserted but without modifying the source file.

Major applications of aspect-oriented programming are debugging and profiling, but it also provides mechanisms to improve modularity and reusability of code. Imagine how easy it is to write a plugin for a given application! Define a joinpoint, cut in at that point et voilá – there you go.

(Remark) Personally I'd love to play around with stuff like that. It's fun and ease and can (partly) replace tools like JUnit, Debugger, Profiler or Assertions. One can simply crosscut her application with own Assertions or debug-messages without breaking existing code.

Declarative programming

Buzz-Words: XML *(eXtensible Markup Language)*, XSL *(eXtensible Style Language)*, Annotations, Generics, generic programming

A completely different paradigma is declarative programming (although functional programming in deed is a simple kind of declarative programming). The idea is to describe the problem — and the computer does the rest. An example of declarative programming is a WYSIWYG GUI-Builder. In fact, what you do is describing the layout — the actual GUI is built by a special library that understands your description. Other applications are XSL (describes Rules for autmatically transforming one machine-readable format into another) or Annotations in Java and Scala. The concept here is that you annotate information (describe something) and a special piece of software processes that information to fulfill a certain task. As mentioned, declarative elements can be found even in Java and Scala. Many web-related languages are in fact designed in the spirit of declarative programming — consider HTML and CSS.

Meta programming

Buzz-Words: introspection, reflection, generic programming

A meta language is a language which can talk about a language and it's structure. Generic programming (like Generics in Java) are in fact a simple application of this paradigm. By annotating a Type-parameter (don't be confused, these are **not** Java-Annotations) to a statement or class-definition you say something about that statement (which Type it is for). Modern programming languages allow for Meta programming using Reflection, that is, it is possible for a program to gain information about its structure at its own runtime. In conjunction with declarative programming (like Annotations in Java) this can lead to very powerfull programs which introspect themselves at runtime and fulfill magical functionality based on the declarations given in their own source code.

Web frameworks

The programming language is not the only technology which we have to think about. There are many frameworks and libraries available which can be used for building rich, powerfull web-applications. Some languages, like Ruby, are closely tied to a framework (**Ruby + Rails = Ruby on Rails**), others don't really need one since they're already highly specialised (PHP). Again others offer endless possibilites (Java). Furthermore do some of these frameworks imply the use of certain software like a Java-Application-Server, etc.

Programming languages

All of the following programming languages allow for structured programming and feature automatic garbage collection. I've tried to portray both features and design philosophy of the several languages. For each language there is a Sample-Hello-World-program. Since I can't code fluently in all of them Samples vary to a certain degree. I tried to emphasize particular features in the Examples which are special to a single language as well as to cover simple Web-Applications, showing how easy or complicated the development of such applications can get.

PHP

Features: object-oriented, closures, reflection, interpreted, scalable, high penetration
Typing: dynamic, weak
Tools: phpDocumentor, phpUnit (test framework), PEAR (package manager)
Who uses it?: Wikipedia, facebook

PHP is a simple but powerfull language which has started as a simple tool for creating personal homepages with a syntax widely borrowed from Shell-Scripts and Perl. In its current major release (PHP 5.3) PHP has grown into a powerfull framework for building rich web applications. It has a huge standard library, object-oriented programming and recently added support for namespaces (somewhat like packages in Java).

The core libraries distributed in a default installation of PHP supports the DOM *(Document Object Model)* (for examining and manipulating XML files), PDO *(PHP Database Objects)* (a standardized API for accessing Databases with builtin support for SQLite and additional support for MySQL, PostgreSQL and many more) and easy-to-use libraries for IO and Networking.

PHP has some very exciting features for adding "magic functionality" and extending the language. For example one does not have to define getters- or setters, magic methods called "__get" and "__set" will do the job.

One major advantage of PHP is, that it's widely spread and supported, if not easy to install. System Administrators will have few to nothing to do setting up the environment for a Software Package written in PHP. PHP is FLOSS *(Free (libre) Open-Source-Software)*. However, the PHP License is incompatible to the GPL, due to restrictions on the usage of the term PHP.

Different Implementations of PHP are available. The original Implementation is written in C and acts as the de facto standard for PHP (since there is no formal specification of the language). Efforts habe been made porting PHP to the JVM *(Java Virtual Machine)* and .NET, solutions exist which translate PHP into Scala. Facebook, which is entirely written in PHP, has developed "HipHop", a piece of software that compiles PHP into native C++.

Sample: **Hello world**

```
echo "Hello, world!";
```

Sample: **Simple Cookie-WebApplication**

```
<?php

function page($title, $body) {
        $title = htmlspecialchars($title);
        return <<<PAGE
<html>
        <head><title>$title</title></head>
```

```
        <body>$body</body>
</html>
PAGE; }

session_start();
if (!isset($_SESSION['count'])) {
        echo page("Biscuits.", "<h1>Welcome!</h1><p>This is the first time I see you.");
        $_SESSION['count'] = 0;
} else {
        echo page("Biscuits.", "<h1>Welcome back!</h1><p>I have seen you {$_SESSION['count']} times before");
        $_SESSION['count']++;
}
session_write_close();
```

Ruby

Features: object-oriented, functional, closures, reflection, interpreted
Typing: dynamic, duck, strong
Tools: gems (package manager)

Ruby is a purely object-oriented programming language featuring dynamic typing. Everything in Ruby is an object, including primitives. Every function is a method. Additionally it allows functional expressions (in fact everything in Ruby is an expression, including statements like if and while).

One of the design principles of Ruby is abbreviated *"POLA"* — *"the principle of least astonishment"*, meaning the developers try to keep the design consistent and understandable. Rubys Syntax is very much inspired by that of Perl and Smalltalk.

Different Implementations of Ruby are available. The original Implementation is written in C and acts as the de facto standard for Ruby (since there is no formal specification of the language). Besides Ruby itself there are implementations like JRuby (targetting the JVM *(Java Virtual Machine)*) and IronRuby (targetting the .NET-platform) - both featuring Just-in-Time-Compilation.

Rails

Rails is a set of tools and libraries designed to ease the development of web-applications. Rails does all configuration and is coonfigured with reasonable defaults. Additionally it separates data model, controlling logic and presentational logic (the *"views"*) by implementing a Model-View-Controler-Pattern.

Rubys great success is mostly that of Rails, the both things are so strongly associated with each other that many people often speak of one things meaning both.

Sample: **Hello World**

```
puts 'Hello world'
```

For a quick introduction into rails see http://www.troubleshooters.com/codecorn/ruby/rails/rapidlearning_rails.htm

Python

Features: object-oriented, closures, reflection, interpreted
Typing: dynamic, duck, strong
Who uses it?: Trac, MoinMoin (a popular wiki), Plone (a powerful CMS *(Content Management System)*)

Python featues imperative, object-oriented and functional programming techniques. It's distributed along with a rich standard library (*"batteries included"*). Pythons syntax is strongly aligned to it's visual formatting, i.e. statements take up a single line, indentation declared blocks, etc. Due to this, Python urges the programmer to maintain a good style.

Different Implementations of Python are available. The original Implementation (CPython) is written in C and acts as the de facto standard for Python (since there is no formal specification of the language). Besides CPython there are implementations like Jython (targetting the JVM *(Java Virtual Machine)*) and IronPython (targetting the .NET-platform) and PyPy (a Python implementation written in Python).

Genshi

Genshi is a Framework for delivering content on the web. Trac is built on-top of Genshi, since it's developed by the same folks (edgewall.org). Genshi features rich support for XML and supports Templating, allowing for the easy implementation of a Model-View-Controller-Pattern.

Sample: **Hello World**

```
print "Hello, world!"
```

Perl

Features: object-oriented, closures, reflection, interpreted
Typing: dynamic, duck, strong
Specials: integrated regular expressions

Perl was designed by a linguist, emphasizing expressivity, it favors language constructs that are concise and natural for humans to read and write. One of the main features of perl is its builtin support for regular expressions, since it was originally built for processing text. Due to its abundance of features and mightiness Perl is also called *"the swiss army chainsaw of programming languages"*.

Perl syntax reflects the idea that *"different things should look different"*, therefor arrays, variables, keywords, etc. all have different sigils (that is, a special symbol attached to its name, like *"$"* for designating variables). However, since there is a variety of ways to achieve one and the same thing Perl code can get rather obfuscated. A funny read is the *"obfuscated perl contest"*.

Sample: **Hello World**

```
print "Hello, world!"
```

Java

Features: object-oriented, generic, reflection, annotations, scalable, high penetration, JVM
Typing: static, strong
Tools: javadoc, JUnit (unit tests), ant (build tool)

Everybody knows Java. Conservative language, strongly object-oriented with primitive datatypes, leading to headache greatly.

Apart from he authors personal resentments against Java, Java has some cool features too. It offers support for generic programming and reflection, introduces a concept called annotations and allows for heavy meta programming at a yet good performance.

Struts

Along with Java comes a wide variety of tools, a proper zoo of libraries and techniques. One of those tools is Struts, developed by the Apache Foundation who also develop Tomcat, Ant and Maven. Struts is a Framework for web applications. However, the author does not have any personal experiences with it.

Sample: **Hello World**

```
public class HelloWorld {
        public static void main(String... args) {
                System.out.println("Hello, world");
        }
}
```

Sample: **Fibonacci-Sequence (efficient implementation)**

```
class Fib {
        public static long fib(int n) {
                long val1 = 0; long val2 = 1; long valN = 1337;
                for (int i = 2; i <= n; i++) {
                        valN = val1 + val2;
                        val1 = val2;
                        val2 = valN;
                }
                return valN;
        }
}
```

Sample: **Ackermann-Function (recursive)**

```
import java.math.BigInteger;
class Ackermann {
        public static BigInteger A(BigInteger m, BigInteger n) {
                if (m.signum() == 0)
                        return n.add(BigInteger.ONE);
                else if (n.signum() == 0)
                        return A(m.subtract(BigInteger.ONE), BigInteger.ONE);
                return A(m.subtract(BigInteger.ONE), A(m, n.subtract(BigInteger.ONE)));
        }
}
```

Sample: **Printing a HelloWorld-Website**

```
import javax.servlet.http.*;
import javax.servlet.*;

public class HelloServlet extends HttpServlet {
        public void doGet (HttpServletRequest req,
                           HttpServletResponse res)
        throws ServletException, IOException {
                PrintWriter out = res.getWriter();

                out.println("Hello, world!");
                out.close();
        }
}
```

AspectJ

120% Java: aspect-oriented

AspectJ is Java with some added functionality called *"aspects"*. They allow defining *"joinpoints"* which can be *"cross cutted"* using an *"advie"*, i.e. executing custom functions at certain points in the execution of a program without modifying the original source code. Thanks to the JVMs integrated support for introspecting Java-class-files it can also be applied to already compiled sources, hence can be used for easily deploying plugins, profiling and debugging.

Spoken funny: Aspect-oriented programming is COMEFROM, the opposite of GOTO, taken to the next level (*höhö*). No seriously, aspect-oriented programming is the logical next step in software engineering, where object-oriented was the last great leap. It redefines principles of abstraction and encapsulation, leading to more strucutred software (if used correctly).

Sample: **Tracing the above Ackermann-Function**

```
aspect TraceAckermann {
        pointcut ack() : execution(BigInteger Ackermann.A(BigInteger, BigInteger));

        before() : ack() {
                System.out.println(System.currentTimeMillis() + ": Entering A("
                        + thisJoinPoint.getArgs()[0] + thisJoinPoint.getArgs()[1];
        }

        after() returning : ack() {
                System.out.println(System.currentTimeMillis() + ": A("
                        + thisJoinPoint.getArgs()[0] + thisJoinPoint.getArgs()[1]
                        + " done.";
        }
}
```

Scala

Features: object-oriented, functional, currying, reflection, annotations, JVM **or** .NET
Typing: static, strong, inferred, structural
Specials: XML integrated, **traits**, **implicits**, singleton pattern integrated, algebraic datatypes (via **pattern matching** and **case classes**)
Tools: scaladoc, scalaTest, JUnit, sbaz (package manager)
Who uses it?: Twitter (after migrating from Ruby)

Scala is a neat language introducing wicked features in a prudential way. Out-of-the-box Scala supports XML natively as part of its Syntax, is as object-oriented as Java or Smalltalk (everything in Scala is an object), features elements of functional programming via closures, higher-order-functions, currying and case-classes. In addition it features even elements of aspect-oriented programming, rendering the retroactive addition of methods to existing classes possible using so-called *"implicits"*. It also features *"traits"*, to some degree allowing for multiple inheritance via the mechanism of *"mixins"*.

Since Scala is compiled for the JVM any existing Java-Library can be used, that is, also existing knowledge can be applied ;-). When the Scala standard-library is in the class-path a compiled Scala program can naturally be run using any any JVM (in other words: once it's compiled it's an ordinary software-package like any other compiled Java-programm).

Long story short: Scala rocks. Scala is Java done right. Scala is the answer to everything. Scala is the origin of all true wisdom; Scala is simply the best (the author likes Scala, although the compiler is terribly slow).

Lift

Lift is a Framework written in Scala for building rich web applications. It uses Apache Maven.

Sample: **Hello World**

```
object HelloWorld extends Application {
        println("Hello, world!");
}
```

Sample: **Sum command-line argumnets**

```
object Main {
  def main(args: Array[String]) {
    try {
      val elems = args map Integer.parseInt
      println("The sum of my arguments is: " + elems.foldRight(0) (_ + _))
    } catch {
      case e: NumberFormatException =>
        println("Usage: scala Main <n1> <n2> ... ")
    }
  }
}
```

Sample: **Factorial function with BigInt, folding and as it's own operator**

```
object FactorialSyntax extends Application {
        def fact(n: Int): BigInt = new Range(1, n, 1).foldLeft (1) (_ * _);
        class Factorizer(n: Int) {
                def ! = fact(n)
        }
        implicit def int2fact(n: Int) = new Factorizer(n)
        println( 10! )
}
```

Sample: **XML-Processing**

```
object XMLPage extends Application {
        val dateString = java.text.DateFormat.getDateInstance().format(new java.util.Date())
        def theDate(title: String) =
                <html>
                        <head>
                                <title>{ title }</title>
                        </head>
                        <body>
                                Hello! Today is { dateString }
                        </body>
                </html>;
        println(theDate("What's the time?"));
}
```

Note that in the above example, XML really is part of the syntax of the programm. In other words: The above code creates XML-Objects, rather than a String containing XML. The above example is also particular interesting for the inclusion of a java library (line 2) and the use of a function inside a function (theDate essentially is a function and the code inside the object-block essentially is a main-function of a class which is implemented using the singleton-pattern via the use of the object-keyword).

Haskell

Features: **purely** functional, currying, structured, generic, compiled
Typing: static, strong, inferred
Specials: very easy extendable (allows operator overloading), algebraic data types, lazy evaluation
Tools: haddock (documentation), cabal (package manager)

Haskell is a purely functional language, allowing for some kind of imperative programming using a mechanism called *"monads"*. There are no side effects due to its intrinsic referential integrity. Thanks to Haskells lazy evaluation it allows for automated parallelization. Since there is no notion of state a Haskell programm always is at a well defined point in its execution, therefore allowing for structural induction, hence greatly enhancing structural verification of a program. Because of this haskell is can greatly be used to specifiy the functionality of programs written in other, less type-safe languages.

Most implementations of Haskell compile into native machine-code, however, interpreters are also available. Haskells flagship implementation GHC *(Glasgow Haskell Compiler)* generates C code and compiles that into machine-code. Other implementations like JHC and YHC do also compile into native machine-code, others, like Hugs, interpret.

Long story short: Haskell is the way to true wisdom (which is Scala). Haskell is a smart language allowing for smart expressions, how ever it is also a tough language, hurting your brain very hard. Knowledge of Haskell truly enriches your programming skills.

Sample: **Hello World**

```
main = putStrLn "Hello, world!"
```

Sample: **Fibonacci-Sequence (efficient implementation)**

```
fib n = fibs !! n
        where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Sample: **Ackermann-Function (recursive)**

```
ackermann :: Integer -> Integer -> Integer
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
```

Sample: **Printing a HelloWorld-Website**

```
import Text.XHtml
import Network.CGI

main = runCGI $ output $ renderHtml $ body << h1 << "Hello World!"
```

Sample: **Simple Cookie-WebApplication**

```
import Network.CGI
import Text.XHtml
import Control.Monad (liftM)
import Data.Maybe (fromMaybe)

hello 0 = h1 << "Welcome!" +++ p << "This is the first time I see you."
hello c = h1 << "Welcome back!" +++ p << ("I have seen you " ++ show c ++ " times before.")

page t b = header << thetitle << t +++ body << b

main = runCGI $ handleErrors cgiMain
        where cgiMain = do
                c <- liftM (fromMaybe 0) $ readCookie "anjuhakoda"
                setCookie (newCookie "anjuhakoda" (show (c+1)))
                output $ renderHtml $ page "Biscuits" $ hello c
```