Can a 40 year old Computer Language do Web Applications?

Using Common Lisp to build Web Applications.

Nicky Peeters
nicky.peeters@pandora.be

June 19, 2003

Abstract

Today's web applications are often built using Java technology. Proponents of Java advocate its many advanced features and repeatedly claim there is some magic link and interdependence between web technology and Java as a programming language.

The fabric of the web however, the open protocols like TCP/IP, HTTP and standards like HTML and XML, is in fact language- , operating-system- , and vendor-neutral. It can be argued that Java is nothing new, and only a clever cocktail of features of several older technologies.

ANSI Common Lisp provides nearly all of the advantages that make Java attractive, including automatic memory management, dynamic object-oriented programming and portability. Several other features including its interactivity and full dynamic typing make Lisp development a very different experience.

It is my thesis that the venerable computer language Common Lisp is at least as capable as an implementation technology for web applications even though conceived and refined long before the internet revolution. I will show how some of Lisp's features, many which are not (yet) available in Java, make it a great platform to build web applications on.

Acknowledgements

I thank Sven Van Caekenberghe[1] and Hein Saris for the opportunity to do this project at Beta Nine Software Engineering[2]. Beta Nine is most of all a *Java/J2EE* shop but are experienced in other languages and technologies and as such are not afraid to experiment with the unknown and offer employees and interns learning opportunities.

I express my gratitude to Sven specifically for his guidance and support throughout the Lisp learning process and the whole project. I learned a lot by just watching him program and reading his beautiful code. Trying to explain what Lisp is and how it works was no sinecure and his advice was most welcome.

I have to mention the discussions I have had with Ivan Melotte, Ronald Jamaer, Jochen Punie, Marco Huygen and Tim Degrande at Beta Nine about software development, the *"holy wars"* we conducted in our dissemination of operating systems, user interfaces and general computer science. They were both entertaining and an essential part of the learning process largely associated with computer geeks and hackers. Thanks guys !

I also thank my fellow interns Bert Heymans and Jurgen Vanhex. When I saw them suffering while programming a proxy server in Java, I could only support them by telling them how cool Lisp was.

Christophe Schats deserves my thanks (and of his Beta Nine colleages) for handling all the paper work associated with internships and other business related issues and last, but not least, for keeping the coffee flowing during our days of writing code. Much appreciated, Christophe !

I thank Francis Vos and Marina Luwel at *De Hogeschool Limburg* for their efforts and help during my years at college.

To end this list I thank Dag Wieers[3], *Linux Guru* and *Hacker Extraordinaire*, who introduced me to *Linux* and *Open Source software* and without whom I would never have ended up at Beta Nine in the first place.

[1]http://homepage.mac.com/svc/
[2]http://beta9.be/
[3]http://dag.wieers.com/

Contents

1 Introduction **5**

1.1 Why Lisp? . . . 5

1.2 Why web applications? . . . 5

2 Lisp **7**

2.1 History and motivation . . . 7

2.1.1 From paper to version . . . 7

2.1.2 Lisp Machines . . . 8

2.1.3 ANSI Standardization : Common Lisp . . . 10

2.2 Language design and Features . . . 12

2.2.1 Design ideas . . . 12

2.2.2 Language features . . . 15

2.2.3 An example Lisp session . . . 17

3 KPAX Web Application Framework **20**

3.1 Web Applications . . . 20

3.1.1 What are Web Applications? . . . 20

3.1.2 KPAX Web Applications . . . 21

3.2 Design and Features . . . 21

3.2.1 Session Management and Security . . . 22

3.2.2 Web Actions . . . 23

3.2.3 Lisp Server Pages . . . 23

3.2.4 Web application definition . . . 23

3.3 Examples . . . 24

3.3.1 Hello World . . . 25

3.3.2 Factorial . . . 26

4 RSS Syndication **30**

4.1 What is RSS? . . . 30

4.2 Design and Features . . . 31

4.2.1 Feed persistancy 32
4.2.2 XML Technology 33

5 Psillog **35**
5.1 The Weblog phenomena 35
5.1.1 What are weblogs? 35
5.1.2 Existing Weblog Implementations 36
5.2 Design and Features 37
5.2.1 Design 37
5.2.2 Features 38
5.2.3 Object Domain Model 38
5.2.4 Database persistency 40
5.3 Psillog Weblog System 41
5.4 Eyeswideshut, a Psillog powered Weblog 44

6 Conclusions **47**
6.1 Lisp 47
6.1.1 First Contact 47
6.1.2 Learning Lisp makes you a better programmer 48
6.1.3 Interactive development environment 49
6.2 KPAX and Web Applications 50
6.3 End Conclusion 51

Chapter 1

Introduction

1.1 Why Lisp?

These days most Computer Science students learn to program in different languages than 10 years ago. Java is the main language in many courses and especially in the workplace. On top of that, many young programmers will be coding in Java for many years to come, there seems to be only one language that matters.

Java programmers are used to features like garbage-collection, runtime type checking and platform independence and young programmers will consider them to be features of their “modern” programming language. Sometimes programmers will use these features to advocate Java as the better language when they hold the occasional “holy wars” between languages.

Lisp incorporates all of these features and more - even though it’s almost 40 years old - and as such is deserves attention by the very people that take them for granted. Not only that, but Lisp is the kind of language that computer scientists admire. Its power and simplicity are second to none , and at the same it’s one of the oldest languages that is still in use today.

There are several books and web pages that show you how to do things in Lisp that cannot be done or are very cumbersome to do in other programming languages. This thesis will do the opposite, it will show how you can use Common Lisp for developing the very Web Applications that we’ve come to associate with technologies like Java/J2EE and .Net.

1.2 Why web applications?

We all use web applications everyday whether we consciously know it or not. The ubiquity of web applications is not always apparent to the everyday web

user. Whenever you go to a site that serves up custom content based on your location or language you've been exposed to a web application.

Popular forms of web applications like weblogs and Wiki's are excellent examples of web applications that are used by ordinary people to publish web content without knowing HTML and other web technologies.

But more complex and advanced examples include applications that perform real-time sales and inventory management across multiple vendors, including both Business to Business and Business to Consumer e-commerce, workflow and supply chain management, and web front-ends for legacy applications.

Web technology is interesting for these applications because the end-user already knows the basics of the client application : the use of a web browser and the use of the internet. On top of that is the internet a cross platform medium. The end user can be on a Macintosh, PC or PDA and still use the same application and get access to the same data on every platform.

A typical example of this huge benefit is the popularity of web based email systems; they enable you to access your email anywhere there's a device with internet and a browser.

Chapter 2

Lisp

2.1 History and motivation

Lisp is more than 40 years old, and has evolved a lot as a programming language through the years. A concise summary of the most important events that shaped Lisp into the language we use today is in order to get a clear picture of what makes Lisp truly remarkable.

2.1.1 From paper to version

The beginnings of Lisp can be traced back to the late 1950's out of the need for artificial intelligence programming. The first major academic workshop on A.I. was held at Dartmouth[1] in 1956. At this workshop John McCarthy became aware that an algebraic LISt-Processing language would be very useful. Hence the name Lisp.

In 1958 McCarthy was using recursion in conjunction with conditional expressions in his definition of List-Processing functions and became more and more convinced of the power of the combination of these two constructs. His earlier work had touched on the subject of List-Processing in the form of a set of subprograms for use with FORTRAN[2] programs. But FORTRAN didn't permit recursive definition, so it became apparent that a new Language was needed.

The desire for an algebraic List Processing Language spurred the 1960 paper *"Recursive Functions of Symbolic Expressions and Their Computa-*

[1] Summer Research Project On Artificial Intelligence held at Dartmouth College in Hanover, New Hampshire USA. See http://artema.fopf.mipt.ru/ai/dartmouth.html

[2] Acronym for FORmula TRANslation. Fortran was the first ever high-level programming language

tion by Machine, Part I" [3] in which he presented his ideas. The paper's introduction stated that McCarthy believed that recursive definitions have advantages both as a programming language and as a vehicle for developing a theory of computation.

With that motto he defined a universal Lisp function that could interpret any other Lisp function, and as such wrote a Lisp interpreter in Lisp. One of the key ideas of Lisp is to use a simple data structure for both code and data. Programs in Lisp are represented as list structures, and interpreted as such. This list notation is called S-expression notation.

The first working Lisp system was the result of a translation of that universal function into assembly language and the link with list-handling subroutines. A prototype interactive Lisp system was demonstrated in 1960 and was one of the earliest examples of interactive computing.

Lisp spread rapidly to a variety of computers during the period of 1960 to 1965 either by bootstrapping from an existing Lisp on another computer or by new implementations. The majority of implementations were almost identical and had a small core (that was hand-coded in assembly) and a compiler; the rest was written entirely in Lisp and compiled in Lisp.

2.1.2 Lisp Machines

Lisp implementations suffered from address-space constraints and limitations of the computer architectures of that time and in the early 1970's the first ideas for machines built specifically to run Lisp emerged. Lisp Machines were born. In essence, a Lisp Machine was :

- a single-user minicomputer-class[4] machine
- specially micro-coded[5] to run Lisp
- capable of supporting a complete Lisp Development Environment in memory.
- running Lisp as the operating system.
- workstation-like with bit-mapped graphics and a mouse.

[3] published in Communications of the ACM, April 1960

[4] obsolete term for a multiprocessing system capable of supporting from 4 to about 200 users simultaneously that were smaller than mainframes and bigger than single-user minicomputers. Equivalent of modern "servers"

[5] microcode is the controlling program of a microprocessor, contains processing techniques to decode the assembler level macroinstructions and then to send control sequences to the microprocessor,

Figure 2.1: A Lisp machine with bit-mapped displays

In the years 1974 through 1980 there were several research efforts into Lisp Machines. At MIT[6] the CONS[7] and CADR[8] Lisp Machines were based on ideas from the Xerox PARC ALTO microprocessor. They were designed to have good performance, a non-prohibitive cost[9], better storage than previous computers, garbage collection, a largely Lisp-coded implementation and a modern programming environment with bit-mapped displays. These research designs where later commercialized by two companies : Lisp Machines Inc. and Symbolics. Both companies produced CADR clones and the Symbolics 3600 line became the industry leader of Lisp machines for 5 years straight.

Other significant research into Lisp machines was conducted by Xerox with the Alto machine. The machine proved to be underpowered for the Lisp environment it used, so it was not widely accepted by its users as a

[6]Massachusetts Institute of Technology. Independent, coeducational university centered on science and technology, USA

[7]list construction operator in Lisp

[8]function that returns the second member of a list

[9]less than 70,000 per machine

Lisp system. The Alto was more successful in the production of the first Smalltalk[10] environment. Xerox released three other Machines called the Dorado, Dolphin and Dandelion which were collectively called "The D-Machines". Although not strictly Lisp Machines, these computers hosted InterLisp environments.

Lisp Machines were responsible for further refinement of the language itself, adding more features and complexity because of the adoption of the machines by its users. The language itself was the reason why Lisp Machines were popular and when general-purpose hardware became good enough to run Lisp, the companies that produced Lisp Machines ran into difficulty.

Lisp machines predated Unix workstations and spearheaded bit-mapped displays and mouse driven interfaces.

2.1.3 ANSI Standardization : Common Lisp

The late 1960's were characterized by experimentation with implementation strategies, and since there was no formal standards process, several major Lisp dialects spawned over the years leading up to the standardization of Lisp. MacLisp was the primary Lisp dialect at the MIT A.I. lab from the late 1960's until the the early 1980's. The Lisp Machine dialect was based on this dialect. InterLisp was the other major dialect during that time period.

The first effort towards Lisp standardization was made in 1969, when Standard Lisp was defined at the University of Utah – a subset of Lisp 1.5 and other dialects.

One of the most important developments in Lisp occurred during the second half of the 1970's: Scheme. Scheme is a simple dialect of Lisp whose design brought to Lisp some of the ideas from programming language semantics developed in the 1960's. Scheme stresses conceptual elegance and simplicity and as such is smaller than Common Lisp. Several features that where pioneered in Scheme can be found in Common Lisp today.

In the late 1970's object-oriented programming concepts started to make a strong impact on Lisp. At MIT, certain ideas from Smalltalk made their way into several widely used programming systems. These systems influenced the design of the Common Lisp Object System (CLOS) that would later be specifically developed for the standardization process.

In April 1981 DARPA[11] organized a "Lisp Community Meeting", in

[10] other dynamically typed object oriented programming language designed at Xerox PARC

[11] Defense Advanced Research Projects Agency, responsible for the development of new technology for use by the military

which different implementation groups got together to discuss the future of Lisp. The main goal of DARPA was to stem the development of even more dialects.

Common Lisp was designed as a description of a family of languages, in such a way that any program written in the language defined would run in any language in the family. The first public edition was published in 1984. The declared goals of the Common Lisp Group were as followed :

- Commonality: Common Lisp originated in an attempt to focus the work of several implementation groups, each of which was constructing successor implementations of MacLisp for different computers. While the differences among the several implementations will continue to force some incompatibilities, Common Lisp should serve as a common dialect for these implementations.

- Portability: Common Lisp should exclude features that cannot be easily implemented on a broad class of computers. This should serve to exclude features requiring microcode or hardware on one hand as well as features generally required for stock hardware, for example declarations. (We use the term stock hardware to describe commercially available, generalpurpose, off-the-shelf computers as opposed to computers specifically designed to support Lisp.)

- Consistency: The interpreter and compiler should exhibit the same semantics.

- Expressiveness: Common Lisp should cull the best experience from a variety of dialects, including not only MacLisp but InterLisp.

- Compatibility: Common Lisp should strive to be compatible with ZetaLisp, MacLisp, and InterLisp, in that order.

- Efficiency: It should be possible to write an optimizing compiler for Common Lisp.

- Power: Common Lisp should be a good system-building language, suitable for writing InterLisp-like user-level packages, but it will not provide those packages.

- Stability: Common Lisp should evolve slowly and with deliberation.

By 1992 a first draft proposed ANSI standard was published. The X3.226 ANSI Common Lisp standard was finalized in December 1994, and formally published about a year later. ANSI Common Lisp was the first object-oriented language to get an ANSI certifcation.

2.2 Language design and Features

Lisp is pretty different in concept and syntax from modern and widely used languages such as Java, Perl and C++. Some differences are purely design concepts which make Lisp unique. On the other hand are there Lisp features that can be found in these newer languages.

It is the combination of design ideas and specific language features that make Lisp attractive as a programming language. Some people say Lisp enables you to write beautiful programs instead of just writing the right program.

2.2.1 Design ideas

Functional Programming Style

Programming languages are often divided into two classes. *Imperative Languages*, which depend heavily on an assignment statement and are basically collections of mechanisms for routing control from one assignment statement to another. *Functional Programming* means writing programs that work by returning values, instead of modifying things.

In Lisp you "define a function" instead of "running a program", and you "apply a function to its arguments". Almost all computation is performed by applying functions to arguments.

For example, consider the task of calculating the sum of the integers in an array. In an imperative language such as Java, this might be expressed using a simple loop, repeatedly updating the values held in an accumulator variable total the array-index i:

```
int integers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total = 0;
for (int i = 0; i < 10; i++) {
  total += integers[i];
}
```

In a functional language, the same program would be expressed without any variable updates. In Lisp we would think of the list as a whole, rather than iterating over their elements. The Lisp function *reduce* deals with the entire data structures as units. (although, of course, the machine will have to process all of the individual elements) :

```
(reduce #'+ '(1 2 3 4 5 6 7 8 9 10))
```

Since we do not have to think about the parts of data structures, we are programming at a higher level of abstraction.

Lisp relies heavily on the concept of functions and arguments, and this is apparent in another design idea : *a function type.*

In Lisp, functions are first class objects– they're a data type just like integers, strings, etc, and have a literal representation, can be stored in variables, can be passed as arguments, and so on. For example, the calculation above can be expressed in this way by using the *apply* function in Lisp. It takes a function as argument and applies it to the rest of the arguments it received :

```
(apply #'+ 1 2 3 4 5 '(6 7 8 9 10))
```

Both Lisp expressions produce the same value of 55; the first expression has a list with the numbers from 1 to 10 as arguments, the second has the + function, numbers 1 to 5 and a list containing the numbers 6 to 10 as its arguments.

Programs are represented as Lists

Everything in Lisp is expressed in terms of symbolic expressions (which are called S-expressions) of two basic types: atom and lists. Atoms are primitive data objects like numbers, strings and characters etc. Lists consist of zero or more S-expressions inside a pair of parentheses, separated by whitespace. This example is a typical list which contains atoms and lists:

```
(1 2 3 '(4 5) 'ATOM ``string'')
```

So an S-expression is in essence an atom or a list of zero or more S-expressions. Whenever we write code in S-expressions, we use prefix-notation.[12] The first element in the list is the name of a form, i.e. a function, operator and the rest are arguments. An example of prefix-notation is the following expressions which subtracts two numbers:

```
(- 10 5)
```

Lisp maintains a *Uniform Representation of Data and Code* and as such the expression :

```
(car '(1 2 3))
```

[12]prefix-notation writes the operator before its operands : 1 + 2 is written as (+ 1 2)

Conditionals

A conditional is an if-then-else construct. We take these for granted now. They were invented by McCarthy in the course of developing Lisp. Following example is a typical conditional :

```
(if (evenp 2)
    'I-AM-EVEN
    'I-AM-ODD)
```

Java has this construct in the form of the *conditional operator* that functions like an if-else statement embedded within an expression :

```
hello = (hello == null) ? "Hello" : "Goodbye";
```

Recursion as Iteration

Recursion existed as a mathematical concept before Lisp of course, but Lisp was the first programming language to support it. In earlier versions of Lisp as well in dialects like Scheme it was the only means to iterate.

A new concept of variables

In Lisp, all variables can be thought of as pointers to a value. Values are what have types, not variables, and assigning or binding variables have the semantics of copying pointers, not what they point to.

Interactive programming

The whole language is always available. There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

Having an interactive environment lets you correct bugs instantly instead of going through endless cycles of editing, compiling, deploying and testing. But even more important is the ease with which you can test programs. You dont have to compile a program and test the whole thing at once. You can test functions individually, and they become the test as well as the implementation.

2.2.2 Language features

Garbage Collection

Lisp has built-in memory management, also known as garbage collection. A programmer does not need to explicitly free dynamically allocated memory and can concentrate on writing the program instead of how the program uses the system it will run on.
Garbage Collection:

- detects garbage during program execution.
- is invoked when more memory is needed.
- is a decision made by run-time system, not program

Dynamically typed

Lisp has a flexible approach to types. As mentioned before, values have types and not variables. In most other languages the opposite is true : you can't use a variable without specifying its type. In Lisp, any variable can hold objects of any type. This means that type is associated with the object/data and not the variable name.

This is called "manifest typing". Lisp enforces runtime type-checking instead of compile-time type-checking like for example C does. This enables a flexible way of programming, but has a runtime cost associated with it.

A simple addition would have to look at the types of its arguments and decide what kind of addition to do at run-time. If we know ahead of time what type of arguments are supplied, we can declare them, and the compiler will hard-wire the addition just as in C. This way, speed will not be an issue when we use Lisp instead of statically typed languages.

Object Oriented : CLOS

The *Common Lisp Object System* offers full object-orientation with classes, subclassing, multiple inheritance and more advanced concepts such as multi-methods and before-, after- and around methods. CLOS is a set of operators for object-oriented programming, but are an integral part of the language and are written in Lisp.

CLOS features :

- built-in classes integrated with type system
- new classes definable, also integrated with types

- classes may inherit from multiple ancestors
- class variables as well as instance variables
- automatic combination of four method types
- methods may specialise on any number of arguments

Interpreted and Compiled

Early Lisp implementations where implemented by interpreters, but Common Lisp is the same language compiled as it is interpreted. There are hardly any implementations of Lisp out there that are purely interpreted systems. All serious implementations include a compiler that produces machine code directly. So Lisp also usually distinguishes between compile-time and runtime.

Still, there are some differences in this regard to so-called ”compiled languages” like C. Most of the time, you don’t explicitly compile a Lisp program and execute it afterwards. Instead you interact with a Lisp development environment that compiles Lisp code on the fly. So Lisp is closer in spirit to a kind of just-in-time compiled language.

Lisp gives you incremental compilation. This means that you can compile one function at a time and be ready to run your program instantly – there is no linkage step. This means that you can make lots of changes quickly and evaluate them for their effect on the program.

Specialized Data Structures and Types

Lists were the first data structures in Lisp. Common Lisp has other data structures that are commonly found in other languages such as arrays (including vectors and strings), structures, sequences (which are lists or vectors) and hash-tables.

Common Lisp also supports advanced numeric types. The arithmetic package includes unlimited size integers, fractions, complex numbers, and a complete floating point library. Conversion between numeric types occurs automatically.

Multiple Values

In Lisp it is possible to write functions that return multiple values, and handle them correctly. This is often used to collect multiple values into a list that can be processed later.

2.2.3 An example Lisp session

You interact with the Lisp system through a built-in piece of code called the top loop, which repeats three simple steps for as long as you run the Lisp system:

1. Read an expression (you provide the expression).
2. Evaluate the expression just read.
3. Print the result(s) of the evaluation.

This is also called the "read-eval-print" loop, and *?* is the typical prompt of this loop. What follows is a typical Lisp session in which we try to show what Lisp can do.

Numbers and string expressions are evaluated to its values (in these cases the value is the same of what is entered):

```
? 666
666
? "this is a string"
"this is a string"
? x
> Error: Unbound variable: X
```

Lisp tried to evaluate the value of x but told us it was unbound. The following assignment associates a value with the symbol x :

```
? (setf x 100)
100
? x
100
? 'foo
FOO
```

If we want to tell Lisp not to evaluate a symbol or expression we need to quote it. This way the symbol *foo* evaluates to itself and not its value. Think of quoting as making something a constant. A more useful expression like an addition and multiplication :

```
? (+ 666 1701 1138)
3505
? (* 10 (+ 10 20))
300
```

This is a good example of evaluation and a function call. First the arguments are evaluated and then the values of the arguments are passed to the function named by the operator(s) (in prefix-notiation). In this example the arguments evaluate to themselves or the evaluation of the expression they represent.

Functions are defined with defun, which usually takes three arguments : a name, a list of parameters and one or more expression that will make up the body of the function. It's the body that will produce the value of the function :

```
? (defun factorial (n)
     (if (zerop n)
         1
         (* n (factorial (1- n)))))
FACTORIAL
? (factorial 0)
1
? (factorial 5)
120
? (time (factorial 666))
(FACTORIAL 666) took 11 milliseconds (0.011 seconds) to run.
Of that, 10 milliseconds (0.010 seconds) were spent in user mode
         0 milliseconds (0.000 seconds) were spent in system mode
         1 milliseconds (0.001 seconds) were spent executing other OS processes.
 208,024 bytes of memory allocated.
```

This function is an example of a recursive function that computes the factorial of an integer. The body of the function is composed of an if-expression.

Since Lisp has great support for lists as data structure, you use it very often to iterate over members of a list and do something with the members of the list. This simple function demonstrates this :

```
? (defun list-processing (list)
    (let ((new-list '()))
      (dolist (member list)
        (push member new-list))
      new-list))
LIST-PROCESSING
? (list-processing '(666 1701 1138))
(1138 1701 666)
```

The function includes a variable declaration : *(let ((new-list '())))*, the iteration in the *(dolist ...)* expression which pushes each member of the original list on the variable *new-list*. The last *new-list* is the return expression.

Processing data with this kind of iteration is very flexible and saves a lot of index-subscripting iteration. For example, if we wanted to multiply the members of 2 lists pairwise we can just "map" the * function to the two lists :

```
? (mapcar #'* '(10 20 30) '(100 100 100))
(1000 2000 3000)
```

These examples maybe difficult to understand at first, but the concepts here are commonly used in almost every Lisp program and show us a glimpse of how powerful and unique Lisp can be.

Figure 2.2: Emacs Lisp session, running OpenMCL Common Lisp

Chapter 3

KPAX Web Application Framework

KPAX is a Common Lisp framework to build web applications. It provides session management, security, web actions and Lisp Server Pages as means to present dynamic content to the user.

3.1 Web Applications

3.1.1 What are Web Applications?

A web application is basically a client/server software application that uses the HTTP protocol as its primary means of communicating with users and/or other systems. The web browser plays the typical client role for the user and communicates with the other tiers of the application. Most web applications are made up of three different tiers :

- *The presentation tier* is responsible for presenting data to a user or system. Browsers like Mozilla and Netscape Navigator reside in the presentation tier.
- *The application tier* is the "engine" that performs the business logic; processing user input, making decisions and producing the data the presentation tier needs. Technologies like CGI, Java Servlets and Server Pages, PHP are all used for the application tier, often deployed in products like JBOSS, Tomcat, Apache or WebLogic.
- *The data tier* stores the data that application tiers uses and acts as a repository for both temporary and permanent data. Databases like

PostgreSQL, XML or flat files are commonly used in various forms of web applications to store data.

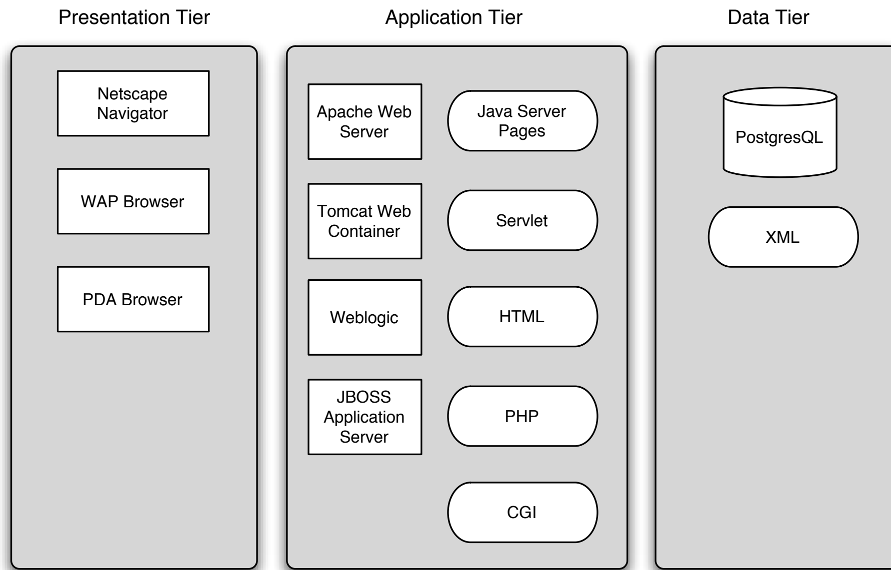

Figure 3.1: Web Application Tiers

3.1.2 KPAX Web Applications

In KPAX, a Web Application is a collection of HTML pages, Lisp Server Pages and web actions that are defined and published in a container context of the underlying web server.

Each defined Web Application has its own business logic functions, web actions and security configuration. This enables KPAX to host several web applications at once and independently stop and start them.

3.2 Design and Features

KPAX was designed to provide everything a typical web application needs with common design ideas like the separation of business logic (web actions) and presentation (Lisp server pages), session management, security and web application definitions.

These features can also be found in popular Java implementations like Jakarta Tomcat and Jetty.

3.2.1 Session Management and Security

Web application developers do not have to worry about Session Management and Security with KPAX. It provides web applications with always enabled session management and a security infrastructure.

Session Management

Session Management is all about remembering state. Sometimes an application needs to know what a user did or what access rights a user has. *HTTP* is by design a stateless protocol. The implication is that web applications do not have information about previous *HTTP* requests by the same user.

KPAX has built-in session management and handles that information through the use of session objects and passes that cookie object back and forth with each request. This cookie contains a session-id. KPAX maintains a list of active sessions and their session-id. With every request a browser makes, state information is sent back to the server.

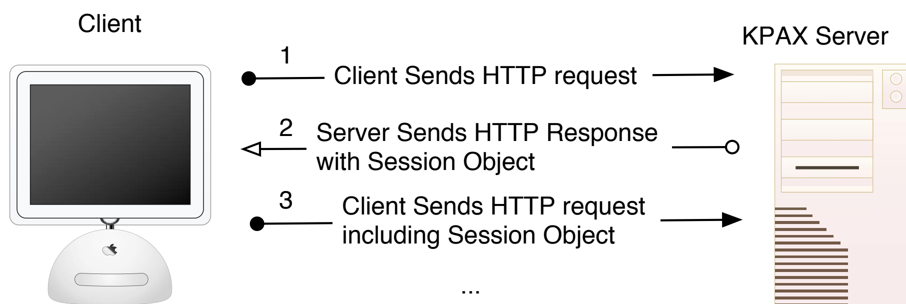

Figure 3.2: Session Object Flow

Every request results in a check for an active session for the session-id in the cookie, and the creation of a session if none is found. KPAX session management is always enabled.
A session gets automatically terminated if it's idle for certain amount of time.

Security

Web applications can be secured in the KPAX framework. Security is by default enabled, and web-users need to be listed in the web application definition. Whenever a secured page, action or file is accessed KPAX will

check if the user was authenticated and if not will show a login form. Security can optionally be disabled per web application.

3.2.2 Web Actions

Web actions are a popular used design paradigm in web applications. In a typical situation a user will fill out a html form and a HTTP request with the contents is sent to a web action which will process the data and forward the processed data to another web page.

KPAX implements web actions by a *webaction handler*. This entity is in essence the equivalent of a *Java Servlet* that decides what Lisp function to execute. KPAX keeps a list of mappings between web actions and Lisp functions. For example : someone accesses the web app at :
http://server:2001/factorial/webaction?action=compute-factorial.

The KPAX *webaction handler* knows it has to handle that request differently (because *webaction* is published entity) and resolves the *show-result* action to its function *compute-factorial* and executes it. Most web actions will forward to a page designed to present processed data.

3.2.3 Lisp Server Pages

Lisp Server Pages are the Lisp equivalent of *Java Server Pages*. They enable a developer to write pages with conventional *HTML* markup and include special tags with Lisp code. When a user asks for a LSP, the server executes the Lisp code and merges the results with the static parts of the page and sends the dynamically composed page back to the browser.

LSP pages are converted into one contain Lisp expression, which is then compiled and cached. HTML markup gets converted to Lisp print expressions, the special LSP tags are removed and the expressions within included.

If the source file containing the LSP code is modified, the next time a request is made for that page the code will be recompiled and recached.

All published LSP files in a KPAX web application are defined in the Web application definition.

3.2.4 Web application definition

KPAX facilitates the easy definition of a web application, and handles all steps to deploy the defined web application and its components based on that definition.
A web application definition consists of the following properties :

- **name**
  The name of the web application. Used in start and stop operations.

- **root**
  A file path location where all files are located (a directory). All other file references within the web app are resolved relative to this root location.

- **prefix**
  The URL context of the web application.
  For example, a web application with prefix "factorial" will be available at the url of *http://server:2001/factorial/*

- **index**
  The name of the index page which must be shown when a user browses to the web application at its prefix. (*http://server:2001/factorial/*)
  This is the equivalent of *Welcome Files* in Tomcat

- **lsps**
  A list of List Server Page files that KPAX serves for this web application.

- **actions**
  The list of actions for this web application, expressed in mappings of actions to functions. The *webaction handler* uses this mapping to execute the right Lisp functions.

- **users**
  A list of instances of *web-user* specifies the users that the web applications knows. All identification and authentication is performed in reference to this list. A web-user has an id (integer), short-name, full-name and a password.

3.3 Examples

Both the *Hello World* and *Factorial* web applications are simple examples, but are made up of the same buildings blocks as more sophisticated web applications:

- They have a main *.lisp* file which contains the *Web Application Definition*, *web actions* and supporting Lisp code.

- Contain several LSP files to submit data and represent data they receive from *web actions* and other files like pictures or *Cascading Style Sheet* files.

3.3.1 Hello World

Web Application Definition

The *helloworld.lisp* file only contains a package statement, and the *Web Application definition* in the *(defwebapp :helloworld ...)* expression.

```
;;; file : helloworld.lisp
(in-package :kpax-user)

(defwebapp :helloworld
  (:root "/Users/tarkin/cvs/svc/kpax/example/helloworld/")
  (:prefix "helloworld")
  (:index "index")
  (:files (list "kpax-movie-poster.jpg"))
  (:lsps (list "kpax-movie"))
  (:actions '())
  (:users (list (make-instance 'web-user
       :id 101
       :short-name "guest"
       :full-name "Guest User"
       :password "trustno1"))))
```

As you can see, the *Hello World* application only serves two LSP files : the index page (*index.lsp*) and one other LSP (*kpax-movie.lsp*), but has no webactions defined.

Index page

The *index.lsp* file :

```
<html>
<head>
 <title>KPAX HelloWorld</title></head>
<body>
<% (standard-header "KPAX HelloWorld" request entity session) %>
<p>Good <% (html (:princ (get-daypart))) %>!
```

```
<p>You can find out more about KPAX on the web site
<a href="http://kpax.beta9.be">http://kpax.beta9.be</a></p>
</body>
</html>
```

This is a typical LSP page. *HTML* markup and text mixed up with special *tags* that indicate that the Lisp code needs to be processed between the tags. In our example above the line :

```
<p>Good <% (html (:princ (get-daypart))) %>!
```

dynamically creates a special greeting based on what part of the day it is. Everything between the

```
<% %>
```

tag construction is executed by the LSP framework.

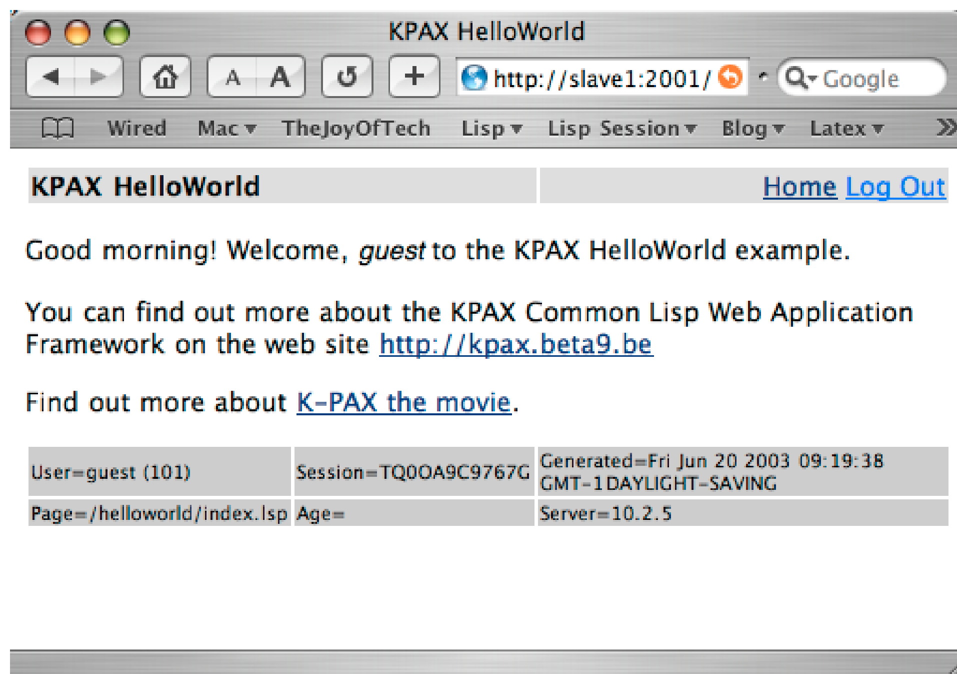

Figure 3.3: generated Hello World Index.lsp

3.3.2 Factorial

The *Factorial* web application follows the same design as *Hello World* but implements one web action which sends its result to an extra LSP file.

Web Action

The definition of the web action and files in the *(defwebapp ...)* expression :

```
(defwebapp :factorial
  (:root "/Users/tarkin/cvs/svc/kpax/example/factorial")
  (:prefix "factorial")
  (:index "index")
  (:lsps (list "result"))
  (:actions (list (cons "compute-factorial" 'compute-factorial)))
  (:users (list (make-instance 'web-user
                               :id 101
                               :short-name "guest"
                               :full-name "Guest User"
                               :password "trustno1"))))
```

The implementation of the web action :

```
(defun compute-factorial (request entity)
  (let* ((number-string (request-query-value "number" request))
         (number (parse-int number-string nil)))
    (fac nil)
    (if (not number)
        (display-message request entity
                         "Error"
                         "Parse Exception")
      (setf fac (fac number)))
    (setf (request-attribute request :number) number
          (request-attribute request :factorial) fac)
    (forward-to-lsp "/result.lsp" request entity)))
```

The HTML form on the *index.lsp* file :

```
<form action="webaction">
  <input type="hidden" name="action" value="compute-factorial">
  <input type="text" name="number" value="0">
  <input type="submit" value="Go!">
<form>
```

Request query values and request attributes

This webaction makes use of two features : *request query values* and *request attributes.*

Request query values are key-value pairs on a HTTP request which carry values of a submitted HTML form (in this case coming from a form on the index page). This data is typically appended as sequence of characters appended to the request URL in what is called a *query string* :

```
http://slave1/fact/webaction?action=compute-factorial&number=6666
```

The values are always strings. If the form is submitted using a POST method, these values are passed as a part of the HTTP request body.

The web action assigns the value of the "number" query value to a variable and parses it into an integer.
Processing the query values in the web action :

```
(let* ((number-string (request-query-value "number" request))
 (number (parse-int number-string nil)))
```

Request attributes are key-value pairs in the body of HTTP request object. The values of request attributes can be anything. In this example the web action computes the factorial for a number and sets a request attribute with key *factorial* and with a value the computed factorial :

```
 (setf (request-attribute request :number) number
   (request-attribute request :factorial) (fac number))
```

The web action forwards this request to the *result.lsp* file and *result.lsp* processes the request attributes from the webaction. The forward to *result.lsp* :

```
(forward-to-lsp "/result.lsp" request entity)
```

Processing the request attributes in *result.lsp* :

```
<% (let ((number (request-attribute request :number))
         (factorial (request-attribute request :factorial))) %>
The factorial of <% (html (:princ number)) %> is
<% (html (:pre (:princ (long-number-to-string factorial)))) %>
<% ) %>
```

KPAX Factorial Result

http://slave1:2001/factorial/webaction?acti

Wired Mac TheJoyOfTech Lisp Lisp Session Blog Latex Dump Psillog

KPAX Factorial Result Home Log Out

The factorial of 666 is

```
101063205684078149339082270812987645175758239832414541134042080 7
```

Computation took 0.01ms.

Compute another factorial

User=guest (101) Session=1C5UPKVIGDB4G Generated=Fri Jun 20 2003 09:29:56 GMT-1DAYLIGHT-SAVING

Figure 3.4: computed Factorial, result.lsp

Chapter 4

RSS Syndication

The *RSS Syndication* web application enables users to manage and view a collection of RSS feeds in their browsers. It is in fact a web application that presents a web page to the user composed of content of other specified websites.

4.1 What is RSS?

RSS stands for Rich Site Summary and is an XML document and format for syndicating news and the content of news-like sites, including major news sites, news-oriented community sites and personal weblogs. But it's not just for news. Pretty much anything that can be broken down into discrete items can be syndicated via RSS: the ”recent changes” page of a wiki, a changelog of CVS checkins, even the revision history of a book. Once information about each item is in RSS format, an RSS-aware program can check the feed for changes and react to the changes in an appropriate way.

A sample RSS XML file (also called RSS feed) :

```
<rss version="0.91">
  <channel>
    <title>XML.com</title>
    <link>http://www.xml.com/</link>
    <description>XML.com features a rich mix of information
 and services for the XML community.</description>
    <language>en-us</language>
    <item>
      <title>Normalizing XML, Part 2</title>
      <link>http://www.xml.com/pub/a/2002/12/04/normalizing.html</link>
```

```
      <description>In this second and final look at applying relational
 normalization techniques to W3C XML Schema data modeling,
 Will Provost discusses when not to normalize, the scope of
 uniqueness and the fourth and fifth normal forms.</description>
    </item>
    <item>
      <title>The .NET Schema Object Model</title>
      <link>http://www.xml.com/pub/a/2002/12/04/som.html</link>
      <description>Priya Lakshminarayanan describes in detail
 the use of the .NET Schema Object Model for programmatic
 manipulation of W3C XML Schemas.</description>
    </item>
    <item>
      <title>SVG's Past and Promising Future</title>
      <link>http://www.xml.com/pub/a/2002/12/04/svg.html</link>
      <description>In this month's SVG column</description>
    </item>
  </channel>
</rss>
```

4.2 Design and Features

The web application allows users to view a summary of the updated content of their favorite sites in the browser. The web application was developed using the same techniques as in the earlier examples (web actions, forms, query values, request attributes) and the goal was to use KPAX for a more sophisticated type of web application which also uses technologies other than web technology.

In essence the *RSS Syndication* has the following features :

- **Management of feeds**
  Adding, editing and deleting user specified feeds through a web user interface.
- **Feed persistency**
  without the use of a database.
- **XML parsing**
- **HTTP connectivity**

The web application uses HTTP connectivity functions built in the server to retrieve the XML files.

4.2.1 Feed persistancy

A user can add and delete feeds in the web interface, and these operations must be remembered by the web application. In this example we need to save the URL for a feed, the title and what user a feed belongs to.

This is a relatively simple situation and a database would be overkill. A flat file seems sufficient in keeping our information saved and up-to-date. Instead of conjuring up a format to write the data into a text file, we will use the flexibility of Lisp to lends us a hand.

All we have to do is define a function *add-feed-for-user* that can be used to add a feed for a user - passing the username, feed-title and feed-url as arguments - and save function calls to this function into a *.lisp* file whenever a feed is added or deleted. Then we have a persistent Lisp representation of all feeds and just have evaluate those expressions when we start our web application.

The *userfeeds.lisp* file (excerpt):

```
(add-feed-for-user "guest"
                   "OSOpinion"
                   "http://www.osopinion.com/OSOlinks2.xml")
(add-feed-for-user "guest"
                   "MacSlash"
                   "http://www.macslash.com/macslash.rdf")
(add-feed-for-user "guest"
                   "The Register"
                   "http://www.theregister.co.uk/tonys/slashdot.rdf")
```

The web application actually holds a hash-table in memory as a global variable with all feeds for a certain user-id. The function *add-feed-for-user* pushes another hash-table (which uses feed-title and feed-url as key and values) as a value onto that *userfeeds* hashtable.
The *add-feed-for-user* function :

```
(defun add-feed-for-user (userid feed-title feed-url)
  (if (gethash userid *user-feeds*)
      (push feed-url (gethash feed-title
                             (car (gethash userid *user-feeds*))))
     (let ((new-feed-hash (make-hash-table :test #'equal)))
```

```
       (push feed-url (gethash feed-title new-feed-hash))
       (push new-feed-hash (gethash userid *user-feeds*)))))
```

The *write-add-feed-calls* function that will write a call to a given stream for each member of the feeds hashtable :

```
(defun write-add-feed-calls (stream)
  (maphash #'(lambda (key value)
               (let ((userid key)
                     (feeds-hashtable (car value)))
                 (maphash
                  #'(lambda (feed-title feed-url)
                      (format stream
                              "(add-feed-for-user \"~A\" \"~A\" \"~A\")~%"
                              userid feed-title (car feed-url)))
                  feeds-hashtable)))
           *user-feeds*))
```

4.2.2 XML Technology

To parse the RSS XML feeds an event-based XML parser written in Lisp was used. This parser was written as part of an implementation of XML-RPC in Lisp by Sven Van Caekenberghe.[1].

Whenever a user logs into the application, the XML file is retrieved through a HTTP connection and parsed into a list representation of the XML tree (called *lxml*) (excerpt):

```
? (get-feed-for-url "http://www.osopinion.com/OSOlinks2.xml")
((:|rss| :|version| "0.91")
 (:|channel|
   (:|title| "osOpinion")
   (:|link| "http://www.osopinion.com")
   (:|description| "Tech Opinion commentary for the people, by the people.")
   (:|language| "en-us")
   (:|image| (:|title| "osOpinion")
   (:|url| "http://www.osopinion.com/art/OSOlogotext.gif")

.....)))
```

[1]http://homepage.mac.com/svc/

These lxml lists are cached for 10 minutes. After that (or when a user performs the *refresh-feeds* web action) all feeds refreshed by getting them through HTTP requests.

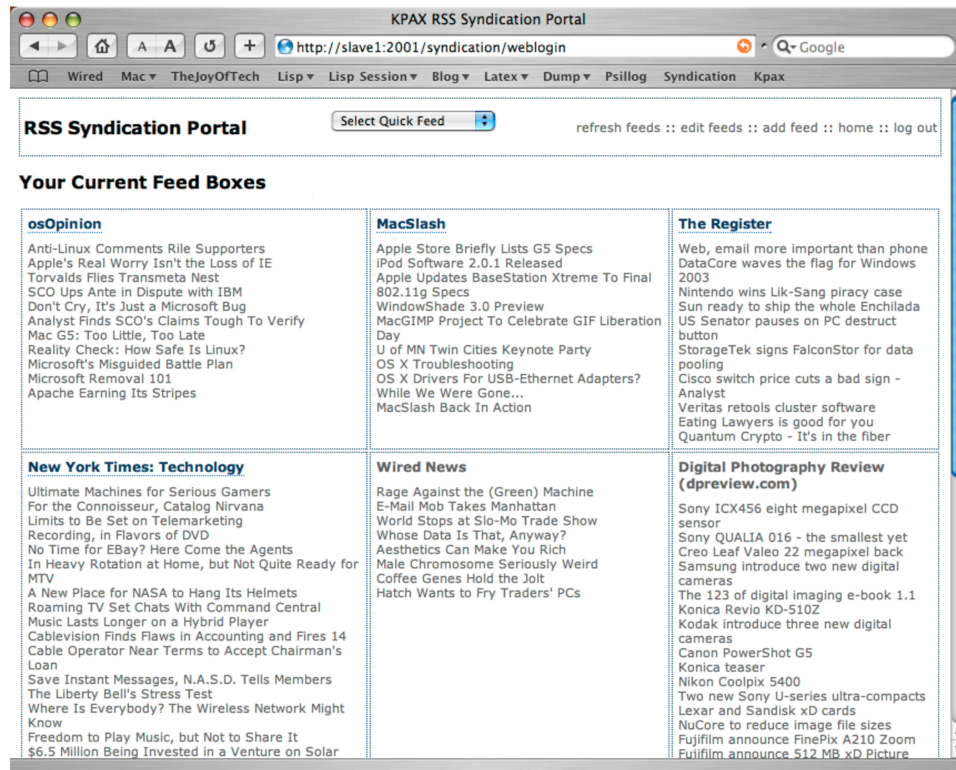

Figure 4.1: Syndication index page with several RSS feeds

Chapter 5

Psillog

5.1 The Weblog phenomena

5.1.1 What are weblogs?

A weblog, or blog for short, is a website or a part of a website. It is usually, but not always, run by a single person and they publish writings on the weblog a few times each day, or less often.

These writings , called “entries” or “posts”, generally appear on the front page of the weblog in reverse chronological order, that is, with the newest entry at the top of the page, with older entries progressively further down. Entries of a certain age often disappear from the front page but all entries are ususally archived on separate pages, perhaps organised by date or topic.

Early weblogs were mostly link-based, a collection of links to other articles on the web that might be interesting along with commentary and references to related articles. Most often made by people who knew how to make web pages. Modern weblogs are updated with custom content management systems to enable easy publishing and more direct communication with readers through features such as comments and tracking other weblogs who reference your weblog.

Weblogs are often called “web journals”, “online diaries” or “news sites”, but weblogs have one central idea : a weblog is part of communities. No weblog stands alone, they are relative to each other and to the world. Most weblogs that gain audiences, connect people together using the Web through common interests and allow people to communicate through the web (as opposed to internet technologies like chatting).

These days people use weblogs to publish digital pictures they take from day to day, let readers track what they eat every day, discuss their movie-watching experiences and lots more.

5.1.2 Existing Weblog Implementations

Most implementations are web based applications where a user needs to register, and where they can manage their weblogs and post entries. Some are free, other require payment and implement more advanced feature sets.

There are several types of weblog software systems:

- Web based systems that generate the HTML files, and upload these to a conventional webserver (at other locations) as static content. (eg. Blogger[1], Blogger Pro). These systems are pre-installed on another server and a user just has to register to start blogging. Sometimes they'll provide web space to host your generated files. Often they lack comment systems because of the need of off-site comment processing.
- Online systems that generate and publish a mixture of dynamic and static content.(eg. Movable Type[2], GreyMatter[3]). These systems usually have to be installed on your own server and require some degree of knowledge about how web applications work. Most of these feature an advanced comment system because they can process comments on-site.
- Software that runs on the users computer, and generates HTML files that are uploaded to a conventional webserver as static content. (eg. Radio Userland[4]). These have the advantage that you can update your site even when you're not online.

Almost all implementations are template driven. A user selects a premade template or uploads its own and all content is processed into HTML files that use these templates. Templates can by anything from plain HTML using custom tags invented by the developers, to *Cascading Style Sheets* that operate on plain HTML markup.

There is usually a tag system to handle the entities that compose a weblog. Tags that insert time-stamps, calendars and custom navigation links can be inserted into the templates and the system will act accordingly. These tag systems differ between implementation and will usually mean there is some kind of HTML and tag parsing engine that handles the template files and supplied content to generate the finished weblog files.

More advanced weblog features are : updating your weblog with a standard email, or special desktop application through a common publishing API,data migration, multiple authors and file-uploading.

[1]http://www.blogger.com/
[2]http://www.movabletype.org/
[3]http://www.noahgrey.com/greysoft/
[4]http://radio.userland.com/

5.2 Design and Features

5.2.1 Design

The Psillog Weblog System is designed to be a dynamic, web based content management system that can a produce a dynamically generated weblog. We have to remember that it was developed to show that such an application can be built using Lisp as a language, and the KPAX Framework as the platform.

It does not feature custom tags and HTML parsing engines or remote publishing API's but uses existing Lisp features to achieve the same results : simple and quick publishing and updating of web content.

Psillog differs (or tries to implement things with Lisp features) from the existing implementations mentioned above in the following ways :

- **LSP tags instead of custom tags**
  Instead of inventing a custom tag format we use the built in LSP tag expressions. For example :
  The following tag would tell the *Movable Type* parsing engine to insert the title of an entry into the page:

  ```
  <$MTEntryTitle$>
  ```

  We tell KPAX to do the same thing using our familiar LSP tags :

  ```
  <%= (:princ (get-title entry) %>
  ```

- **Real objects instead of a tag parsing engine**
  We handle real *CLOS* objects instead relying on parsing engine to do the work. The example above used the *get-title* accessor to access the title of the entry object.

- **Functional Programming style generation of HTML**
  With Lisp it is easy to define a function that generates HTML components based on an object. The following LSP tag generates a HTML table from a list containing entry objects :

  ```
  <% (generate-entries-table entries categories) %>
  ```

- **CSS as the sole template system**
  Dynamically generated means no template processing. The LSP files and HTML markup are the templates and as such rely on *Cascading Style Sheets* to provide the layout.

5.2.2 Features

Psillog sports the following features:

- Web based Content Management System which includes adding, deleting and editing of weblog entries, categories and comments of entries.
- Dynamically generated index and archive pages, based on content created by the Content Management System
- Object Oriented design.
- PostgreSQL Relational Database data storage.
- Lisp based ;-)

5.2.3 Object Domain Model

The Object Domain model in essence has three objects. These are all implemented with the CLOS Object system and have the class *database-object* as ancestor. The *database-object* class has methods that look up what SQL statements to execute for each object in order to save, retrieve or delete it :

1. Entries
2. Categories
3. Comments

The class definition for the entry object :

```
(defclass entry (database-object)
   ((blog-id :accessor get-blog-id :initarg :blog-id :initform 1)
   (category-id :accessor get-category-id :initarg :category-id :initform -1)
   (author-id :accessor get-author-id :initarg :author-id :initform -1)
   (title :accessor get-title :initarg :title :initform "")
   (text :accessor get-text :initarg :text :initform "")
   (text-more :accessor get-text-more :initarg :text-more :initform "")
   (keywords :accessor get-keywords :initarg :keywords :initform "")
   (created-on :accessor get-created-on :initarg :created-on :initform nil)
   (modified-on :accessor get-modified-on :initarg :modified-on :initform nil)
   (modified-by :accessor get-modified-by :initarg :modified-by :initform nil))
   (:metaclass db-access-class))
```

The database column mappings for the entry object :

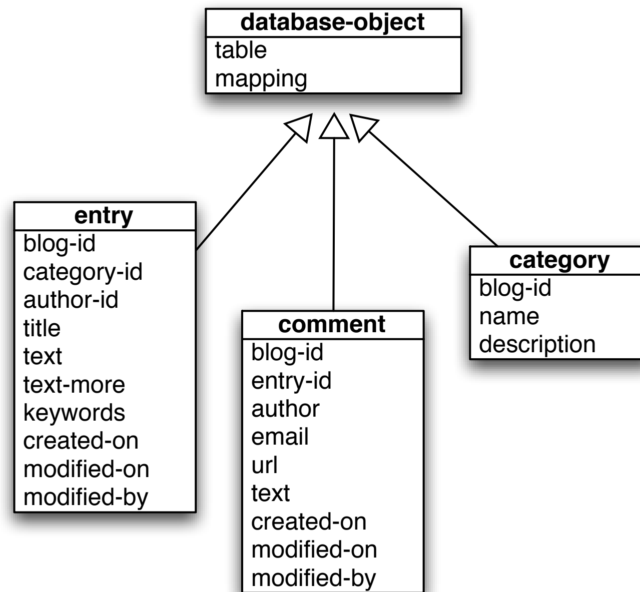

Figure 5.1: Object Domain Model

```
(defmapping category
  "category"
  ((id "id")
   (blog-id "blog_id")
   (name "name")
   (description "description")))
```

In LSP files we would typically get these objects as request attributes and use the accessors to get to the data en present it at the user, in the following example an entry object is manipulated:

```
<%= (convert-line-breaks (get-text entry)) %>
<% (unless (equal "" (get-text-more entry)) %>
 <span class="extended">
    <%= (convert-line-breaks (get-text-more entry)) %><br />
 </span>
<% ) %>
```

Webactions typically update objects with query values from a form :

```
(defun save-category (request entity)
```

```
(with-request-query-values request
 (id name description)
 ("category-id" "category-name" "category-description")
 (let ((category (if (equal id "-1")
                     (make-instance 'category)
                   (get-object-with-id 'category id))))
   (setf (get-name category) name
         (get-description category) description)
   (save-object category)
   (setf (request-attribute request :category) category
         (request-attribute request :message)
         "<font color=red>Your Category has been saved</font>"))
 (forward-to-lsp "edit-category.lsp" request entity)))
```

5.2.4 Database persistency

We use a relational database in order to provide persistency for the objects used in the web application.

PostgreSQL

PostgreSQLis a relational database that is developed by an international group of open-source software proponents known as the PostgreSQL Global Development group. This means anyone can download the source, read and modify it as longs as they publish the changes back to the developers. PostgreSQL as a database has the following features :

- Object-relational, every table defines a class and inheritance between tables is possible.
- Standards compliant, it implements most of the widely used SQL92 standard an a lot of SQL99 standards.
- Transaction processing, supporting multiple concurrent users with data protection.
- Referential Integrity through full support of primary and foreign key relationships and triggers.
- Extensive data types including dates, geometric data, Booleans...

Lisp Database Layer

Psillog makes uses of an intermediate database layer. It uses a PostgreSQL module that implements the client part of the PostgreSQL socket-level frontend/backend protocol. The module is capable of type coercions from a range of SQL types to the equivalent Lisp type.

On top of the socket-level protocol implementation a Lisp abstraction interface was written for another project, which handles data storage and retrieval of objects and execution of custom SQL code. It also provides connection pooling. Once a database spec (which is a list of the database, username, password and hostname) is created, the database can be accessed and accessed :

```
;; create db-spec
(db:setup-db :dbspec '("psillog" "tarkin" "foo" "localhost"))
;; start db-connection pooling using the spec
(db:start-db)
```

With this abstraction layer it is simple to retrieve,save and delete objects that are defined in the object model. This layer has knowledge of the database mappings defined in the object model and thus knows what SQL statements to execute.

```
;; saves a category
(save-object category-object)
;; deletes the category
(delete-object category-object)
;; retrieve the object with id ``25'' and assign it to a variable
(setf entry-25 (get-object-with-id 'entry 25))
```

Custom SQL can be executed using a similar abstraction interface, the following example will return a count of a specified object :

```
(defun count-objects (object)
  (db:with-database
   (let ((sql (format nil "select count(*) from ~a" object)))
     (util:logm :sql sql)
     (caar (db:resultset-tuples (db:sql-exec sql))))))
```

5.3 Psillog Weblog System

This is a content management part of the application. It has a different web application definition and security configuration than the actial weblog

and has a main webaction (the action equivalent of an index page). Other features include fully validated HTML and extensive use of *Cascading Style Sheets* to provide the layout templates.

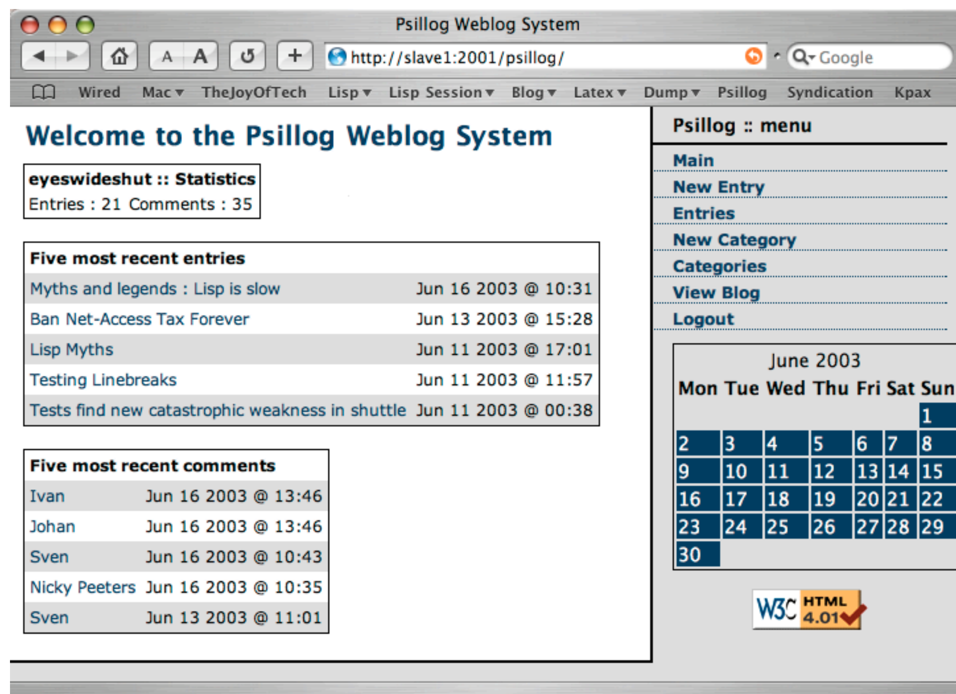

Figure 5.2: Psillog main page

The web appliction definition:

```
(defwebapp :psillog
  (:root (util:pathname-parent *load-truename*))
  (:prefix "psillog")
  (:main '("main-writer" . main-writer))
  (:files (list "psillog.css" "images/notepad.gif"
               "images/eyeswideshut.jpg"))
  (:lsps (list "new-blog" "edit-blog"
              "new-entry" "edit-entry" "entries" "entry" "delete-entry"
              "categories" "edit-category" "delete-category"
              "new-category" "psillog" "archives" "monthly"
              "preview-comment" "edit-comment" "delete-comment"))
  (:actions '(("save-blog" . save-blog)
             ("save-entry" . save-entry)
```

```
              ("save-category" . save-category)
              ("edit-entry" . edit-entry)
              ("edit-category" . edit-category)
              ("edit-comment" . edit-comment)
              ("delete-entry" . delete-entry)
              ("delete-category" . delete-category)
              ("delete-comment" . delete-comment)
              ("entries" . entries) ("categories" . categories)
              ("archive" . archive)
              ("comment" . comment)
              ("show-entry" . show-entry)))
(:users (list (make-instance 'web-user
                              :id 101
                              :short-name "guest"
                              :full-name "Guest User"
                              :password "blog")))
(:post-install 'create-initial-blog))
```

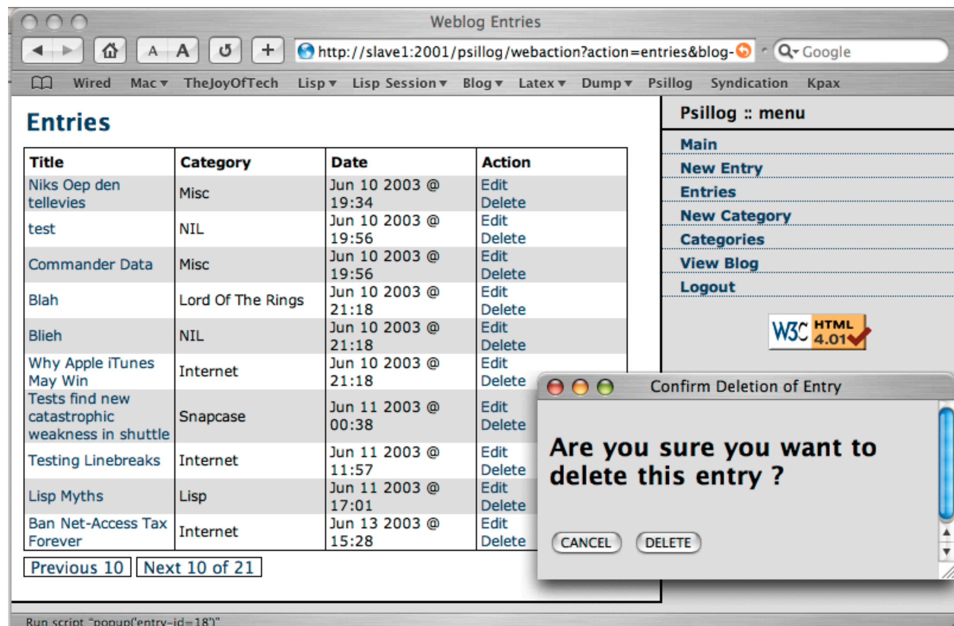

Figure 5.3: Entry management

5.4 Eyeswideshut, a Psillog powered Weblog

This is the example weblog. The main parts are the index page, archive page and comment system with previewing capabilities. The weblog is implemented as a *seperate view* on the content of Psillog, without security.

Other notable features include the seperation of content and presentation through the use of *Cascading Stylesheets*
The web appliction definition :

```
(defwebapp :eyeswideshut
  (:root (util:pathname-parent *load-truename*))
  (:prefix "eyeswideshut")
  (:insecure t)
  (:main '("main-reader" . main-reader))
  (:files (list "psillog.css" "eyes.css" "stupidbrowsers.css"
                "images/eyeswideshut.jpg" "images/notepad.gif"
                "images/psillog.jpg" "faq.html" "about.html"))
  (:lsps (list "index" "entry" "preview-comment" "archives"))
  (:actions '(("comment" . comment) ("show-entry" . show-entry)
              ("archive" . archive)
              ("show-month-archive" . show-month-archive)
              ("show-category-archive" . show-category-archive))))
```

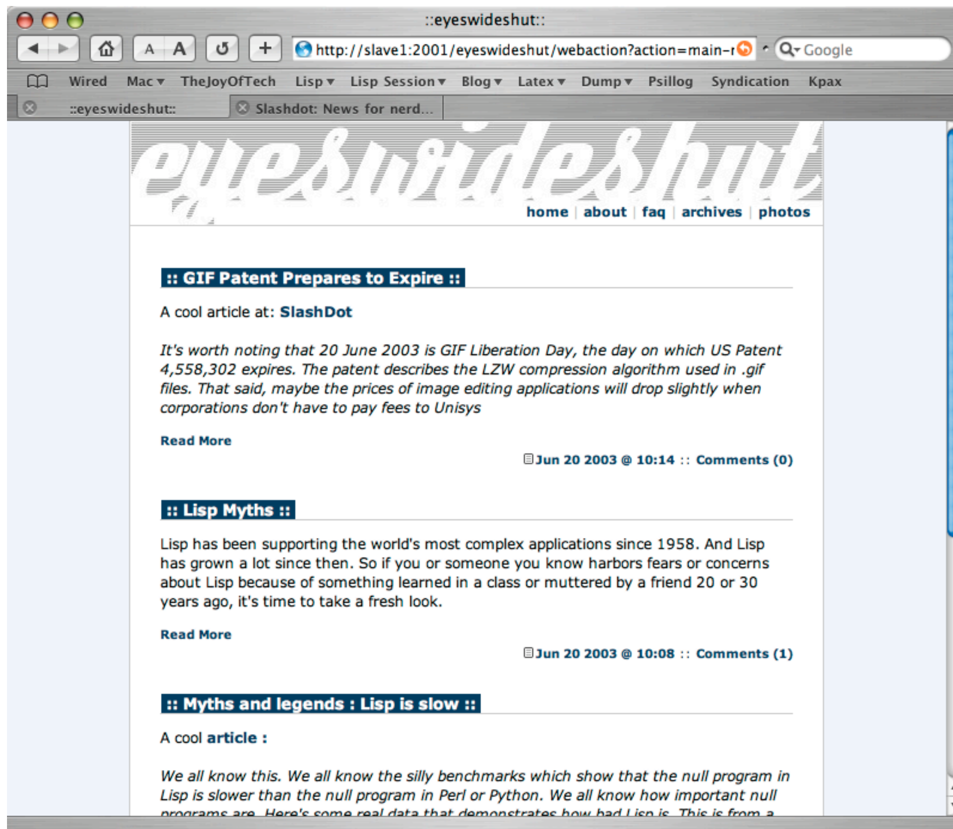

Figure 5.4: Eyeswideshut Weblog

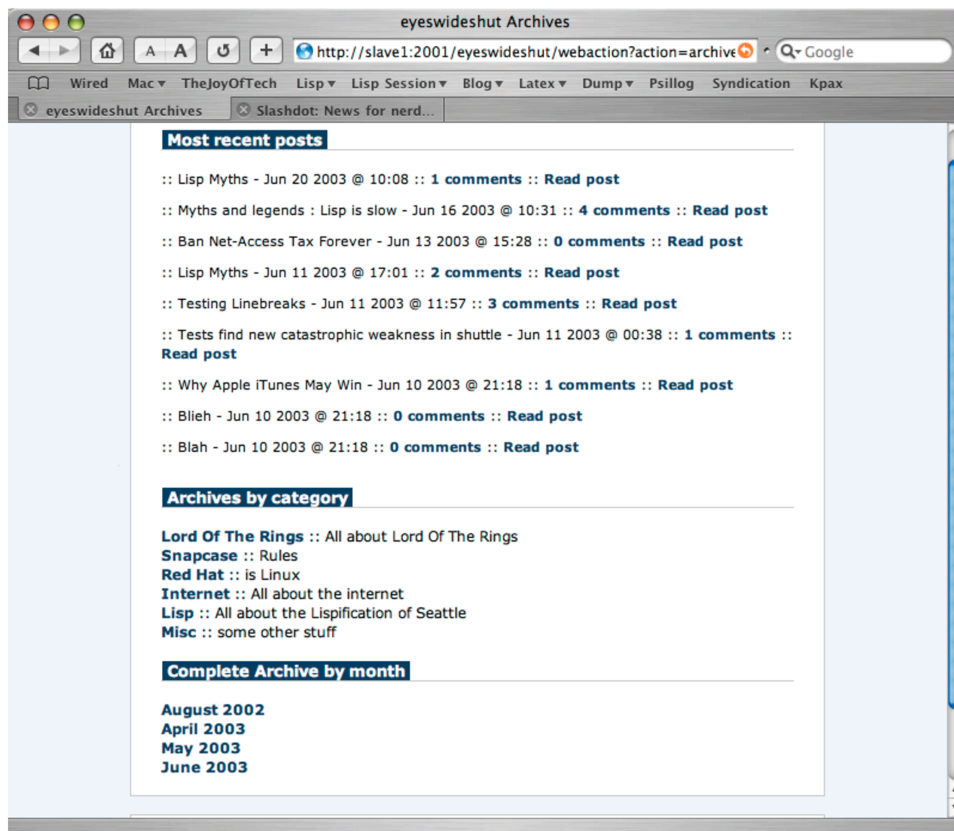

Figure 5.5: Eyeswideshut archives

Chapter 6

Conclusions

The initial goals were twofold. First, to evaluate Lisp as a programming language and alternative to Java. Second, to evaluate Lisp and the KPAX framework as a tool for web applications, keeping Java in mind since most web applications these days are written in Java using Java technologies like Servlets and Java Server pages and because of the exposure Java gets in the current Computer Science education.

Because Lisp offers several different approaches of language design, application design and development cycle, it would be interesting to see how such an approach would measure up to the traditional languages and edit-compile-run-debug development cycles.

These are my personal conclusions with these goals in mind, as a direct result of learning Lisp and using it together with KPAX as an existing Lisp based Web Application Framework (and other Lisp implementations of technologies available to Java) to write several typical web applications.

6.1 Lisp

6.1.1 First Contact

First contact with Lisp was pretty scaring. I remember looking at the code of my favorite Linux window manager *Sawmill*[1] a couple of years ago, and not understanding a thing about it. All those parenthesis felt pretty obfuscating and identifying "methods" (I was used to Java), variable assignments and iteration structures seemed almost impossible.

When I actually started to learn Lisp at the start of this project, the same feelings recurred. Nothing felt familiar to Java or other languages.

[1] url

But syntax is only that : syntax. Once I learned and understood the syntax, that hurdle went away. Not only that, but when you learn Lisp, you actually learn **why** it has the syntax it does, and **what** the advantages are. That's more than I can say about my knowledge of other languages.

Although I didn't use it extensively in the development of web applications, the ability to write Lisp programs that actually write Lisp programs makes sense. Other than that, being able to use Lisp syntax to implement persistency in the *RSS Syndication* example was fun and enlightening.

6.1.2 Learning Lisp makes you a better programmer

Eric S. Raymond, a famous Emacs hacker and free software advocate, states in *"How To Become A Hacker"* :
"LISP is worth learning for a different reason the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot."

I agree with that assumption. In the process of learning Lisp I learned a lot about how languages are designed, and what specific language features actually mean and got a better understanding of programming languages in general (types, object systems, interpreters, virtual machines, compilers, run-time, compile-time, ...)

Several features of Lisp seem so natural to me, that I wonder why some programming languages do not implement them. Mapping functions to members of lists and hash-tables are one example, passing functions as arguments (which is needed for mapping to work) is another.

In my opinion they deliver the same results as in other languages, but do it more logical, in less code and sometimes more efficient.

I really have become accustom to dynamic typing. Not having to worry about type declarations enables prototyping an idea in less time, and makes code more readable in my opinion because it isn't cluttered with lengthy lists of declarations.

And because in Lisp you usually work on compact functions and can test these functions quickly in the interactive development environment, you can catch type errors sooner, rather than later. And in the end, Lisp doesn't stop you if you need strong type declarations for design or speed efficiency.

It's just a matter of using the right tool or idea for the job with Lisp.

6.1.3 Interactive development environment

I found the interactive development cycle to be a huge benefit of using Lisp to develop web applications. No need to write special test programs to see if your *generate-entries-table* function will generate correct HTML when it is called in the LSP file, just evaluate the function in the Emacs Lisp Listener and you *can see* what HTML code is produced (excerpt):

```
? (psillog-menu)
<div CLASS="psillog-menu">Psillog :: menu</div>
<div CLASS="menu-item"><a HREF="/psillog/">Main</a></div>
<div CLASS="menu-item"><a HREF="new-entry.lsp">New Entry</a></div>
...
<div CLASS="menu-item"><a HREF="/eyeswideshut/">View Blog</a></div>
<div CLASS="menu-item"><a HREF="weblogout">Logout</a></div>
"</div>"
```

This makes debugging web applications and HTML code very easy and straightforward. Another perfect example is the development of database storage and retrieval functions. Again, you don't need to write special test programs if you want to know if your SQL is correctly processed and the right objects are returned. This way, your functions are the tests when you evaluate them.

```
? (get-object-with-id 'entry 3)
#<ENTRY 3>
? (count-objects 'entry)
21
? (get-title (get-object-with-id 'entry 11))
"Test"
```

The interactive nature of the Lisp environment means you can redefine a function in a running environment without the need of bringing it down. You can even add new functionality by defining it in the environment. This means faster bug-fixing and less downtime because of the elimination of patching procedures,recompilation and distribution of code.

```
tarkin@slave1: ~/cvs/svc/kpax — ⌘2
      (forward-to-lsp "archives.lsp" request entity))))
;Compiler warnings :
;   Undefined function REQUEST-QUERY-VALUE, in ARCHIVE.
;   Undefined function GET-MONTHS-WITH-ENTRIES, in ARCHIVE.
;   Undefined function SETF::|COMMON-LISP-USER::REQUEST-ATTRIBUTE| (3 references), in ARCHIVE.
;   Undefined function PAGE-SOME-OBJECTS (2 references), in ARCHIVE.
;   Undefined function FORWARD-TO-LSP, in ARCHIVE.
ARCHIVE
? (in-package :kpax-user)
#<Package "KPAX-USER">
? (defun archive (request entity)
  (let ((type (request-query-value "type" request)))
    (when (equal "monthly" type)
      (setf (request-attribute request :months-with-entries) (get-months-with-entries 1)
            (request-attribute request :recent-posts) (page-some-objects 'entry `(= blog-id 1) 1 10 "created_on
" "desc")
            (request-attribute request :categories) (page-some-objects 'category `(= blog-id 1)))
      (forward-to-lsp "archives.lsp" request entity))))
ARCHIVE
? 10.0.0.45 - - [Fri, 20 Jun 2003 08:20:11 GMT] "GET /eyeswideshut/webaction?action=archive&type=monthly HTTP/1
.1" 200 -1
10.0.0.45 - - [Fri, 20 Jun 2003 08:20:11 GMT] "GET /eyeswideshut/eyes.css HTTP/1.1" 200 6886
10.0.0.45 - - [Fri, 20 Jun 2003 08:20:12 GMT] "GET /eyeswideshut/images/psillog.jpg HTTP/1.1" 200 21033
10.0.0.45 - - [Fri, 20 Jun 2003 08:20:24 GMT] "GET /eyeswideshut/ HTTP/1.1" 200 -1
^D.0.0.45 - - [Fri, 20 Jun 2003 08:20:26 GMT] "GET /eyeswideshut/webaction?action=archive&type=monthly HTTP/1.1
" 200 -1
? (+ 1 2)
3
? Entry-id: 3010.0.0.45 - - [Fri, 20 Jun 2003 08:23:18 GMT] "GET /eyeswideshut/webaction?action=show-entry&entr
y-id=30&target=comments HTTP/1.1" 200 -1
^D
? 10.0.0.45 - - [Fri, 20 Jun 2003 08:24:41 GMT] "GET /eyeswideshut/webaction?action=main-reader HTTP/1.1" 200 -
1
```

Figure 6.1: redefining a function in the running KPAX

6.2 KPAX and Web Applications

As a framework for developing web application, KPAX can do everything that application servers like Tomcat can do. It provides session management and security for free and has easy ways to develop, deploy, start and stop web applications.

It comes with the Lisp equivalents of *Java Servlets* and *Java Server Pages* so we can use the same design paradigms to build our precious web applications. On top of that, KPAX comes with a surplus value : Lisp ! For all the reasons stated above, the ability to use Lisp makes it in many ways superior to develop web applications.

I had no trouble doing what I needed to do to get my web applications working in KPAX. The two *Hello World* and *Factorial* have all the essentials you need, and the same ideas and concepts were used in *RSS Syndication* and *Psillog*.

But KPAX and Lisp aren't J2EE either, sometimes you'll have develop part of the infrastructure to get things done (eg. the database layer and XML parser) and the lack of a huge Application Programming Interface like the ones that Java, .NET and Python have can complicate more advanced development situations a great deal.

But the combination of Lisp and the KPAX framework was powerful enough to convince me that Lisp has a bright future. I would certainly use that combination in further projects and try to expand my knowledge of Lisp and web applications because of this project.

6.3 End Conclusion

This project was truly an extensive learning experience, and because of Lisp it was an extremely fun experience. Some Lisp features make you gloat because of how elegant they are, how easy they provide you with the solution to your problems, and because they're often very unique to Lisp.

When you read Lisp programs written by experienced Lisp programmers (usually called *Lisp Hackers*) you will notice not only logical and efficient solutions to problems, but above all a beautiful code.

In my humble opinion, Lisp drives you to try and write elegant programs, learn more about Lisp and programming languages in general and above all : Lisp keeps programming fun !

Bibliography

[1] E. Gat. Lisp as an alternative to java. 2000.

[2] P. Graham. *ANSI Common Lisp*. Prentice Hall, New Jersey, 1996.

[3] P. Graham. The roots of lisp. 2002.

[4] C. Gunter. *Semantics of Programming Languages*. The MIT Press, Cambridge Massachusetts, 1992.

[5] S. Kamin. *Programming Languages, and Interpreter-based approach*. Addison-Wesley Publishing Company Inc., New York, 1990.

[6] B. MacLennan. *Principles of Programming Languages*. Holt, Rinehart and Winston, New York, 1987.

[7] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. 1960.

[8] G. Steele and R. Gabriel. The evolution of lisp.

[9] D. Touretzky. *COMMON LISP: A Gentle Introduction to Symbolic Computation*. The Benjamin/Cummings Publishing Company,Inc., New York, 1990.