Chair of
Software Engineering

A Comprehensive Eiffel Web Framework

Semester Thesis

By: Peizhu Li
Supervised by: Marco Piccioni
Prof. Bertrand Meyer

Student Number: 02-925-899

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

The EiffelWeb library provides an essential set of classes for CGI handling and HTML code generation, but concerning general server-side processing and content generation in Web application development domain, its functionalities can be extended greatly.

With this semester project, we developed a Web Framework as an extension to the EiffelWeb library, which introduces a more flexible request handling and dispatching mechanism, adds session management, user authentication support, and elaborates a template based HTML content generation solution with enhanced form validation and manipulation functionalities. As demonstrated with two sample applications, it effectively reduces Web application development complexity and increases extendibility.

Acknowledgement

I would like to thank my supervisor **Marco Piccioni** for his competent support and very helpful advices, especially for his initial advices and diverse references for the design, his expertise with Eiffel language and the IDE, and his patience, which helped a lot and saved me very much time along the development practice.

I also would like to thank the students from the SS2007 Software Engineering class, especially **Christian Regg**, **Bhardwaj Sandeep** and **Lucas Serpa Silvas**: some of their ideas for the CSARDAS project have given helpful input to this project and eventually led me to this framework solution. In addition, I reused their neat and nice style sheet designed for CSARDAS project in the "**Computer Science Event List**" sample application.

Contents

1. Introduction

EiffelWeb is the current web library for Eiffel, which introduces three sets of classes to help developing CGI web applications. The first set of classes focuses on maintaining a web context (`CGI_ENVIRONMENT`, `CGI_FORMS`), the second set takes care of input/output (encapsulated in `CGI_INTERFACE`), and the third set focuses on structured HTML code generation. The CGI application's entry class, `CGI_INTERFACE`, is designed to inherit from `CGI_ENVIRONMENT` and `CGI_FORMS`, so the Web context is initialized and made accessible inside `CGI_INTERFACE` in the first place; For simple CGI applications, we just need to inherit from `CGI_INTERFACE`, perform the necessary processing based on actual request, and return the result `HTML_PAGE` to the client: it is a rather straightforward solution, which is good for simple web sites, but does not provide too much support for more complex ones.

Often there are a lot of dynamic content and business logics that need to be processed on the server-side, together with a tight binding to a persistence layer. To give more support on Web application development and a better separation to different developer concerns, a restructuring and extension of the current EiffelWeb library becomes very helpful.

This semester thesis work implements a Framework based on the EiffelWeb library, which provides the necessary generalization, encapsulates a request string based dispatching mechanism and brings session management functionalities inside the framework, with extended user authentication and form validation/manipulation support. As demonstrated with the "**Computer Science Event List**" application, coupling between different aspects involved in web application development is greatly reduced, so that user can focus more on the web application business logic and content generation rather than request dispatching, state management and several other "plumbing" tasks.

By doing so we hope to have effectively improved the web application development library support of the Eiffel language.

2. System Design

The design of the Framework is mainly based on the following four concepts:

- Leaving static content and HTML presentation layer out of Eiffel code as much as possible, while letting Eiffel focus on server-side processing and dynamic content generation;
- Realizing a more flexible request dispatching mechanism by introducing an application configuration file, encapsulating request handling objects into a separate layer, and extracting application level variables from the Eiffel code;
- Implementing session support into the Framework, making session handling transparent to developers;
- Implementing a basic yet extendable user management module into the Framework, enhancing form validation and manipulation support

An overview of the system design is illustrated as fig 2.1, necessary details are described in the following sections.

2.1 Architecture

The request URL is formalized to include a string handler part, which identifies a specific Handler type, and a string request part, which identifies a specific request need to be processed by the identified handler.

A string handler to type handler mapping should be defined in the application configuration file, together with a default HTML template file if necessary. Based on this mapping relation, when a request is received at the entry point, the `REQUEST_DISPATCHER` class, which inherits from `CGI_INTERFACE`, will read application configuration through a `CONFIG_READER`, initialize a corresponding session object through a `SESSION_MANAGER`, and finally instantiate a corresponding `REQUEST_HANDLER` object and dispatch the request to it for further processing. A reference to the current dispatcher object will be passed to the handler as the web context – so all environment variables and form data can be accessed by it.

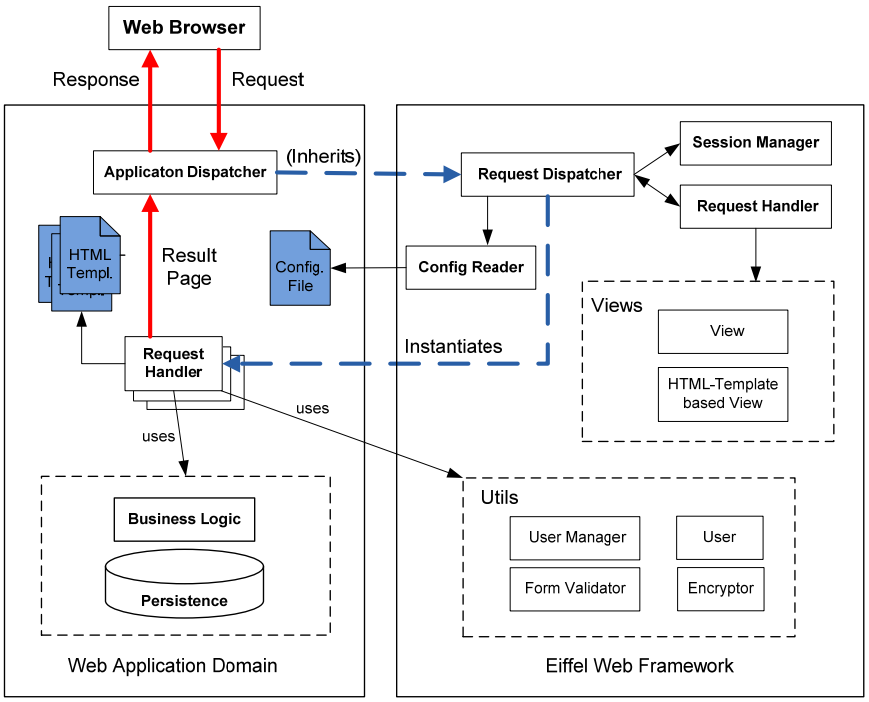

Fig 2.1 System Architecture

In the Web application domain, developers need to make their APPLICATION_DISPATCHER class inherit from REQUEST_DISPATCHER, implement common processing routines in it or in a deferred REQUEST_HANDLER class, and then implement specific request handlers for particular requests. These handlers can make use of USER_MANAGER, FORM_VALIDATOR, ENCRYPTOR and different VIEW implementations from the Framework and interact with the application Business Logic and Persistence Layer, applying server-side processing to produce the result HTML content as response to the request from Web client.

2.2 Configuration File

With this implementation, a typical URL request appears in a form like:

*http://[host]/[app_path]?[**request_string**]&cmd=[**command_string**][&...]*

the ***request_string*** will be used to identify the corresponding handler, and the ***command_string*** to distinguish actual requests for the handler. The configuration file is mainly designed to handle such a ***request_string***, ***command_string*** to handler class name, html template mapping and necessary session parameters; some often used web application variables are introduced also, and a traditional Linux/Windows application configuration file is adopted. All defined variables will be retrieved by a CONFIG_READER and can be used by the request handlers later; however, you can simply omit most of them in case they are not necessary for your application.

The configuration file is supposed to be in the same directory as the Eiffel Web application, with the same name but "*.conf*" as the extension. Follows an example configuration file used by the '**Computer Science Event List**' application. The application path is

```
/cgi-bin/informatics_events.exe
```

So the configuration file should be *informatics_events.conf* and saved under */cgi-bin/* directory.

Here is the [**general**] section of the configuration file explained:

```
[general]                                   ; general parameters for the web application
App_path=/cgi-bin/informatics_events.exe    ; path to the web application
default_request=event                       ; request_string in case not specified in url
default_command=list                        ; command_string in case not specified in url
notfound_request=other                      ; request_string when specified map not found
notfound_command=notfound                   ; command_string when map not found
stylesheet=/info_events/css/style01.css     ; default style-sheet to include
javascript=/info_events/sorttable.js        ; default java-script to include
image_path=/info_events/images/             ; path to image files
error_template_page=..\info_events\errorpage.html   ; error template page
```

default_request, ***default_command***, ***notfound_request*** and ***notfound_command*** are supposed to be defined, so that the request dispatcher can still forward the request to an expected handler in case request string is not specified or a corresponding handler is not defined in the configuration file. Otherwise, as a default action, a *"404 – page not found"* page will be returned to the browser.

A [**database**] section is also included:

```
[database]
host=localhost                  ; database server host name
port=3360                       ; port
socketfile=/tmp/mysql.sock      ; socket file in case there is a socket connection
database=informatics            ; database name
username=informatics            ; username
password=_informatics           ; password
```

Then comes the [**session**] section for the directory in which session files will be stored, session expiration time (in seconds) and the length of new generated session ids:

```
[session]
session_files_folder=..\info_events\sessions\       ; folder to save session files
expiration=600                                      ; in seconds, 15 minutes in case not specified
session_id_length=12                                ; length of generated session ids
```

Follows a [**constant**] section, where you can define constants and access them in your request handler later:

```
[constants]
app_data_folder=..\info_events\data\  ; path to save user/event files
users_file_name=users                 ; filename to save user accounts
event_list_data_file_name=events      ; filename to save events
event_id_generator_data_file_name=event_id   ; filename to save latest event id
```

Finally the important [**Request_Handler**] sections, each of which should specify a request string and the corresponding handler class name; additionally a default HTML template file and HTML templates corresponding to different commands if necessary:

```
[Request_Handler]
request=user                    ; request_string
handler=USER_HANDLER            ; corresponding Handler class name
default_template=..\info_events\user.html          ; default template for USER_HANDLER
userform_template=..\info_events\adduser.html      ; template for 'userform' command
saveuser_template=..\info_events\adduser.html      ; template for 'saveuser' command
details_template=..\info_events\userdetails.html   ; template for 'details' command
loginform_template=..\info_events\login.html       ; template for 'loginform' command
login_template=..\info_events\login.html           ; template for 'login' command
list_template=..\info_events\userlist.html         ; template for 'list' command
activate_template=..\info_events\userlist.html     ; template for 'activate' command
suspend_template=..\info_events\userlist.html      ; template for 'suspend' command
delete_template=..\info_events\userlist.html       ; template for 'delete' command

[Request_Handler]
request=event                   ; request_string
handler=EVENT_HANDLER           ; corresponding Handler class name
default_template=..\info_events\event.html      ; default template for EVENT_HANDLER
```

However, only ***request*** entry and ***handler*** entry must be defined for a successful handler instantiation and control transfer. The template for a specific request is defined by a [***command_string***]**_template** string (for example: "*activate_template*"). This section is supposed to be repeated until all request, handler mappings and necessary command, template mapping relations are defined.

2.3 Request Dispatching

Based on the actual ***request_string*** specified in the URL, the corresponding request handler's class name will be looked up from the configuration file by a CONFIG_READER object, and then such a handler object will be instantiated using Eiffel reflection, initialized and passed over the control for further processing of the received request. Besides, a session object will be initialized either with a previously saved state if current session can be identified or with a new state if current session is unrecognized.

By introducing such a request dispatcher layer into the framework, the extendibility for both the Web Framework and the Application is greatly improved. More specific behaviour can be easily introduced by inheriting from the REQUEST_DISPATCHER class if necessary. Handlers are on a separated layer from the framework, and application developers can focus on their specific request handlers implementation while letting the framework do the dispatching transparently, which relies only on a configuration file which can be updated easily.

2.4 HTML Result Page Generating

With the HTML technologies going more and more towards a standardization, especially **XHTML**, **CSS** and **JavaScript**, and with the most used browsers being able to correctly use them, a lot of work including certain client side interactions and many content presentations can be well placed inside the HTML domain itself. This should help providing a better

separation of tasks between web designers and web developers. With this in mind, we have focused on the HTML view and suggested a HTML template based solution for the resulting web page generation.

To address the extendibility to other possible presentation implementations (like rss, csv, pdf), a HTML_PAGE derived VIEW class is introduced in the framework as a mid-layer, then a HTML_TEMPLATE_VIEW class is implemented focusing on the HTML presentation. After experienced Web designers designed the web site as they used to, as Eiffel Web application developers we can inject different markers into those designed HTML pages corresponding to different dynamic contents, organize dynamic contents into different sections, or extract them into separate files, and then with HTML_TEMPLATE_VIEW provided functionalities we can effectively update them, and finally reassembly them into a valid HTML page for the client as expected.

Three types of markers are introduced in current implementation:

- **Content Marker**: *{#NAME_OF_THE_MARKER#}*
  The marker itself can be replaced by a given string or content from a text file;
- **Normal Section Marker**: we can mark a part of the HTML code as a section by inserting
  ```
  <!--##A_SECTION_MARKER_NAME##-->
  ```
  at the beginning of the segment and insert
  ```
  <!--##/A_SECTION_MARKER_NAME##-->
  ```
  at the end, or mark it as in a commented-out state by inserting
  ```
  <!--##A_SECTION_MARKER_NAME##
  ```
  at the start and
  ```
  ##/A_SECTION_MARKER_NAME##-->
  ```
  at the end, similar to the HTML syntax for comments. Marked sections can be switched on of off when required (commented out or not);
- **Alternative Section Marker**: in case several sections are mutually exclusive with each other, at the same time only one section stays active, others should be commented out, you can name those sections the same but with an indexed integer suffix starting with 0, and let the first section be active and others commented out in the template:
  ```
  <!--##A_SECTION_MARKER_NAME_0##-->
     ... (...HTML code...)
  <!--##/A_SECTION_MARKER_NAME_0##-->
  <!--##A_SECTION_MARKER_NAME_1##
     ... (...HTML code...)
  ##/A_SECTION_MARKER_NAME_1##-->
  ```
  By giving the marker name and an index number you can activate a specific section while deactivating all other alternative sections.

Routines like marker cleanup and replace marker with corresponding form data are also supplied for convenience. By introducing such a marker based HTML solution we tried to be as less invasive as possible of the HTML presentation layer, but still maintaining full control over the dynamic content, and reducing the HTML content to be embedded into the Eiffel code to the minimum.

2.5 Request Handling

A reference to the actual dispatcher object is included in the handler for necessary access to the web context (environment, form data, application configuration, session data etc.). After necessary interaction with business logic and persistence layers, a result HTML page is supposed to be prepared and returned to the dispatcher, as response to the web Client.

Some utility classes from the web framework, like FORM_VALIDATOR, USER_MANAGER, and diverse implementation of the VIEW class are supposed to give a better support for request processing and result page generation.

2.6 Session Management

An extendable session object, which can be identified with a unique id, is supposed to be serialized to a file in a user-defined folder on the server, using its unique id as the filename. Based on such a concept, a hash table is used as the container for session variables, which enables developers to conveniently set objects into a session object, check, and retrieve them from a restored session. Then with the SESSION_MANAGER class, the Session object for the current request will be transparently initialized by the request dispatcher, either read from the serialized file, or initialized as new, and will be saved to the corresponding file before sending the response back to the client.

To track client session status more effectively even without cookie enabled, a URL-rewriting based mechanism is implemented in this framework. As a default, in case cookies are enabled on the client browser, the session id will be saved in a cookie on the client; otherwise a session id will be assigned, and all URL links in the result HTML page will be rewritten automatically to include a session id, so that client state can be maintained effectively on the server side.

2.7 User Authentication

As user profiles are widely used in Web applications, a module with extendable user management functionalities is implemented in the Web Framework, to simplify the user profile related development.

Similar to the session management design, a USER class with only username and password properties is defined in the Framework, together with a USER_MANAGER class and a file serialization based implementation (USER_MANAGER_FILE_IMPL). Developers can specialize the User class and can either make use of the current implementation for simple user management functionalities, or supply their own implementations for the persistence layer, still taking advantage of the user management functionalities supplied by the framework.

2.8 Form Processing

A FORM_VALIDATOR class is introduced in the Framework by wrapping necessary functionalities supplied by CGI_FORM class and providing more form validation related functionalities. A reference to the actual request dispatcher object is used to access current

web context. For each specific HTML form, developers can implement corresponding form validation routines more easily inside the handler objects.

Besides, together with the HTML page generating mechanism introduced before, we can easily manipulate forms inside Eiffel, no matter to update form elements or to restore form values to a specific state.

2.9 Others

Additionally, an `ENCRYPTOR` class and an `ENIGMA` algorithm implementation are included in the Framework for some basic encrypt/decrypt functionalities. It's used in the user management module and leaves it to the user to replace it with a custom implementation for a better encryption if necessary.

3. Implementation

The implementation of the Framework is illustrated as fig 3.1.

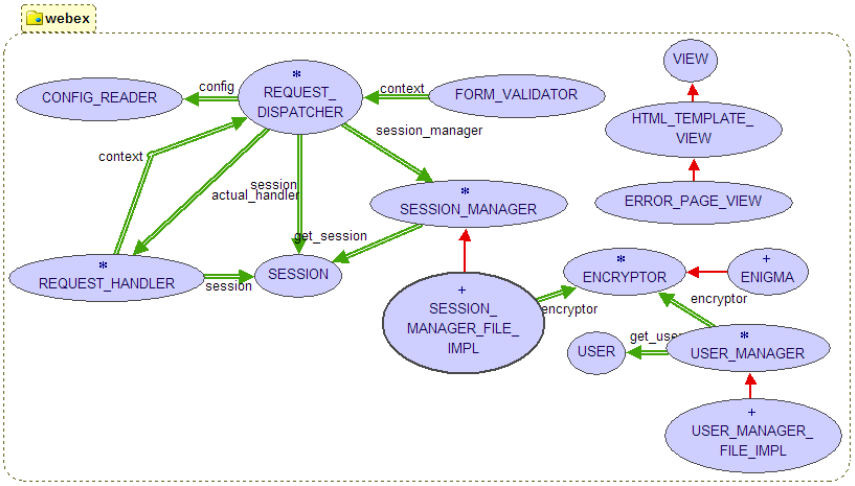

Fig 3.1 BON diagram for the Web Framework Implementation

Some implementation details of main classes are described in following sections. For more information please refer to the source code and generated API documentation.

3.1 CONFIG_READER Class

A simple CONFIG_READER that tightly binds to the proposed configuration file design is implemented. Because Eifel Web application runs on process level for each URL request, using *read_configuration* CONFIG_READER reads predefined parameters, constant variables, and only the corresponding handler class name and template file path for the given request string and command string. *get_template_file* and *get_handler_type* are supplied for lookup other handler/template entries based on given *request-string* and *command-string*.

Configuration entries are defined as public attributes for direct access.

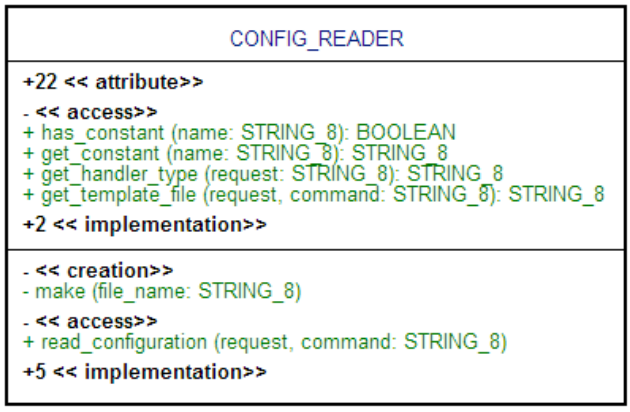

Fig 3.2 Implementation of CONFIG_READER

3.2 REQUEST_DISPATCHER Class

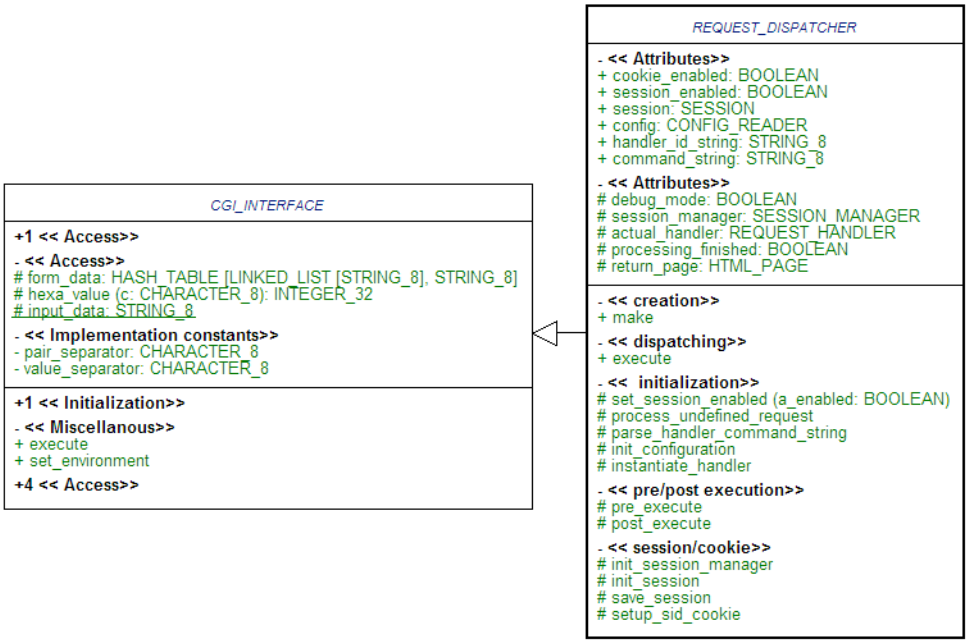

Fig 3.3 Implementation of REQUEST_DISPATCHER

The deferred REQUEST_DISPATCHER class inherits from CGI_INTERFACE and is the application's entry point. In order to get Eiffel include handlers implemented in the application domain into compilation and get the reflection based handler instantiation work properly, it is necessary to write a creation procedure ({REQUEST_DISPATCHER}.*make*) and define a local variable for each used request handler class. In the INFORMATICS_DISPATCHER class for "**Computer Science Event List**" application 3 handlers are used:

```
class
	INFORMATICS_DISPATCHER
inherit
	REQUEST_DISPATCHER

redefine make end
create make

feature -- creation
	make is
		-- simply call parent's make procedure, but define a local variable with
		each implemented handlers here to get all handlers compiled into the
		application, so that polymorphism works properly
	local
		event_handler: EVENT_HANDLER
		user_handler: USER_HANDLER
		general_handler: GENERAL_HANDLER
	do
		PRECURSOR {REQUEST_DISPATCHER}
	end
	…
end
```

By *instantiate_handler* REQUEST_DISPATCHER will instantiate and initialize a handler object corresponding to current request string, then inside *execute*, call the handler's *process_request* feature, get the prepared result HTML page, setup response header, save session and send response back to the client.

Dispatcher level common routines can be injected into the processing flow by redefining *pre_execute* and *post_execute* features, which will be executed respectively before/after *process_request* call of the instantiated handler object in *execute*. The *processing_finished* tag can be used to indicate whether further processing is still necessary or current *return_page* is already well set as the response to actual request. Or, the dispatcher's behaviour can be customized by redefining relevant routines from REQUEST_DISPATCHER.

If session support is enabled (it is by default), to tell whether cookies are enabled by the browser but avoid using additional testing requests/responses, the session id cookie is included in every response header, and checked upon every incoming request. If a valid, not expired session id is found in cookie, cookie support is assumed to be enabled.

3.3 REQUEST_HANDLER Class

In a user derived class the *initialize* feature must be redefined, include all instantiation routines in *make* creation feature, because with reflection in Eiffel the creation feature is not called and here in the Framework the *initialize* routine is supposed to do all necessary initialization includes the context.

The *process_request* routine is separated again into *pre_processing*, *handling_request*, *post_processing*, and with variable *processing_finished* to tell whether further processing is necessary, to make injection of common processing routines easier.

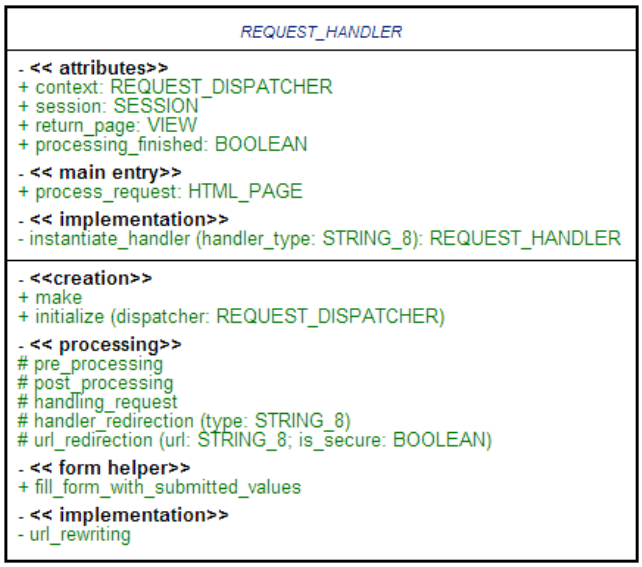

Fig 3.4 Implementation of REQUEST_HANDLER

Before the prepared *return_page* is returned, *session_enabled* and *cookie_enabled* conditions from current web *context* object will be inspected and if necessary, *url_rewriting* will be called for session support under cookie disabled circumstances.

Besides, routine *url_redirection* and *handler_redirection* are implemented to realize URL redirections inside REQUEST_HANDLER, and a help function for forms, *fill_form_with_submitted_values* is implemented to check and replace all markers in a HTML_TEMPLATE_VIEW based result page with corresponding values in case a variable with the same name is defined in the request.

3.4 SESSION and SESSION_MANAGER Classes

An *object_list* hash table container is used inside SESSION implementation to let user be able to add session variables conveniently. The deferred SESSION_MANAGER class is implemented for general session lookup, session id generating, session retrieving, saving and clean up operations.

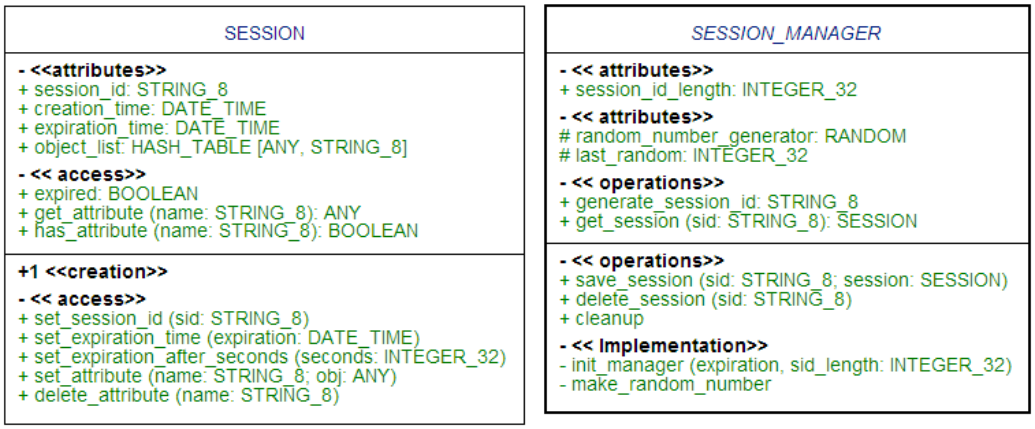

Fig 3.5 Implementation of SESSION and SESSION_MANAGER

In the current web framework implementation, a file serialization based solution for session storage is implemented with SESSION_MANAGER_FILE_IMPL class.

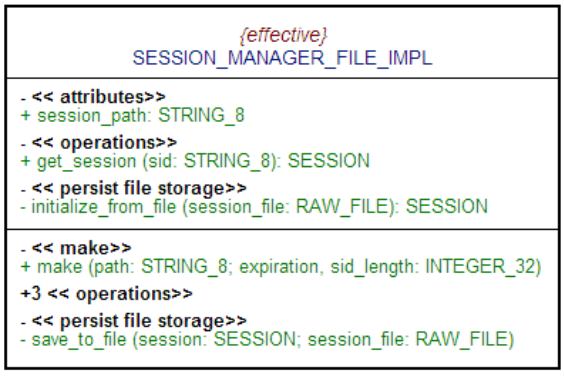

Fig 3.6 Implementation of SESSION_MANAGER_FILE_IMPL

3.5 USER and USER_MANAGER Classes

As illustrated in Fig 3.7, a simple USER class including only *username* and *password* attributes is implemented and also a file serialization based USER_MANAGER implementation as class USER_MANAGER_FILE_IMPL. However user can extend the USER class according to his own needs but still take advantage of this USER_MANAGER implementation if no special request is concerned.

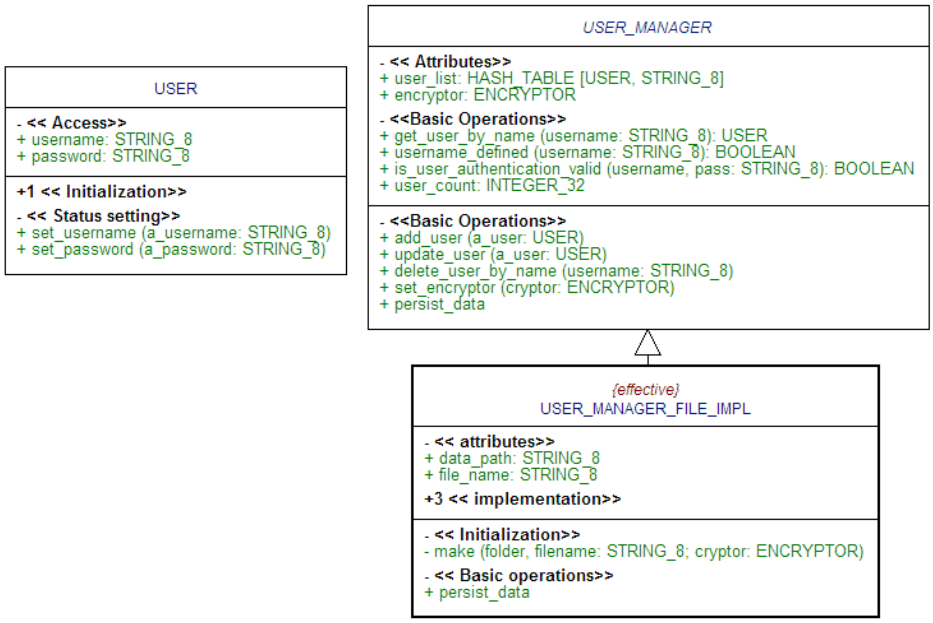

Fig 3.7 Implementation of USER and USER_MANAGER

An *encryptor* object is included in USER_MANAGER and as used in USER_MANAGER_FILE_IMPL, the password is encrypted before user profile is saved and decrypted after retrieved from saved file.

3.6 VIEW and HTML_TEMPLATE_VIEW Classes

In current Framework, HTML_TEMPLATE_VIEW has been focused on the marker based HTML rewriting concept as described before. *cleanup_tags* and *cleanup_unused_sections* can be used in *post_processing* in your derived REQUEST_HANDLER class to cleanup the HTML page before sending it back to the DISPATCHER if necessary.

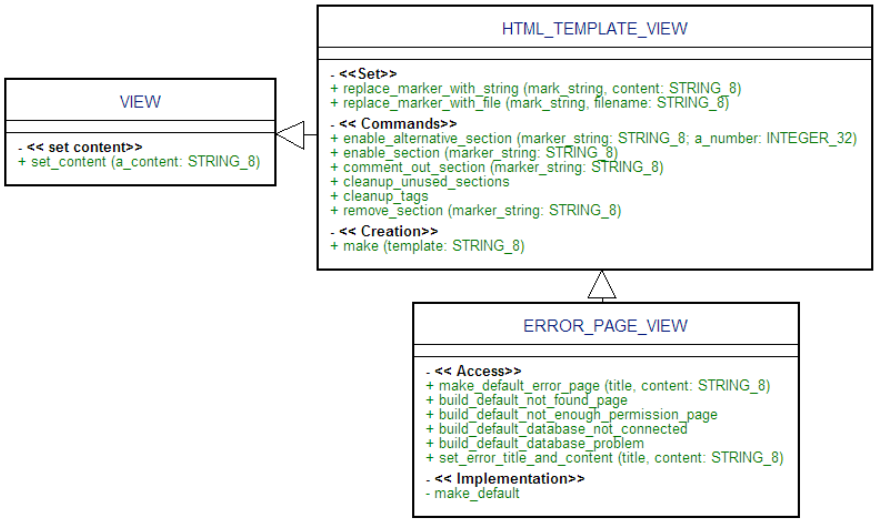

Fig 3.8 Implementation of VIEW and HTML_TEMPLATE_VIEW

Beside, a template based simple ERROR_PAGE_VIEW is implemented for creating HTML pages displaying some typical error messages.

3.7 FORM_VALIDATOR Class

As Fig 3.9, necessary attributes and features from CGI_FORM class and a reference to the actual REQUEST_DISPATCHER object is wrapped into a FORM_VALIDATOR class, together with some extended features especially for form validation.

Most features have combined a *field-defined* check and a *value-valid* check together for a better failure tolerance by form validation. A *last_value* STRING attribute is used to buffer the last value of a checked field. User can easily write a routine to validate a form input, collect values and generate customized error messages as demonstrated in the "**Computer Science Event List**" sample application.

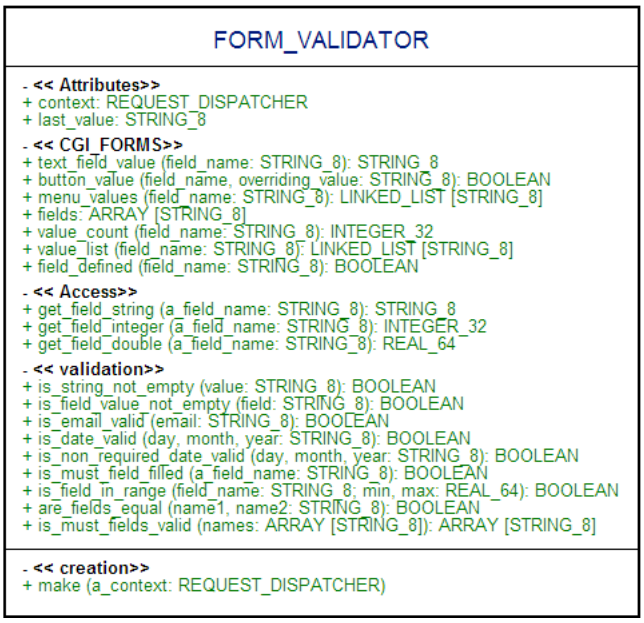

Fig 3.9 Implementation of FORM_VALIDATOR

3.8 ENCRYPTOR and ENIGMA Classes

A very simple ENCRYPTOR class and an ENIGMA implementation is included in this version of the Framework implementation. User can extend his own ENCRYPTORs for a securer encryption.

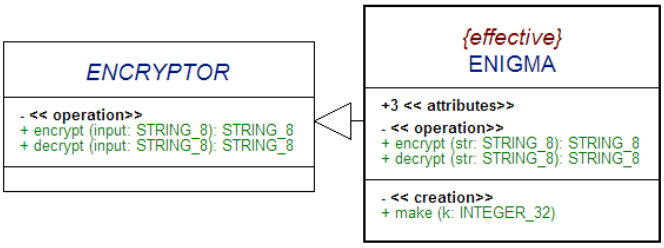

Fig 3.10 Implementation of ENCRYPTOR

4. Testing Applications

Because the expected interaction between a Web server and the Eiffel Web application is not straightforward to get emulated, instead of conventional test cases for each class, two example applications are programmed and used along the developing process for both functionality and Framework applicability testing.

4.1 "CSS Zen garden" revisited

At the beginning of the web framework development work, a simple application based on a revised XHTML page as from **CSS Zen Garden** (www.csszengarden.com) is written to test the basic functionality of the Framework and also demonstrate the power of Cascade Style Sheet in HTML domain.

With this simple application, most important functionalities, including multiple requests dispatching and handling based on configuration file, session support with and without cookie enabled, form validation and manipulation, user authentication and basic encryption, are demonstrated and tested along the development.

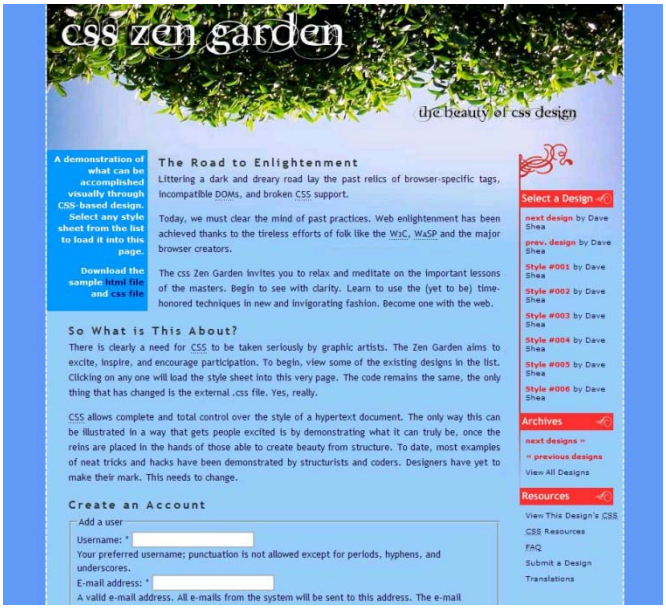

Fig 4.1 "CSS Zen Garden" revisited

4.2 "Computer Science Event List" application

In the later phase of the development, the "**Computer Science Event List**" application has been refactored by using the proposed web framework. It helped also very much for the later stage restructuring and code optimization of the Framework. To focus on the usage of the framework, a tutorial following the implementation of this application provides a quick-start guide. Please refer to the tutorial and included source code for more information.

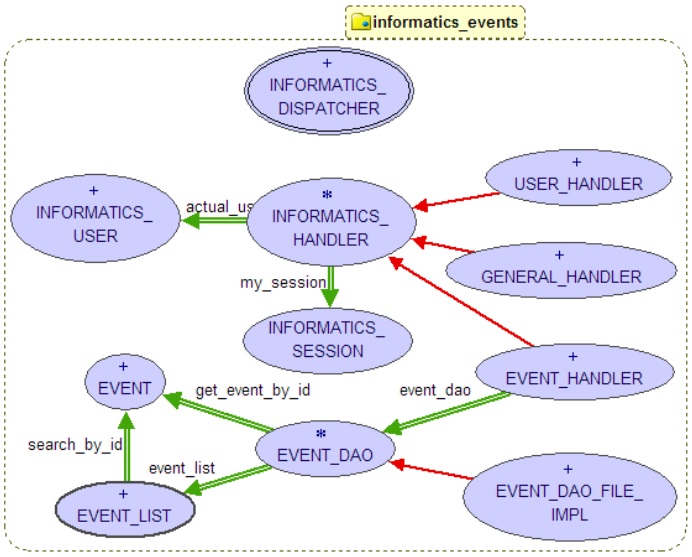

Fig 4.2 Computer Science Event List Implementation

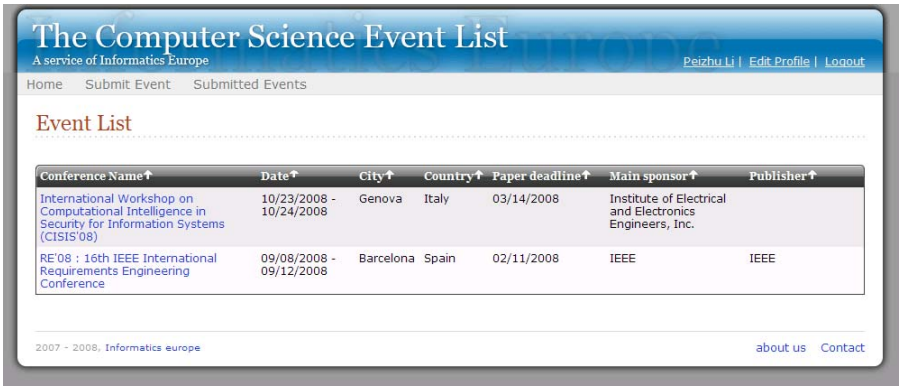

Fig 4.3 The Computer Science Event List

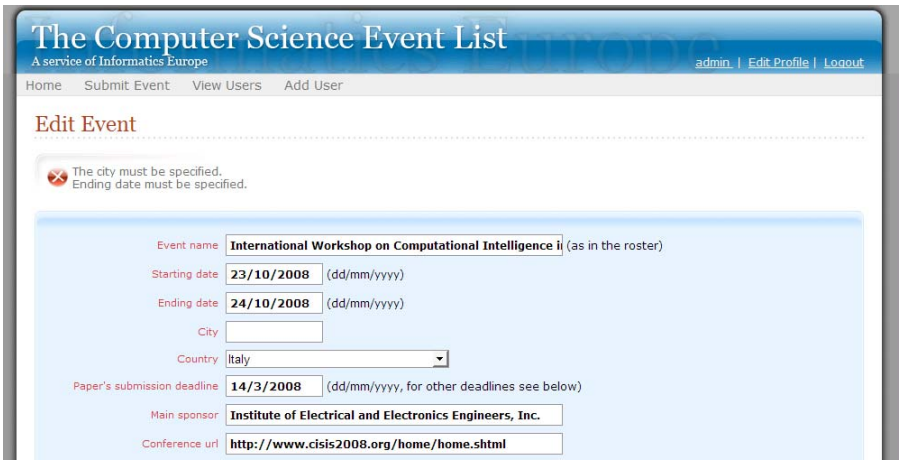

Fig 4.4 Event Submission Form Validation

5. Summary and Future Work

5.1 Summary

As shown in the sample applications, the proposed web framework provides a clear separation of concerns and a better level of support for web application development when compared with the previous EiffelWeb library. Supplied functionalities may also be further extended and optimized.

5.2 Future work

Due to the limited time-frame, some ideas are not implemented in the current Framework but might be worth some further consideration:

- XML format fits better to the configuration file. Either with an own XML parser extension or a third party library, migrate current application configuration file and CONFIG_READER class into a XML format version should still improve the extendibility.
- HTML_TEMPLATE_VIEW can be further optimized by utilizing a small parser and including a marker index in the class if performance issues arise.
- Another extension that can be implemented into the Web library is an AJAX interface, to further delegate the dynamic content in the result page.

6. References

[1] Software Engineering Course Project CSÁRDÁS, http://se.inf.ethz.ch/teaching/ss2007/252-0204-00/project.html

[2] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997

[3] Eiffel language online documentation: http://docs.eiffel.com/

[4] Seth Ladd with Darren Davison, Steven Devijver and Colin Yates: Expert Spring MVC and Web Flow, Apress, 2006

[5] Spring Framework Online Documentation, the Web, http://static.springframework.org/spring/docs/2.0.x/reference/spring-web.html

[6] MEWS – "More Eiffel Web Support" Project, http://mews.origo.ethz.ch/

[7] Gobo Eiffel Test Project, http://www.gobosoft.com/eiffel/gobo/getest/index.html

[8] AutoTest Tool, http://se.inf.ethz.ch/people/leitner/auto_test/

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[10] Struts framework: http://struts.apache.org/