A Framework for Automatic Web Application Generation

Marc Battyani
Fractal Concept
http://www.fractalconcept.com/

Abstract

Web sites have evolved from static documents to simple applications (eCommerce) and now to complete applications. Today, frameworks like J2EE and .Net have emerged for writing these applications made of tens to hundreds of object classes. These huge frameworks deal mostly with the business logic (middle-ware) but suffer from the limitations of their programming languages.
The framework presented here shows how the unique qualities of Common Lisp can boost the productivity for writing web applications by more than an order of magnitude compared to those systems. This framework automatically generates the presentation, modification, validation, storage and data integrity layers of all the object classes of an application. It provides session management, web controls like those in ASP.NET, and co-browsing. Internally, it makes extensive use of the Meta Object Protocol, CLOS generic functions, lexical closures, and on the fly code generation and compilation.

Contents

1 Introduction

Web sites have evolved from the presentation of static documents to simple applications like e-Commerce but now there is a growing demand for complete general applications. Potentially all desktop and client/server applications will be transformed into web applications. We can already see this with the ASP (Application Service Provider) model which tries, with very mitigated success, to make the switch from software sales to service subscription. This trend has started with the current technologies, but these technologies are too limited, have a low productivity and high design and maintainability costs. New products, like Microsoft's .NET and the various J2EE application servers like IBM's websphere and BEA's weblogic, have emerged for writing these applications but these huge frameworks provide only incremental improvements over the previous generation of tools. For instance they all put a strong emphasis on the Web-services concept which, for example, looks like Microsoft's COM/DCOM mechanisms on the Windows platform. But this is only just another RPC protocol that will maybe add some inter-operability but won't fundamentally change the problem.
In a similar way, the new languages of the .NET framework, based on a common virtual machine, are only imitating and (slightly) extending the Java model. This is why even though these giants will take the major part of the market there will remain some place for alternative technologies.
The idea of automatically generating web applications is not new and all framework designers say they are doing it. But, as these frameworks are based on statically compiled languages without many introspective features, the automation generally consists of a number of code generation wizards. These wizards provide a User Interface for defining objects, interfaces, etc. and from there they generate source code files that have to be completed and then compiled, linked and tested. We will show here that, with Common Lisp, it is possible to go much further in the application generation and that almost all the data management can be done automatically. We base this on our experience of a previous framework, that we have made for building Win32 applications, which has been able to remove 90% to 97% of the hand written code in typical business applications. We also want the applications to be really interactive applications and not just a collection of forms that users have to fill and submit only to get an error message at the end because one field was not correct.

2 The Problem Domain

Web applications can take many forms. As our goal is to have really usable and useful integrated framework, rather than just another collection of thin layers and modules, we have to restrict the problem domain to a specific class of web applications.
The engineering constraints for the framework are the following:

Corporate Acceptability The framework must be acceptable for applications in a corporate environment.

Data Centric The applications targeted are based on data objects. The management of the data objects (creation, modification, storage in a database) is the focus of this framework.

Partitionable Concurrency Model The data objects and their users are partitionable into disjoint sets with a number of objects and users small enough to be managed on a single computer even if there are lots of users at the same time.

In addition to these general constraints we have set a list of functionalities that we want the framework to provide:

Data Integrity The data in the database must be in a stable coherent state at any time. The framework must prevent users to enter data that violate integrity constraints.

UI Views Generation As we will have lots of object classes, we want User Interface views to be generated automatically from the class meta information.

View Coherence Different views of the same object must display the same information even for distinct users. This will also allow collaborative work.

Application Navigation The framework must provide all the navigation needed in the application.

Flexibility Object classes must be easily modifiable and the framework must be able to convert the existing objects in the database if needed.

Portability The framework and the applications must be able to work on different OS (Unix, linux, Windows 2K) and on any ANSI compliant Common Lisp implementation with the Meta Object Protocol[1]. This is why we try to not use any non ANSI feature of the Lisp implementations. The most notable exceptions are well circumscribed to sockets interface and multiprocessing extensions.

Performance and Scalability The applications generated must be fast enough for real use and must be scalable.

Pragmatism This is not an academic project so pragmatism and usability is preferred to theoretical purity.

These constraints and functionalities, have induced most of the technical choices for the framework.

3 Why Common Lisp?

Let's begin by a brief and (over-)simplified history of web application development. When we look at the programming languages used for web applications, it is interesting to note

[1] Thought the M.O.P. is not in the ANSI standard it is generally considered as a de facto standard.

that they are mainly scripting languages. Perl was the popular language that has been used for the first generation of web applications (CGI scripts). With the second generation of web applications came dedicated languages like ASP (VBscript + JScript) and PHP. The scripts execution was now in the HTTP servers processes thus giving a better performance than the CGI model which spawned a new process for each request. The next generation has seen more powerful languages like Java, C#,coming with the J2EE or .NET frameworks. The application server has now its own separate processes and is not in the HTTP server processes anymore. This language evolution has been following the increase in complexity of the web applications. Any scripting language is powerful enough to manage a shopping cart in an eCommerce application. But who[2] would try to write large complex applications with them ?
Today, the state of the art in web application developpment are the J2EE and .NET frameworks based on Java or look-alike languages like C#/J#. They are based on VM architecture (which can have JIT compilers) and have huge libraries of functions for everything. They want to be much more than scripting languages but in reality they are mainly used like scripting languages by combining calls to pre-existing libraries.
So now why not use these languages for developping those new web applications? After all, nobody would never be blamed for using the more mainstream tools for a task. The answer is that though Common Lisp suffers from only one problem it has many advantages. The problem is that Common Lisp is not well known[3] or worse is badly known. A lot of people either have no clue about it or worse think it is slow and bloated with an illisible syntax. They are wrong but the problem is here. Fortunately the many advantages are quite real. Here are some of them:

Performance: CL is compiled to native code giving very good performance. Much better than the JIT compilers.

Stability: CL is a stable and mature ANSI standard. It does not change every 6 months.

Excellent Macro System: the macros of CL enable the realisation of specialized domain-specific languages embedded in CL. We use macros a lot for HTML, predicates and rules code generation.

Most Powerful Object System : the CL object system and its Meta Object Protocol is critical for this application. The generic functions and method combinations (before, around, after methods) allow for a clean design without having to create utility classes.

Lexical Closures : they provide a very simple an elegant way to manage the system state and the user interaction.

[2] Of course, for any scripting language, we can find people that assure they can write any complex application with it
[3] For more information on Common Lisp look at [1–2]

Dynamic Code Generation and Compilation : the ability to generate specialized code and to compile it is used in conjunction with the macros: for the HTML views generation, for the database storage, for the data integrity rules, etc

Debugging and Dynamic Modification of a Running System: there is no Edit → Compile → Link → Restart cycle. The code can be modified without stopping a running system. This alone is already a real boost in productivity.

Resilience to Changes: the uniform syntax, the generic functions, the macros allow for major changes without full redesign of the code.

4 General Architecture

The general architecture is a classical one with HTTP servers used as a front end to handle the communication with the user, Lisp application servers to handle the application logic and database servers to store the application data.

HTTP Front End: For serving the HTTP requests, we have chosen to use Apache[4] with mod_lisp[4][5] rather than processing them directly in Lisp for several reasons:
First for corporate acceptance. It is always easier to tell corporations that you are based on Apache rather than on a Lisp server. But also for technical reasons. We are focused on making web applications not whole web sites. Those applications will generally be a part of a general web site made with classical tools (PHP, JSP, ASP, etc). We also don't want to use Lisp processing power to serve static content like HTML or JPEG files, to handle the HTTP protocol or deal with SSL. For scalability, we want to be able to redirect the HTTP requests to several Lisp processes on different computers. Finally, for security, we want the Lisp applications to be on a local network not directly connected to the Internet.

Lisp Application Servers: The HTTP front ends transmit the decoded requests to the Lisp application servers through mod_lisp[5]. The different application servers will be linked to different URI prefixes so that all the requests from a user will be sent to the same server. This is a consequence of our choice of a partitionable concurrency model. The Lisp servers will process the logic of the application. The data integrity of the objects is managed in the Lisp servers.

Data Storage: The corporate acceptability constraint has made us choose to use SQL databases for the data storage. In our previous frameworks (win32) we stored data in blobs made of Lisp s-expr. This works very well but IT managers don't like it. So we switched[5] to SQL tables, loosing some flexibility in the process. Now that XML rules, we could probably switch back to the blob storage if we encode the s-expr in XML and put them in an SQL/XML database We use CL-SQL[6] to manage the SQL database connections.

[4] mod_lisp[5] is an Apache[4] module for using Lisp processes to handle the HTTP requests
[5] Different data stores can be present at the same time and we have kept the s-expr store for some config objects

5 Objects Descriptions

Object classes are at the core of this framework. We use the Meta Object Protocol to define a meta-class so that we can add a lot of meta information to the classes and their slots. We also use the MOP to redefine the slots access. This enables us to verify the values constraints, trigger rules and disabling predicates when the slot is written.
Here are some of the slots that we added to the meta-class:

guid:	We give a GUID to the class to assure its unicity among applications.
instanciable:	The class is instanciable or not.
user-name:	The name of the class for the users.
sql-name:	We can force an SQL table name for the classe.
description:	Comments are always welcomed...
version:	We keep the previous version for generating the convertion code.
direct-views:	The list of the direct views of the class.
direct-rules:	The list of the class rules used for data integrity.
direct-functions:	The list of the class functions callable by the user.

and so on.
All the names and textual information for the users are given in several languages (English, French, German, Spanish and Italian for now).
We also added a lot of slots to the slot-definitions. Here are some slots of them:

user-name:	The name of the class for the users.
sql-name:	We can force an SQL column name for the slot.
description:	Comments are still always welcomed...
value-type:	The type of the value for the slot. It can be a type that does not really exists in CL. Like fixed decimal for instance.
list-of-values:	The slot contains a list of values.
unique:	The values of the slot must be unique.
stored:	The values of the slot must be saved to the database.
indexed:	The values of the slot must be indexed in the database.
null-allowed:	Null values (nil) are allowed for the slot.
unit:	The unit of the value.
read-only-groups:	The user groups for which the slot is read only in the views.
hidden-groups:	The user groups for which the slot is hidden in the views.
disable-predicate:	The predicate used to find if the slot is disabled.
value-constraint:	The predicate used to find if a value is allowed for the slot.
void-link-text:	The text to display if the slot is nil.
view-type:	The type of widget to use to display the slot in a view if it's not the default one.

and so on.
Most of the information added is used for two purposes. Giving hints on how to store the slot in the database, how to display and interact with it in a view and to ensure that the data is always valid. We have shadowed the defclass macro so that it can accept the augmented

class and slot definitions and create the many methods and functions needed for handling all the added functionalities.
To maintain a consistent state of the objects we have 2 mechanisms. The first one is a value constraint predicate for each slot that will not allow the slot value to be set to a value not satisfying it. The second one is a global set of rules for the object class. These rules are fired when the slot values change and can perform some actions like recomputing other slot values. If a rule creates a constraint violation, the object state is restored to its previous values and the initial value change that triggered the rules is not allowed.
The objects also keep a list of closures that are called when the object state changes so that the views can change the presentation of the object when the object change.
A documentation is generated from the meta-class information.

6 Session and State Management

As the HTTP protocol is stateless, a session management system is needed. The most commonly used alternatives to do this are by using cookies, URI encoding or both. To be completely flexible, we have chosen to use URI encoding with property lists. Each URI is a property list converted to a string and then encoded with a variant of base64. This allows us to put lots of informations in the URI, like the session ID, a page to get, a function to call and various parameters.
On the server side we maintain a session object with the history of the session, the time of the last action (for timeout), the user identification, etc. When there is a request we look at the session-id in this encoded URI. If there is no session-id or the session-id does not correspond to an existing session object, we instantiate a session object, re-encode the URI with the same property list (except for the session-id property which is set to the new one) and send a redirect to this new URI to the browser.
In case of an SSL connection the sessions can be linked to the SSL session-id.

7 UI Management

The user interface is handled by the HTML browsers, but we decided that the usual paradigm of form submitting (pull only mode) could not apply for this framework as it is not compatible with the views coherence and data integrity constraints. This is why we use a combination of classic pull mode navigation as well as a less classic push mode using remote scripting.

7.1 Navigation

All the navigation in the application is generated by the framework. They are 2 types of navigation:
Client initiated navigation (pull mode) is provided by HTML links (`<A>` tags) which have property lists encoded in URI as href. So when the user clicks such a link, the framework

receives and decodes the link URI to retrieve the property list and then performs the action encoded in the property list.
A typical page link property list looks like this:

```
(:page "home" :lang :fr :session-id "HycTS6fO")
```

If a page needs some parameters they are just added to the property list like this:

```
(:page "obj-view" :lang :fr :session-id "HycTS6fO"
 :object-id "5JugJG76fHGfD" :display :full)
```

Server initiated navigation (push mode) is provided by remote scripting. It can be a reply to a user action like clicking a function link or initiated by a data change on the server.

7.2 HTML Generation

For HTML generation we use the now classic LHTML[6] representation. The HTML is represented by forms where a tag is represented by a keyword or a list (keyword &rest attributes) if the tag has attributes. The tag is closed at the corresponding s-expr end. Forms not starting by a keyword are considered normal lisp forms and are left unchanged.
An example of LHTML is:

```
(:table (:tr ((:td :width "30%")"first line")
             ((:td :width "70%") variable)))
```

This simple example is transformed in the following code by the html macro:

```
(progn
  (write-string "<TABLE><TR><TD WIDTH=\"30%\">first line</TD>
</TR><TD WIDTH=\"70%\">"
                html:*html-stream*)
  (princ variable html:*html-stream*)
  (write-string "</TD></TABLE>"
                html:*html-stream*))
```

As you can see the LHTML code is compiled to Lisp code. There is no run-time interpretation to write the HTML. The consecutive calls to `#'write-string` and `#'write-char` with string or char constants are even concatenated into one unique call to `#'write-string`.
This representation has many advantages such as allowing to mix HTML generation code with lisp code and to define macro tags that takes parameters and expand to complex HTML. We have tags like `:jscript` for inserting java-script in the HTML or `:insert-file` that will read a file and insert its content in place of the tag or even complete DHTML widgets/controls like `:tab` that will expand to a tabbed layout.

[6] John Foderaro (Franz inc.) published it with its html-gen macro[7]. For other HTML gen macro see [8]

```
(html
  (:jscript "alert('hello');")
  ((:table :bgcolor "#C0C0C0")
   (loop for (a b c) in list
         do (html (:tr (:td a)(:td b)(:td c))))))
```

is compiled to

```
(progn
  (write-string
     "<SCRIPT LANGUAGE=\"JavaScript\"
TYPE=\"text/javascript\">alert('hello');</SCRIPT>
<TABLE BGCOLOR=\"#C0C0C0\">"
     html:*html-stream*)
  (loop for (a b c) in list do
        (progn
         (write-string "<TR><TD>" html:*html-stream*)
         (princ a html:*html-stream*)
         (write-string "</TD><TD>" html:*html-stream*)
         (princ b html:*html-stream*)
         (write-string "</TD><TD>" html:*html-stream*)
         (princ c html:*html-stream*)
         (write-string "</TD></TR>" html:*html-stream*)))
  (write-string "</TABLE>" html:*html-stream*))
```

7.3 Views

For each object class, the framework dynamically generates and compiles, when needed, object views based on the meta class information (slot properties, user permissions, etc). These view are fully functional and nice enough for direct use. They can be customized with a CSS style sheet. The figure **figure 1** shows a fully automatic view.

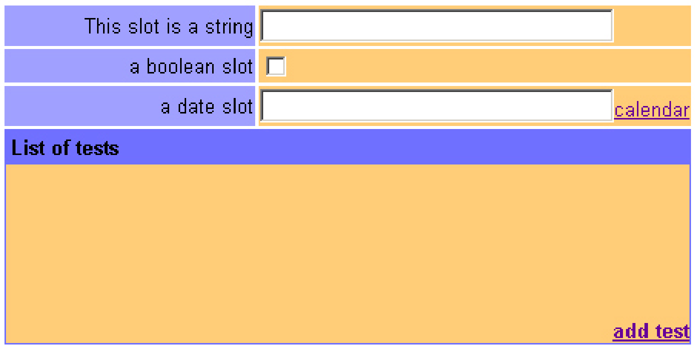


Figure 1 An automatic view

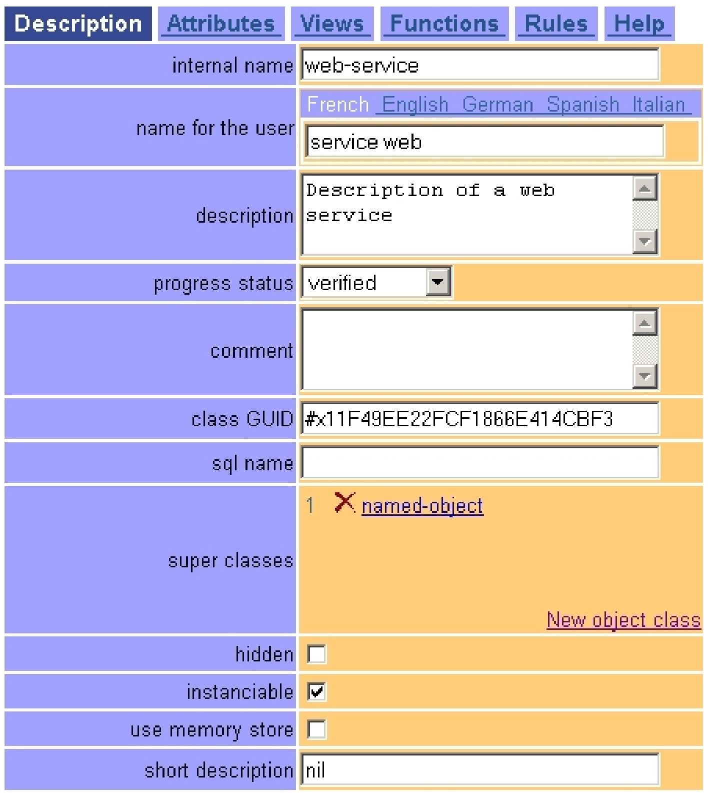
For some very often used or complex objects, the views can be semi-automatically generated by a view description in LHTML representation using macro tags that get expanded into

HTML generation code based on the slot attributes. Those macro tags also generate the server side lisp code for handling the HTML representation, modification and updating of the slot. For instance (:slot-edit name) will look at the meta information for the "name" slot generate closures to deal with changes of the value of the slot "name" and will expand to the following DHTML:

```
<INPUT TYPE="text" NAME="G1024" CLASS="dvcve"
 onchange='fire_onchange("G1024",G1024.value);'>
```

Those views use remote scripting to get and set values updates from the application server. That way we can ensure that the values presented are always up to date. If a user or the program changes a slot value all the views that represent it will get a notification that the value has changed and will reload the new value. This system allows collaborative work on the same objects by different users, providing an enhanced co-browsing functionality. The remote scripting also allows making server based widgets and modifying only parts of the HTML page without reloading it.
Here is an example of a semi-automatic view:

```
(make-instance 'interface::object-view :object-class 'class-info
               :country-language :en :name "class-en" :source-code
 `(((:tab :class "tabf")
    ("Description"
     ,@(interface:make-std-object-slots-view
        'class-info
       '(name user-name description class-status comment
         guid sql-name direct-superclasses hidden
         instanciable use-memory-store short-description)
         :en nil))
     ("Attributes"
      ((:slot-list direct-slots :height "500px"
          :col-fn interface:std-list-col-fn :class "dvl")
          (:table (:tr ((:td :class "dvch2") "Slots")))))
     ("Views"
      ((:slot-list direct-views :height "500px"
          :col-fn interface:std-list-col-fn :class "dvl")
          (:table (:tr ((:td :class "dvch2") "Views")))))
     ("Functions"
      ((:slot-list direct-functions :height "500px"
          :col-fn interface:std-list-col-fn :class "dvl")
          (:table (:tr ((:td :class "dvch2") "Functions")))))
     ("Rules"
      ((:slot-list direct-rules :height "500px"
          :col-fn interface:std-list-col-fn :class "dvl")
          (:table (:tr ((:td :class "dvch2") "Rules")))))
     ("Help"
      (:object-view :object (object-help interface:*object*)))) :br
    ,@(interface:make-std-object-functions-view
        'class-info :en nil) :br))
```



internal name	web-service
name for the user	service web
description	Description of a web service
progress status	verified
comment	
class GUID	#x11F49EE22FCF1866E414CBF3
sql name	
super classes	1 ✗ named-object
hidden	☐
instanciable	☑
use memory store	☐
short description	nil

Figure 2 The semi-automatic view

The figure **figure 2** shows the view generated from this description. The `:tab` macro tag is expanded to DHTML code that implements a tabbed pane layout. The `user-name` slot contains a translated-string object whose default view is a multi-languages tabbed pane layout that will be inserted as the value pane for the slot. The `make-std-slots-views` function takes a class and a list of slots and render them as a name/value table. The `:slot-list` tag inserts a server based DHTML list widget. In the "Help" pane here is another object view embedded in this view (the `:object-view` tag). We can insert any view into general HTML code, and insert views as well as HTML into other views.

8 Documents Generation

Business applications generally need to produce good quality paper documents in addition to the HTML pages.
For simple documents, we use our CL-PDF[9] library to generate these documents directly from the Lisp process. For more complex documents, we use a combination of CL-PDF for some blocks, and pdfTeX to provide the general layout and include the PDF blocks as graphics.

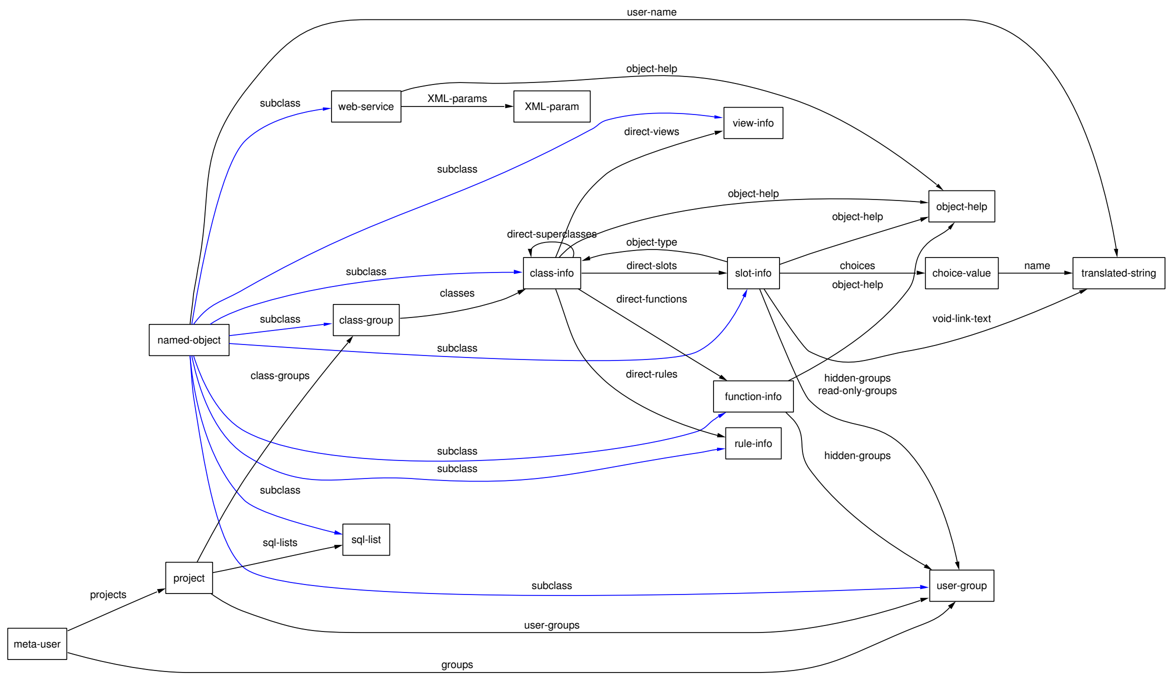


Figure 3 The graph of the meta application classes

9 The Meta Application

Of course the first application written with the framework has been the framework application itself.

The framework application provides users with a UI to create, view, modify and store object classes descriptions. The lisp code representing those classes is generated from those descriptions and the classes can be dynamically tested in a memory store. This test has fully functional views and navigation. Graphs of classes at various levels (application, class group, class neighbourhood) can be generated (in PDF with GraphViz[10]) and displayed. Clicking on the classes' graphs nodes opens a view of the class meta data. See figure **figure 3** for the graph of the framework application classes.

10 Web Services

In addition to managing the interaction with the users, the framework also supports being called as a web service with XML-RPC requests. We will probably add SOAP support in the future, but for now XML-RPC has always been enough. The HTML generation macro works also very well for generating XML.

11 Conclusion

The framework is still in an alpha stage but is already usable. In addition to the meta-application it is being tested on three real applications - A communication network statistics and maps generation software for a bank, a traceability software for the food industry and a benefits management software.
From the first tests we have done, it looks very promising. For the applications that fit well in the problem domain, we estimate that we have an increase of more than an order of magnitude in productivity, with a much better result quality and usability than the existing alternatives.
The choice of Common Lisp has been comforted by the result. The full framework has less than 16K lines of code and, though it is not completely finished yet, it already has functionalities without equivalent in other frameworks. This compares rather well to the 0.5 to several millions of lines of code of the J2EE[7] and .NET frameworks. The first application, the meta-application, has only 1100 line of code for a fully functional web application.
Common Lisp has allowed us to cleanly and quickly implement our ideas and even to test several of them for different parts of the framework.

References

1 *ALU, Association of Lisp Users.*
http://www.lisp.org/.
2 *Allegro CL, a commercial Lisp system.* Franz inc.
http://www.franz.com.
3 *Lispworks, a commercial Lisp system.* Xanalys inc.
http://www.lispworks.com.
4 *Apache, the most popular web server.*
http://httpd.apache.org/.
5 Battyani, M. *Mod_lisp, an Apache module for writing web applications in Lisp.* Fractal Concept.
http://www.fractalconcept.com/asp/html/mod_lisp.html.
6 Rosenberg, K. *CL-SQL, a Common Lisp library for using SQL databases.*
http://clsql.med-info.com/.
7 Foderaro, J. *html-gen, an HTML generation macro.* Franz inc.
http://opensource.franz.com/aserve/aserve-dist/doc/htmlgen.html.
8 Bradshaw, T. *htout, an HTML generation macro.*
http://www.tfeb.org/lisp/hax.html#HTOUT.

[7] JBoss the open source J2EE framework has more than 451KLOC

9 Battyani, M. *CL-PDF, a stand alone Common Lisp library to generating PDF files.* Fractal Concept.
http://www.fractalconcept.com/asp/html/cl-pdf.html.

10 *GraphViz, a graph drawing tools*. ATT.
http://www.research.att.com/sw/tools/graphviz/.

11 Rosenberg, K. *cl-lml, an HTML generation macro.*
http://lml.b9.com/.