Scheme in The Real World: A Case Study

Michael Bridgen, Noel Welsh, Matthias Radestock
LShift
Hoxton Point
6 Rufus Street
London, N1 6PE, UK
{mikeb, noel, matthias}@lshift.net

ABSTRACT

Scheme is the core technology in the back-end of the New Media Knowledge web site developed by LShift. This case study examines how we integrated Scheme into a commercial web development environment, and how we used unique features of Scheme to simplify and speed up development.

1. INTRODUCTION

New Media Knowledge (NMK, [1]) is a business resource for individuals and companies working in interactive digital media. NMK has recently acquired a grant to redevelop its web site. Lateral (a design company) and LShift combined to deliver the work, with Lateral creating the interface and LShift doing the back-end work. This case study details the technology LShift used to implement the website. In particular, we examine how we used Scheme to manage complex user interaction via the World Wide Web.

2. OVERVIEW

The main functions of the NMK web site are:

- Provide a calendar of events that NMK run, and a means for users to register for them; and
- Support a user community with contributed content.

The website is implemented as a standard three-tier system (see figure 1). The main technologies used in each tier are:

- presentation: Cocoon [2], Castor [3] and XSLT [4]
- business logic: SISC [19], JavaBeans [5], Object-Relational Bridge (OJB) [6]
- database: SQL Server [7]

The basis of the system is Java. This choice was a function of a number of factors: There are a number of tools available for Java that we have in-house experience with; Java has plenty of ready-made libraries available for standard tasks, like database connectivity and logging; Java application servers are well understood by support staff. Using Java gives us a wide choice of tools; on the other hand, most of these tools are more complicated than we really need. We are effectively trading some extra configuration and glue-code work for a widely familiar, pre-fabricated platform.

2.1 Presentation

The presentation layer uses Cocoon and XSLT. We chose XSLT as we have found it is usually close enough to HTML to be easily modified by designers. Additionally, we've found XSLT significantly faster and more enjoyable to develop with than other Java-based web presentation layers (e.g. JSP [8], FreeMarker [9]).

Cocoon is used to map URLs to actions. Cocoon uses a configuration document describing 'pipelines' that pair request criteria (a pattern for the URL, for example) with the procedure to generate the response through a sequence of actions and transformations. In the case of NMK, the most common steps are Scheme actions and XSL transformations. The Cocoon pipelines are kept very short at one or two stages maximum. This is a deliberate design decision; we have found programming with long Cocoon pipelines to be tedious and error prone: XSLT has poor abstraction facilities and long chains of XSLT transformations require maintaining too many complex contracts. In the NMK system, the input to XSLT is always XML representing a data object and the output is always a view of that object, usually HTML.

There can be performance problems with XSLT but this is not expected to be an issue with the load the NMK website experiences. Cocoon provides a fairly extensive infrastructure for caching inputs, stylesheets and results, and component pooling, which help in any case.

The majority of the business logic is written in Scheme and runs in the SISC Java-based Scheme interpreter. This case study concentrates on the business logic, which is where the most interesting action takes place.

2.2 From database to XML

There are two tasks we have to take care of before we can write the business logic:

- Getting data out of the database; and
- Transforming output into XML so the XSLT transformations can work on it.

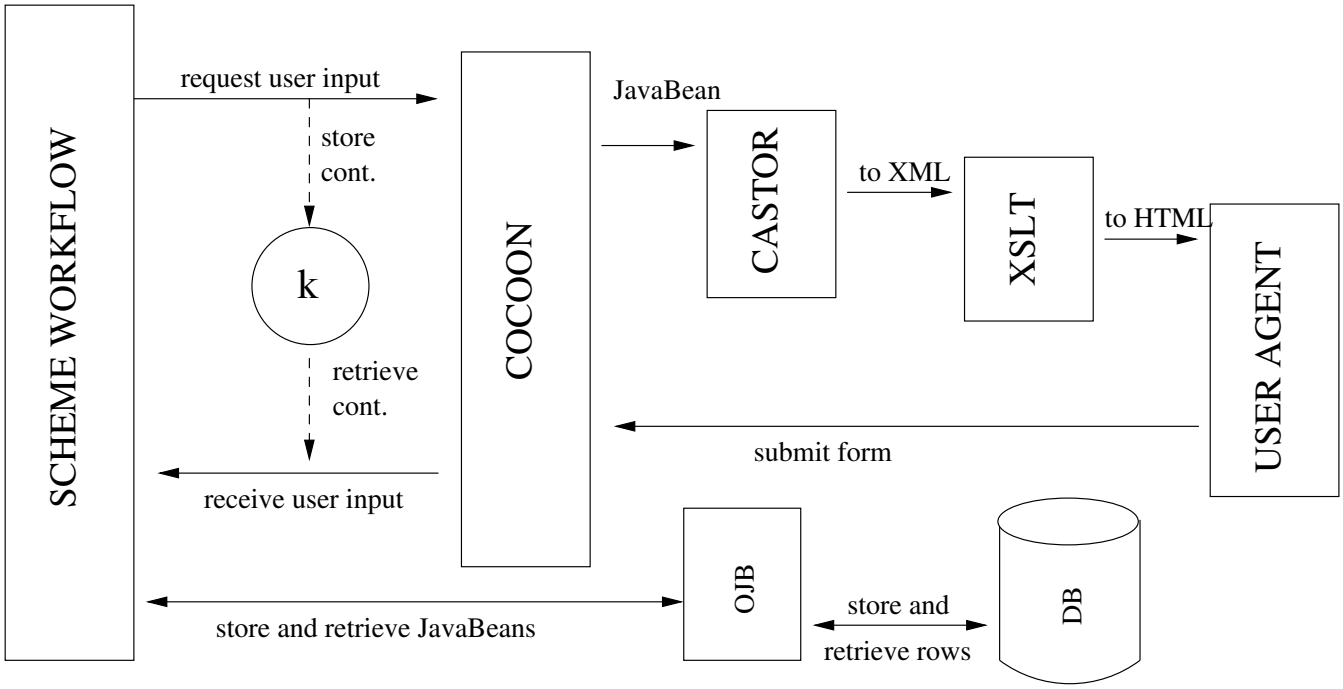

Figure 1: Overview of the components in the NMK website.

Each task is tedious to do manually. Luckily, there are ready-made Java tools that do the job for us.

The Apache Object-Relational Bridge (confusingly, "OJB") is used to map JavaBeans to database tables. It requires a bit of initial configuration, but once completed we only interact with a small API to load and save data. All other code deals solely with manipulating the JavaBeans. OJB provides a JDO [10] interface; we decided to forgo this in favour of the simpler OJB-specific API as there are only a few operations we need.

OJB also takes care of filling collections of dependent objects for us, though this causes complications as we will see later.

The choice of SQL Server is arbitrary (NMK already had a SQL Server license available). Through the use of the OJB object-relational mapping the business logic is insulated from the details of the underlying database. We could, in principle, use any database with a JDBC driver [11].

Transforming data into XML is done using the Castor framework. Castor generates JavaBean source code from XML Schema documents [12], as well as the code to marshal the JavaBeans to XML. Castor requires a bit of up-front configuration and some compromises on data types, but thereafter is invisible, as it generates standard JavaBeans. Most of the configuration is to get Castor to generate JavaBeans that are compatible with OJB (e.g., to use the correct collection classes). We manipulate and eventually transform the same JavaBeans we populate from the database. Castor allows us to add fields that are not reflected in the database so this does not limit our functionality.

3. SCHEME IN CONTEXT

Holding everything together is Scheme. Before discussing the particulars of the Scheme code, it is necessary to discuss how we integrate Scheme with Java and Cocoon.

3.1 The SISC Foreign Function Interface

SISC has a comprehensive Java Foreign Function Interface (FFI). Java classes can be instantiated and methods on those classes called as normal generic functions in SISC's object system. The integration with JavaBeans is slightly different, though as we will see later this offers benefits we can exploit. In short, if an object is called with a list of symbols, those symbols are interpreted as accessor functions to call on the object and successive objects returned from previous calls. So

(*bean* '(field1 field2 ...))

is equivalent to the Java code

```
bean.getField1().getField2()... ;
```

When we call a bean with a list of symbols and a value, that value is stored into the bean. So

(*bean* '(field1) *value*)

is equivalent to the Java code

```
bean.setField1(value);
```

Java methods are incorporated within SISC's generic procedures mechanism:

```
;; define a generic procedure for a Java method ...
(define method-one
  (generic-java-procedure '|methodOne|))

;; and now call it
(method-one obj)
```

In SISC, Java methods are dispatched on the runtime type of *all* arguments, not just the this argument. This means that we can do

some things in Scheme with Java methods that are much harder to accomplish in Java alone. For access control, for instance, we implement a class that has a static method for each of $(predicate \times class)$, where the predicates are like

```
static boolean isEditor(Article o, User u) {...}
static boolean isEditor(Meeting o, User u) {...}
```

If we declare isEditor as a generic procedure, then using it with different objects will 'just work'. In contrast, in the Java code that prepares content for indexing there are overloaded methods for creating index entries, but we have to do the dispatching ourselves. Java overloading uses the static type of the arguments, so to encode this pattern in Java requires a chain of instanceof tests[1].

```
public Document createDocument(Article o) {
  ...
}
public Document createDocument(Meeting o) {
  ...
}
public Document createDocument(User o) {
  ...
}

public Document createDocument(Indexable o) {
  return
    (o instanceof Article) ?
      createDocument((Article)o) :
    (o instanceof Meeting) ?
      createDocument((Meeting)o) :
    (o instanceof User)    ?
      createDocument((User)o) :
    null;
}
```

The generic procedure system in SISC is similar in feeling to that in Dylan [13] and CLOS [14].

SISC also supports defining classes that implement Java interfaces, by using the Java reflection API to create dynamic proxies. This is especially useful for creating event listeners, as these usually end up being anonymous classes. We can use a proxy with a closure as a listener to achieve the same effect; for example, using the Swing Timer class to schedule re-indexing:

```
(define-java-proxy (timer-task fn)
  (<swing.action-listener>)
  (define (action-performed proxy event)
    (fn event)))

(start (java-new <swing.timer>
                 (->jint 3600000)
                 (timer-task
                   (lambda (proxy event)
                     (re-index)))))
```

3.1.1 SISC and J2EE

To provide a persistent environment for the interpreter, we use the SISC servlet. The SISC servlet, as a standard feature, runs a REPL on a nominated port. Being able to run arbitrary code in the same context as the application is extremely handy for debugging and incrementally developing code in a running system.

The J2EE application server includes an application context that can be configured at deployment time. This is useful for storing application variables, like database connections, and deployment-specific file locations. All that is needed are a few wrappers to make it convenient in Scheme:

```
(define initial-ctx (make <javax.naming.InitialContext>))
(define local-ctx (lookup initial-ctx (->jstring "java:comp/env")))
(define (lookup-context-var name)
  (->string (lookup local-ctx (->jstring name))))

(define *SECRET-KEY* (lookup-context-var "secretKey"))
```

3.2 Transactions and the Back Button

A core part of the NMK website is presenting data to the user and allowing them to edit it. The editing cycle is basically structured as edit ↔ preview → commit. Within each cycle we want multiple browsers and the back button to work as expected. For a full description of the problem and general solution, see [20], and more recently [18]. There are a few issues here to get this all working correctly.

3.2.1 Scheme, interrupted

Naturally we use continuations so we can structure the program code clearly despite the inverted control of HTTP. We hide the continuations behind a `channel` abstraction. The channel API boils down to:

```
;; channel × symbol × bean → symbol × input
;;
;; Go get some data from the user
(channel-call channel action data)

;; Send some data to the user (never returns)
(channel-send channel action data)
```

The function *channel-call* provides the main abstraction for communicating with the user. It stuffs away the continuation in the user's HTTP session, and hands the (probably incomplete) JavaBean to Cocoon along with a stylesheet to use (the `action`). Cocoon serialises the JavaBean and runs it through the stylesheet. Usually the stylesheet produces a form, populated with data from the XML-ised JavaBean; submitting the form picks up the continuation from the HTTP session and takes up where it left off. This corresponds to the 'shortcut' in figure 1. We number the continuations per user and use the number as part of the URL for subsequent requests, so the user can branch at or revisit any point. This is an effective way of cloning similar content, for instance.

Scheme continuations get stored in the user session of the J2EE [15] application server. Persistence is accomplished by simply storing the serialised continuations in a database; many application servers deal with session migration by serialising the sessions and restoring them on the migration target. This means SISC web applications can be made **resilient** and **load-balanced** by simply configuring the appropriate session persistence/migration features of the application server. We have run experiments with Tomcat [16] involving restarting of the application server in the middle of a workflow, and it all works beautifully.

Using continuations in this way we are effectively righting the inverted interaction that HTTP tries to force on us. The usual model is to wait for the user to ask for something, figure out what they were trying to accomplish and provide an appropriate response. In our righted model, when we need data from the user, we just ask for

[1] or a modified virtual machine - see [17]

it - at least from the Scheme programmer's point of view. The following pseudo-code shows the advantages of this system - here we display some data to the user and save it following their approval:

```
(let-values (((action input) (channel-call channel 'show data)))
  (save data))
```

We do not need to do any parsing of URLs or form parameters to figure out what state the computation is at.

There are some implications of storing the continuation in the user session that have to be worked around:

- The continuation is only available while the user session lasts. We want continuations to expire at *some* point, and the typical session lifetime of around 30 minutes is reasonable. We keep workflow steps short to avoid the situation in which the user's session times out while they are filling in a form.
- When a user session is serialised, the first continuation is expensive as the closure captures all the lexical environments up the call-chain minus the top one; subsequent continuations to be serialised are relatively cheap, as they will share references. SISC only serialises the names of top-level definitions, so there is no chance of inadvertently serialising the entire state of the application.
- User sessions are isolated - we must avoid mutating application-level objects, so that users cannot cannot have different application states captured in their stored continuations.

3.2.2 SISC and Cocoon

The Scheme code is invoked by Cocoon, using a small class (SchemeAction) that implements one of the Cocoon plug-in interfaces. SchemeAction uses a SISC interpreter to evaluate a specified function. In the Cocoon configuration, for example, we declare a workflow for adding an article:

```
<map:match pattern="article/add/*">
  <map:call resource="scheme-workflow">
    <map:parameter name="src"
                   value="article-add" />
    <map:parameter name="kont" value="{1}" />
  </map:call>
</map:match>
```

The `pattern` attribute maps a URL starting with article/add to workflow. Whatever appears where the wildcard is, is passed to Scheme as a 'continuation number' (`kont`)so it can retrieve the continuation from the user's session. `src` parameter corresponds to the Scheme procedure *article-add-start*.

3.2.3 Avoid Mutation

Continuations make supporting multiple browsers and the back button easy, but we must be careful how we manage state. If we mutate values that could be captured by a continuation the changes will persist when the user moves backward. However our infrastructure relies on getting and setting properties in JavaBeans, an inherently stateful operation. Hence we always clone any JavaBeans we are using, and only update values in the clones.

This also applies to collections of dependent objects. Unfortunately, Java cloning operations are shallow so we have to clone collections ourselves:

```
(define (meeting-clone meeting)
  (let ([meeting (clone meeting)])
    (meeting '(meeting-file-as-reference)
             (clone (meeting '(meeting-file-as-reference))))
    meeting))
```

3.3 Code structure

We can divide the interesting Scheme code into three types of function:

- action - these do the interaction with the user, and dispatch to other actions or return depending on the result; and
- population - these take the user input and fill in the fields of a JavaBean.
- workflow - these fulfill the contract with Cocoon, usually by calling an action;

3.3.1 Actions: There and Back Again

Actions perform interaction with the user. All actions have the same type:

$$bean \times environment \times channel \rightarrow bean$$

This enables us to compose actions as simply as making a function call. A simple example is:

```
(define (meeting-preview meeting env channel)
  (with-input-from-channel (channel 'preview meeting => input)
    [ok meeting]
    [edit (meeting-edit meeting env channel)]
    [cancel #f]))
```

We have defined a macro, *with-input-from-channel*, that encapsulates the common pattern of using *channel-call* - get some data from the user, then dispatch to another action or return based on the button they pushed.

Notice the call to *meeting-edit*, which hides an interaction with the user. We use this facility a lot to go on 'excursions': reusable, atomic interactions such as uploading a file. We can make these interactions as complex as we need and control will return to where we left off. In effect we have function call semantics just like in a normal program.

3.3.2 Population

Filling a JavaBean with data from the database or a web form is a very common operation. For the former we use OJB and a few wrappers in Scheme for querying, updating and deleting objects. All database actions are done at the beginning or end of the workflow, to keep it transactional.

```
(define (article-add-start env channel input)
  (let ([article (make <nmk.Article>)])
    (article '(group) (group-info-by-id (input-parameter input 'id)))
    (let ([article (article-add article env channel)])
      (if article
          (begin
            (create-object article)
            (channel-send channel 'confirm-add article))
          (channel-send channel 'cancel jnull)))))
```

Here, the call to *article-add* populates the JavaBean, asking the user for more information in the process; once it has a result, it either saves the JavaBean to the database and sends a confirmation message to the user, or sends a cancel message.

We can use some features of Scheme and the SISC FFI to help us out with populating JavaBeans from user input. There are three main steps:

- Extract data from the request parameters
- Convert and validate the data if necessary
- Populate the JavaBean with converted data

A few simple utilities suffice:

```
;; (string → any) × (list-of symbol) × input
;; → (list-of any)
;;
;; Given a list of field names, returns a list of the inputs with
;; those names, converted by the conversion function.
(define (input-parameters/conversion conversion fields input)
 (map
  (lambda (field)
    (conversion (input-parameter input field)))
  fields))

;; bean × (list-of symbol) × (list-of (union any #f))) → bean
;;
;; Populate the given fields of the bean with the given converted
;; values
(define (fill-object obj fields values)
  (for-each (lambda (field value)
              (if value
                  (obj field value)))
            (map list fields)
            values)
  obj)
```

With these two utilities all we need to supply is the list of fields for each JavaBean, and the conversion functions to apply.

```
(define (meeting-edit-fill meeting username input)
  (define text-fields
    '(title abstract recurrence venue-title venue-description
      speakers sponsor-introduction |reportURL|))
  (define markup-fields '(body))
  (define time-fields '(start-time finish-time))
  ...
  (let ([meeting (meeting-clone meeting)])
    (meeting '(modified) (make <DateTime>))
    (fill-meeting meeting ->jstring text-fields input)
    (fill-meeting meeting string->markup markup-fields input)
    (fill-meeting meeting (safe-conversion string->jtime)
                          time-fields
                          input)
    ...
    (meeting '(modified) (make <DateTime>))
    meeting))
```

Converting between input strings and XML Schema/Castor, Java and JDBC types occasionally proves troublesome. In particular, expressing times (of the day) in each has its own peculiarities, leading to some circuitous functions

```
(define (string->jtime s)
  (if (string-null? s)
      (java-null <castor.Time>)
      (let ([res  (make <castor.Time>)]
            [c    (get-instance <Calendar>)])
        (c '(time) (parse time-format (->jstring s)))
        (for-each (lambda (set-field get-field)
                    (res (list set-field)
                         (->jshort (->number
                                    (jget c (<Calendar> get-field))))))
                  '(hour minute second)
                  '(|HOUR_OF_DAY| |MINUTE| |SECOND|))
        (set-utc res)
        res)))
```

3.3.3 Error handling and reporting

We distinguish between application errors and workflow errors. The former are run-time errors, and percolate through to the normal error mechanisms through SISC to Java to be reported by Cocoon. The latter are errors in the usage of the application; e.g., a user trying to edit something they are not permitted to edit. Workflow errors are wrapped in a JavaBean and sent through the normal Cocoon channel. We use Java resource bundles to localise the error messages, and the resource file to use is stored in an application variable so it can be set at deployment time.

3.3.4 Don't Repeat Yourself

A side-effect of using Scheme alongside other languages is that it is easy to end up with duplicated procedures.

Our strategy for avoiding duplication with Java was to write the common part in Java and refer to it in Scheme. For example, Cocoon is configured to map URL patterns corresponding to workflows to Scheme actions, and other URLs to Java classes that query for JavaBeans directly. This replicates some query-building requirements across Scheme and Java; we have minimised this by putting common query fragments into static methods of a utility class. We use a class QueryHelper to get query criteria in Java and Scheme:

```
Criteria crit =
  QueryHelper.makeDateAndTagCriteria(
       this.dateField,
       this.tagField,
       this.date,
       this.tag);
OjbQueryImpl query =
  new OjbQueryImpl(extentClass, crit);
  return query;
```

```
(define (object-by-date-and-tag
         type
         date-field
         tag-field
         date
         tag)
  (object-by-criteria
      type
      (make-date-and-tag-criteria
        <nmk.QueryHelper>
        (->jstring date-field)
        (->jstring tag-field)
        (to-date (string->jdate date))
        (->jstring tag))))
```

Another place that code can be duplicated is in the symmetry between *presenting* actions and *restricting* actions; we do not want to give the user choices that they cannot follow through on. In the NMK system, the presentation is done with XSLT while the restriction is computed in Scheme. Luckily, our channel abstraction lets us pass parameters to the XSLT, so we can calculate both in Scheme. Further, we can do it using the same, inlined data structure.

```
(define (article-preview-actions edit-func preview-func)
  (define (article-change-status-helper status)
          ...)
  (define (article-edit-helper)
          ...)
  (let* ([administrator-and-editor-pending-or-wip
          `((approve . ,(article-change-status-helper 'live))
            (comment . ,(article-change-status-helper 'wip))
            (later   . ,(article-change-status-helper 'pending))
            (reject  . ,(article-change-status-helper 'hidden))
            (edit    . ,(article-edit-helper))
            (cancel  . ,(lambda args #f)))]
         [administrator-and-editor-actions
          `((pending . ,administrator-and-editor-pending-or-wip)
            (wip     . ,administrator-and-editor-pending-or-wip)
            (live
             (edit    . ,(article-edit-helper))
             (approve . ,(article-change-status-helper 'live))
             (save    . ,(article-change-status-helper 'wip))
             (remove  . ,(article-change-status-helper 'hidden))
             (pull    . ,(article-change-status-helper 'pending))
             (cancel  . ,(lambda args #f))))]
         [user-pending-or-wip
          `((save    . ,(article-change-status-helper 'wip))
            (submit  . ,(article-change-status-helper 'pending))
            (edit    . ,(article-edit-helper))
            (cancel  . ,(lambda args #f)))])
    `((administrator . ,administrator-and-editor-actions)
      (editor        . ,administrator-and-editor-actions)
      (author        (pending . ,user-pending-or-wip)
                     (wip     . ,user-pending-or-wip))))
```

Beyond all the quasi-quoting, it is simply a structure with the allowed actions per user type and the resulting function to call. With this data structure in hand, we can ask what things is the user allowed to do:

```
(define (deep-assq keys alist)
  (and (pair? alist)
       (let ([key-car (car keys)]
             [key-cdr (cdr keys)])
         (cond [(assq key-car alist)
                => (lambda (val)
                     (if (null? key-cdr)
                         (cdr val)
                         (deep-assq key-cdr (cdr val))))]
               [else #f]))))

(define (available-actions user)
  (let ([actions
         (deep-assq (list (user-role user) (article-status article))
                    preview-actions)])
    (and actions (map car actions))))
```

... and dispatch to the appropriate function:

```
(let ([proc (deep-assq (list (user-role user)
                             (article-status article)
                             action)
                       preview-actions)])
  (and proc
       (proc article env channel)))
```

4. CONVENTIONS

Applying consistent conventions throughout the code base makes it much easier to understand the code. Some of the conventions we used include:

A noun-verb naming scheme, and standard verbs (add, edit, remove). The code uses consistent names like *meeting-edit* and *article-preview*, and the URLs to access these functions are simply `/meeting/<id>/edit` and `/article/<id>/preview` respectively. In addition to the readability benefits this provides a weak form of namespaces, as we do not use the SISC module system.

A common pattern for initiating actions. Each workflow begins as a function named *<action>-start* which initialises the JavaBean and then calls the actual action (named using our noun-verb scheme). We leverage this convention to reduce the integration with Cocoon to supplying a list of action names, which we process with a macro.

5. CONCLUSION

Our experience with Scheme has been very positive and the NMK back-end is known as one of the cleanest and best designed projects at LShift. We put our success down to two points:

Using existing Java frameworks where possible. We do not write any code to deal with the drudge work of connecting to a database or generating HTML. Interfacing with Java is occasionally painful (using the date/time API, for example) but in general it is more than worth it. Additionally we can express the user interface as XSLT, which web designers are already familiar with, by integrating with Cocoon.

The features of the Scheme language. Continuations allow us to work around HTTP's inverted control, and write code in a natural style. Higher-order functions and the SISC FFI allow us to write very expressive, compact code. Writing functions to fit some standard contracts means we can compose units of user interaction into complex workflows.

6. REFERENCES

[1] New Media Knowledge Website, http://www.nmk.co.uk/.

[2] Cocoon website, http://xml.apache.org/cocoon2/.

[3] Castor website, http://castor.exolab.org/.

[4] XSLT Recommendation, http://www.w3.org/TR/xslt.

[5] The JavaBeans specification is available from http://java.sun.com/products/ejb/docs.html.

[6] OJB website, http://db.apache.org/ojb/.

[7] Microsoft SQL Server product information, http://www.microsoft.com/sql/.

[8] The JSP specification is available from http://java.sun.com/products/jsp/download.html#specs.

[9] Freemarker website, http://freemarker.sourceforge.net/.

[10] The JDO specification is available from http://www.jcp.org/aboutJava/communityprocess/first/jsr012/.

[11] The JDBC specification is available from http://java.sun.com/products/jdbc/download.html.

[12] Information about XML Schema can be found at http://www.w3.org/XML/Schema.

[13] Detail at http://www.gwydiondylan.org/drm/drm_48.htm#HEADING48-0.

[14] Detail at http://www.lisp.org/table/references.htm#ansi.

[15] Information on Java 2 Enterprise Edition at http://java.sun.com/j2ee/docs.html.

[16] Tomcat website, http://jakarta.apache.org/tomcat/index.html.

[17] C. Dutchyn. Multi-dispatch in the java virtual machine: Design and implementation, 2001.

[18] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions, 2003.

[19] S. G. Miller. SISC Scheme Interpreter website, http://sisc.sourceforge.net.

[20] C. Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. *ACM SIGPLAN Notices*, 35(9):23–33, 2000.