NEAR EAST UNIVERSITY

GRADUATE SCHOOL OF APPLIED SCIENCES

A COMPARATIVE ANALYSIS OF PROGRAMMING TECHNIQUES FOR INTERACTIVE WEBPAGE DESIGN

Mohammad Elfawair

Master's Thesis

Department of Computer Engineering

Nicosia – 2007

ABSTRACT

The selection of the programming technique to be used for interactive webpage design should be according to the conventional criteria including readability, maintainability, performance, and memory used during execution. This thesis compares two programming approaches for interactive webpage design, namely regular programming with embedded scripting and scripting with embedded programming, and it provides recommendations to software designers about which of these programming techniques is preferable for designing interactive web applications. In general, regular programming languages are preferred for programming logic, and scripting languages are preferred for business logic. In business applications such as banking systems, both programming and business logic are required, so combining programming languages and scripting languages is needed to draw the benefits of both techniques. Unfortunately, there are no conventional benchmarks for the composition of programming languages and scripting languages. Therefore, a special benchmark algorithm was designed for evaluating two approaches for combining regular programming and scripting. This benchmark was implemented to compare the two techniques based on the conventional criteria of readability, maintainability, program size, performance, memory used, and skills needed for programming. The benchmark designed demonstrates that a scripting language with embedded programming language is the preferable approach for the interactive web application design.

ACKNOWLEDGMENTS

First, I thank Dr. Adil Amirjanov. Under this guidance, I successfully overcame many difficulties. In each discussion, Dr. Amirjanov answered my questions patiently, and I felt the progress from his advices. I asked him many questions, and he always answered my questions in detail.

Special thanks to the Faculty of Engineering for their helping me acquire good qualifications that will help me in both my professional and personal life.

I also would like to thank my friends who helped me for the past four years and the ABAC club for their supporting me during my research.

Finally, I wish to thank my family, especially my parents. Without their endless support and love for me, I would never have achieved my goals.

Dedicated to my mother and father …

CONTENTS

ABSTRACT i

ACKNOWLEDGMENTS ii

DEDICATION iii

CONTENTS iv

LIST OF FIGURES vii

LIST OF TABLES viii

1 INTRODUCTION 1

1.1 Contribution 2

1.2 Thesis Overview 3

2 PROGRAMMING LANGUAGES 4

2.1 Procedural Programming 4

2.2 Examples of Procedural Programming Languages 5

2.3 Object Oriented Programming(OOP) 6

2.4 Major Tenets of Oriented Programming 8

2.4.1 Encapsulation 8

2.4.2 Inheritance 8

2.4.3 Polymorphism 8

2.4.4 Dynamic Binding 9

2.5 Examples of OOP languages 9

2.6 Procedural Programming vs. Object Oriented Programming 11

2.7 Summary 12

3 SCRIPTING LANGUAGES **13**

3.1 Procedural Scripting Languages 14

3.2 Object-oriented Scripting Languages 14

3.3 Characteristics of Scripting Languages 15

3.3.1 Loose Typing 15

3.3.2 Interpretation 18

3.4 Examples of Scripting Languages 20

3.5 Programming Languages vs. Scripting languages 21

3.6 Summary 25

4 DESIGN OF A BENCHMARK FOR INTERACTIVE WEBPAGE DESIGN **26**

4.1 Conventional Benchmark Problems 27

4.1.1 Sum-File Benchmark 27

4.1.2 Binary-trees Benchmark 27

4.1.3 Partial-Sums Benchmark 28

4.2 Benchmark Algorithm Design 30

4.3 Summary 33

5 PROGRAMMING LANGUAGES WITH EMBEDDED SCRIPTS **35**

5.1 Java Servlets 36

5.2 Advantages of Servlets over CGI 36

5.2.1 Efficiency 36

5.3 Life Cycle of a Servlet 37

5.3.1 Servlet Initialization 38

5.3.2 Interacting with Clients 39

5.3.3 Destroying a Servlet 42

5.4 Database Connectivity 43

5.4.1 Creating a JDBC Application 43

5.4.2 Using the Statement Interface 44

5.5 Implementing a Benchmark Problem 45

5.6 Summary 46

6 SCRIPTING WITH EMBEDDED PROGRAMMING LANGUAGES **47**
6.1 Java Server Pages (JSP) 48
6.2 Advantages of JSP 49
6.3 JSP Syntax 49
6.3.1 Directives 49
6.3.2 Declarations 50
6.3.3 Expressions 51
6.3.4 Scriptlets 51
6.4 The Benchmark Problem 52
6.5 Comparison of Programming Techniques 52
6.5.1 Readability 53
6.5.2 Maintainability 54
6.5.3 Program Size 54
6.5.4 Compilation Time 55
6.5.5 Performance 55
6.6 Summary of Results 55
6.7 Summary 56

7 CONCLUSIONS **58**
7.1 Future Work 59

REFERENCES **62**

APPENDICES **63**

APPENDIX A Implementation of the Benchmark (Method 1) **64**
A.1 Source listing: `myapplication1.html` 64
A.2 Source listing: `MyServlet.java` 66

APPENDIX B Implementation of the Benchmark (Method 2) **69**
B.1 Source listing: `myapplication2.html` 69
B.2 Source listing: `Myjsp.jsp` 71

APPENDIX C Source listing: `MyClass.java` **74**

LIST OF FIGURES

2.1 Categories of programming languages 4

3.1 Categories of scripting languages 14

4.1 Block diagram of the benchmark algorithm 31

4.2 Flowchart of the special benchmark algorithm 33

5.1 Life cycle of a servlet from loading until unloading 38

LIST OF TABLES

4.1 Performance of the *Sum-File* benchmark algorithm 27

4.2 Performance of the *Binary-trees* benchmark algorithm 28

4.3 Implementation of the Partial-Sum benchmark algorithm 29

6.1 Comparison of servlets and JSP 56

Chapter 1

INTRODUCTION

The selection of which programming approach to use for solving a problem is important for creating software products efficiently. This thesis studies two programming approaches for implementing interactive webpages, regular programming with embedded scripting and scripting with embedded programming. It provides recommendations to programmers about which of these programming approaches is preferable for designing and implementing interactive webpages, especially for business applications. This work has been motivated by the growing number of interactive web applications in the world.

Programming languages can be divided into two broad categories:

1. *Procedural programming languages* that consist of procedures (or functions), each of which consists of programming statements those statement are executed by calling that procedure. C is as an example of this type of programming languages.

2. *Objectoriented programming languages* contain objects that are made up of attributes and methods. Attributes are modified usually via object methods. C++ and Java are examples of such type of programming languages.

On the other hand, scripting languages represent a very different style of programming than programming languages. Scripting languages assume that there already exists a collection of useful components written in other languages. Scripting languages are, in general, not intended for writing applications from scratch. They are rather intended for gluing together existing components.

Scripting languages can be divided into two groups:

1. Procedural-scripting languages: Sometimes used as a synonym for imperative scripting (specifying the steps the program must take to reach the desired state), based upon the concept of the procedure call.

2. Object-Oriented scripting: Object-oriented programming (OOP) is a programming paradigm that uses *objects* to design computer programs. OOP utilizes several techniques from previously established paradigms, including inheritance, modularity, polymorphism, and encapsulation. Many popular programming languages nowadays support OOP such as ActionScript, Java, Javascript, C#, Visual FoxPro, VB.Net, C++, Python, Perl, PHP, Ruby, and Objective-C.

In recent years, object-oriented programming has become especially popular in scripting languages. For example, Python and Javascript are scripting languages built upon OOP principles. Languages such as Perl and PHP have been supplemented with object oriented features since earlier versions.

Programming languages with embedded scripts combine both programming languages and scripting languages. This means that program statements contain scripting statements, usually HTML tags, to achieve better performance and to gain benefits from both programming and scripting languages. Java servlets are an example of such a technique, where Java programs run on the web server and generate HTML pages for the clients to interact with it. The servlet life cycle and its advantages are discussed later in this thesis.

Scripting languages with embedded programming languages is the second technique for combining programming and scripting languages together, where a scripting language statements (usually HTML tags) contains programming language statements. Java Server Pages (JSP) is an example of this technique, where documents contains Java code.

1.1 Contribution

This thesis provides recommendations about what general programming approach should be used for particular interactive web application design problem. The goal of this thesis was achieved by designing and implementing a special benchmark problem, which is used for comparing programming languages with embedded scripts and scripting lan-

guages embedded with programming languages. The comparison uses the criteria of reliability, maintainability, program size, compilation time, format, memory used, and performance for particular applications.

1.2 Thesis Overview

The remaining chapters of this document are organized as follows:

- *Chapter 2* discusses major types of programming languages. It describes procedural programming languagesand object-oriented programming languages and compares them.

- *Chapter 3* discusses scripting languages, main types of scripting languages, procedural scripting languages, and object-oriented scripting languages, which use objects as building blocks. It discusses scripting languages characteristics and compares different types of scripting languages.

- *Chapter 4* discusses benchmark problems for comparing programming languages and scripting languages according to CPU time, memory used, and code size. This chapter also discusses the design of a benchmark algorithm that contains programming logic and presentation logic. The implementation of this algorithm is discussed in detail in *Chapter 5* and *Chapter 6*.

- *Chapter 5* discusses software design using programming languages with embedded scripting languages. This chapter also discusses Java servlets as an example of this concept.

- *Chapter 6* discusses scripting languages with embedded programming languages, Java Server Pages (JSP) as an example of this technology, advantages of JSP over existing technologies, and the basics of JSP syntax. This chapter also compares JSP and Java servlets from both a qualitative perspective (readability, simplicity, maintainability) and a quantitative perspective (performance, memory use).

- Concluding remarks of this thesis and future work are given in *Chapter 7*.

Chapter 2

PROGRAMMING LANGUAGES

This chapter introduces the concept of a programming language, which is used to model and implement software systems. It is possible to divide programming languages into two main categories as shown in *Figure 2.1*. Procedural programming languages consist of functions, where each function comprises programming statements that are executed by calling that function. Object-oriented programming (OOP) languages depend on *objects* that contain data and methods, where data are modified usually via object methods.

This chapter discusses the concepts of procedural programming and object-oriented programming. It discusses major tenets of OOP, namely encapsulation, inheritance, polymorphism, and data binding, and it also compares procedural and object-oriented languages according to structure, efficiency, and ability to debug.

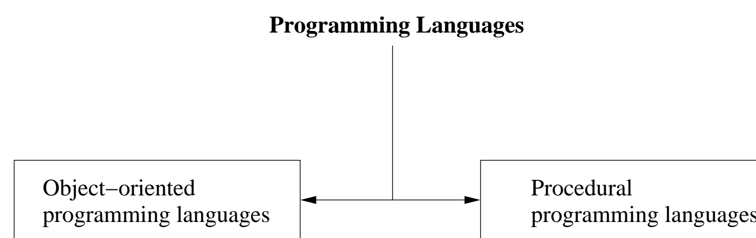

Figure 2.1: Categories of programming languages

2.1 Procedural Programming

Procedural programming is sometimes used as a synonym for imperative programming, i.e., specifying the steps that a program must take to produce the desired result. However,

it can also refer to a programming paradigm based upon the concept of the procedure call. Procedures, also known as routines, subroutines, methods, or functions contain a series of computational steps to be executed. Any given procedure might be called at any point during a program's execution by other procedures or recursively [Ziring, 2007].

Procedural programming is often a better choice than sequential or unstructured programming in situations that involve moderate complexity or require significant ease of maintainability [Wikipedia, 2007e].

Procedural languages have the following advantages:

- The ability to reuse the same code at different places in a program without the need to duplicate.
- They provide an easier way to keep track of program flow than a collection of ”GOTO” or ”JUMP” statements.
- They provide the ability to be strongly modular or structured.

Procedural languages have the following disadvantages:

- It is sometimes difficult to reason about procedural programs.
- Procedural programs are difficult to parallelize.
- Procedural programs are at a lower level of abstraction compared to some other paradigms such as OOP and others, and, because of this, they can be very much less productive than OOP languages, for example.

2.2 Examples of Procedural Programming Languages

C is a low-level block-structured language that provides good support for system programming. C is renowned as the language of the UNIX operating system, but in fact, it is widely used in all kinds of computing platforms [Ziring, 2007; Prechelt, 2003; Wikipedia, 2007b].

The C programming language has fair arithmetic support, simple data structures, subroutines, conventional flow control constructs, naked memory pointers, simple but useful I/O facilities, and a powerful macro preprocessor. Primitive data types supported

in modern standard C are: several sizes of integers, reals, characters, pointers, and arrays [Ziring, 2007; Prechelt, 2003].

Advantages of C are that it is popular, and there are lots of code available written in it. Its disadvantages are that it can be complex to learn and debug pointers, lacks features available in C++, and that it is more difficult to parallelize than FORTRAN [Ziring, 2007; Wikipedia, 2007c].

Fortran 77 is a programming language designed for scientists. Hence it is intended for scientific computing. Fortran 77 has been very popular among scientists, but it lacks features making certain tasks such as the allocation of variable-sized arrays and text handling difficult. The main advantage of Fortran 77 is that it is well-suited for numeric computation. However, it is unpopular outside science circles and lacks features [Ziring, 2007; Wikipedia, 2007c,b].

Fortran 90 and Fortran 95 are improvements over Fortran 77. They provide modern features such as dynamically allocatable arrays. However, they are less commonly used than Fortran 77, even though they are more powerful than Fortran 77 in terms of features.

Fortran 90/Fortran 95 are easier to learn than C for a programmer with Fortran 77 experience, and they provide similar functionality. Fortran 90/Fortran 95 are more powerful than Fortran 77, increasingly common, and more amenable to parallel programming than Fortran 77. The main disadvantage of Fortran 90/Fortran 95 is that they are less common than Fortran 77, and free compilers are not as available as they are for Fortran 77 [Ziring, 2007; Wikipedia, 2007c,b].

2.3 Object Oriented Programming(OOP)

Object-oriented programming (OOP) utilizes *objects* in the design of computer programs. OOP makes heavy use of several software engineering concepts from earlier paradigms including inheritance, modularity, polymorphism, and encapsulation. Even though the idea of OOP originated in the 1960s, OOP was not in common use for mainstream software application development until the 1990s. Today, many popular programming languages have OOP support, such as ActionScript, Java, Javascript, C#, Visual FoxPro, VB.Net, C++, Python, Perl, PHP, Ruby, and Objective-C [Wikipedia, 2007a].

Object-oriented programming may be modeled as a collection of cooperating objects,

as opposed to the traditional view that models a program as a sequential list of instruction executed on some CPU. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Therefore, each object can be seen as an independent entity with a distinct responsibility in the overall picture of things [Wikipedia, 2007a,c].

The OOP approach has a number of advantages:

- **Simplicity**: Software objects tend to model real-world objects, so the complexity of a system is reduced, and the program structure may become clearer than modeling using, for example, the procedural approach.

- **Modularity**: Each object is a separate entity, whose implementation are decoupled from other parts of the system. Therefore, internal modification of an object, for example, does not affect the rest of the system.

- **Modifiability**: This advantage is related to modularity. Making minor changes in the data representation or the methods in an object-oriented program does not have any effect on the rest of the system, since the public interface of the modified class remains unchanged.

- **Extensibility**: Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying existing ones.

- **Maintainability**: Since objects facilitate modularity, objects can be maintained individually, making locating and fixing problems (such as during integration) easier.

- **Reusability**: The same objects can be reused in different programs. If the object can be used in multiple projects, naturally, that object will be debugged and its weaknesses will be eliminated much more efficiently than than designing and implementing different objects for each similar task, thereby increasing the robustness of that object [Wikipedia, 2007a].

Despite these advantages, there is a slight cost in terms of efficiency in using OOP compared to procedural languages. Since objects are normally referenced by pointers and memory allocated dynamically, there is a small space overhead to store the pointers,

and a small speed overhead to find space on the heap at runtime to store objects. For dynamically bound methods, there is an additional time penalty, as the method to be executed is determined at runtime, by searching through the object hierarchy using the class precedence list for a given object. However, the benefits of using objects overweigh the little time cost[Cawsey, 2007].

2.4 Major Tenets of Oriented Programming

OOP is based on a set of software principles.

2.4.1 Encapsulation

In programming, *encapsulation* is the process of combining elements to create a new entity. For example, a procedure is a type of encapsulation because it combines a series of computer instructions under a single name, by which those series of instructions are referred to in other parts of a computer program. Similarly, a complex data type, such as a record or class, relies on encapsulation. Object-oriented programming languages rely heavily on encapsulation to create high-level objects. Encapsulation is closely related to abstraction and information hiding.

2.4.2 Inheritance

The concept of *inheritance* represents the *is-a* relationship between different classes. Inheritance allows a class to have the same behavior as another class (i.e., its parent) and further extend that behavior to fit a more specific need. For example, triangle is a specific type of shape. Therefore, triangle may inherit features from shape and extend the capabilities of shape to fit the needs of the triangle itself.

2.4.3 Polymorphism

The concept of *polymorphism* refers to the ability of processing objects differently depending on their data type. In more specific terms, polymorphism is the ability to redefine methods for derived classes. For example, given a base class *shape*, polymorphism enables the programmer to define different methods that compute the area of a given shape

such as a *triangle*, *rectangle*, or *circle*, where triangles, rectangles, and circles would be subclasses of *shape*. A single interface would exist for computing the area of a given shape (say, *area()*), and no matter what the type of the object, the specific *area()* method defined for that object would be executed, and the correct result would be generated.

2.4.4 Dynamic Binding

The concept of *dynamic binding* is the property that enables a language to determine which specific operation to perform depending on the type of a given object. There may be several different classes of objects that can receive a given message. An expression may denote a general object type that may have more than one possible class, and the particular class of that object can only be determined at runtime. New classes may be created that can receive a particular message, without modifying the code that sends that message.

One important reason for having dynamic binding is that it provides a mechanism for selecting between alternatives at runtime, and this approach is, by its nature, more robust than explicitly selecting objects using conditional statements or pattern matching.

2.5 Examples of OOP languages

The syntax of C++ is similar to C, with various extensions that were added to support classes, inheritance, and other object-oriented features such as multiple inheritance, strong typing, dynamic memory management, templates, polymorphism, exception handling, and overloading. Some newer C++ systems also offer runtime type identification and separate namespaces.

C++ also supports the usual features expected of an application language: a variety of data types including strings, arrays and structures, full I/O facilities, data pointers and type conversion [Ziring, 2007; Wikipedia, 2007b].

C++ has the advantage that is a powerful language, can link C code easily to C++ code. [Ziring, 2007] Moreover, C++ is pragmatic, and, both free and commercial compilers are available for it on many computing platforms. On the downside, C++ has only been standardized, and there are still variations between compilers [Ziring, 2007; Prechelt, 2003; Wikipedia, 2007b].

Java is a full-featured, portable object-oriented language designed at Sun Microsystems. In terms of syntax, Java is fairly similar to C++. Java also supports inheritance, strong type checking, modularity, exception handling, polymorphism, concurrency, dynamic loading of libraries, arrays, string handling, garbage collection, and an extensive standard library. The newest version of the language, Java 1.5, includes generics, annotations, auto-boxing, variable arguments, as well as many additional standard libraries [Ziring, 2007; Prechelt, 2003; Wikipedia, 2007b].

The fundamental structural component of a Java program is still the class. All data and methods in Java are associated with some class. Unlike C++, Java has no true pointers, no true multiple inheritance, no operator overloading, and no macro preprocessor. Instead of multiple inheritance, Java supports the definition and inheritance of multiple stateless ”interfaces.” [Ziring, 2007; Wikipedia, 2007b].

The Java standard library packages include extensive I/O facilities, a comprehensive GUI toolkit, collection classes, date/time support, cryptographic security classes, distributed computation support, network interfaces, CORBA support, XML support, and system interfaces [Ziring, 2007].

Java is typically compiled to platform-independent byte-code. These byte-codes must be interpreted by a Java Virtual Machine (JVM), which may choose to compile the byte-codes further into native machine instructions [Ziring, 2007]

Ada is a block-structured language with many object-oriented programming features. It was intended to support large-scale programming and promote software reliability. Some of Ada's features include nested procedures, nested packages, strong typing, multi-tasking, templates, exception handling, and abstract data types [Ziring, 2007; Wikipedia, 2007b].

Primitive data types supported by Ada include a variety of numeric types, booleans, characters, references, and enumerated symbols. Arrays, records (structures), and strings are Ada's composite types. Since Ada emphasizes program safety, Ada is a strongly typed language [Benchmarks, 2007].

2.6 Procedural Programming vs. Object Oriented Programming

In procedural programming languages, instructions to be executed are grouped into procedures [Subramanian, 2007; Wikipedia, 2007a]. The procedure is the unit of code in procedural programming languages, and the unit of data are the structures. Functions and structures are not connected, and the programmer can only use primitive data types such as integers, characters, floating-point values, and strings [Subramanian, 2007].

In object-oriented programming, objects comprise both data and functions that act upon that data. Therefore, objects are the unit of code in object-oriented programming. Since data can be encapsulated in a high-level structure, the class, a programmer can create new data types unlike as in the procedural programming languages. In addition, the implementation of any given class/object can be reused by other applications, since the class/object is an independent entity with a well-defined border [Subramanian, 2007].

Procedural programming breaks down a task into a collection of data structures and methods. It allows a more direct control over the hardware it is being run upon. Assembly code, one step above the basic machine code that processors interpreter, is procedural. Procedural languages allow the programmer to write more time-efficient code, and thus are more suitable for time-critical applications such as 3D games, statistics engines, and the like.

Object-oriented programming is a more structured and organized programming methodology than procedural programming. Data structures and code are merged to form classes, which contain methods and variables. Well-defined interfaces between classes are constructed, facilitating code reuse and ease of modification and extensibility.

OOP makes it possible to organize a program into classes. With classes, the programmer can create individual objects that execute functions and set variables, thereby creating the behavior of a program [Subramanian, 2007].

In general, object-oriented code will always be less efficient than its procedural equivalent. Indeed, any program that can be written in an object-oriented language can be written in a procedural language to express the same functionality. However, the advantage of object-oriented design is that, as an application's functionality becomes more and more complex, the difficulty of programming does not necessarily increase. However, the inverse is true for procedural programming. In addition, because of the highly

structured approach to design, OOP is an order of magnitude easier to debug than the equivalent debug procedural code [Subramanian, 2007].

Another important difference between object-oriented and non-object-oriented languages is that object-oriented languages need to map the common relational database model to their own class structure to provide database access.

2.7 Summary

This chapter describes procedural and object-oriented programming languages, and it compares these two programming approaches and gives examples of both. The next chapter will describe scripting languages.

Chapter 3

SCRIPTING LANGUAGES

Scripting languages such as Perl, Visual Basic, and UNIX shells represent a very different style of programming than programming languages. Scripting languages assume that there already exists a collection of useful components written in other languages. Scripting languages are not usually intended for writing applications from scratch. They are rather intended for gluing together existing components. For example, Tcl and Visual Basic can be used to arrange collections of user interface controls on the screen, and UNIX shell scripts are used to assemble *filters* in a pipelined fashion, where each stage of the pipeline completes a part of the overall task.

Scripting languages are often used to extend the features of components, but they are rarely used for applications that need complex algorithms and data structures. Such features are usually provided by external components that scripts fuse together. This is why scripting languages are classified as glue languages or system integration languages [Kramer and Magee, 1998; Ziring, 2007].

In order to simplify the task of connecting components, scripting languages tend to be typeless. That is, all things look and behave the same so that they are interchangeable. For example, in Tcl, Visual Basic, or Bash, a variable can hold a string one moment in a given program and an integer later in the code. Code and data are often interchangeable, so that a program can write another program and execute it dynamically. Scripting languages are often string-oriented, since this provides a uniform representation for many different types of data [Kramer and Magee, 1998; Ziring, 2007].

Scripting languages can be divided into two main categories as shown in *Figure 3.1*.

Similar to regular procedural programming languages, procedural scripting languages make it possible to write functions in the traditional sense. Object-oriented scripting languages, on the other hand, support object-oriented principles such as classes in varying degrees.

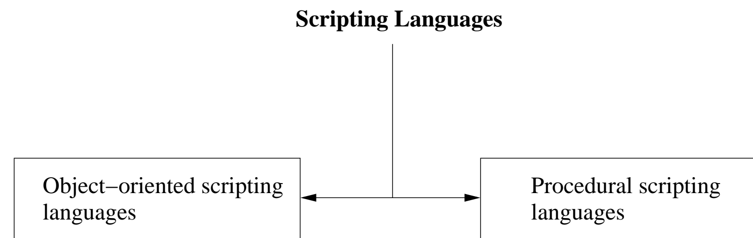

Figure 3.1: Categories of scripting languages

3.1 Procedural Scripting Languages

Procedural scripting languages consist of blocks that contain statements that are executed via function calls. Therefore, functions are the unit of code of this type of languages. As in regular programming languages, functions in procedural scripting languages contain a series of computational steps to be carried out. Any given function might be called at any point during a program's execution, including recursive calls.

3.2 Object-oriented Scripting Languages

Object-oriented programming has become especially popular in scripting languages in recent years. Many scripting languages now support object-oriented constructs and facilities such as classes, inheritance, and polymorphism. The class is the unit of code of this type of scripting languages, Python and Javascript are scripting languages built on OOP principles. In addition, languages such as Perl and PHP have been adding object oriented features since Perl 5 and PHP 4 [Ousterhout, 1998; Bezroukov, 2007; Ziring, 2007; Wikipedia, 2007c].

3.3 Characteristics of Scripting Languages

There are a few fundamental characteristics that define the general nature of scripting languages, namely loose typing and interpretation [Ousterhout, 1998].

3.3.1 Loose Typing

Strong typing means that type rules are enforced without exception. Types of all variables must be known at compile time, i.e. types are statically bound. In the case of variables that can store values of multiple types, incorrect type usage is detected at runtime. In scripting languages, where there are no type rules applied, a variable can hold multiple values of incompatible types throughout the life cycle of a program. A variable, for example, can hold an integer and later a string. However, in strongly typed languages, a general type may be used to contain values that belong to subtypes of that general type. For example, a shape object can point to a triangle at one time and a circle at another time.

Scripting languages represent all types of data the same way, for example, using strings. In a way, it can be said everything is a string. In such languages, meaning is derived at the time of usage. For example, if two strings are being added using the addition operation, then the values are treated as numbers. If two strings are being concatenated, then strings are kept as is, without conversion. The advantage of this approach is that reuse of existing variables is made easier, since the programmer need not worry about types of variables or the declaration of variables. A variable can be declared and used at the same time, without any prior declaration of type as in compiled languages. On the contrary, if the type of a variable needs to be known, the programmer becomes helpless, since no facilities exist for determining the type of the data contained in a given variable in a scripting language.

In scripting languages, the interpreter internally keeps track of the type of the data assigned to a variable, and it can alter the data type of the variable as new data is assigned to the variable. Due to automatic type conversion in scripting languages, the complexity associated with typed language tends to be hidden from the programmer. Hence the programmer can effectively treat the language as typeless.

A type-free language makes it easier to hook components possibly written in different

languages than a language that enforces any sort of typing, especially strong typing. In typeless scripting, there are no a priori restrictions on how data can be used, and all components and values are represented in a uniform fashion. Thus any component or value can be used in any situation. That is, components designed for one purpose can be used for totally different purposes never intended by the designer at the outset. For example, in UNIX shells, all filter programs read a stream of bytes from an input device and write a series of bytes to an output device. Any two programs can be connected together by attaching the output of one program to the input of the other using *pipes*. For example, the following shell command creates three filters to count the number of lines containing one or more occurrences of the word `scripting` in it (Example modified from [Ousterhout, 1998]):

```
cat report.txt | grep "scripting" | wc
```

The `cat` program reads the text in file `report.txt` in the current working directory and prints to the default *standard output* device, which is the screen. The `grep` program reads its input from the default *standard input* device, which is the keyboard, and prints out all lines in `report.txt` containing `scripting` and writes its output to the `standard output` device. The `wc` program counts the number of lines of the stream of bytes in its input stream. By connecting these three programs using pipes, the input of one program is redirected to the output of the next program, thereby connecting the programs together, something that would be unnecessarily difficult to do for any of the three external programs used here as example. Providing the ability to connect any program to any other program instead of providing the basic capability of redirection from the standard input and output devices would have been costly in terms of standardization and programming. Therefore, programs written in this fashion, however complex, can be externally linked by the redirection mechanism provided by the shell scripting language. This way, it is possible to link any series of external programs (such as `wc`, `grep`, or `cat` in this case) to implement the desired functionality, since there is no limitation to which programs connected to which others.

Conversely, the strongly typed nature of programming languages discourages reuse. Typing encourages programmers to create a variety of incompatible interfaces. Each in-

terface requires data of specific types, and the compiler prevents any other types of objects from being used with the defined interface, even when that would be useful. Therefore, in order to use a new object with an existing interface, conversion code must be written to translate between the type of the data and the type expected by the interface. This, in turn, requires recompiling part or all of the application, and doing so is impossible in the common case where the application is distributed in binary form and not in source code form.

To demonstrate the advantages of a type-free language, consider the following Tcl command (Example reproduced from [Ousterhout, 1998]):

```
button .b -text Hello! -font {Times 16} -command {puts hello}
```

This command creates a new button control that displays a text string in a 16-point Times font and prints a short message when the user clicks on that button. It mixes six different types of things in a single statement: a command name (`button`), a button control (`.b`), property names (`-text`, `-font`, and `-command`), simple strings (`Hello!` and `hello`), a font name (`Times 16`) that includes a typeface name (Times) and a size in points (16), and a Tcl script (puts hello). Tcl represents all of these things uniformly with strings. In this example, the properties may be specified in any order and unspecified properties are given default values. In this particular example, there are more than 20 properties have been left unspecified.

The implementation of the same example requires 7 lines of code in two methods when implemented in Java. Using C++and Microsoft Foundation Classes (MFC), the same implementation requires about 25 lines of code, divided in three procedures. For example, setting the font requires several lines of code in MFC: (Example reproduced from [Ousterhout, 1998]):

```
CFont *fontPtr=new CFont();
fontPtr->CreateFont( 16, 0, 0,0,700, 0, 0, 0, ANSI_CHARSET,
           OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
           DEFAULT_PITCH|FF_DONTCARE, "Times New Roman" );
buttonPtr->SetFont( fontPtr );
```

Strong typing affects how much of this code has been written. To set the font of a button, the `SetFont` method must be invoked, but this method must be passed a pointer to a `CFont` object that must be created and initialized. In order to initialize the `CFont` object, its `CreateFont` method must be invoked, but `CreateFont` has a rigid API that requires 14 different arguments to be specified. In Tcl, the essential characteristics of the font can be used immediately with no declarations or conversions. Furthermore, Tcl allows the behavior for the button to be included directly in the command that creates the button. However, C++and Java require it to be placed in a separately declared method.

Since there are no types to check, it might seem that the type-free aspect of scripting languages could allow errors to remain undetected. In practice, however, scripting languages are as safe as programming languages. For example, an error will occur if the font size specified for the button example above is a non-integer. The difference is that scripting languages do their error checking at the last possible moment, that is, exactly when a value is used. Strong typing allows errors to be detected at compile-time, so the cost of runtime checks is avoided [Ousterhout, 1998].

3.3.2 Interpretation

A key difference between scripting languages and programming languages is that scripting languages are usually interpreted whereas programming languages are usually compiled. Some languages are both compiled and interpreted, such as Java. Lisp is similar to Java, however, in Lisp, the source code can be directly interpreted in order to be executed [Steele, 1990].

Interpreted languages make rapid development possible by eliminating compilation. Some interpreters also make applications more flexible by allowing expert users to program the applications as they use that application. For example, many synthesis and analysis tools for integrated circuits include a Tcl interpreter. Users of the programs can write Tcl scripts to specify their designs and control the operation of the tools.

Moreover, interpreters also allow powerful effects to be achieved by generating code on the fly. For example, a Tcl-based web browser can parse a webpage by translating the HTML for the page into a Tcl script using a few regular expression substitutions. It then executes the Tcl script to render the page on the screen.

In general, scripting languages are less efficient than programming languages, in part because they use interpreters instead of compilers but also because their basic components are chosen for power and ease of use rather than an efficient mapping onto the underlying hardware. For example, scripting languages often use variable-length strings in situations where a programming language would use a binary value that fits in a single machine word, and scripting languages often use hash tables where programming languages use indexed arrays.

Fortunately, the performance of a scripting language is usually not a major issue. Applications for scripting languages are generally smaller than applications for programming languages, and the performance of a scripting application tends to be dominated by the performance of the components it glues together, where these components are implemented in a programming language.

Scripting languages work at a higher level of abstraction than programming languages, since a single scripting statement does more work on average than a regular programming language statement. A typical statement in a scripting language executes hundreds or thousands of machine instructions. A typical statement in a programming language executes about five machine instructions.

Part of this difference is because scripting languages use interpreters, which are less efficient than the compiled code for programming languages. But much of the difference is because the primitive operations in scripting languages have greater functionality. For example, in Perl it is about as easy to invoke a regular expression substitution as it is to invoke an integer addition. In Tcl, a variable can have *traces* associated with it so that setting the variable causes side effects. For example, a trace might be used to keep the variable's value updated continuously on the screen.

In [Ousterhout, 1998], it is shown that, in every comparison between scripting and regular programming, the scripting version required less code and development time than the programming version. The difference varies from a factor of 5 to a factor of 10. The benefits of scripting also depend on the application.

3.4 Examples of Scripting Languages

Javascript is a loosely typed scripting language with both object-oriented and block-structured features. The syntax of Javascript is similar to that of C or Java, but it is simpler and not as rich either. Primitive data types include integers, reals, strings, and associative arrays. Since Javascript is a loosely typed language, any variable can contain data of any type, and conversions between types is mostly automatic. The language definition includes extensive facilities for controlling and manipulating parts of web pages, especially HTML forms [Ousterhout, 1998],

Perl is an interpreted scripting language with wide support for data manipulation and rapid application development. Perl is block-structured, but it also supports object-oriented programming. Perl does not have records or structs as in C. Instead, it provides associative arrays (hashes) to serve the same and all similar purposes. Similarly, Perl supports object-oriented programming, but does not enforce an object storage format [Bezroukov, 2007].

Perl has gained a great deal of popularity and has grown as a language since the advent of the Word Wide Web, because it is an effective language in which to write backend common gateway interface (CGI) scripts. Perl is a very suitable language for such tasks, because it has extensive string manipulation facilities, file I/O, control structure, and a database interface [Bezroukov, 2007].

PHP is an interpreted server-side scripting language for webservers. It was designed to support simple, fast server-side web development. The syntax of PHP is quite similar to that of Perl, with some aspects of Bourne shell, Javascript, and C [Bezroukov, 2007; Prechelt, 2003].

Variables in PHP are weakly typed, and the language does not support strong typing. PHP supports a modest complement of primitive data types such as integers, floats, and strings. It also supports heterogeneous multi-dimensional associative arrays. PHP offers some object-oriented functionality, allowing the programmer to define classes with member variables and methods, and simple inheritance. The language includes an extensive set of operators and built-in functions for manipulating strings, numbers, and arrays.

Python is an interpreted, object-oriented language. It is intended to be highly effective and extensible. The syntax of Python is statement-oriented. Block structure is specified

with indentation. Python supports a good set of primitive and composite data types: integers, floats, complex numbers, strings, lists, and associative arrays called dictionaries. Data values are typed, but strong type checking is not enforced. Like most scripting interpreters, Python does have the ability to execute a string as code. Python also has exceptional-handling constructs similar to those in Modula-3. Classes in Python can use single and multiple inheritances.

Python is supported by a feature-rich standard library; it includes extensive string manipulation, I/O, parsing, date handling, low-level networking services, high-level protocol and data format handling, image I/O, and a variety of operating system-specific services.

Python is normally interpreted, but the Python interpreter can compile scripts and modules into portable binary form and execute this form instead. Several Unix-specific and portable graphics and GUI libraries also exist for Python [Bezroukov, 2007; Prechelt, 2003].

Visual Basic scripting (VBScript) edition is a subset dialect of Visual Basic. It is an interpreted, procedural language intended for creating application extension scripts and for adding interactivity to webpages. VBScript's syntax is similar to that of Visual Basic. Statements are bounded by newlines, and normal Basic keywords are used for control structures and code modularity [Bezroukov, 2007; Prechelt, 2003].

VBScript supports a modest set of data types such as strings, dates, Booleans, arrays, and object references. Variables in VBScript are typed, but the interpreter does not enforce strong typing. VBScript supports subroutines and functions, and it can interact with objects provided by its environment. VBScript does not support the definition of new object classes, and it also does not support overloading or polymorphism as in object-oriented languages. Latest versions of VBScript support a dictionary object, an associative array for storing string data.

3.5 Programming Languages vs. Scripting languages

There are currently two high-level programming techniques that are used for implementing software systems. Conversely, low-level assembly programming has a much more restricted area of application. Most current general-purpose programming languages,

with the popular exception of Java, require programmers to manage memory manually instead of letting the execution environment do so by garbage collection. So the programmer needs to explicitly declare pointer variables, allocate memory dynamically, and deallocate unused memory in order not to cause buffer overruns. The programmer has to manage memory in such a way that no memory leaks occur. For example, systems with long life cycle, such as servers, cannot tolerate memory leaks [Liang, 2005].

Scripting languages work at a higher level than compiled languages. There is no need for the programmer to manage memory at runtime or statically using the features of the language or by introducing extra logic or by taking advantage of programming techniques to keep track of memory use. Moreover, scripting languages are typeless, which means that variables can be declared and assigned values of different types without worrying about whether the types match. That is, at one instant, a variable in a script can hold a string. At another time, it can hold an integer or a character.

Scripting languages are not compiled but interpreted. So there is no need for a separate compilation process. Since scripting languages are typeless, there is no need to check whether all variable assignments are proper or not. That is, the language can provide data structures such as lists and associative arrays that can keep any type of data, usually represented as string.

Due to the typeless nature of scripting languages, it is easier to make modifications and test new versions of a script compared to a regular compiled language, since all modifications can be tested right after they are made. In addition, a single interpreter process may be able to run multiple scripts of the same language, thereby reducing memory load on a given machine [Liang, 2005]. Java is an interesting exception worthy of mention here. Java is a compiled language. However, to execute a Java program, an interpreter, a Java virtual machine, is needed. Since Java is a compiled language, modifications to a Java program cannot be readily tested as in the case of a scripting language, such as Perl or Bash. Instead, the modified program must be recompiled, before being interpreted within a virtual machine. Unlike many programming languages, memory management in Java is done automatically by a garbage collector. The existence of garbage collection then suggests that the language itself need not explicitly support pointer types, thereby making Java a higher-level language compared to C++. Another notable example is Lisp,

which is interpreted but can be bytecode compiled as Java. In Lisp, there is also no need for explicit pointers [Steele, 1990].

Scripting languages are designed for gluing applications together. They provide a higher level of programming than either assembly or compiled programming languages. Scripting languages enforce much weaker typing than programming languages. Scripting languages sacrifice execution speed for development speed [Ziring, 2007].

Strong typing in regular programming languages makes the management of complex algorithms and data structures easier than typeless scripting languages. A system developed using a programming programming language can run 10 to 20 times faster than a scripting equivalent, since statically typed languages do less number of runtime type checks [Mischook, 2007]. Since computers nowadays are so fast and scripting languages are so efficient, for most business applications, the speed difference that once existed between compiled programming languages and scripting languages no longer exists [Mischook, 2007].

Scripting languages are usually better for implementing business logic since they are interpreted and provide great flexibility. However, we need a methodology to compare the efficiency of using a combination of scripting languages and programming language in order to choose the best approach for combining both programming approaches. One method for carrying out this comparison is to use benchmark problems, which is the subject of this thesis.

A scripting language is not a replacement for a programming language or vice versa. Each is suited well to different sets of tasks. For gluing and system integration, applications can be developed 5 to 10 times faster with a scripting language. Conversely, programming languages require large amounts of code to connect the pieces of a system. This connection of pieces, on the other hand, can be done directly with a scripting language whenever technically possible. Usually, there is no need for large or complicated amount of coding for calling external programs from a script. For example, in Bash, a programmer can call external executables written in lower level languages such as C [Cawsey, 2007].

Finally, considering the advantages and disadvantages of compiled languages and interpreted languages, it makes sense to write today's software systems using several

languages, selecting the best tool for each smaller part of the overall task [Deitel, 2003]. There is a set of issues to consider in order to decide whether to use a scripting language or a regular programming language for a particular software task [Ousterhout, 1998]:

- Whether the main task is to connect together pre-existing components
- Whether the application needs to manipulate different types of data
- Whether the application needs to have a graphical user interface
- Whether the application needs to do lots of string manipulation
- Whether the application's functionality needs to evolve rapidly over time
- Whether the application needs to be extensible

If the answer is in the affirmative to all of these questions, then it means that a scripting language will likely be the better choice for implementing that application. If an application implements complex algorithms and data structures, needs to manipulate large amounts of data, and the functionality of that application are well-defined and does not need to change rapidly, then it is likely better to use a regular programming language for implementation.

Scripting and programming are complementary activities, when used together, they can help create a powerful programming environment. Regular programming languages can be used to create components requiring complex data structures and functionality, and these components can then be assembled using scripting languages. For example, Visual Basic is attractive for Windows programmers, because it is possible to write ActiveX components in C, and less sophisticated programmers can then use these components in their Visual Basic applications. In UNIX or GNU/Linux environments, it is easy to write shell scripts in Bash that call applications written in C. Another interpreted language, Tcl, provides programmers the ability to extend the language by writing C code that implements new commands [Mischook, 2007].

3.6 Summary

In this chapter, scripting languages, types of scripting languages, and main characteristics of scripting languages are discussed. This chapter cites pertinent examples of scripting languages, and it also compares scripting languages with programming languages. The next chapter introduces the special benchmark algorithm designed for this thesis in order to compare the techniques of programming with embedded scripting and scripting with embedded programming.

Chapter 4

DESIGN OF A BENCHMARK FOR INTERACTIVE WEBPAGE DESIGN

In general, benchmarks are designed to emulate a particular type of workload on a given system so that the performance of that system can be compared to other similar systems, whether it is software or hardware. For example, two CPUs with different architecture can be compared based on floating-point performance. Two separate compiler implementations of C++ can be compared to find out how fast and efficiently executable code is generated. [Wikipedia, 2007d]

There are many benchmark problems that were designed for comparing single programming languages and single scripting languages for based on features such as CPU usage and memory usage. No benchmark algorithms exist, on the other hand, for comparing a composition of programming languages and scripting languages. In this chapter, the design and the algorithm for a benchmark for comparing the composition of programming and scripting languages is discussed. This algorithm is used for comparing the techniques of combining programming and scripting languages for interactive webpage design. *Chapter 5* and *Chapter 6* discuss the implementation of the two approaches in detail.

4.1 Conventional Benchmark Problems

There are many different types of conventional benchmark problems that perform various measurements. The following subsections list three benchmarks that are pertinent to this thesis.

4.1.1 Sum-File Benchmark

The sum-file benchmark provides a measure of line-oriented I/O and string conversion [Benchmarks, 2007]. A program implementing this benchmark has to read (N=21,000) lines of integers from standard input, parse, and sum these integers one line at a time. Finally, the implementation should print out the sum of those integers. The implementation should only use built-in line-oriented I/O functions instead of any custom-made code for the same purpose. No line should exceed 128 characters, including the newline, and Reading one line at a time, the programs should run in constant space [Benchmarks, 2007].

Table 4.1: Performance of the *Sum-File* benchmark algorithm (Source [Benchmarks, 2007; Shaw et al., 1981])

Languages	CPU time (sec)	Memory use (KB)	Code size (GZip bytes)
C	5.21	324	180
Fortran	43.27	468	115
C++	8.83	824	141
Java	7.40	11520	249
ADA	9.85	460	224
Javascript	113.10	20400	58
PHP	41.99	5456	130
Python	17.03	2376	61
Perl	14.88	1448	68

Table 4.1 shows implementations of the sum-file benchmark algorithm in a number of regular programming languages and scripting languages:

4.1.2 Binary-trees Benchmark

In this benchmark, the implemented program must allocate and deallocate a large number of binary trees to a maximum depth of 16 [Benchmarks, 2007].The implementation involves intense traversal of nodes, allocation, and deallocation of nodes. In addition,

at each node, the benchmark computes a checksum of the node and possibly deallocates that node during traversal. The benchmark also allocates a binary tree that is supposed to live on while others are allocated and deallocated.

Table 4.2: Performance of the *Binary-trees* benchmark algorithm (Source: [Benchmarks, 2007; Shaw et al., 1981])

Languages	CPU time (sec)	Memory use (KB)	Code size (GZip bytes)
C	3.77	4528	690
Fortran	43.92	10700	810
C++	4.45	6988	525
Java	6.50	23412	587
Ada	5.14	6592	939
Javascript	140.52	136884	451
PHP	173.25	81648	477
Python	92.98	16044	402
Perl	243.00	37568	475

Table 4.2 shows the performance characteristics of several implementations:

4.1.3 Partial-Sums Benchmark

The partial-sums benchmark computes partial sums (N=25000) of a number of series using the power, square-root, and trigonometric functions. Programs implementing this benchmark need to use naïve iterative double-precision algorithms to calculate the following series using looping:

- $\sum \left(\frac{2}{3}\right)^k$ for $k = 0, ...$: In this case, the power function must be used to compute this series.
- $\sum k^{-0.5}$ for $k = 1, ...$: In this case, power or square-root function must be used to compute this series.
- $\sum \frac{1}{k(k+1)}$
- $\sum \frac{1}{k^2(\sin k)^2}$
- $\sum \frac{1}{k^3(\cos k)^2}$
- $\sum \frac{1}{k}$

- $\sum \frac{1}{k^2}$
- $\sum \frac{-1^{k+1}}{k}$
- $\sum \frac{-1^{k+1}}{2k-1}$

According to [Benchmarks, 2007], implementations of this algorithm in several languages have the performance characteristics given in *Table 4.3*. Most programming languages are efficient in CPU time and memory use. However, these languages need to perform type checking, since each variable must have a type. *Table 4.3* also shows that regular programming languages (C, Fortran, C++, Java, and Ada) have a relatively lower CPU time compared to scripting languages (Javascript, PHP, Python, and Perl).

Table 4.3: Implementation of the Partial-Sum benchmark algorithm (Values reproduced from [Benchmarks, 2007; Shaw et al., 1981])

Languages	CPU time (sec)	Memory use (KB)	Code size (Gzip bytes)
C	4.16	444	383
Fortran	4.10	504	509
C++	3.21	948	693
Java	11.88	9308	454
Ada	6.82	440	648
Javascript	58.09	66516	364
PHP	17.81	5504	351
Python	27.69	2396	410
Perl	16.57	1456	370

However, in terms of memory use, the performance division is not as clear as it is in the case of CPU time. The memory use of compiled languages (C, Fortran, C++, and Ada) are relatively much smaller than that of scripting languages listed in the table. However, Java has a much higher memory use value than all scripting languages, except Javascript. This is likely due to the fact that Java is interpreted and is an object-oriented language. Similarly, Javascript has a relatively very large memory use value due to being an interpreted object-oriented language.

Interestingly, code sizes in *Table 4.3* do not differ by large amounts due to the simplicity of the partial-sum benchmark. For example, both C and Perl implementations needed almost exact number of lines of code to implement the partial-sum benchmark.

4.2 Benchmark Algorithm Design

Interactive webpages enhance the interaction between users and webpages, receiving data from users, performing operations on users data, and interacting with databases. Such sites interact with the user usually through either a text-based or graphical user interface (GUI).

Interactive webpage design is used in many fields such as banking, e-learning, and e-commerce. To mimic the interactions and computations of an interactive webpage in order to design a benchmark, the banking system was chosen.

Banking is a domain of interactive webpage design, since it involves deposits, withdrawals, and other operations that require interaction between clients and the bank system. Calculating the amount of interest for a company that has many customers is one such operation performed by a banking system. The algorithm designed models this operation. The algorithm combines business logic and programming logic.

- *Business logic:* all operations for interacting with users, using interfaces, buttons, text fields, changing colors, images, and validating user's parameters.
- *Programming logic:* implementing algorithms, performing processing, interactive with databases, searching, calling functions.

The block diagram in *Figure 4.1* illustrates the benchmark algorithm designed. In this diagram,

- Block 1 represents the interaction with users for collecting information –This is the business logic part of the benchmark.
- Block 2 represents the calculations of the amount of interest for a company – This is the programming logic part of the benchmark.
- Block 3 represents the saving of the amount of interest to secondary storage, a database file and retrieving this value –Again, this is the programming logic of the benchmark.
- Block 4 is about printing the result of the interest calculations to user by drawing a table that contains the results –Again, this is part of the business logic of the benchmark.

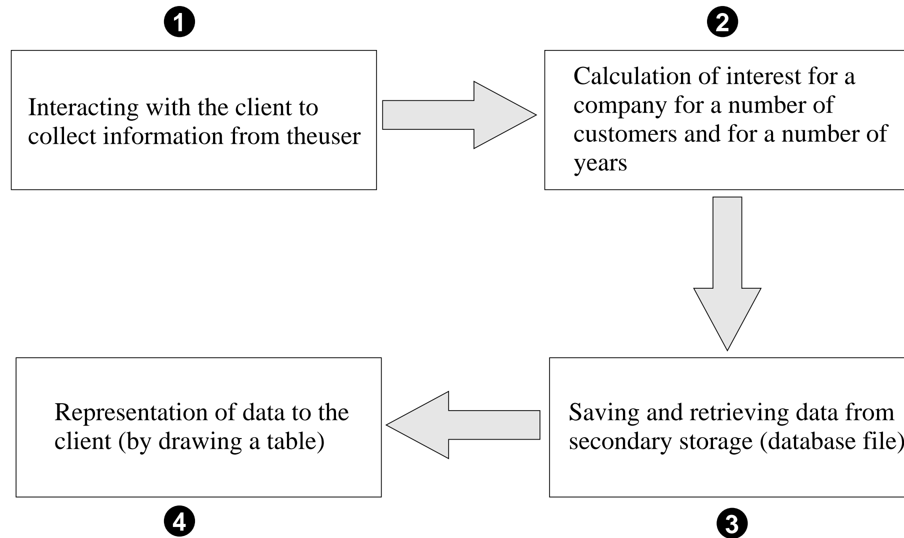

Figure 4.1: Block diagram of the benchmark algorithm designed

The algorithm contains the following steps:

1. Get parameters from the user. Four parameters are required. Parameter 1 is a string, and the remaining three are numerical values.

2. Check the validity of the parameters entered by the user and check the type of the variables to confirm that parameters match the expected types needed by the algorithm.

3. Perform the calculations on the operands. Actually, any conventional benchmark can be used at this stage [Bodoff, 2007]. For this thesis, the problem of calculating the amount of interest for a company after a period of time has been designed as benchmark. In this step, the amount of interest for a company that is assumed to have many customers will be calculated according to the following formula:

   Amount of interest = $P(1+I)^N$ Where P is the deposit amount, I is the interest rate for year (constant =0.05%), N is the number of years.

   The algorithm for this process is as follows:

   ```
   FOR num=1 TO num-client
      Amount of interest = P(1+I)^N
   END
   ```

   In the algorithm above, the user will enter the name of the company, number of the

customers, number of years, and the amount of deposit.

4. Save permanently onto secondary storage. This step saves the company name, number of clients of the company, amount of deposit, number of years, and total amount of interest in a database file into two database tables, *company table* and *customer table*.

5. Retrieve the data that is saved in *Step 4* using the proper SQL statements, which select all information saved by *Step 4* from the two tables.

6. Print the results to the user. After selecting all information from the two tables that contain all data about the amount of interest of companies, this step draws a table on the screen and prints the data retrieved in this table.

The flowchart of the algorithm is shown in *Figure 4.2*. *Step 1*, *Step 2*, and *Step 6* are the business logic part of the algorithm, in particular, receiving data from users, printing data, validating parameters supplied, and customizing of the user interfaces.

Step 1 receives the parameters from the user. A graphical user interface is used that contains four text fields and a button for sending the data to the program.

Step 2 validates the parameters entered by the user in *Step 1*. The benchmark program should check the types of the parameters at this stage.

Step 6 prints the results of interest computations to the user by drawing a table on screen and printing specific data values in the table.

Step 3, *Step 4*, and *Step 5* are the programming logic part of the algorithm, which involves performing computations on operands, calling subroutines, and interacting with external storage (files and databases).

Step 3 can compute any of the 19 benchmark problems [Benchmarks, 2007]. In the case of this thesis, the problem of calculating the amount of interest in a company after a period of time has been used. The algorithm of the benchmark has been provided above. The algorithm involves computing values for a number of customer clients using the power function.

Step 4 stores the results on external storage. This step saves the company name, the customer's ID, customer's deposit among, and number of years in a database file for permanent storage. There are two tables for saving data. The company table keeps the

company name and number of clients. The customer table stores the company name of the customer, amount of deposit, number of years, and the total amount of interest. The company name field of the customer table will be a foreign key to company name field of the company table.

Step 5 retrieves the results previously saved in the tables, draws a table on the user's screen, and prints the all rows of databases on the user screen.

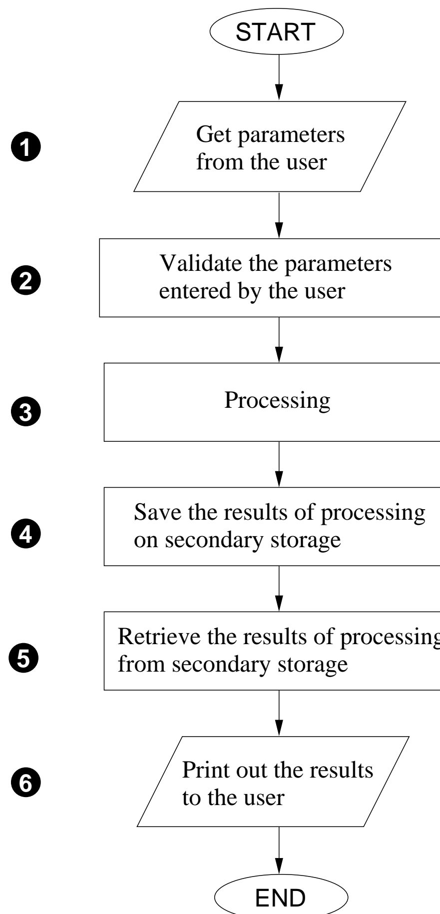

Figure 4.2: Flowchart of the processing done by the special benchmark algorithm designed and implemented for this thesis

4.3 Summary

This chapter describes the special benchmark algorithm designed for this thesis in order to compare the two programming approaches for combining programming logic and business logic for the design of interactive webpages. Combining programming languages and programming languages is needed to get benefits from programming lan-

guages in programming logic and scripting languages in business logic.

In this chapter, a number of benchmark problems for comparing programming languages and scripting languages have been described. This chapter emphasizes that, in general, regular programming is better for programming logic (implementing algorithms, work with databases, execute operations that need high processing speed) and that scripting languages are better for business logic (user interfaces, manipulating strings, validating variables) [Prechelt, 2000].

There are two approaches possible for combining programming and scripting languages. Programming languages with embedded scripting languages and scripting languages with embedded programming languages. The next two chapters describe the implementation of this benchmark algorithm in order to show which approach is better.

Chapter 5

PROGRAMMING LANGUAGES WITH EMBEDDED SCRIPTS: Implementation of the Benchmark Problem

Programming languages with embedded scripts combine both programming and scripting. This means that the program statements contain scripting statements, usually HTML tags, to achieve better performance and to draw benefits from both regular programming and scripting approaches.

Java is an object-oriented programming language that is platform-independent. Java is multithreaded. That is, several operations can be executed concurrently without deadlocking. A program written in Java runs on any computing platform whether that platform is GNU/Linux, Windows, UNIX, or Mac OS, given that a Java interpreter (Java Virtual Machine) exists for that platform [Wikipedia, 2007a].

In this thesis, an application for this concept has been implemented using a Java *servlet*. A servlet is a Java class that runs on a webserver and allows the embedding of HTML tags.

In this chapter, the Java servlet technology is discussed. A servlet is a Java class that runs on the server. Servlets have many advantages over other technologies. A Java servlet, by definition, is platform independent just as a Java program is. Similarly, it

is object-oriented and multithreaded.

This chapter also discusses the life cycle of servlets. It compares servlets and other competing technologies, describes Java servlets and database connectivity. The benchmark problem has been implemented on on a 667 Mhz Pentium III machine with 256 MB of RAM and 20 GB hard disk, running the Microsoft Windows XP (Service Pack 1) operating system.

5.1 Java Servlets

The Java Servlet API allows a programmer to add dynamic content to a webserver using the Java platform. The generated content is commonly HTML, but it may be other type of data such as XML. Servlets are the Java counterpart to non-Java dynamic web content technologies such as CGI and ASP. Servlets can maintain state across many server transactions using HTTP cookies, session variables or URL rewriting [Yourdon, 1996].

A servlet is a dynamically live object that receives a request (`ServletRequest`) and generates a response (`ServletResponse`) based on that request. The servlet API defines HTTP subclasses of the generic servlet request (`HttpServletRequest`) and response (`HttpServletResponse`), besides a session object (`HttpSession`) that tracks multiple requests and responses between the webserver and a client [Layon, 2004; Wikipedia, 2007c; Bodoff, 2007].

5.2 Advantages of Servlets over CGI

Compared to traditional CGI and similar technologies, Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper [Wikipedia, 2007c; Bodoff, 2007].

5.2.1 Efficiency

With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process.

Similarly, in traditional CGI, if there are N simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are N threads but only a single copy of the servlet class. Servlets also have more alternatives than do regular CGI programs for optimizations such as caching previous computations, keeping database connections open, and the like.

Besides the convenience of reusing one's existing knowledge of Java, servlets provide extensive functionality for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

Java servlets allow certain types of operations that are difficult or impossible with the regular CGI approach. For example, servlets can talk directly to the webserver, but regular CGI programs cannot. This ability simplifies operations that need to look up images and other data stored on the server. Servlets can also share data among each other, making database connection pools relatively easy to implement. They can also maintain state between requests, simplifying operations such as session tracking and caching of previous computations.

Servlets are written in Java, hence follow a standard API. Therefore, servlets written one type of server, say, I-Planet Enterprise Server, can run virtually unchanged on another server running Apache, Microsoft IIS, or Web Star. Servlets are supported directly or via a plugin on almost every major webserver platform nowadays.

Given that a webserver already exists, adding servlet support to it is either free or cheap. For example, if the Apache is the choice for webserver, not only that it is free of charge, extending the server with servlet support can be done free of charge as well [Wikipedia, 2007c; Bodoff, 2007].

5.3 Life Cycle of a Servlet

Each servlet has the life cycle shown in *Figure 5.1*. First, a webserver loads and initializes the servlet. Second, the servlet handles zero or more client requests, remaining alive in memory. Third, the webserver removes the servlet by unloading it from memory – Some servers may remove servlets only when they shut down. The following subsections describe this life cycle in more detail [Bodoff, 2007].

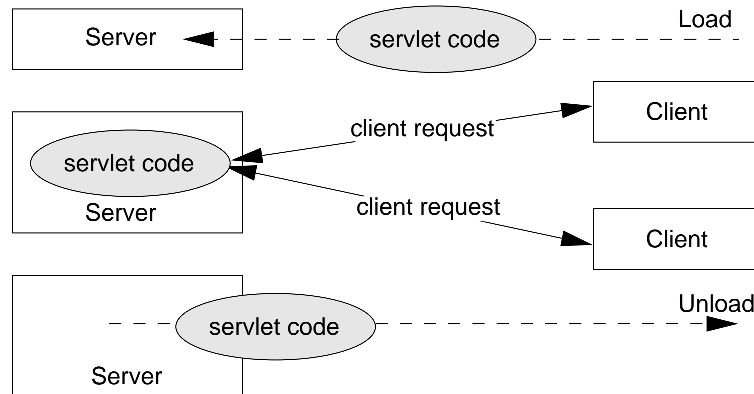

Figure 5.1: Life cycle of a servlet from loading until unloading by the server (reproduced from [Bodoff, 2007])

5.3.1 Servlet Initialization

When a webserver loads a servlet, the server runs the servlet's `init()` method. Initialization completes before client requests are handled and before the servlet is destroyed [Wikipedia, 2007c; Bodoff, 2007].

Even though most servlets are run on multithreaded webservers, servlets have no concurrency issues during servlet initialization. The webserver calls the `init()` method once on loading the servlet, and it does not call the `init()` method again unless the server is reloading that servlet. A server cannot reload a servlet until after destroying that servlet by calling the `destroy()` method.

The `init()` method provided by the `HttpServlet` class initializes the servlet and logs its initialization. Following is an example of the structure of an `init()` method:

```
public class BookDBServlet ... {
   private BookstoreDB books;
   public void init(ServletConfig config) throws ServletException {
      // Store the ServletConfig object and log the initialization
      super.init( config );
      // Load the database to prepare for requests
      books = new BookstoreDB();
   }
   ...
}
```

The `init()` method calls the `super.init()` method to manage the `ServletConfig` object and sets a private field. If the `BookDBServlet` used an actual database, instead of simulating one with an object, the `init()` method would have been more complex. Following is the general structure of the `init()` for this case:

```
public class BookDBServlet ... {
   public void init( ServletConfig config ) throws ServletException {
      // Store the ServletConfig object and log the initialization
      super.init( config );
      // Open a database connection to prepare for requests
      try {
         databaseUrl = getInitParameter("databaseUrl");
         ... // Get user and password parameters the same way
         connection = DriverManager.getConnection( databaseUrl,
                                                   user, password );
      }
      catch( Exception e ) {
         throw new UnavailableException( this,
                "Could not open a connection to the database");
      }
   }
   ...
}
```

5.3.2 Interacting with Clients

An HTTP servlet handles client requests through its `service()` method. The `service()` method responds to standard HTTP client requests by dispatching each request to a method designed to handle that particular request.

The methods in the `HttpServlet` class that handle client requests need two arguments, (1) an `HttpServletRequest` object, which encapsulates the data from the client, and (2) an `HttpServletResponse` object, which contains the response to the client.

An `HttpServletRequest` object provides access to HTTP header data, such as cookies found in the request and the HTTP method with which the request was made.

The HttpServletRequest object also allows the programmer to get access to the arguments sent by the client as part of the request.

The getParameter() method returns the value of a parameter whose name is provided as argument. If the parameter could have more than one value, then the

getParameterValues() method is used. The getParameterValues() method returns an array of values for the specified parameter. The getParameterNames() returns the names of all the parameters sent from the client.

For HTTP GET requests, the getQueryString() method returns a string of the raw data received from the client. This data must be parsed manually to obtain the parameters and their values.

An HttpServletResponse object is used to return data to the client. When the particular output stream associated with the HttpServletResponse object is closed, then the server knows that the response is complete.

To handle HTTP requests in a servlet, HttpServlet class must be extended and the servlet methods that handle the HTTP requests must be overridden. The methods that handle these requests are doGet() and doPost(). That is, handling GET requests involves the overriding of the doGet() method. The following is an example (reproduced from [Wikipedia, 2007c]) that shows how this has been done for the sample BookDetailServlet.

```
public class BookDetailServlet extends HttpServlet {
   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
         throws ServletException, IOException {
      ...
      // set content-type header before accessing the Writer
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();
      // Then write the response
      out.println( "<html>" +
                   "<head><title>Book Description</title></head>" +
                   ...);
      // Get the identifier of the book to display
```

```
      String bookId = request.getParameter( "bookId" );
      if (bookId != null) {
         // and the information about the book and print it
         ...
      }
      out.println( "</body></html>" );
      out.close();
   }
   ...
}
```

The servlet extends the HttpServlet class and overrides the doGet() method. Within the doGet() method, the getParameter() method obtains the expected argument for the servlet. The doGet() method uses a Writer from the HttpServletResponse object to return (text) data to the client. Before accessing the Writer, the example implementation given above sets the content-type header. Then the Writer is closed at the end of the doGet() method to signify the end of the response.

```
public class ReceiptServlet extends HttpServlet {
   public void doPost( HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException {
      ...
      // Set content type header before accessing the Writer
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();
      // Then write the response
      out.println( "<html>" + "<head><title> Receipt </title>" + ...);
      out.println( "<h3>Thank you for purchasing your books from us "
                   + request.getParameter("cardname") + ...);
      out.close();
   }
   ...
}
```

Handling POST requests involves the overriding of the `doPost()` method. The above example shows the structure of the code for the sample `ReceiptServlet`.

The servlet extends the `HttpServlet` class overriding the `doPost()` method. Within the `doPost()` method, the `getParameter()` method obtains the expected servlet argument. The `doPost()` method uses a `Writer` from the `HttpServletResponse` object to return (text) data to the client. Before accessing the `Writer`, the method sets the `content-type` header. Finally, at the end of the `doPost()` method, the `Writer` is closed to signify the end of the response.

5.3.3 Destroying a Servlet

The `destroy()` method defined by the `HttpServlet` class destroys the servlet and logs that destruction event. To destroy any resources specific servlet, this `destroy()` method must be overridden. The `destroy()` method must undo any initialization and synchronize the servlet's persistent state with the current in-memory state. The following is an example of how the `destroy()` method is implemented.

```
public class BookDBServlet extends GenericServlet {
   private BookstoreDB books;
   ... // the init method
   public void destroy() {
      // Allow the database to be garbage collected
      books = null;
   }
}
```

A server calls the `destroy()` method. If the servlet happens to be handling any long-running operations, `service()` methods might still be running when the server calls the `destroy()` method. This means that the programmer is responsible for ensuring that those service threads run to completion successfully. The example `destroy()` method assumes that the servlet has no long-running operations. Therefore, it expects all interactions with the client to be completed when the `destroy()` method is called [Layon, 2004].

5.4 Database Connectivity

In interactive webpage designs, database access is needed. Servlets can be connected to databases using JDBC, which enables Java programs to execute SQL statements. Hence, Java programs can interact with any SQL-compliant databases. Since nearly all relational database management systems (DBMS) support SQL, JDBC makes it possible for a programmer to write a single database application that can run on multiple platforms and interact with different database systems [Zakhour et al., 2007].

5.4.1 Creating a JDBC Application

This involves several steps [Zakhour et al., 2007]. This involves loading the driver and then making the connection. Loading the driver(s) involves one line of code. For example, the following single line will load the JDBC-ODBC Bridge driver.

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

```
Connection conn = DriverManager.getConnection( url, "myLogin",
                                               "myPassword" );
```

The second step is the establishment of a connection to the database, using the driver that must have already been loaded as shown above. The programmer must pay attention to the specification of the URL (first parameter to the `getConnection()` method) so that it is correct for a given database driver. If, for example, the JDBC-ODBC Bridge driver is used, the JDBC URL will start with the `jdbc:odbc:` prefix. The rest of the URL is the data source name or database system. So, on the other hand, ODBC is used to access an ODBC data source called ”NEU,” JDBC URL should be `jdbc:odbc:NEU`. In the example code above, ”myLogin” is the login name and ”myPassword” is the password for that login in order to connect to the DBMS.

In Java, an instance of the `Statement` interface is used to send SQL queries to a DBMS. First, a `Statement` object needs to be created by calling the `createStatement()` method of the `Connection` interface. To send a SELECT statement, the `executeQuery()` is used. For statements that create, modify, or remove tables, the `executeUpdate()`

method is used. The The following code snippet shows how to create a `Statement`.

```
Statement stmt = conn.createStatement();
```

When a SQL statement is executed, all connections must be closed using the `close()` method, as shown in the example code framework below.

```
// Create a connection
conn = DriverManager.getConnection( url, " ", " " );
stmt = conn.createStatement( ... ); // Create some SQL statement
stmt.executUpdate(); // Execute the SQL statement created above
stmt.close(); // Close the statement
conn.close(); // Close the connection
```

5.4.2 Using the Statement Interface

Entering data into a table requires that data are inserted in the same order that the database columns have been defined. The following code demonstrates this, using the `executeUpdate()` method interface.

```
Statement stmt = con.createStatement();
stmt.executeUpdate( "INSERT INTO Table_Name +
                    "VALUES( '    ', , ,0, 0 )" );
```

JDBC returns results in a `ResultSet` object. Therefore, an instance of the `ResultSet` interface must be declared to hold the results. The following code demonstrates the declaration and the storage of the results from an earlier defined SQL statement, where `Table_Name` is the name of the table from which data is being requested.

```
ResultSet rs = stmt.executeQuery( "SELECT * FROM Table_Name" );
```

In the code above, the variable, `rs`, an instance of `ResultSet`, is used to access each row and retrieve the values according to their types. The method `next()` moves the cursor to the next row and makes that row the currently active row.

A specific version of a get method is used to retrieve the value in each column. For example, the method for retrieving a value of SQL type VARCHAR is `getString()`, the method for retrieving floating-point values is `getFloat()` and so on, as demonstrated in the following example code, where value columns are given as `String_value` and `Float_value`.

```
String query = "SELECT * FROM Table_Name";
ResultSet rs = stmt.executeQuery( query );
while (rs.next()) {
   String s = rs.getString( "String _value" );
   float n = rs.getFloat( "Float_value" );
   System.out.println( s + "    " + n );
}
```

5.5 Implementing a Benchmark Problem

A Java servlet is used to implement the benchmark algorithm that is discussed in *Chapter 4*. First, a servlet class named `MyServlet` has been created java is created , and it aggrrgates a user-defined class named `MyClass`.

`MyClass` models the benchmark problem designed for this thesis. It contains four attributes. The first attribute is a string and others are integers. This class comprises nine methods including the constructor.

The servlet is invoked from an HTML page, in the case of the implementation for this thesis, the HTML page is named `myapplication1.html` (See *Appendix A.1*.) To run the benchmark, the following address must be entered into the address box of a browser:

```
http://localhost:8080/myapplications/servlets/myapplication1.html
```

The file `myapplication1.html` contains a form that is to be filled by the user. Then the data are posted to `MyServlet`. Java in turn creates an instance of the `MyClass` class and sets its attributes according to the values posted from the user.

`MyServlet` calculates the amount of the interest for the specified company, for a given deposit and the number of customers, number of years that the user has entered. It then saves the results of this processing in a database file. The database is created using

the Microsoft Access 2003 DBMS.

Two database tables are created. One of the tables, company table, is for strong company data, and it has the following fields:

- Company name (type: text) primary key
- Number of client (type: integer)

The second table, client table, contains information about clients, and it has the following fields:

- Client ID (type: text) –Primary key
- Name of company (type: text) – This field is the foreign key
- Deposit amount (type: long integer)
- Number of years (type: integer)
- Total amount of deposit

Then `MyServlet` retrieves the content of these database tables and prints this content to the user by drawing a table and placing the retrieved data in it.

5.6 Summary

In this chapter, the technique of combining programming with embedded scripting is described. As an application of this technique, Java servlets are discussed. Java servlets are object-oriented, multithreaded, and platform-independent. This chapter also describes the life cycle of a Java servlet and database connectivity using JDBC. The next chapter describes the second approach for combining scripting with embedded programming.

Chapter 6

SCRIPTING WITH EMBEDDED PROGRAMMING LANGUAGES: Comparisons of Implementation of a Benchmark Problem

Scripting languages with embedded programming languages is the second technique for combining programming and scripting languages together. In this thesis, scripting language statements are HTML tags that contain programming language statements. Java Server Pages (JSP) are used as an application of this technique. JSP pages are documents that contain Java source code. Microsoft's ASPs (Active Server Pages) is also an application of this technique, but ASP is not object-oriented or multithreaded, and it runs only on Microsoft operating system platforms [Moreira et al., 2000].

This chapter describes the JSP language. It lists the advantages of JSP, provides the basic JSP syntax, and compares JSP with other technologies. In addition, this chapter describes the implementation of the benchmark problem described in *Chapter 4*.

This chapter finally compares JSP with servlets using some tests to show which is the preferable technique for programming interactive webpages. The testing of the qualitative and quantitative parameters was done according to the following criteria:

1. **Qualitative parameters:**

   (a) *Readability* is measured by an expert programmer by assessing the ease to read and understand individual program statements or a group of statements [Sebesta, 2006; Kennedy et al., 2004].

   (b) *Simplicity* is measured by assessing the number of statements to express the same algorithm (the program size) [Sebesta, 2006; Kennedy et al., 2004].

   (c) *Maintainability* is measured by assessing the independencies to update the program for business logic which usually changes very often and for programming logic which is more stable [Sebesta, 2006; Kennedy et al., 2004].

2. **Quantitative parameters** are measured in the conventional way by processing time and memory usage [Parker et al., 2006].

6.1 Java Server Pages (JSP)

Java Server Pages (JSP) is a server-side technology, and this technology is an extension of the Java servlet technology. Java Server Pages provide a dynamic scripting capability that works in conjunction with HTML code. In this way, the page logic is separated from the static elements, namely the design and display of the page, to help make the HTML have extra functionality such as dynamic database queries [Yourdon, 1996; Webopedia, 2007].

A Java server page is translated into a Java servlet before being run. As in the case of a Java servlet, the translated JSP processes HTTP requests and generates responses. However, the JSP technology provides a more convenient approach to coding a servlet. Translation occurs only the first time a given application is run. The translation is triggered by the `.jsp` filename extension in a URL, and, because of this translation into a servlet, JSPs are fully interoperable with servlets. Therefore, the output from a servlet can include output from either a servlet or a JSP and forward output to a servlet or a JSP. [Shaw et al., 1981; Hall, 2007; Yourdon, 1996; Webopedia, 2007].

6.2 Advantages of JSP

There are a number of advantages to implementing in JSP an application of scripting languages with embedded programming languages, compared to doing the same using servlets only.

The logic for generation dynamic content is a natural part of servlets, and this is closely related to the static presentation templates that implement the user interface. So, even minor modifications to the user interface result in the recompilation of the servlet. However, this tight coupling of presentation and content results in brittle and inflexible applications. On the other hand, with JSP, the logic for generating dynamic content is separate from the static presentation templates, because the dynamic content is handled by external JavaBeans components. Then, when a presentation template is modified, the JSP engine automatically recompiles and reloads the JSP page into the webserver.

Due to their interpreted nature, JSP pages can be moved across computing platforms and across webservers, without any changes, thereby providing the "write once, run anywhere" capability. In addition, dynamic content can be presented in any format, depending on the type of the browser or application platform, ranging from conventional HTML/DHTML to XML and WML.

Since JSP is a high-level abstraction of servlets, it leverages the servlet API, bringing all the advantages of servlets with it, and, JSP is also object-oriented [Yourdon, 1996].

6.3 JSP Syntax

The JSP syntax can be grouped into three categories, namely directives, scripting elements, and standard actions. Since standard actions can be any legal Java statement, the following subsections describe only JSP directives and scripting elements.

6.3.1 Directives

JSP directives are messages for the JSP engine. Directives do not directly produce any visible output, but they inform the engine about what to do with the rest of a given JSP page. JSP directives are enclosed within the `<%@ ... %>` tag. The two primary directives are `page` and `include`.

Page Directive

The `page` directive is typically found at the top of a JSP page. There can be any number of `page` directives within a JSP page, but each attribute/value pair must be unique. Unrecognized attributes or values cause a translation error. The example `page` directive given below imports the types declared within the included packages for scripting, and it sets page buffering to 16K.

```
<%@ page import="java.util.*, com.foo.*" buffer="16k" %>
```

Include Directive

The `include` directive allows the separation of content into more manageable parts. For example, the `include` directive can be used to including a page header or footer that must be common to multiple pages. The page to be included can be a static HTML page or more JSP content. For example, the directive includes the contents of the indicated file (`copyright.html`) at any location within the JSP page.

```
<%@ include file="copyright.html" %>
```

6.3.2 Declarations

JSP declarations allow the definition of page-level variables to save information and methods that will be used in the rest of a JSP page. As mentioned in *Section 6.2*, encapsulating logic-intensive operations in JavaBeans components helps in readability and reusability. Declarations in JSP are specified using the `<%! ... %>` tag. All content within this tag must be a valid Java statement. For example, the code below defines an integer variable, initializing it to zero.

```
<%! int i=0; %>
```

Moreover, the programmer may define methods, as in Java. For example, if the default initialization event in the JSP life cycle needs to be modified, then the programmer

must override the `jspInit()` method, as in the example code below:

```
<%! public void jspInit() {
       // Initialization code goes here
    }
%>
```

6.3.3 Expressions

In JSP, the result of the evaluation of an expression is converted to a string and directly included within the output page. In general, expressions in JSP are used to display the values of variables or simple values of variables or return values of get-methods on a JavaBean. JSP expressions are specified within `<%= ... %>` tags –Here, there is no need to include semicolons, as demonstrated in the code piece below..

```
<%= fooVariable %>
<%= fooBean.getName() %>
```

6.3.4 Scriptlets

JSP code fragments (*scriptlets*) are specified within `<% ... %>` tags. The Java code contained in each such tag is run when the request is serviced by the JSP page. A programmer can include any valid Java statement in a scriptlet tag. For example, the following example code displays the string `''Hello''` within `H1`, `H2`, `H3`, and `H4` tags, combining the use of expressions and scriptlets:

```
<% for (int i=1; i<=4; i++) { %>
    <H<%=i%>>Hello</H<%=i%>>
<% } %>
```

6.4 The Benchmark Problem

JSP are used to implement the benchmark algorithm that is discussed in *Chapter 4*. The JSP code is called from an HTML page, called `myapplication2.html`. To run this page, the following URL must be entered into the browser:

```
http://localhost:8080/myapplications/jsp/myapplication2.html
```

The file `myapplication2.html` contains a form that is to be filled by the user, then it will be posted to `Myjsp.jsp`. Then `Myjsp.jsp` creates an instance of `MyClass` class and sets its attributes to the values posted from the user –the first 4 attributes. The JSP code in `Myjsp.jsp` calculates the amount of the interest for the specified name of the company for a given deposit, and the number of customers, for number of years that the user has entered via the user interface. It then saves the results of this processing in a database file called `mydatabase`. that is Then `Myjsp.jsp` retrieves the contents of database tables and prints it to the user.

6.5 Comparison of Programming Techniques

The comparison approach in this thesis follows the conventional method, which includes qualitative parameters and quantitative parameters [Sebesta, 2006]:

1. Qualitative parameters: readability, simplicity and maintainability.

2. Quantitative parameters: performance (time of processing) and memory use.

It is known that the most accurate benchmark is the one customized to the particular group of the problems being solved. In this work, the problem domain is interactive web application development using a mixture of regular programming and scripting, about which there are no known benchmarks available. Existing benchmarks only compare single programming languages (not a composition of the languages) using quantitative parameters mentioned a little earlier. Thus, a special benchmark was designed for the assessment of composing scripting languages and programming languages.

Qualitative parameters are assessed expert human voting. To assess the qualitative parameters, comparisons of the source code for each programming technique should be performed.

Quantitative parameters are assessed by measuring the amount of time and the amount of memory used for the program. In the case of this thesis, this measurement was done using the *Task Manager* of the Windows XP operating system. To use the Task Manager, the user needs to press *Ctrl + Alt + Delete* and then go to the *Processes* tab. In this tab, every process is listed with its user name, CPU time, and memory usage. Therefore, these values are readily available.

The test for quantitative parameters was done 3 times, and then the average was calculated. The number of tests according to statistics is based on the value of standard deviation, which very small for this measurement, because the data obtained is statistically stable.

6.5.1 Readability

Readability means that a piece of code is easily understood when read by a programmer who is already familiar with the language of that code. Java servlet technology combines both business logic and programming logic together. So a Java servlet programmer must be familiar with Java programming and webpage design. Note that the page ranges given below are inclusive.

- `MyServlet.java` (*Appendix A.2*):

  Programming logic starts at line 1 and lasts until line 14. It then restarts at line 23 and goes until line 97, then from 104 until 110. Business logic starts at line 15, ends at line 21. It then continues from line 98 until line 103. As can be seen from the source code, programming logic and business logic are intertwined from the beginning until the end of the program.

  This is an example of how business logic is combined with programming logic. Switching from one logic to the other in this fashion would be confusing.

- `Myjsp.jsp` (*Appendix B.2*):

  In JSP, on the other hand, programming logic is separated from the business logic, and the programmer only needs to take care of programming logic. Programming logic in the benchmark JSP code starts at line 1, by importing the needed packages, and finishes at line 2. It then restarts at line 19 and continues until line 92, then

restarts from line 101 until line 108. Therefore, as a result of this separation, between programming logic and business logic, the JSP page is more readable than the servlet implementation in `MyServlet.java`.

The business logic in this JSP implementation starts at line 3 and lasts until 17. It then restarts at line 94 and ends at line 99. Finally, there is a section of business logic at lines 109 and 110.

In the servlet implementation in `MyServlet.java`, the business logic is embedded within regular Java code, so there is no clear separation between the two logics. Conversely, in the JSP implementation in `Myjsp.jsp`, it can be seen from the source code that programming logic and business logic are interleaved as in the servlet implementation, except that business logic statements are not embedded inside programming logic. Therefore, the readability of JSP code is higher than servlet code.

6.5.2 Maintainability

Maintainability refers to the ease of adding new capability to existing code in the face of new needs, fixing errors that come up during regular use, or adapting existing code to previously unforeseen problems. Since maintainability also depends on readability, JSP code is simpler than the Java servlet implementation due to the clear separation between programming logic and business logic.

6.5.3 Program Size

As stated earlier, JSP separates programming logic from the business logic, which is not the case with Java servlets. Because of this separation, JSP pages require less size than Java servlet programs. For example, the Java servlet implementation of the benchmark with embedded business logic (`MyServlet`) requires 3.99 KB of memory, and the JSP implementation of the benchmark with interleaved business logic statements (`Myjsp`) requires 3.66 KB of memory.

6.5.4 Compilation Time

Java servlets are regular programs, so they need be compiled before execution by the programmer. JSP, on the other hand, is a server-side script that is interpreted on demand when a user loads a JSP page into his or her browser.

6.5.5 Performance

To test the performance of both programming techniques, the benchmark algorithm discussed in *Chapter 4* is implemented using both approaches. The benchmark implementation using regular programming with embedded scripting (`MyServlet.java`) requires an average time of 4.06 seconds to execute. The benchmark implementation using scripting with embedded programming (`Myjsp.jsp`) requires an average time of 3.24 seconds to execute. Therefore, the time needed for executing the servlet is larger than that needed to execute the JSP implementation.

6.6 Summary of Results

Table 6.1 below summarizes the results obtained from the execution of the benchmark using the two programming techniques. The table lists the major differences between the Java servlet implementation of the benchmark and the JSP implementation. This information in this table may help a web programmer who wishes to use Java technology to choose the proper programming approach, whether to use programming with embedded scripting or scripting with embedded programming.

JSPs are used when most of the content sent to the client contains business logic, and only a small portion of the content is generated with Java code, i.e., using programming logic. On the other hand, servlets are commonly used when a small portion of the content sent to the client is business logic.

Scripting languages with embedded programming such as the case in JSP, divides programming logic and business logic distinctly even though statements of both are interleaved within the same overall body of code. So as *Table 6.1* shows, programming languages with embedded scripting is more readable, and maintainable [Liang, 2005]. Scripting with embedded programming also needs less amount of programming, even

Table 6.1: Comparison of servlets and JSP using both qualitative and quantitative criteria. This table summarizes the results of the study reported in this thesis.

PARAMETERS	SERVLET	JSP
Readability	Low	High (because of the division of programming logic and business logic) [Liang, 2005]
Maintainability	Low	High (easy to maintain and adaptation),readability is high [Liang, 2005]
Program size (Simplicity)	Bigger (combining programming logic and business logic), `MyServlet` requires 3.99 KB	Smaller (separate business logic from programming logic), `Myjsp` requires 3.66 KB
Compilation time	Higher	Precompiled by server (no need to compile)
Format	Programs embedding html tags	HTML document embedding java codes
Performance	Needs more time to execute `MyServlet` requires 4.06 sec	Needs less time to execute `Myjsp` (3.24 sec)
Programming skills	Java and HTML programmer	Only HTML
Memory used	`MyServlet` requires 15.316 KB	`Myjsp` requires 17.856 KB

though in the case of the benchmark developed, the difference in code size was small. In addition, this programming approach requires less time to execute than the programming with embedded scripting approach.

Since processing in a scripting language such as JSP can be done externally in JavaBeans, a web designer who is not very familiar with Java programming can also develop interactive applications by using existing JavaBeans components to develop dynamic HTML pages. Therefore, the scripting with embedded programming approach is preferable to the programming with embedded scripting approach, according to the criteria mentioned earlier.

6.7 Summary

This chapter describes the second technique of combining programming languages and scripting languages. Java Server Pages (JSP) are used as an implementation of this technique. In addition, the basics of JSP syntax are described. This chapter also lists the advantages of using JSP compared to using servlets. In addition, database interaction using JDBC is described.

This chapter also compares the two techniques of combining programming and script-

ing languages, according to the qualitative parameters (readability, maintainability, simplicity) and quantitative parameters (CPU time, memory used). According to this comparison, scripting languages with embedded programming languages are preferable to programming with embedding scripting.

The next chapter gives the conclusion of this study, and it mentions possible improvements to the work reported in this thesis.

Chapter 7

CONCLUSIONS

Interactive webpages and online applications are prevalent in many fields nowadays, such as banking, e-learning, e-commerce, and both programming logic and presentation logic are used to implement them in order to draw the strengths of both programming techniques. Then the question of whether programming logic should be used inside presentation logic or whether presentation logic should be used in programming logic becomes of importance for programmers from a number of perspectives such as reliability, maintainability, performance, and the like. Therefore, there is a need for standard benchmarking of the two approaches in order to be able to decide which approach should be preferred and under what circumstances.

Conventional benchmarks exist for comparing programming and scripting languages. However, there are no benchmarks for comparing the overall performance of scripting with regular programming and programming with scripting. The aim of this thesis was to develop and implement such a benchmark.

According to conventional benchmarks, programming languages are better for programming logic and scripting languages are better for business logic. However, when these two logics are combined, it is unclear which approach is the advantageous one.

In this thesis, a special benchmark algorithm was designed that follows programming logic and business logic for the comparison of the composition of programming languages and scripting languages according to conventional set of criteria that includes readability, maintainability, program size, performance, and memory used.

Programming languages with embedded scripting languages is the first technique

of combining programming regular languages and scripting languages. Java servlets were discussed as an application for this technique, since the Java technology is object-oriented, cross-platform, and multithreaded technique. The benchmark algorithm for this programming approach was designed and implemented as a Java servlet.

Scripting languages with embedded programming languages is the second technique of combining programming languages and scripting languages. Java server pages (JSP) was applied as an application of this technique, and the benchmark algorithm for this programming approach was designed and implemented in JSP.

It was shown with an interactive web application that a scripting language with an embedded programming language has better readability for being able to distinguish programming logic and business logic. Maintainability of the code also is better as a result of high readability, which means that program simplicity is high. In addition, there is no need for compiling in this technique, and performance is high. So, as a result, this approach is preferable to programming languages with embedded scripting languages for interactive webpage design.

JSPs were used to divide clearly the business (interconnection to the clients) logic and programming (processing) logic. As a result of this approach, the major recommendation is that, when the interconnections to the clients are needed to be established more often and should be more flexible than making some calculation in interactive web application, then JSP should be used.

7.1 Future Work

This work investigates only one application of interactive webpage design, which is a banking system. However, to prove the claim that scripting language with embedded programming language is the preferable approach to interactive webpage design, additional benchmarks should be designed. Thus, other applications of interactive websites should be investigated for a stronger generalization of the claim of this thesis, and different types of benchmarks should be designed and implemented for comparing the composition of programming and scripting languages.

REFERENCES

Intel Benchmarks. Intel pentium4 computer language benchmarks game. Accessed 05/07/2007, 2007. <http://shootout.alioth.debian.org/gp4>. ◇ pp: 10, 27, 28, 29, 32

Nikolai Bezroukov. A slightly skeptical view on scripting languages. Accessed 05/07/2007, 2007. <http://www.softpanorama.org/Articles/a_slightly_skeptical_view_on_scripting_languages.shtml>. ◇ pp: 14, 20, 21

Stephanie Bodoff. Java servlets technology. Accessed 05/07/2007, 2007. <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html>. ◇ pp: 31, 36, 37, 38

Alison Cawsey. Advantages and disadvantages of oop. Accessed 05/07/2007, 2007. <http://www.cee.hw.ac.uk/˜alison/ds98/node47.html>. ◇ pp: 8, 23

Deitel & Deitel. *Java: How to Program*. Prentice Hall, 2003. ◇ pp: 24

Marty Hall. Servlets and java server pages (jsp) 1.0: A tutorial. Accessed 05/07/2007, 2007. <http://www.apl.jhu.edu/˜hall/java/Servlet-Tutorial/>. ◇ pp: 48

Ken Kennedy, Charles Koelbel, and Robert Schreiber. Defining and measuring the productivity of programming languages. *Int. J. High Perform. Comput. Appl.*, 18(4):441–448, 2004. doi: http://dx.doi.org/10.1177/1094342004048537. ◇ pp: 48

Jeff Kramer and Jeff N. Magee. Analysing dynamic change in software architectures: a case study. pages 146–154, oct 1998. ◇ pp: 13

Douglas A. Layon. *Java for Programmers*. Prentice Hall, 2004. ◇ pp: 36, 42

Y. Daniel Liang. *Introduction to Java Programming*. Prentice Hall, 2005. ◇ pp: 22, 55, 56

Stefan Mischook. Scripting vs. programming: Is there a difference? Accessed 05/07/2007, 2007. `<http://www.killersites.com/blog/2005/scripting-vs-programming-is-there-a-difference/>`. ◇ pp: 23, 24

J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1): 21–56, 2000. ◇ pp: 47

John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998. `<citeseer.ist.psu.edu/article/ousterhout97scripting.html>`. ◇ pp: 14, 15, 16, 17, 18, 19, 20, 24

Kevin R. Parker, Thomas A. Ottaway, and Joseph T. Chao. Criteria for the selection of a programming language for introductory courses. *International Journal of Knowledge and Learning*, 2(1/2):119–139, 2006. ◇ pp: 48

Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33 (10):23–29, 2000. ◇ pp: 34

Lutz Prechelt. Are scripting languages any good? A validation of perl, python, rexx, and tcl against C, C++, and Java. *Advances in Computers*, 57:207–271, 2003. ◇ pp: 5, 6, 9, 10, 20, 21

Robert W. Sebesta. *Concepts of Programming Languages*. Pearson Education, 2006. ◇ pp: 48, 52

Mary Shaw, Guy T. Almes, Joseph M. Newcomer, Brian K. Reid, and William A. Wulf. A comparison of programming languages for software engineering. *Software: Practice, Experience*, 11(1):1–52, 1981. ◇ pp: 27, 28, 29, 48

Guy L. Steele. *COMMON LISP: the Language, 2nd edition*. Digital Press, 1990. ISBN 1-55558-041-6. ◇ pp: 18, 23

Ashok Subramanian. Object-oriented programming vs. procedural programming. Accessed 05/07/2007, 2007. `<http://www.umsl.edu/~subraman/concepts1.html>`. ◇ pp: 11, 12

Webopedia. What is jsp? Accessed 05/07/2007, 2007. <http://www.webopedia.com/TERM/J/JSP.html>. ◇ pp: 48

Wikipedia. Object-oriented programming. Accessed 05/07/2007, 2007a. <http://en.wikipedia.org/wiki/Object_oriented_programming>. ◇ pp: 6, 7, 11, 35

Wikipedia. Comparison of programming languages. Accessed 05/07/2007, 2007b. <http://en.wikipedia.org/wiki/Comparison_of_programming_languages>. ◇ pp: 5, 6, 9, 10

Wikipedia. Java servlets. Accessed 05/07/2007, 2007c. <http://en.wikipedia.org/wiki/Java_Servlet>. ◇ pp: 6, 7, 14, 36, 37, 38, 40

Wikipedia. Benchmark (computing). Accessed 05/07/2007, 2007d. <http://en.wikipedia.org/wiki/Benchmark_%28computing%29>. ◇ pp: 26

Wikipedia. Procedural programming. Accessed 05/07/2007, 2007e. <http://en.wikipedia.org/wiki/Procedural_programming_languages>. ◇ pp: 5

Edward Yourdon. Java, the web, and software development. *IEEE Computer*, 29(8):25–30, 1996. ◇ pp: 36, 48, 49

Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. Jdbc(tm) database access. Accessed 05/07/2007, 2007. <http://java.sun.com/docs/books/tutorial/jdbc>. ◇ pp: 43

Neal Ziring. Dictionary of programming languages. Accessed 05/07/2007, 2007. <http://cgibin.erols.com/ziring/cgi-bin/cep/cep.pl>. ◇ pp: 5, 6, 9, 10, 13, 14, 23

APPENDICES

APPENDIX A

Implementation of the Benchmark (Method 1)

Implementation of benchmark algorithm using programming languages with embedded scripting languages, namely Java servlets.

A.1 Source listing: `myapplication1.html`

```
<html>
<head>
<title>
myapplication1
</title>
</head>
<body>
<h3 align="center">My Application1</h3>
<p align="CENTER"><b>
This program is used to calculate the amount of interest of  a company
</b></p>
<form  action="/myapplications/myservlet" method="POST">
<p align="center">Company name
   <input type="text" name ="param1" size=13>
</p><br>
Customer No.<input type="text" name ="param2"size=13><p><br>
Deposit
Amount<input type="text" name ="param3" size=12></p>
<br>
No. of years<input type="text" name ="param4"size=18>
<blockquote>
  <blockquote>
    <blockquote>
      <blockquote>
```

```
        <blockquote>
          <blockquote>
            <blockquote>
              <blockquote>
                <blockquote>
                  <input type="submit" value="Click"><br>
                </blockquote>
              </blockquote>
            </blockquote>
          </blockquote>
        </blockquote>
      </blockquote>
    </blockquote>
  </blockquote>
</blockquote>
<p align="center"> </p><br>
</form>
</body>
</html>
```

A.2 Source listing: MyServlet.java

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import mypackage.MyClass;

public class MyServlet extends HttpServlet {

   public void doPost( HttpServletRequest req, HttpServletResponse resp )
               throws ServletException, IOException {

      String url = "jdbc:odbc:mysource";
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.println( "<html><head><title>myservlet</title></head><body>" );
      out.println( "<h3>My Servlet</h3>" );
      out.println( "<table border=2><th>Company name</th>" +
                   "<th>Number of clients</th>" +
                   "<th>Deposit</th>" +
                   "<th>Years</th>" +
                   "<th>Total amount of interest</th>" );

      MyClass ob = new MyClass();

      String p1 = req.getParameter( "param1" );
      int p2 = Integer.parseInt( req.getParameter( "param2" ) );
      int p3 = Integer.parseInt( req.getParameter( "param3" ) );
      int p4 = Integer.parseInt( req.getParameter( "param4" ) );

      double i = 0.05;
      double amount=0;

      for ( int j=1; (j <= p2); j++ ) {
         amount += p3 * (Math.pow( ( 1 + i ), p4 ));
      }

      ob.setAttribute1( p1 );
      ob.setAttribute2( p2 );
      ob.setAttribute3( p3 );
      ob.setAttribute4( p4 );
      ob.setAttribute5( amount );


```

```
44        try {
45           Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
46        }
47        catch( Exception exp ) {
48           out.println( exp.getMessage() );
49        }
50
51        try {
52           Connection conn = DriverManager.getConnection( url, " ", " " );
53           Statement stmt = conn.createStatement();
54           stmt.executeUpdate( "insert into company values('" + p1 + "','" + p2 +
55                               "')");
56           stmt.close();
57           conn.close();
58        }
59        catch( SQLException ex ) {
60           out.println( "SQLERROR:" + ex.getMessage() );
61        }
62
63        try {
64           Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
65        }
66
67        catch( Exception exp ) {
68           out.println( exp.getMessage() );
69        }
70
71        try {
72           Connection conn = DriverManager.getConnection( url, " ", " " );
73           Statement stmt = conn.createStatement();
74           stmt.executeUpdate( "insert into " +
75                               "customer( companyname, deposit, year, total) " +
76                               "values('" + p1 + "','" + p3 + "','" + p4 +
77                               "','" + amount + "')" );
78           stmt.close();
79           conn.close();
80        }
81        catch( SQLException ex ) {
82           out.println( "SQLERROR:" + ex.getMessage() );
83        }
84
85        try {
86           Connection conn2 = DriverManager.getConnection( url," "," " );
87           Statement stmt2 = conn2.createStatement();
88           String s = "select company.companyname," +
89                      "company.numberofclients," +
90                      "customer.deposit," +
```

```
91                      "customer.year," +
92                      "customer.total " +
93                      "FROM company,customer " +
94                      "WHERE company.companyname = customer.companyname";
95          ResultSet rset = stmt2.executeQuery(s);
96
97          while (rset.next()) {
98             out.println("<tr>");
99             out.println("<td>"+(rset.getString(1))+"</td><br>");
100            out.println("<td>"+(rset.getInt(2))+"</td><br>");
101            out.println("<td>"+(rset.getInt(3))+"</td><br>");
102            out.println("<td>"+(rset.getInt(4))+"</td><br>");
103            out.println("<td>"+(rset.getDouble(5))+"</td><br>");
104         }
105      }
106      catch( SQLException ex ) {
107         out.println( "SQLERROR:" + ex.getMessage() );
108      }
109   }  /* end of doPost */
110 }  /* end of class */
```

APPENDIX B

Implementation of the Benchmark (Method 2)

Implementation of the benchmark algorithm using scripting languages with embedded programming languages, namely Java Server Pages.

B.1 Source listing: `myapplication2.html`

```
<html>
<head>
<title>
myapplication2
</title>
</head>
<body>
<h3 align="center">My Application2</h3>
<p align="CENTER"><b>
This program is used to calculate the amount of interest of  a company
</b></p>
<form method="post" action="myjsp.jsp">
<p align="center">Company name
   <input type="text" name ="param1" size=13>
</p><br>
Customer No.<input type="text" name ="param2"size=13><p><br>
Deposit
Amount<input type="text" name ="param3" size=12></p>
<br>
No. of years<input type="text" name ="param4"size=18><blockquote>
  <blockquote>
    <blockquote>
      <blockquote>
        <blockquote>
```

```
          <blockquote>
            <blockquote>
              <blockquote>
                <blockquote>
                  <input type="submit" value="Click"><br>
                </blockquote>
              </blockquote>
            </blockquote>
          </blockquote>
        </blockquote>
      </blockquote>
    </blockquote>
  </blockquote>
</blockquote>
<p align="center"> </p><br>
</form>
</body>
</html>
```

B.2 Source listing: `Myjsp.jsp`

```
<%@ page import="java.sql.*"%>
<%@ page import="mypackage.MyClass"%>
<html>
<head>
<title>
Myjsp
</title>
</head>
<body>
<h3 align="CENTER">My JSP</h3>

<table border=2 align="CENTER">
   <th>Company name</th>
   <th>Number of clients</th>
   <th>Deposit</th>
   <th>Years</th>
   <th>Total amount of interest</th>
<%
MyClass ob = new MyClass();
String url = "jdbc:odbc:mysource";

String p1 = request.getParameter("param1");
int p2 = Integer.parseInt(request.getParameter("param2"));
int p3 = Integer.parseInt(request.getParameter("param3"));
int p4 = Integer.parseInt(request.getParameter("param4"));

double i = 0.05;
double amount=0;

for ( int j=1; (j <= p2); j++ ) {
   amount+= p3 * ( Math.pow(( 1 + i ), p4 ));
}

ob.setAttribute1( p1 );
ob.setAttribute2( p2 );
ob.setAttribute3( p3 );
ob.setAttribute4( p4 );
ob.setAttribute5( amount );

try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
}
catch( Exception exp ) {
```

```
44      out.println( exp.getMessage() );
45   }
46
47   try {
48      Connection conn = DriverManager.getConnection(url," "," ");
49      Statement stmt = conn.createStatement();
50      stmt.executeUpdate("insert into company values('"+p1+"','"+p2+"')");
51      stmt.close();
52      conn.close();
53   }
54   catch( SQLException ex) {
55      out.println( "SQLERROR:" + ex.getMessage() );
56   }
57
58   try {
59      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
60   }
61   catch( Exception exp ) {
62      out.println( exp.getMessage() );
63   }
64
65   try {
66      Connection conn = DriverManager.getConnection(url," "," ");
67      Statement stmt = conn.createStatement();
68      stmt.executeUpdate( "insert into " +
69                          "customer( companyname, deposit, year, total)" +
70                          "values('" + p1 + "','" + p3 + "','" + p4 +
71                          "','" + amount + "' )" );
72      stmt.close();
73      conn.close();
74   }
75   catch( SQLException ex) {
76      out.println( "SQLERROR:" + ex.getMessage() );
77   }
78
79   try {
80      Connection conn2 = DriverManager.getConnection(url," "," ");
81      Statement stmt2 = conn2.createStatement();
82      String s = "SELECT company.companyname,
83                         company.numberofclients,
84                         customer.deposit,
85                         customer.year,
86                         customer.total
87                  FROM company,customer
88                  WHERE company.companyname = customer.companyname";
89      ResultSet rset = stmt2.executeQuery(s);
90
```

```
91  while (rset.next()) {
92  %>
93
94     <tr>
95     <td><% out.println((rset.getString(1)));%></td><br>
96     <td><% out.println((rset.getInt(2)));%></td><br>
97     <td><% out.println((rset.getInt(3)));%></td><br>
98     <td><% out.println((rset.getInt(4)));%></td><br>
99     <td><% out.println((rset.getDouble(5)));%></td><br>
100
101 <%
102 }
103
104 }
105 catch( SQLException ex) {
106    out.println( "SQLERROR:" + ex.getMessage() );
107 }
108 %>
109 </body>
110 </html>
```

APPENDIX C

Source listing: MyClass.java

The source code of `MyClass.java` is needed by both implementation methods.

```
package mypackage;

public class MyClass {
   String attribute1;
   int attribute2, attribute3, attribute4;
   double attribute5;

   public void MyClass() {}

   public void setAttribute1( String parameter1 ) {
      attribute1 = parameter1;
   }

   public String getAttribute1() {
      return( attribute1 );
   }

   public void setAttribute2( int parameter2 ) {
      attribute2 = parameter2;
   }

   public int getAttribute2() {
      return( attribute2 );
   }

   public void setAttribute3( int parameter3 ) {
      attribute3 = parameter3;
   }

   public int getAttribute3() {
      return( attribute3 );
```

```java
    }

    public void setAttribute4( int parameter4 ) {
        attribute4 = parameter4;
    }

    public int getAttribute4() {
        return( attribute4 );
    }

    public void setAttribute5( double parameter5 ) {
        attribute5 = parameter5;
    }

    public double getAttribute5() {
        return( attribute5 );
    }

}  /* end of class */
```