WMR 2006

First International Workshop on Web Maintenance and Reengineering

24-March-2006, Bari, Italy

co-located with the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)

Theme and goals

Traditionally, in the software engineering field, a lot of effort is dedicated to design/model, project, and implement software. In fact the importance of designing a robust and well written software system is known and recognized by the industry and the scientific community. However, the "activities" related to general software maintenance (including re-engineering and reverse engineering), and evolution are less addressed. Moreover, the techniques for re-engineering and software maintenance are mostly focused on traditional software rather than Web software (i.e., Web sites, Web applications, or Web services). This workshop/working session aims at evaluating, identifying, and discussing the following example themes:

- Are traditional techniques fully applicable to Web software?
- What are the particular challenges posed by Web software in terms of maintenance, understanding and evolution?
- Which technique is best for XY Web platform? (Where XY represents a Web platform/language/etc of choice)
- Can software measurements be used to increase the quality of Web software maintenance? How it is possible?
- What are the best metrics for Web applications? Do all metrics are meaningful for every Web platform/language/etc?
- What techniques/methods may be useful in order to control Web software changes, versions or migration?
- How reengineering methods can be used to increase the quality of new Web applications?
- Can Web reverse engineering be used to simplify or guide Web software maintenance (including reengineering)?
- Can reverse engineered information (models, source code, etc.) be used to maintenance, evolve and reengineer Web software?

Topics

The goal of the workshop is to identify the most practical and effective (i.e., covering widespread implementation platforms and languages) techniques for web reengineering, evolution and maintenance. We encourage original submissions in any field of Web maintenance and reengineering. Areas of particular interest include (but are not limited to):

- Maintainability analysis and prediction
- Software architecture recovery and evolution
- Machine learning approaches for software maintenance
- Model-driven Web software engineering
- Software restructuring, refactoring and renovation
- Feature identification, extraction and analysis
- Slicing and change analysis
- Reverse engineering techniques
- Techniques, environment and technologies for reengineering
- Evolutionary algorithms or intelligent systems to support reengineering and manintenance
- Effort and cost estimation
- Metrics-based rules for detecting design flaws
- Monitoring the evolution of a system with metrics
- Testing techniques for maintenance and evolution
- Defect rate and reliability prediction
- Aspect-oriented programming on Web software maintenance

Organizers

Andrea Trentini, Alessandro Marchetto and Carlo Bellettini
{andrea.trentini, alessandro.marchetto, carlo.bellettini}@unimi.it
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano,
via Comelico 39/41, I-20135 Milano,
Italy.

Program committee

WMR 2006 Program

The Core NSP Type System

Dirk Draheim (Freie Universität Berlin, Germany)

Gerald Weber (University of Auckland, New Zealand)

Style-based Architectural Analysis for Migrating a Webbased Regional Trade Information System

Simon Giesecke (Carl von Ossietzky University, Germany)

Johannes Bornhold (Carl von Ossietzky University, Germany)

A framework for Web Applications Testing through Object-Oriented approach and XUnit tools

Alessandro Marchetto (Università di Milano, Italy)

Andrea Trentini (Università di Milano, Italy)

Supporting the Evolution of Service Oriented Web Applications using Design Patterns

Manolis Tzagarakis (University of Patras Campus, Greece)

Michail Vaitis (University of the Aegean, Greece)

Nikos Karousos (University of Patras, Greece)

Towards Empirical Validation of Design Notations for Web Applications: An Experimental Framework

Paolo Tonella (ITCirst, Trento, Italy)

Filippo Ricca (ITCirst, Trento, Italy)

Massimiliano Di Penta (University of Sannio, Benevento, Italy)

Marco Torchiano (Politecnico di Torino, Italy)

User-Centred reverse engineering: Genesis-D project

Luca Mainetti (Università di Lecce, Italy)

Roberto Paiano (Università di Lecce)

Andrea Pandurino (Università di Lecce, Italy)

The Core NSP Type System

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
draheim@acm.org

Gerald Weber
Department of Computer Science
University of Auckland
g.weber@cs.auckland.ac.nz

ABSTRACT

A good deal of software development and maintenance costs for Web applications stem from the fact that the untyped, flat message concept of the CGI interface has its footprint in the commonly used Web programming models and Web development technologies. Still it is necessary to reengineer large legacy Web applications that have been developed without the help of an improved Web technology. Current web application frameworks offer support to deal with client page type errors dynamically, however, no static type checks are provided by these tools. Furthermore, they do not allow for detecting potential web page description errors at compile time. In this paper, the static semantics of a new typed server pages approach is defined as an algorithmic, equi-recursive type system with respect to an amalgamation with a minimal imperative programming language and a collection of sufficiently complex programming language types.

1. INTRODUCTION

In [6] a strongly typed server pages technology NSP (Next Server Pages) has been proposed. The NSP type system has also been exploited in the design and implementation of the reverse engineering tool JSPick [5]. JSPick can recover the design and type structure of a web presentation layer that is based on server pages technology.

In this paper the type system of NSP is defined formally. Server pages technologies are widely used in the implementation of ultra-thin client applications. Unfortunately the low-level CGI programming model shines through in these technologies, especially user data is gathered in a completely untyped manner. The NSP contributions target stability and reusability of server pages based systems. The findings are programming language independent.

Important contributions of NSP has been the following:

Parameterized server pages. A server page possesses a specified signature that consists of formal parameters, which are native typed with respect to a type system of a high-level programming language. A server page signature is termed web signature in the sequel.

Statically ensured client page type and description safety. The type correct interplay of dynamically generated forms and targeted server pages is checked at compile-time. It is checked at compile-time if generated page descriptions are always valid with respect to a defined client page description language.

Support for complex types in writing forms. New structured tags are offered for gathering arrays and objects of user defined types.

Functional decomposition of server pages In NSP a server-side call to a server page is designed as a parameter-passing procedure call, too. This helps decoupling architectural issues and implementing design patterns.

Higher order server pages. Server pages may be actual form parameters. This enables improved web-based application architecture and design.

Exchange of objects across the web user agent. Server side programmed objects may be actual form parameters and therefore passed to client pages and back, either as messages or virtually as objects.

The NSP concepts are reusable, programming language independent results. They must be amalgamated with a concrete programming language. The NSP concepts are designed in a way that concrete amalgamations are conservative with respect to the programming language. That is the semantics of the programming language and especially its type system remain unchanged in the resulting technology. In [6] the NSP concepts are explained through a concrete amalgamation with the programming language Java. As a result of conservative amalgamation the NSP approach does not restrict the potentials of the considered programming language in any way, for example in the case of Java the Servlet session facility for state handling is available as a matter of course.

The NSP coding rules [6] give an informal explanation of NSP type correctness. They are easy to learn and will help in everyday programming tasks, but may give rise to ambiguity. This paper formally defines the type system of Core NSP, which is the amalgamation of NSP concepts with a minimal imperative programming language. This enables precise reasoning about the NSP concepts.

2. CORE NSP GRAMMAR

In this section the abstract syntax of Core NSP programs is specified[1]. A Core NSP program is a whole closed system of several server pages. A page is a parameterized core document and may be a complete web server page or an include server page:

```
system ::= page | system system
page ::= <nsp name="id"> websig-core </nsp>
websig-core ::= param websig-core
param ::= <param name="id" type="param-type"/>
websig-core ::= webcall | include
webcall ::= <html> head body </html>
head ::= <head><title> strings </title></head>
strings ::= ε | string strings
body ::= <body> dynamic </body>
include ::= <include> dynamic </include>
param-type ::= t ∈ 𝕋 ∪ ℙ
supported-type ::= t ∈ 𝔹_supported
```

There are some basic syntactic categories. The category id is a set of labels. The category string consists of character strings. The category parameter-type consists of the possible formal parameter types, i.e. programming language types plus page types. The category supported-type contains each type for which a direct manipulation input capability exists. The respective Core NSP types are specified in section 4.

Parameterized server pages are based on a dynamic markup language, which combines static client page description parts with active code parts. The static parts encompass lists, tables, server side calls, and forms with direct input capabilities, namely check boxes, select lists, and hidden parameters together with the object element for record construction.

```
dynamic ::= dynamic dynamic | ε | string | ul
| li | table | tr | td | call | form | object
| hidden | submit | input | checkbox | select
| option | expression | code
```

Core NSP comprises list and table structures for document layout. All the XML elements of the dynamic markup language are direct subcategories of the category dynamic, which means that the grammar does not constrain arbitrary nesting of these elements. Instead of that the manner of use of a document fragment is maintained by the type system. We delve on this in section 3.

```
ul ::= <ul> dynamic </ul>
li ::= <li> dynamic </li>
table ::= <table> dynamic </table>
tr ::= <tr> dynamic </tr>
td ::= <td> dynamic </td>
```

The rest of the static language parts address server side page calls, client side page calls and user interaction. A call may contain actual parameters only. The call element may contain no element, denoted by $\varepsilon_{\mathtt{act}}$.

```
call ::= <call callee="id"> actualparams </call>
actualparams ::= ε_act | actualparam actualparams
actualparam ::=
<actualparam param="id"> expr </actualparam>
form ::= <form callee="id"> dynamic </form>
object ::= <object param="id"> dynamic </object>
hidden ::= <hidden param="id"> expr </hidden>
submit ::= <submit/>
input ::=
<input type="supported-type" param="id"/>
checkbox ::= <checkbox param="id"/>
select ::= <select param="id"> dynamic </select>
option ::= <option> <value> expr </value>
<label> expr </label> </option>
```

Core NSP comprises expression tags for direct writing to the output and code tags in order to express the integration of active code parts with layout parts. The possibility to integrate layout code into active parts is needed. It is given by reversing the code tags. This way all Core NSP programs can be easily related to a convenient concrete syntax.

```
expression ::= <expr> expr </expr>
code ::= <code> com </code>
com ::= </code> dynamic <code>
```

The imperative sublanguage of Core NSP comprises statements, command sequences, an if-then-else construct and a while loop.

```
com ::= stat | com ; com
| if expr then com else com | while expr do com
```

The only statement is assignment. Expressions are just variable values or deconstructions of complex variable values, i.e. arrays or user defined typed objects.

```
stat ::= id := expr
expr ::= id | expr.id | expr[expr]
```

Core NSP is not a working programming language. It posseses only a set of most interesting features to model all the complexity of NSP technologies. Core NSP, in contrast, aims to specify the typed interplay of server pages, the interplay of static and active server page parts and the non-trivial interplay of the several complex types, i.e. user defined types and arrays, which arise during dynamically generating user interface descriptions.

[1]Nonterminals are underlined. Terminals are not emphasized. Every nonterminal corresponds to a syntactic category. In the grammar a syntactic category is depicted in bold face.

3. CORE NSP TYPE SYSTEM STRENGTH

The grammar given in section 2 does not prevent arbitrary nestings of the several Core NSP dynamic tag elements. Instead necessary constraints on nesting are guaranteed by the type system. Therefore the type of a server page fragment comprises information about the manner of use of itself as part of an encompassing document.

As a result some context free properties are dealt with in static semantics. There are pragmatic reasons for this. Consider an obvious example. In HTML forms must not contain other forms. Furthermore some elements like the ones for input capabilities may only occur inside a form. If one wants to take such constraints into account in a context free grammar, one must create a nonterminal for document fragments inside forms and duplicate and appropriately modify all the relevant production rules found so far. If there exist several such constraints the resulting grammar would quickly become unmaintainable. For that reason the Standard Generalized Markup Language (SGML) supports the notions of exclusion and inclusion exception. Indeed the SGML exception notation does not add to the expressive power of SGML [26], because an SGML expression that includes exceptions can be translated into an extended context free grammar [10]. The transformation algorithm given in [10] produces $2^{2^{|\mathbb{N}|}}$ nonterminals in the worst case. This shows: if one does not have the exception notation at hand then one needs another way to manage complexity. The Core NSP way is to integrate necessary information into types.

Furthermore in NSP the syntax of the static parts is orthogonal to the syntax of the active parts, nevertheless both syntactic structures must regard each other. Again excluding wrong documents already by abstract syntax amounts to duplicate production rules for the static parts that may be contained in dynamic parts.

4. CORE NSP TYPES

In this section the types of Core NSP and the subtype relation between types are introduced simultanously. There are types for modeling programming language types, and special types for server pages and server page fragments. The Core NSP types are given by a family of recursively defined type sets. Every type represents an infinite labeled regular tree.

The subtype relation formalizes the relationship of actual client page parameters and formal server page parameters by adopting the Barbara Liskov principle [11]. A type A is subtype of another type B if every actual parameter of type A may be used in server page contexts requiring elements of type B. The subtype relation is defined as the greatest fix point of a generating function. The generating function is presented by a set of convenient judgment rules for deriving judgements of the form $\vdash\ S < T$.

4.1 Programming Language Types

In order to model the complexity of current high-level programming language type systems, the Core NSP types comprise basic types $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$, array types $\mathbb{A}$, record types $\mathbb{R}$, and recursive types $\mathbb{Y}$. $\mathbb{B}_{primitive}$ models types, for which no null object is provided automatically on submit. $\mathbb{B}_{supported}$ models types, for which a direct manipulation input capability exists. The set of all basic types $\mathbb{B}$ is made of the union of $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$. Record types and recursive types play the role of user defined form message types. The recursive types allow for modeling cyclic user defined data types. The types introduced so far and the type variables $\mathbb{V}$ together form the set of programming language types $\mathbb{T}$.

$$\mathbb{T} = \mathbb{B} \cup \mathbb{V} \cup \mathbb{A} \cup \mathbb{R} \cup \mathbb{Y}$$
$$\mathbb{B} = \mathbb{B}_{primitive} \cup \mathbb{B}_{supported}$$
$$\mathbb{B}_{primitive} = \{\texttt{int}, \texttt{float}, \texttt{boolean}\}$$
$$\mathbb{B}_{supported} = \{\texttt{int}, \texttt{Integer}, \texttt{String}\}$$
$$\mathbb{V} = \{X, Y, Z, \ldots\} \cup \{\texttt{Person}, \texttt{Customer}, \texttt{Article}, \ldots\}$$

For every programming language type, there is an array type. According to subtyping rule 1 every type is subtype of its immediate array type. In commonly typed programming languages it is not possible to use a value as an array of the value's type. But the Core NSP subtype relation formalizes the relationship between actual client page and formal server page parameters. It is used in the NSP typing rules very targeted to constrain data submission. A single value may be used as an array if it is submitted to a server page. Judgment rule 2 is the preserving subtyping[2] rule for array types.

$$\mathbb{A} = \{\ \texttt{array of}\ T \mid T \in \mathbb{T} \setminus \mathbb{A}\}$$

$$\frac{}{\vdash T < \texttt{array of}\ T} \tag{1}$$

$$\frac{\vdash S < T}{\vdash \texttt{array of}\ S < \texttt{array of}\ T} \tag{2}$$

The usage of some Z Notation [20] for record types will ease writing type operator definitions and typing rules later on: a record type is a finite partial function from a set of labels to the set of programming language types.

$$\mathbb{R} = \textbf{Label} \ \mathrel{+\!\!\!\!\!\!\;+\!\!\!\!\rightarrow}\ \mathbb{T}$$

$$\frac{T_j \notin \mathbb{B}_{primitive} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{3}$$

$$\frac{\vdash S_1 < T_1 \ldots \vdash S_n < T_n}{\vdash \{l_i \mapsto S_i\}^{i \in 1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{4}$$

Rule 4 is just the necessary preserving subtyping rule for records. The establishing subtyping rule 3 states that a shorter record type is subtype of a longer record type, provided the types are equal with respect to labeled type variables. At a first site this contradicts the well-known rules for subtyping records [3] or objects [1]. But there is no contradiction, because these rules describe hierarchies of feature support and we just specify another phenomenon: rule 3

[2]We informally distinguish between establishing subtyping rules and preserving subtyping rules. The establishing subtyping rules introduce initial NSP specific subtypings. The preserving subtyping rules are just the common judgements that deal with defining the effects of the various type constructors on the subtype relation.

models that an actual record parameter is automatically filled with null objects for the fields of non-primitive types that are not provided by the actual parameter, but expected by the formal parameter.

The Core NSP type system encompasses recursive types for modeling the complexity of cyclic user defined data types. Type variables may be bound by the recursive type constructor μ. Overall free type variables, that is type variables free in an entire Core NSP system resp. complete Core NSP program, represent opaque object reference types.

$$\mathbb{Y} = \{\ \mu\, X\ .\ R \mid X \in \mathbf{V}\ ,\ R \in \mathbb{R}\ \}$$

$$\frac{\vdash S[{}^{\mu X.S}/_{X}] < T}{\vdash \mu X.S < T} \qquad \frac{\vdash S < T[{}^{\mu X.T}/_{X}]}{\vdash S < \mu X.T} \tag{5}$$

We have chosen to handle recursive types in an equi-recursive way [7]. Core NSP types represent finite trees or possibly infinite regular trees [4]. Type equivalence is not explicitly defined, it is given implicitly by the subtype relation. The subtype relation is defined as the greatest fixpoint of a monotone generating function on the universe of type trees [7]. The Core NSP subtyping rules provide an intuitive description of this generating function. Thereby the subtyping rules for left folding and right folding (5) provide the desired recursive subtyping. Beyond this only one further subtyping rule is needed, namely the rule 6 for introducing reflexivity.

$$\frac{}{\vdash T < T} \tag{6}$$

4.2 Server Page Types

In order to formalize the NSP coding rules the type system of Core NSP comprises server page types $\mathbb{P}$, web signatures $\mathbb{W}$, a single complete web page type $\Box \in \mathbb{C}$, document fragment types $\mathbb{D}$, layout types $\mathbb{L}$, tag element types $\mathbb{E}$, form occurence types $\mathbb{F}$ and system types $\mathbb{S}$. A server page type is a functional type, that has a web signature as argument type. An include server page has a dynamic document fragment type as result type, and a web server page the unique complete web page type.

$$\mathbb{P} = \{\ w \rightarrow r \mid w \in \mathbb{W}\ ,\ r \in \mathbb{C} \cup \mathbb{D}\ \}$$

$$\mathbb{W} = \mathbf{Label} \ \mathrel{+\!\!\!\!+\!\!\!\!\rightarrow}\ (\mathbb{T} \cup \mathbb{P}) \qquad \mathbb{C} = \{\Box\}$$

A web signature is a record. This time a labeled component of a record type is either a programming language type or a server page type, that is the type system supports higher order server pages. Noteworthy a clean separation between the programming language types and the additional NSP specific types is kept. Server page types may be formal parameter types, but these formal parameters can be used only by specific NSP tags. Server pages deliberately become no first class citizens, because this way the Core NSP models conservative amalgamation of NSP concepts with a high-level programming language. The preserving subtyping rule 4 for records equally applies to web signatures. The establishing subtyping rule 3 must be slightly modified resulting in rule 7, because formal parameters of server page type must always be provided, too.

Subtyping rule 8 is standard and states, that server page types are contravariant in the argument type and covariant in the result type.

$$\frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7}$$

$$\frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \rightarrow R < w' \rightarrow R'} \tag{8}$$

A part of a core document has a document fragment type. Such a type consists of a layout type and a web signature. The web signature is the type of the data, which is eventually provided by the document fragment as part of an actual form parameter. If a web signature plays part of a document fragment type it is also called form type. The layout type constrains the usability of the document fragment as part of an encompassing document. It consists of an element type and a form occurence type.

$$\mathbb{D} = \mathbb{L} \times \mathbb{W} \qquad \mathbb{L} = \mathbb{E} \times \mathbb{F}$$

$$\frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)} \tag{9}$$

Subtyping rule 9 is standard for products and applies both to layout and tag element types. An element type partly describes where a document fragment may be used. Document fragment that are sure to produce no output have the neutral document type $\circ$. Examples for such neutral document parts are hidden parameters and pure Java code. Document fragments that may produce visible data like string data or controls have the output type $\bullet$. Document fragments that may produce list elements, table data, table rows or select list options have type **LI**,**TD**, **TR** and **OP**. They may be used in contexts where the respective element is demanded. Neutral code can be used everywhere. This is expressed by rule 10.

$$\mathbb{E} = \{\ \circ, \bullet, \mathbf{TR}, \mathbf{TD}, \mathbf{LI}, \mathbf{OP}\}$$

$$\frac{T \in \mathbb{E}}{\vdash \circ\ <\ T} \tag{10}$$

The form occurrence types further constrain the usability of document fragments. Fragments that must be used inside a form, because they generate client page parts containing controls, have the inside form type $\Downarrow$. Fragments that must be used outside a form, because they generate client page fragments that already contain forms, have the outside form type $\Uparrow$. Fragments that may be used inside or outside forms have the neutral form type $\Updownarrow$. Rule 11 specifies, that such fragments can play the role of both fragments of outside form and fragments of inside form type.

$$\mathbb{F} = \{\ \Downarrow, \Uparrow, \Updownarrow\ \} \qquad \mathbb{S} = \{\ \Diamond, \surd\ \}$$

$$\frac{T \in \mathbb{F}}{\vdash \Updownarrow < T} \tag{11}$$

An NSP system is a collection of NSP server pages. NSP systems that are type correct receive the well type $\diamond$. The complete type $\surd$ is used for complete systems. A complete system is a well typed system where all used server page names are defined, i.e. are assigned to a server page of the system, and no server page names are used as variables.

5. TYPE OPERATORS

In the NSP typing rules in section 7 a central type operation, termed form type composition $\odot$ in the sequel, is used that describes the composition of form content fragments with respect to the provided actual superparameter type. First an auxiliary operator $*$ is defined. If applied to an array the operater lets the type unchanged, otherwise it yields the respective array type.

$$T* \equiv_{\texttt{DEF}} \begin{array}{ll} \texttt{array of } T & , T \notin \mathbb{A} \\ T & , \mathit{else} \end{array}$$

The form type composition $\odot$ is the corner stone of the NSP type system. Form content provides direct input capabilities, data selection capabilities and hidden parameters. On submit an actual superparameter is transmitted. The type of this superparameter can be determined statically in NSP, it is called the form type (section 4.2) of the form content. Equally document fragments, which dynamically may generate form content, have a form type. Form type composition is applied to form parameter types and describes the effect of sequencing document parts. Consequently form type composition is used to specify typing with respect to programming language sequencing, loops and document composition.

$$w_1 \odot w_2 \equiv_{\texttt{DEF}}$$
$$\begin{array}{llll} \bot & ,\boldsymbol{if} & \exists (l_1 \mapsto T_1) \in w_1 \bullet \exists (l_2 \mapsto T_2) \in w_2 \bullet & \\ & & l_1 = l_2 \wedge P_1 \in \mathbb{P} \wedge P_2 \in \mathbb{P} & \\ \bot & ,\boldsymbol{if} & \exists (l_1 \mapsto T_1) \in w_1 \bullet \exists (l_2 \mapsto T_2) \in w_2 \bullet & \\ & & l_1 = l_2 \wedge T_1 \sqcup T_2 \ \mathit{undefined} & \\ & & & \\ & \multicolumn{2}{l}{(\texttt{dom } w_2) \lhd w_1 \cup (\texttt{dom } w_1) \lhd w_2} & \\ \cup & & l \mapsto (T_1 \sqcup T_2)* \mid & ,\boldsymbol{else} \\ & & (l \mapsto T_1) \in w_1 \wedge (l \mapsto T_2) \in w_2 & \end{array}$$

If a document fragment targets a formal parameter of a certain type and another document fragment does not target this formal parameter, then and only then the document resulting from sequencing the document parts targets the given formal parameter with unchanged type. That is, with respect to non-overlapping parts of form types, form type composition is just union. With antidomain restriction notation [20] this is specified succinctly in the $\odot$ operator definition.

Two document fragments that target the same formal parameters may be sequenced, if the targeted formal parameter types are compatible for each formal parameter. NSP types are compatible if they have a supertype in common. The NSP subtype relation formalizes when an actual parameter may be submitted to a server page: if its type is a subtype of the targeted formal parameter. So if two documents have targeted parameters with compatible types in common only, the joined document may target every server page that fulfills the following: formal parameters that are targeted by both document parts have an array type, because of sequencing a single data transmission cannot be ensured in neither case, thereby the array items' type must be a common supertype of the targeting actual parameters. This is formalized in the $\odot$ operator definition: for every shared formal parameter a formal array parameter of the least common supertype belongs to the result form type. The least common supertype of two types is given as least upper bound of the two types, which is unique up to the equality induced by recursive subtyping itself.

The error cases in the $\odot$ operator definition are equally important. The $\odot$ operator is a partial function. If two document fragments target a same formal parameter with non-compatible types, they simply cannot be sequenced. The $\odot$ operator is undefined for the respective form types. More interestingly, two document fragments that should be composed must not target a formal server page parameter. This would result in an actual server page parameter array which would contradict the overall principle of conservative language amalgamation.

Form type composition can be characterized algebraically. The web signatures form a monoid ($\mathbb{W}$, $\odot$, $\emptyset$) with the $\odot$ operator as monoid operation and the empty web signature as neutral element. The operation $(\lambda v.v \odot w)_w$ is idempotent for every arbitrary fixed web signature w, which explains why the typing rule 23 for loop-structures is adequate.

6. ENVIRONMENTS AND JUDGEMENTS

In the NSP type system two environments are used. The first environment Γ is the usual type environment. The second environment Δ is used for binding names to server pages, i.e. as a definition environment. It follows from their declaration that environments are web signatures. All definitions coined for web signatures immediately apply to the environments. This is exploited for example in the system parts typing rule 45.

$$\begin{array}{lll} \Gamma : \textbf{Label} \twoheadrightarrow\!\!\!\!\!\!+\ (\mathbb{T} \cup \mathbb{P}) & = \mathbb{W} \\ \Delta : \textbf{Label} \twoheadrightarrow\!\!\!\!\!\!+\ \mathbb{P} & \subset \mathbb{W} \end{array}$$

The Core NSP identifiers are used for basic programming language expressions, namely variables and constants, and for page identifiers, namely formal page parameters and server pages names belonging to the complete system. In some contexts, e.g. in hidden parameters or in select menu option values, both page identifiers and arbitrary programming language expressions are allowed. Therefore initially page identifiers are treated syntactically as programming language expressions. However a clean cut between page identifiers and the programming language is maintained, because the modeling of conservative amalgamation is an objective. The cut is provided by the premises of typing rules concerning such elements where only a certain kind of entity is allowed; e.g. in the statement typing rule 15 it is prevented that page identifiers may become program parts. The Core NSP type system relies on several typing judgements:

$$\begin{array}{ll}
\Gamma \vdash e : \mathbb{T} \cup \mathbb{P} & e \in \textbf{expr} \\
\Gamma \vdash n : \mathbb{D} & n \in \textbf{com} \cup \textbf{dynamic} \\
\Gamma \vdash c : \mathbb{P} & c \in \textbf{websig-core} \\
\Gamma \vdash a : \mathbb{W} & a \in \textbf{actualparams} \\
\Gamma, \Delta \vdash s : \mathbb{S} & s \in \textbf{system}
\end{array}$$

Eventually the judgment that a system has complete type is targeted. In order to achieve this, different kinds of types must be derived for entities of different syntactic categories. Expressions have programming language types or page types. Both programming language code and user interface descriptions have document fragment types, because they can be interlaced arbitrarily and therefore belong conceptually to the same kind of document. Parameterized core documents have page types. The actual parameters of a call element together provide an actual superparameter, the type of this is a web signature and is termed a call type. All the kinds of judgements so far work with respect to a given type environment. If documents are considered as parts of a system they must mutually respect defined server page names. Therefore subsystem judgements have to be given additionally with respect to the defintion environment.

7. TYPING RULES

The notion of Core NSP type correctness is specified as an algorithmic type system. Compared to a declarative version extra premises are needed in some of the typing rules, in some premises slightly bit more complex type patterns have to be used. However in the Core NSP type system these extra complexity fosters understandability. The typing rule 12 allows for extraction of an identifier typing assumption from the typing environment. Rules 13 and 14 give the types of selected record fields respectively indexed array elements.

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{12}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{13}$$

$$\frac{\Gamma \vdash e : \ \texttt{array of}\ T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{14}$$

Typing rule 15 introduces programming language statements, namely assignments. Only programming language variables and expression may be used, i.e. expressions must not contain page identifiers. The resulting statement is sure not to produce any output. It is possible to write an assignment inside forms and outside forms. If it is used inside a form it will not contribute to the submitted superparameter. Therefore a statement has a document fragment type which is composed out of the neutral document type, the neutral form type and the empty web signature. The empty string, which is explicitly allowed as content in NSP, obtains the same type by rule 16.

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \ : \ ((\circ, \Updownarrow), \emptyset)} \tag{15}$$

$$\frac{}{\Gamma \vdash \varepsilon : ((\circ, \Updownarrow), \emptyset)} \tag{16}$$

Actually in Core NSP programming language and user interface description language are interlaced tightly by the abstract syntax. The code tags are just a means to relate the syntax to common concrete server pages syntax. The code tags are used to switch explicitly between programming language and user interface description and back. For the latter the tags may be read in reverse order. However this switching does not affect the document fragment type and therefore the rules 17 and 18 do not, too.

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash <\texttt{code}>c</\texttt{code}> \ : \ D} \tag{17}$$

$$\frac{\Gamma \vdash d : D}{\Gamma \vdash </\texttt{code}>d<\texttt{code}> \ : \ D} \tag{18}$$

Rule 19 introduces character strings as well typed user interface descriptions. A string's type consists of the output type, the neutral form type and the empty web signature. Another way to produce output is by means of expression elements, which support all basic types and get by rule 20 the same type as character strings.

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{19}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash <\texttt{expr}>e</\texttt{expr}> \ : \ ((\bullet, \Updownarrow), \emptyset)} \tag{20}$$

Composing user descriptions parts and sequencing programming language parts must follow essentially the same typing rule. In both rule 21 and rule 22 premises ensure that the document fragment types of both document parts are compatible. If the parts have a common layout supertype, they may be used together in server pages contexts of that type. If in addition to that the composition of the parts' form types is defined, the composition becomes the resulting form type. Form composition has been explained in section 5.

$$\frac{\begin{array}{c} d_1, d_2 \in \textbf{dynamic} \\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \\ L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash d_1 \ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{21}$$

$$\frac{\begin{array}{cc} \Gamma \vdash c_1 : (L_1, w_1) & \Gamma \vdash c_2 : (L_2, w_2) \\ L_1 \sqcup L_2 \downarrow & w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash c_1 ; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{22}$$

The loop is a means of dynamically sequencing. From the type system's point of view it suffices to regard it as a sequence of twice the loop body as expressed by typing rule 23. For an if-then-else-structure the types of both branches must be compatible in order to yield a well-typed structure. Either one or the other branch is executed, so the least upper bound of the layout types and least upper bound of the form types establish the adequate new document fragment type.

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \ : \ (L, w \odot w)} \tag{23}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \quad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1 \texttt{else}\ c_2\ :\ D_1 \sqcup D_2} \tag{24}$$

Next the typing rules for controls are considered. The submit button is a visible control and must not occur outside a form, in Core NSP it is an empty element. It obtains the output type, the inside form type, and the empty web signature as document fragment type. Similarly an input control obtains the output type and the inside form type. But an input control introduces a form type. The type of the input control is syntactically fixed to be a widget supported type. The param-attribute of the control is mapped to the control's type. This pair becomes the form type in the control's document fragment type. Check boxes are similar. In Core NSP check boxes are only used to gather boolean data.

$$\frac{}{\Gamma \vdash <\texttt{submit}/>\ :\ ((\bullet, \Downarrow), \emptyset)} \tag{25}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \begin{array}{l} <\texttt{input type} = "T"\ \texttt{param} = "l"/>: \\ ((\bullet, \Downarrow), \{(l \mapsto T)\}) \end{array}} \tag{26}$$

$$\frac{}{\Gamma \vdash \begin{array}{l} <\texttt{checkbox}\quad \texttt{param} = "l"/>\ : \\ ((\bullet, \Updownarrow), \{(l \mapsto \mathit{boolean})\}) \end{array}} \tag{27}$$

Hidden parameters are not visible. They get the neutral form type as part of their fragment type. The value of the hidden parameter may be a programming language expression of arbitrary type or an identifier of page type.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \begin{array}{l} <\texttt{hidden param} = "l" > e </\texttt{hidden}>\ : \\ ((\circ, \Downarrow), \{(l \mapsto T)\}) \end{array}} \tag{28}$$

The select element may only contain code that generates option elements. Therefore an option element obtains the option type **OP** by rule 30 and the select element typing rule 29 requires this option type from its content. An option element has not an own param-element. The interesting type information concerning the option value is wrapped as an array type that is assigned to an arbitrary label. The type information is used by rule 29 to construct the correct form type.

$$\frac{\Gamma \vdash d :\ \ (\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}}{\Gamma \vdash \begin{array}{l} <\texttt{select}\quad \texttt{param} = "l" > d \\ </\texttt{select}>\ :\ ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{29}$$

$$\frac{\Gamma \vdash v\ :\ T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \mathbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \qquad <\texttt{value}> v </\texttt{value}> \\ \qquad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of}\ T)\}) \end{array}} \tag{30}$$

The object element is a record construction facility. The enclosed document fragment's layout type lasts after application of typing rule 31, whereas the fragment's form type is assigned to the object element's param-attribute. This way the superparameter provided by the enclosed document becomes a named object attribute.

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash \begin{array}{l} <\texttt{object param} = "l" > d </\texttt{object}>\ : \\ (L, \{(l \mapsto w)\}) \end{array}} \tag{31}$$

The form typing rule 32 requires that a form may target only a server page that yields a complete web page if it is called. Furthermore the form type of the form content must be a subtype of the targeted web signature, because the Core NSP subtype relations specifies when a form parameter may be submitted to a server page of given signature. Furthermore the form content's must be allowed to occur inside a form. Then the rule 32 specifies that the form is a vizible element that must not contain inside another form.

$$\frac{\Gamma \vdash l : w \to \Box \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash <\texttt{form}\quad \texttt{callee} = "l" > d </\texttt{form}> : ((e, \Uparrow), \emptyset)} \tag{32}$$

Now the layout structuring elements, i.e. lists and tables, are investigated. The corresponding typing rules 33 to 37 do not affect the form types and form occurrence types of contained elements. Only document parts that have no specific layout type, i.e. are either neutral or merely vizible, are allowed to become list items by rule 33. Only documents with list layout type may become part of a list. A well-typed list is a vizible element. The rules 35 to 37 work analogously for tables.

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{\Gamma \vdash <\texttt{li}> d </\texttt{li}>\ :\ ((\mathbf{LI}, F), w)} \tag{33}$$

$$\frac{\Gamma \vdash d : ((\mathbf{LI} \vee \circ, F), w)}{\Gamma \vdash <\texttt{ul}> d </\texttt{ul}>\ :\ ((\bullet, F), w)} \tag{34}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{\Gamma \vdash <\texttt{td}> d </\texttt{td}>\ :\ ((\mathbf{TD}, F), w)} \tag{35}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TD} \vee \circ, F), w)}{\Gamma \vdash <\texttt{tr}> d </\texttt{tr}>\ :\ ((\mathbf{TR}, F), w)} \tag{36}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TR} \vee \circ, F), w)}{\Gamma \vdash <\texttt{table}> d </\texttt{table}>\ :\ ((\bullet, F), w)} \tag{37}$$

As the last core document element the server side call is treated. A call element may only contain actual parameter elements. This is ensured syntactically. The special sign $\varepsilon_{\texttt{act}}$ acts as an empty parameter list if necessary. It has the empty web signature as call type. Typing rule 40 makes it possible that several actual parameter elements uniquely provide the parameters for a server side call. Rule 38 specifies, that a server call can target an include server page only. The call element inherits the targeted include server page's document fragment type, because this page will replace the call element if it is called.

$$\frac{\Gamma \vdash l : w \to D \qquad \Gamma \vdash \mathit{as} : v \qquad \vdash v < w}{\Gamma \vdash <\texttt{call}\quad \texttt{callee} = "l" > \mathit{as} </\texttt{call}>\ :\ D} \tag{38}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{39}$$

$$\frac{\Gamma \vdash \mathit{as} : w \qquad \Gamma \vdash e : T \qquad l \notin (\mathit{dom}\ w)}{\Gamma \vdash \begin{array}{l} <\texttt{actualparam}\quad \texttt{param} = "l" > e \\ </\texttt{actualparam}> \mathit{as}\ :\ w \cup \{(l \mapsto T)\} \end{array}} \tag{40}$$

With the typing rule 41 and 44 arbitrary document fragment may become an include server page, thereby the document fragment's type becomes the server page's result type. A document fragment may become a complete web page by typing rules 42 and 44 if it has no specific layout type, i.e. is neutral or merely visible, and furthermore is not intended to be used inside forms. The resulting server page obtains the complete type as result type. Both include server page cores and web server page cores start with no formal parameters initially. With rule 43 parameters can be added to server page cores. The rule's premises ensure that a new formal parameter must have another name than all the other parameters and that the formal parameter is used in the core document type-correctly. A binding of a type to a new formal parameter's name is erased from the type environment.

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash <\texttt{include}>\, d <\texttt{/include}> : \emptyset \to D} \tag{41}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset) \; t \in \textbf{strings} \; d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} <\texttt{html}> \\ \quad <\texttt{head}> \\ \quad\quad <\texttt{title}>\, t <\texttt{/title}> \\ \quad <\texttt{/head}> \\ \quad <\texttt{body}>\, d <\texttt{/body}> \\ <\texttt{/html}> : \emptyset \to \Box \end{array}} \tag{42}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \to D \qquad l \notin (dom\ w)}{\begin{array}{l} \Gamma \backslash (l \mapsto T) \vdash \\ <\texttt{param}\ \ \texttt{name} = "l" \ \ \texttt{type} = "T" / > \\ c : (w \cup \{(l \mapsto T)\}) \to D \end{array}} \tag{43}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\begin{array}{l} \Gamma \backslash (l \mapsto P), \{(l \mapsto P)\} \ \vdash \\ <\texttt{nsp}\ \ \texttt{name} = "l" >\, c <\texttt{/nsp}> \ : \ \Diamond \end{array}} \tag{44}$$

A server page core can become a well-typed server page by rule 44. The new server page name and the type bound to it are taken from the type environment and become the definition environment. An NSP system is a collection of NSP server pages. A single well-typed server page is already a system. Rule 45 specifies system compatibility. Rule 46 specifies system completeness. Two systems are compatible if they have no overlapping server page definitions. Furthermore the server pages that are defined in one system and used in the other must be able to process the data they receive from the other system, therefore the types of the server pages defined in the one system must be subtypes of the ones bound to their names in the other's system type environment.

$$\frac{\begin{array}{c} s_1, s_2 \in \textbf{system} \quad (dom\ \Delta_1) \cap (dom\ \Delta_2) = \emptyset \\ ((dom\ \Gamma_2) \lhd\ \Delta_1) < ((dom\ \Delta_1) \lhd\ \Gamma_2) \\ ((dom\ \Gamma_1) \lhd\ \Delta_2) < ((dom\ \Delta_2) \lhd\ \Gamma_1) \\ \Gamma_1, \Delta_1 \vdash s_1 : \Diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \Diamond \end{array}}{\begin{array}{l} ((dom\ \Delta_2) \unlhd \Gamma_1) \cup ((dom\ \Delta_1) \unlhd \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2 \ \vdash \\ s_1 s_2 \ : \ \Diamond \end{array}} \tag{45}$$

$$\frac{\begin{array}{c} (dom\ \Delta) \cap bound(s) = \emptyset \\ \Gamma, \Delta \vdash s : \Diamond \qquad \Gamma \in \mathbb{R} \end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{46}$$

Typing rule 46 specifies when a well-typed system is complete. First, all of the used server pages must be defined, that is the type environment is a pure record type. Second server page definitions may not occur as bound variables somewhere in the system.

THEOREM 7.1. *Core NSP type checking is decidable.*

Proof(7.1): Core NSP is explicitly typed. The Core NSP type system is algorithmic. Recursive subtyping is decidable. The least upper bound can be considered as a union operation during type checking - as a result a form content is considered to have a finite collection of types, which are checked each against a targeted server page if rule 32 is applied.□

8. RELATED WORK

WASH/HTML is a embedded domain specific language for dynamic XML coding in the functional programming language Haskell, which is given by combinator libraries [23][24]. In [24] four levels of XML validity are defined. Well-formedness is the property of correct block structure, i.e. correct matching of opening and closing tags. Weak validity and elementary validity are both certain limited conformances to a given document type definition (DTD). Full validity is full conformance to a given DTD. The WASH/HTML approach can guarantee full validity of generated XML. It only guarantees weak validity with respect to the HTML SGML DTD under an immediate understanding of the defined XML validity levels for SGML documents. In the XHTML DTD [21] exceptions only occur as comments - in XML DTDs no exception mechanism is available - however these comments become normative status in the corresponding XHMTL standard [22]; they are called element prohibitions. In [18][2] it is shown that the normative element prohibitions of the XHMTL standard [22] can be statically checked by employing flow analysis [15][17][16].

There are a couple of other projects for dynamic XML generation, that garuantee some level of user interface description language safety, e.g. [8][9][12]. We delve on some further representative examples. In [25] two approaches are investigated. The first provides a library for XML processing arbitrary documents, thereby ensuring well-formedness. The second is a type-based translation framework for XML documents with respect to a given DTD, which garuantees full XML validity. Haskell Server Pages [14] garuantee well-formedness of XML documents. The small functional programming language XMλ [19] is based on XML documents as basic datatypes and is designed to ensure full XML validity [13].

9. CONCLUSION

The best practice of the proven 3GL programming languages – to define a programming system as the interplay of statically typed components – has not yet been adopted to the development of Web interfaces. With respect to Software design, this problem is tackled by the introduction of proprietary concepts in several commercial Web technologies, like the concept of object wrappers for the form data in the SAP technology BSP (Business Server Pages). Dealing with type errors is supported by web applications frameworks like Struts or IBM Websphere, too, however, only dynamic concepts are offered.

There are several initiatives that propose a statically typed approach to web application development. With NSP, web development with server pages is addressed. A precise description of the type system of NSP is desired, because it (i) can be used as the specification for implementations of NSP concepts, (ii) allows for precise reasoning about web interaction and therefore (iii) deepens the understanding of the interplay between web pages, forms and Web scripts. Therefore, in this paper the core type system of NSP has been given as a Per Martin-Löf style type system.

10. REFERENCES

[1] M. Abadi and L. Cardelli. A Theory of Primitive Objects - Untyped and First-Order Systems. *Information and Computation*, 125(2):78–102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789.

[2] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*. ACM, 2001.

[3] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1997.

[4] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[5] D. Draheim, E. Fehr, and G. Weber. JSPick - A Server Pages Design Recovery Tool. In *Proceedings of CSMR 2003 - 7th European Conference on Software Maintenance and Reengineering*. IEEE Press, 2003.

[6] D. Draheim and G. Weber. Strongly Typed Server Pages. In *Proceedings of The Fifth Workshop on Next Generation Information Technologies and Systems*, LNCS, pages 29–44. Springer, June 2002.

[7] V. Gapayev, M. Y. Levin, and B. C. Pierce. Recursive Subtyping Revealed. In *International Conference on Functional Programming*, 2000. To appear in Journal of Functional Programming.

[8] A. Gill. HTML combinators, version 2.0. 2002. http://www.cse.ogi.edu/ andy/html/intro.htm.

[9] M. Hanus. Server side Web scripting in Curry. In *Workshop on (Constraint) Logic Programming and Software Engineering (LPSE2000)*, July 2000.

[10] P. Kilpeläinen and D. Wood. SGML and Exceptions. Technical Report HKUST-CS96-03, Department of Computer Science, University of Helsinki, 1996.

[11] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[12] E. Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, 2000.

[13] E. Meijer and M. Shields. XMλ - A Functional Language for Constructing and Manipulating XML Documents. 2000. http://www.cse.ogi.edu/$\sim$mbs, Draft.

[14] E. Meijer and D. van Velzen. Haskell Server Pages - Functional Programming and the Battle for the Middle Tier. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.

[15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[16] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the ACM SIGPLAN '95 Conference on Principles of Programming Languages*, pages 367–378, 1995.

[17] J. Palsberg and M. I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[18] A. Sandholm and M. Schwartzbach. A type system for dynamic web documents. In T. Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 290–301. ACM Press, 2000.

[19] M. Shields and E. Meijer. Type-indexed rows. In *Proceedings of the 28th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 261–275. ACM Press, 2001.

[20] J. Spivey. *The Z Notation*. Prentice Hall, 1992.

[21] The W3C HTML working group. Extensible HTML version 1.0 Strict DTD. W3C, 2000. http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd.

[22] The W3C HTML working group. XHTML 1.0 The Extensible HyperText Markup Language. W3C, 2000. http://www.w3.org/TR/xhtml1/.

[23] P. Thiemann. Modeling HTML in Haskell. In *Practical Applications of Declarative Programming (PADL '00)*, LNCS, January 2000.

[24] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4):435–468, July 2002.

[25] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or typebased translation? *ACM SIGPLAN Notices*, 34(9):148–159, September 1999. Proceedings of ICFP'99.

[26] D. Wood. Standard generalized markup language: Mathematical and philosophical issues. In J. van Leeuwen, editor, *Computer Science Today. Recent Trends and Developments*, LNCS, pages 344–365. Springer, 1995.

Style-based Architectural Analysis for Migrating a Web-based Regional Trade Information System

Simon Giesecke
Software Engineering Group
Carl von Ossietzky University
26111 Oldenburg, Germany
giesecke@informatik.uni-oldenburg.de

Johannes Bornhold
Software Engineering Group
Carl von Ossietzky University
26111 Oldenburg, Germany
johannes.bornhold@informatik.uni-oldenburg.de

ABSTRACT

In this paper, we present the MIDARCH method for selecting a middleware platform in Enterprise Application Integration (EAI) and migration projects. Its specific contribution is the use of architectural styles (MINT Styles) as a vehicle for binding architectural knowledge. In addition, an ongoing case study is presented which applies the MIDARCH method to a web-based regional trade information system. The project involves the integration of three subsystems, which have been developed rather independently in the past, two of which are already web-based. The major motivation for migrating the system is to improve evolvability of the system and to make it more apt for the supply to a larger number of customers.

Keywords

ArchiMate, Architectural Description Languages, Architectural Style, Cocoon, Enterprise Application Integration, Java, Web Migration, xADL

1. INTRODUCTION

Software reengineering is concerned with the transformation of legacy software systems. Many reengineering projects aim to modernise systems that are based on outdated technologies, e.g. mainframe systems, that are no longer properly maintained. The transformation target of many business information systems are web-based platforms. Today, we are in the situation that reengineering projects are also concerned with systems that are already web-based, but need to be transformed for a particular reason. Typical reasons are:

- The employed implementation technologies are already outdated themselves.
- Requirements have changed and the chosen architecture is no longer adequate.
- Multiple systems are to be integrated.
- The implementation technologies have been used in an inadequate way.

These reasons are not exclusive to web-based software systems. In particular the latter reason appears like a generic maintainability problem. However, in the case of implementation technologies for web-based systems, such as Sun's Java Server Pages [24] or Apache Cocoon [1], many systems have been developed with only a premature understanding of the architectural style endorsed by these technologies.

The migration of such systems is often not possible without modifying the internal structure of the participating systems, because they do not expose adequate interfaces. Migration projects therefore offer an opportunity to restructure the participating systems, and enabling the integration using advanced middleware techniques. Such approaches are also termed Enterprise Application Integration (EAI), which is a special case of migration that involves multiple, heterogeneous systems.

In this paper, we propose the MIDARCH method for supporting the migration business information systems based on architectural descriptions and architectural styles that are induced by the middleware used (Middleware INTegration Styles, MINT Styles). The main feature of the method is the use of MINT Styles as a vehicle for enabling reuse of architectural design knowledge across multiple migration projects. In addition, we describe a case study of an application of the method which involves a migration project concerning a web-based regional trade information system, and present first results of the case study.

1.1 MIDARCH Research Project

The major goal of the MIDARCH research project [10] is the development and validation of a software engineering method for migrating business information systems based on architectural descriptions that exploit the benefits of architectural styles which are endorsed by the middleware used. The method is called MIDARCH (MIDdleware style based ARCHitectural integration). Architectural styles capture architectural knowledge and provide the basis to reason about families of related software architectures. The method supports the transfer of knowledge from one integration project to another by creating and analysing architectural descriptions that are explicitly based on some architectural style. Through this, experiences from one integration project do not remain constrained to the specifics of the concrete architecture of the subject system, but can be related to the MINT Style. Thus, integration

WMR '06 Bari, Italy

knowledge can be reused in other projects that consider the use of the same style.

1.2 Overview

In the remainder of the paper, we provide the details of fundamental topics that are required for the rest of the paper in Section 2. The setting of the case study is described in Section 4. The research approach taken is outlined in Section 5, which includes the outline of the general procedure of the MIDARCH method. Preliminary results of the ongoing case study are presented in Section 6. The paper ends with a conclusion (Section 7).

2. FOUNDATIONS

In this section, we discuss some foundations we deem necessary to understand the remainder of the paper. First, we introduce the general research areas of Enterprise Application Integration (Section 2.1) and Service-oriented Architectures (Section 2.2), which form the conceptual basis of our approach. Afterwards, we discuss the role of architectural styles for our research (Section 2.3) and present the Architectural Description Languages we use for modelling our case study (Section 2.4).

2.1 Enterprise Application Integration

Enterprise Application Integration (EAI) is a special form of software reengineering, concerning the integration of legacy business information systems. The term Enterprise Application Integration is essentially used in two different meanings: In one view, EAI is used in a restricted sense to denote a specific approach to the integration of information systems which employs off-the-shelf EAI components and is non-invasive with respect to the subsystems to be integrated [18]. EAI in this first view always leads to loosely integrated systems. In this view, EAI is distinct to (invasive) migration.

The other view on EAI refers to any approach to the integration of information systems at the application level as EAI [11], which is the view we take as well. In this view, EAI is a special case of migration that involves multiple, heterogeneous systems. However, in our work, we focus the aspect of the middleware that is used for integration. We have a wide view of middleware, i.e. we regard any software layer that it used for enabling communication of (often, but not necessarily, remote) subsystems or components as a middleware platform. In the case of web-based applications, e.g. Apache Cocoon or Servlet containers are considered middleware platforms.

In [11], three architectural levels are distinguished: Business architecture, application architecture and technology architecture. Integration at each of these levels is described as inter-organisational processes, Enterprise Application Integration and middleware integration, respectively. We are concerned with the latter two levels: The selection of a middleware platform provides the infrastructure for the implementation of the application architecture, and thus for achieving Enterprise Application Integration.

2.2 Service-oriented Architectures

Service-oriented Architectures (SOAs) [12, 20] can both be regarded on a concrete, technical and on an abstract, conceptual level. The first possibility involves the realization of components using specific technologies creating a service infrastructure as web services and the use of technologies such as WSDL, SOAP, UDDI, etc. [29]. The conceptual view generalises this approach and does not necessarily require a specific service-oriented realization, but uses services at the elements of architectural description. One option for implementing a conceptual service-oriented architecture is, of course, using explicitly service-oriented technologies, but other technologies may be used as well. In the latter case, a well-founded mapping of service-oriented concepts to the concepts endorsed by the implementation technology should be provided (cf., e.g., [17]).

2.3 Architectural Styles

Architectural styles [23] and architectural patterns [3] are similar concepts, which we deem equivalent for the purposes of this paper and only use the former term, in order to better distinguish them from lower-level design patterns [7].

Classical general-purpose architectural styles are the pipe-and-filter, blackboard, layered, and event-based styles [23], and variants thereof. These styles are often used in an informal manner to establish a common vocabulary for architectural design elements. In the context of ADLs, formal specifications of architectural styles are used, which define families of architectures or impose constraints on concrete architectural configurations. A special case of architectural styles are those induced or endorsed by an implementation platform, especially by middleware platforms which make use of high-level abstractions [6]. We refer to such architectural styles as MINT Styles.

2.4 Architectural Description Languages

Over the last 15 years, many Architectural Description Languages (ADLs) have been developed with different goals and approaches [22]. There is no broad agreement on a definition of an ADL, for example there have been some debates on whether UML qualifies as an ADL – be it UML as such or a specific usage of UML [8, 21]. We do not intend to provide a rigorous definition here either. However, we briefly describe two ADLs, which play some role in our research project: xADL 2.0 and ArchiMate.

2.4.1 xADL

xADL 2.0 [5] (we will use the brief form xADL in the following) is an ADL which evolved from a traditional line of ADLs at the University of California at Irvine. xADL is a collection of extensions to the xArch [4] core ADL, which is meant to be a "standard, extensible XML-based representation for software architectures" [4]. xADL was designed in a modular and extensible fashion which is based on the modularity and extensibility of XML and XML-Schema [28]. Tool support is available on different levels. On the syntactical level, xADL benefits from its XML basis. Generic tools can be used out of the box, e.g. XML validators can validate xADL and also custom extensions. Another example are syntax-based editors, which can understand the schemas and adopt to custom extensions automatically. Specific xADL tools [26] are a data binding library and a generator which automatically generates a custom data binding library from a XML-Schema. Additionally some higher-level tools are available.

2.4.2 ArchiMate

ArchiMate [19] is not a traditional ADL, insofar as it does not focus exclusively on software architectures, but is used to describe enterprise architectures, which place – in the definition of ArchiMate's developers – software architecture in the context of the organisation(s) using the software. An enterprise architecture is "a coherent whole of principles, methods, and models that are used in the design and realisation of an enterprise's organisational structure, business processes, information systems, and infrastructure" [19, p. 3]. The language closely resembles the UML, and can be mapped onto the UML, but it is not merely an extension of the UML metamodel. ArchiMate supports a layered modelling approach in essentially three layers: business, application and technology architecture (similar to [11]). It is based on the concepts of SOA, so services play a central role on each of the layers.

3. MIDARCH-METHOD

In this section we describe the generic MIDARCH method. The activities proposed by the generic MIDARCH method are shown in Figure 1. The activities shown are quite coarse-grained and must be described on a more fine-grained level to be effectively implementable. Furthermore, no backsteps that might be necessary are indicated in the figure, but the application of the method will be very iterative in practise.

The steps can be structured into four activities, which consist of several subactivities:

Activity 1: Scoping and Goal Definition The first activity consists of two subactivities: Scope Definition and Requirements Elicitation.

Define Scope Scope Definition involves creating a list of (sub)systems to be integrated.

Determine Current and Future Requirements Requirements Elicitation involves the determination of the future requirements on the system, which motivate the need for the integration, in detail, as well as the current requirements on the system. Current requirements may already be documented, but it must be ensured that they are documented in a form that can be compared to the future requirements.

As part of the requirements delta, high-level goals of the integration are identified, which are important for the second activity.

The requirements elicitation process is influenced by the scope determined in the previous step, e.g. because current requirements can only be determined on the basis of a specific system scope.

Afterwards, it must be determined if the requirements match the functionality of the systems to be integrated. In this case, scoping must be reconsidered. There may either be functionality missing, in which case it must be determined whether another (internally or externally available) system can be considered in the integration. If some functionality is not available in an existing system, it must be planned to be newly implemented. There may also be (sub)systems that are not needed to fulfil the future requirements.

Activity 2: Preparation The second activity consists of two subactivities preparatory with respect to Activity 3.

Develop Project-Specific Quality Model A project-specific quality model is developed, which is focused on the migration goals identified in the previous activity.

Model Current Architecture The current architecture is modelled using a suitable modelling language/method. One goal of the overall research project is to evaluate the suitability of different architectural description languages for this purpose. While probably no single modelling language is suited for modelling any system, we contribute to the body of knowledge on the use of modelling languages, and thus provide support for the selection of modelling languages in the future. Suitability here involves the ability to express distinctive features of the current and future architectures (which should be modelled using the same notation and method to ensure commensurability) and to analyse the system or architecture characteristics that occur in the quality model.

Activitiy 3: Architecture Exploration The third activity models and explores different architectural alternatives. It may be considered the core of the method and consists of four subactivities.

Choose/Model MINT Style In each iteration of this activity, one or more MINT Style(s) may be considered. At least in the first iteration, multiple styles should be considered to enable a meaningful assessment in the fourth subactivity. In the method description, we assume that only one style is considered for better readability.

The style description may be either taken from a taxonomy of styles or may be specifically created. One goal of the research project is to provide a taxonomy of MINT Styles and an initial body of knowledge which supports the selection of styles with suitable quality characteristics.

A further goal of the research project is to evaluate the usefulness of different levels of rigour of style descriptions, most importantly informal style descriptions that may include example architectures as opposed to formal style descriptions in an architectural description language as a constraint for concrete architectures in the same language.

Model Candidate Architecture The candidate architecture is modelled on the basis of the chosen style and the current architecture to reflect future requirements.

Evaluate Candidate Architecture The Candidate Architecture is evaluated against the quality model using a scenario-based architectural evaluation method such as ATAM [16].

Assess Evaluation Results The evaluation results of the candidate architectures developed so far and the current architectures are assessed. If the results are found to sufficiently support the integration goals, the activity ends, otherwise further styles and architectures must be considered.

Activity 4: Architecture Selection and Adoption The last activity is not considered within the method in detail, but is included here to make the method complete within the context of its intended application.

Choose Target Architecture Based on the results of the previous activity, a target architecture is chosen, which is based on the best architecture(s) that were identified in the last step of Activity 3. If necessary, details which have been left out in the previous activity are amended.

Adopt Target Architecture The systems are integrated and possibly modified according to the chosen target architecture.

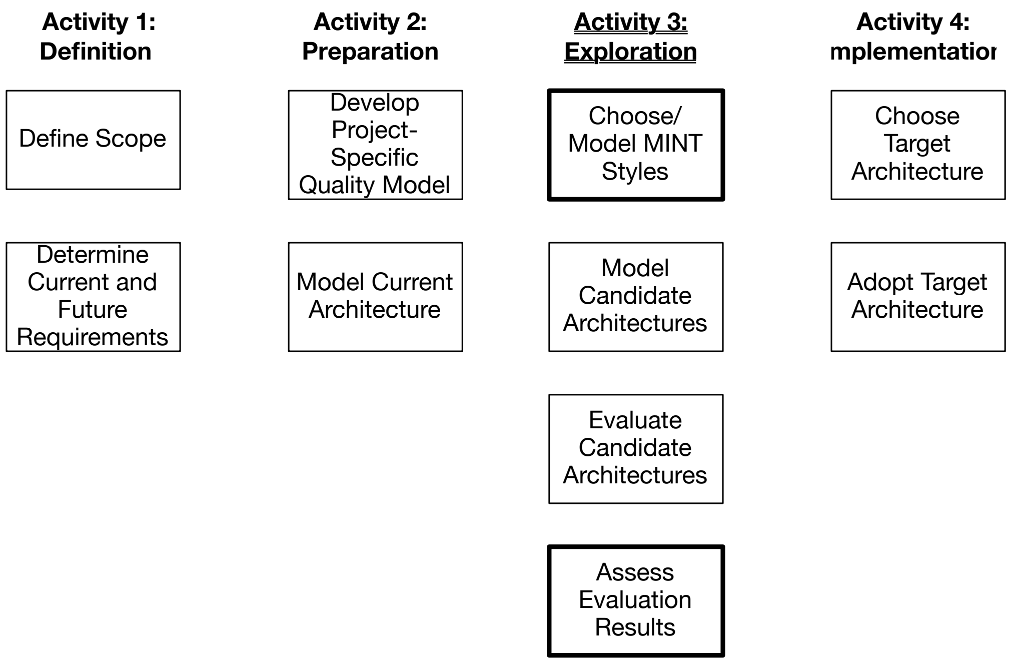

Activity 1: Definition
Define Scope
Determine Current and Future Requirements
Activity 2: Preparation
Develop Project-Specific Quality Model
Model Current Architecture
Activity 3: Exploration
Choose/ Model MINT Styles
Model Candidate Architectures
Evaluate Candidate Architectures
Assess Evaluation Results
Activity 4: mplementatio
Choose Target Architecture
Adopt Target Architecture

Figure 1: Activities of the MIDARCH method

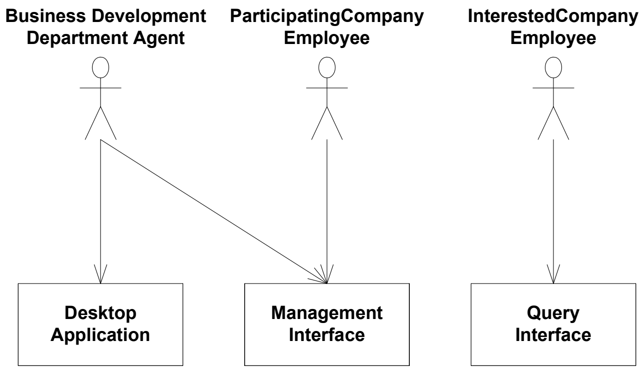

Business Development Department Agent
ParticipatingCompany Employee
InterestedCompany Employee
Desktop Application
Management Interface
Query Interface

Figure 2: User roles and their relationships to the system's interfaces

4. CASE STUDY

In this section we give a brief overview of the regional trade information system which is the subject of the case study (Section 4.1) and the migration goals which shall be achieved (Section 4.2).

4.1 System Purpose

The trade information system is provided as a supporting tool for sustainable regional development. The general idea behind this system is to make information on the economic potential of a region available to companies to increase regional business collaboration.

As indicated in the introduction, the subject system of the case study is separated into three subsystems which have been developed rather independently in the past. Each of these subsystems currently provides a distinct user interface. Two of these interfaces are already web-based. There are three roles of users accessing these interfaces. The relationships of user roles and interfaces are shown shown in Figure 2. The business development departments of the counties and municipalities in the covered region collectively form the current customer, to which our cooperation partner provides the service.

First, an access-controlled web interface is used to collect and manage the data about the participating regional companies. It has two main groups of users. The first group represents the agents at the business development departments. These users can administrate the data of their district's companies and manage the users of the second group, which represent the participating companies themselves.

Second, a web-based query interface is publicly available. It presents information about companies which are located in the covered region. The ability to query this information by different filter criteria facilitates finding potential collaboration partners among regional companies, and thereby supports building regional business networks. The data about each company consists of statistical and address data as well as information on offered technologies, special skills and cooperation interests.

As a kind of glue to the data-management interface the presentation of each company's data contains a hyperlink to edit it. Through this link, new company users can use a registration mechanism to request a login to the system and the necessary rights to edit their data records. In addition the business development department agents have the ability to export their data in a spreadsheet file format.

The third user interface is provided by a desktop application which goes back to a point in time before the development of the other subsystems. Only part of the functionality offered via this interface is still in use. It allows to manage private additions to the data records, which are used only internally by the business development departments. Currently, the new management subsystem provides an export facility which allows the users of the desktop application to manually download the up to date data in the desktop application's proprietary file format and afterwards import it to update their locally stored data.

The desktop application was originally also used to manage the data, which is now managed through the web-based user interface. Originally, the data was sent by the business development departments to the service provider by email and the service provider manually combined the data fragments to feed the query subsystem.

4.2 Migration Goals

There are three main migration goals: First, the system shall be made ready for use by multiple customers. Second, it shall be made more evolvable. Third, the availability of the system should be improved.

The first goal must be seen in the context that this system was originally developed to be used in a single instance for a single region and therefore no effort was made to support multi-customer capabilities and customer-specific customisation needs. In the future, this system shall be offered to multiple customers (i.e., other regions). This means on the one hand that a greater effort must be put on support the adaptability to special customer needs with a manageable amount of human resources. On the other hand, special care must be taken in the product development process to either support hosting of multiple instances and a (semi)automated update-mechanism to new releases of the product, or to add multi-customer capabilities to a single instance of the system.

The second major goal is to increase the system's evolvability [9, ch. 2.2.5.2]. The system itself and its parts evolved over time. Adoption to new requirements has become a challenging task which requires involved developers to be familiar with many parts of the current system. Evolvability is enabled at the architectural level, which must be adequately reflected in the system's implementation.

The third goal is to increase the availability of the system, e.g. by introducing redundant components. Availability becomes more important when more customers are using the system.

5. APPROACH

In this section we describe the MIDARCH method's adaptation to the case study. This section is like section 3 structured according to the activities of the MIDARCH method.

Activity 1: Scoping and Goal Definition

Define Scope In the case study we selected the three interfaces *QueryInterface*, *ManagementInterface* and *DesktopApplication* which are described in Section 4 and the subsystems they depend on.

Determine Current and Future Requirements In the case study, the requirements are elaborated on the basis of different internal documents. These documents contain information on the long-term vision for the software system, non-functional requirements and use cases.

Activity 2: Preparation

Develop Project-Specific Quality Model We use an approach based on GQM (goal/question/metric) [27] to create the quality model. Software quality has different aspects: the internal (cf. [25]) and external (cf. [2]) quality of the software architecture description itself, and the internal and external quality of the software system it represents.

Model Current Architecture We plan to use ArchiMate and xADL (see Section 2.4) to model both the current and the candidate architecture in the case study. ArchiMate provides us with the ability to see the architecture in an organisational context and to connect the application domain with both the business and the technology domain. With xADL, on the other hand, we are able to model the architecture on the application level in detail and to integrate the xADL architecture description with the implementation artefacts. The connection between both languages is done on the application level, enhanced with relations to the other levels (within the ArchiMate description) and related to the development artifacts (within the xADL description).

Activitiy 3: Architecture Exploration

Choose/Model MINT Style In the case study, we are exploring the suitability of xADL to describe architectural styles.

Model Candidate Architecture This architecture shall be modelled in xADL and ArchiMate analogously to the model of the current architecture from Activity 2.

The last two steps of this activity **Evaluate Candidate Architecture** and **Assess Evaluation Results** do not need any special adaption to the case study.

Activity 4: Architecture Selection and Adoption

Choose Target Architecture In the case study, one detail which must be added to the chosen target architecture is the information which implementation artifacts correspond to the architecture components, interfaces and connectors. This shall be achieved through the Java extensions which are part of xADL.

Adopt Target Architecture A prototype of the chosen architecture is created which shall show how the xADL model of this architecture can be connected to the implementation artifacts and thus be integrated with the future steps in the development process. This is also the last step taken in the case study. The adoption of the examined systems to the target architecture is a task out of the scope of the case study.

6. PRELIMINARY RESULTS

In this section we describe the current state of the case study. Activity 1 has been virtually completed and we are currently in Activity 2. The current architecture has been partially modelled, i.e. coarse-grained components, their dependencies and the information flow have been identified. These are described in Section 6.1. An example of an ArchiMate model of a part of the system is presented in Section 6.2. In this model, the coarse-grained components are refined and linked with information on the business and technology levels. Finally, we describe the current middleware technologies and their usage in Section 6.3, and identify potential problems with respect to the migration goals.

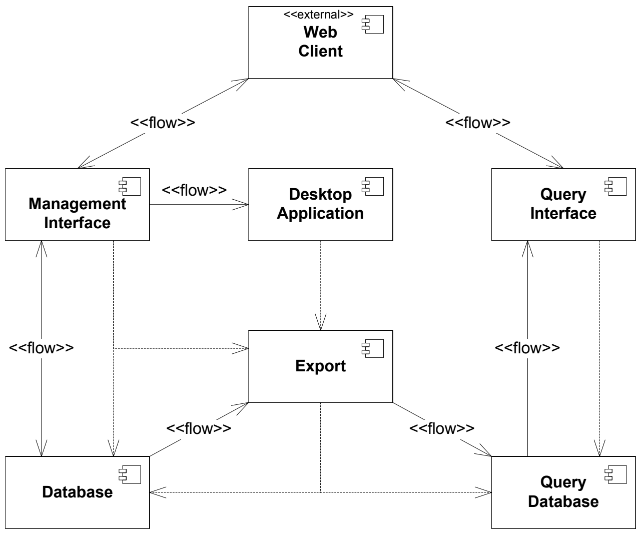


Figure 3: Component dependencies and information flow

6.1 Dependencies and Information Flow

An overview of the component structure and the information flow of the subject system is shown in Figure 3. The two viewpoints are combined in this diagram, because at this abstraction level there are only few elements and the diagram can still be understood.

The different parts of the system have been developed at different times. The oldest part is the desktop application which was originally used to manage the data. The agents at the business development departments used this system to collect the data of their region. Periodically they sent their data to the Application Service Provider (ASP) of the query interface where this data was merged manually and fed into the query database. Now, the direction of the information flow is inverted. The desktop application is updated from the database of the management interface. It contains an export facility which creates a snapshot of the data in the desktop application's proprietary file format. The users can download the file and use it to update their local application. This is shown in Figure 3 by the flow lines from *Database* to *ManagementInterface* and from there continued to *DesktopApplication*. Because the *Export* is needed to create the file, there is a dependency from *DesktopApplication* to *Export*.

The data management interface is the youngest part of the system. It has a database of its own. Note the dependency from *ManagementInterface* to *Database*. The users (agents at the business development departments and employees of the participating companies) can update the data through its web-based interface. The collected data is then transferred periodically into the *QueryDatabase*. This results in a delay before updates of the data are reflected in query results. The information flow is shown by the bidirectional flow lines between *WebClient*, *ManagementInterface* and *ManagementInterface*, *Database*. The propagation to the *QueryInterface*, and thus to the query results, can be read by the directed flows from *Database* to *Export* continued to *QueryDatabase* and finally reaching *QueryInterface*. There is no information flow in the reverse direction.

The last part is the query interface. For historical reasons it has its own database and a slightly different database schema than the management interface. As shown in Figure 3 it is only direct dependent on the *QueryDatabase*. But as mentioned above, its data is updated by the information provided by the *Export*, so to be useful over a longer time, it needs the *Export*, this can be concluded from the flow line between *Export* and *QueryDatabase*.

6.2 ArchiMate Model

Figure 4 shows an example excerpt from an ArchiMate model of the current architecture, which shows the parts necessary for the registration of new users. Its layout is based on an example given in [14]. When a new company's employee requests a login to manage the data about his company, he is in the role *Company* and uses the *RegistrationService* to request his new login. This service is realised on the business layer by the business process *Registration* which depends on the *PostOfficeService*. This service is realised on the application layer by the *PostOffice* and responsible for informing the right person in the *BusinessDevelopmentDepartment* role (*BDD*) to *Check* and possibly *Accept* this request. In the bottom part, this figure shows that the *PostOffice* component needs an available *EmailService* which, in the current case, is realised by a *MailServer* on the technology layer which is installed on some device that in not further specified. This example demonstrates ArchiMate's ability to show the relations between the different architectures on the business, application and technology layers.

6.3 Middleware Technologies and Their Usage

We focus on the subsystems of the regional trade information system which already provide a web-based interface, i.e. the query and management interfaces. Both subsystems are based on Apache Cocoon [1] but use the technology in different ways.

Apache Cocoon is a "a web development framework built around the concepts of separation of concerns and component-based web development" [1]. Cocoon is designed as a Java Servlet. Requests are processed in a pipeline in which several components (*filters*) are hooked together, i.e. it uses a variant of the pipe-and-filter architectural style. Within the pipeline, filters communicate via a stream of SAX events. The entry to the pipelined processing is a *generator* followed by an arbitrary number of *transformers* and finalised by a *serialiser* which typically serialises the SAX events into an HTML output. Apart from this basic concept, Cocoon has the facility to read from and to serialise to many data formats like XML, graphic formats, etc. Many extension filters are provided off-the-shelf, which can be integrated into the pipeline and further support the development of web applications. With regard to the regional trade information system the most important extensions are a framework for form handling and extended control flow support (CForms).

The query interface of the regional trade information system does not use special extensions of Cocoon. Most of its functionality is embedded in XSP documents (a Cocoon-specific language similar to Java Server Pages) which allow Java code to be embedded into XML documents. From these XSP documents, direct queries to the underlying database are made and the results are written into an XML representation of the query result which is then further processed by the following filters. These filters transform the query result into an appropriate HTML representation.

From a very abstract point of view, the management interface works in a similar fashion. The first filters perform some operations on the data and the following filters transform the result into a HTML rep-

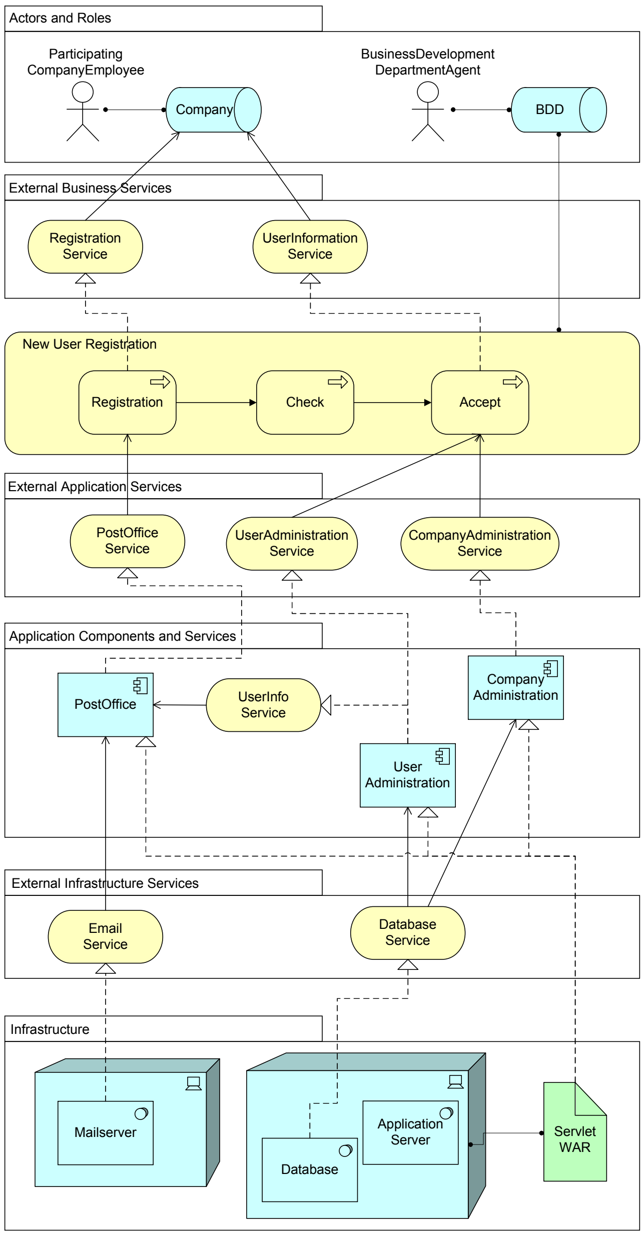


Figure 4: Partial ArchiMate Model of the Case Study System

resentation. Differences appear with a closer look to the first part of the pipeline. The management interface makes use of Cocoon's form framework, which allows for better handling of form data and constraints, and of Cocoon's control flow framework, which allows to send forms with a blocking function call and to formulate control flows in an explicit, closed form. This framework is realised through a JavaScript API which provides access to underlying Java objects. The second difference in comparison to the query interface is the way, data is accessed. In the management interface, all data queries and manipulations are performed with Java objects which map to the underlying data storage (object-relational mapping using Hibernate [13]).

Potential Problem Areas

There are several problems with the current architecture with respect to the migration goals. First, especially the query interface is closely coupled to its underlying database, which is one of the reasons why it still uses a database of its own with an old schema. Because of this, it is technically hard to adopt new requirements that have an impact the database schema, and the effort is difficult to estimate.

Second, the mechanism which transforms the intermediate results to a HTML representation has been identified as another difficulty in practice. An own proprietary language has been developed for the intermediate results which has grown over time and is nearly unmaintainable now.

As a third problem area there are many tight couplings within the Java implementation of the data model, so that requirement changes often result in changes at many different places of the implementation which makes it harder to parallelise development tasks. For this reason, it is not easy to isolate the data tier from the presentation tier in the query and management subsystems, which is why we did not split up the coarse-grained *QueryInterface* and *ManagementInterface* components is Figure 3.

Part of the implementation of the query interface has been reused in the management interface, but has been modified afterwards. Modifications must be ported manually in every case.

7. CONCLUSION

In this paper, we presented the MIDARCH method for integrating heterogeneous business information systems on the architectural level. Many integration projects are performed ad-hoc, i.e. without using a systematic method specifically supporting the integration process. Reuse of experience from other projects thus remains entirely implicit. A few other methods for integration projects have been proposed: Kazakov [15] proposed a semi-automated method for software integration, which requires specifications of the involved software components in the SHIQ description logic.

We do not specifically aim to automate integration efforts, but primarily intend to make reuse of integration knowledge more effective. Tool support for this process is a subsidiary part of the overall research project.

Methods for the development and composition of web services, e.g. Semantic Web approaches, are not in the focus of our work, since we are dealing with pre-web-service legacy applications.

We presented the current state of a case study which evaluates the MIDARCH method. One of the next steps is the modelling of the current and the endorsed usages of Cocoon explicitly as a MINT Style. This and future case studies will provide feedback that will be used to improve the method, and contribute to the knowledge base on the quality characteristics of architectural styles that is necessary for effective application of the method.

Additional future work includes the creation of a taxonomy of middleware platforms based on the MINT Styles they endorse. This taxnomy would allow the stepwise refinement of integration techniques within the exploration process (Activity 3 of the MIDARCH method).

8. ACKNOWLEDGEMENTS

Thanks to Bernd Kramer and Kai Bruns at Regio GmbH, Oldenburg. This work has been partially supported by the German Research Foundation (DFG), grant GRK 1076/1.

9. REFERENCES

[1] Apache Foundation. Apache Cocoon. `http://cocoon.apache.org/`, 2006.

[2] F. P. M. Biemans, M. M. Lankhorst, W. B. Teeuw, and R. G. van de Wetering. Dealing with the complexity of business systems architecting. *Systems Engineering*, 4(2):118–133, 2001.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] E. Dashofy, D. Garlan, A. van der Hoek, and B. Schmerl. xArch, 2006. `http://www.isr.uci.edu/architecture/xarch/`.

[5] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[6] E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st international conference on Software engineering*, pages 13–22. IEEE Computer Society Press, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002.

[9] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[10] S. Giesecke. A method for integrating enterprise information systems based on middleware styles. In *International Conference on Enterprise Information Systems (ICEIS'06) Doctoral Symposium*, 2006. Accepted for publication.

[11] W. Hasselbring. Information system integration. *Commun. ACM*, 43(6):32–38, 2000.

[12] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[13] JBoss Labs. Hibernate, 2006. `http://www.hibernate.org/`.

[14] H. Jonkers, M. M. Lankhorst, R. van Buuren, S. Hoppenbrouwers, M. M. Bonsangue, and L. W. N. van der Torre. Concepts for modeling enterprise architectures. *Int. J. Cooperative Inf. Syst.*, 13(3):257–287, 2004.

[15] M. Kazakov and H. Abdulrab. Semi-automated software integration: An approach based on logical inference. In *3rd International Conference on Enterprise Information Systems (ICEIS)*, pages 527–530, 2004.

[16] R. Kazman, M. Klein, and P. Clements. Atam: A method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, 2000.

[17] I. Krüger and R. Mathew. Systematic development and exploration of service-oriented software architectures. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 177–187. IEEE Computer Society Press, 2004.

[18] R. Land and I. Crnkovic. Software systems integration and architectural analysis – a case study. In *Proceedings of the International Conference on Software Maintenance*, pages 338–. IEEE Computer Society, 2003.

[19] M. Lankhorst et al. *Enterprise architecture at work*. Springer, 2005.

[20] C. M. MacKenzie et al. Reference model for service oriented architecture 1.0. Public Review Draft wd-soa-rm-cd1, OASIS SOA Reference Model TC, Feb. 2006. `http://www.oasis-open.org/committees/download.php/16628/wd-soa-rm-pr1.p%df`.

[21] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[22] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[23] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[24] Sun Microsystems. Java Server Pages Technology. `http://java.sun.com/products/jsp/`, 2006.

[25] W. B. Teeuw and H. van den Berg. On the quality of conceptual models. In S. W. Liddle, editor, *Proc. ER'97 Workshop on Behavioral Models and Design Transformations*, 1997.

[26] University of California at Irvine. xADL 2.0 – A highly extensible architecture description language for software and systems. `http://www.isr.uci.edu/projects/xarchuci/index.html`.

[27] R. van Solingen and E. Berghout. *The goal/question/metric method : a practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[28] World Wide Web Consortium. Extensible markup language (XML), 2006. `http://www.w3.org/XML/`.

[29] O. Zimmermann, M. R. Tomlinson, and S. Peuser. *Perspectives on Web Services*. Springer, 2005.

A framework for Web Applications Testing through Object-Oriented approach and XUnit tools

Alessandro Marchetto and Andrea Trentini
Dipartimento di Informatica e Comunicazione,
Università degli Studi di Milano
Via Comelico 39, 20135 Milano, Italy
marchetto, trentini@dico.unimi.it

ABSTRACT

Nowadays Web applications quality, reliability and dependability are important factors because software glitches could block entire businesses and cause major embarrassment. Web applications are complex and heterogeneous software, based on several components, often written in many different languages and potentially distributed over the Web. Thus, testing Web applications may be a complex task. This paper presents the OO-based framework used in our WAAT project (Web Applications Analysis and Testing) to test traditional Web applications composed of Web documents, objects and server components (e.g., applications written in HTML, Javascript, PHP4/5, etc.).

Our Web testing model named OTMW (OO Testing Model of WAAT project) is inspired by the conventional *category partition* testing method applied to Web software through the use of a reverse engineered OO model used to describe the architecture of existing applications. OTMW tests Web software using three different layers of test: unit, integration and system testing. This paper describes the set of techniques used by OTMW in every testing layer. To achieve this result this paper describes the OO model used (based on UML class and state diagrams) and it defines the reverse engineering techniques used to analyze software and to describe them through the model. Moreover, the paper proposes a method to identify software units and sequences of units to test applications components and their interactions. Furthermore, it describes an approach to define test cases using the reverse engineered models with a technique based on the subdivision of input data in classes of equivalence. Finally, this paper presents tools used to perform some empirical experiments to evaluate the power, effectiveness and flexibility of the OTMW approach.

Keywords

Web Applications,Object-Oriented, Testing

1. INTRODUCTION

Web applications have become the core business for many companies in several market areas. The development, distribution and control of on-line services (on-line retail, on-line trading and so on) can be the mean to and/or the object of business. The growth of the World Wide Web led to the expansion of application areas for new on-line services. For example, many businesses have at least some Web presences with the relative e-commerce (buy/sell, CRM, products information) functionalities. Web applications quality, reliability and functionality are important factors because any software glitch could block an entire business and determine strong embarrassments. These factors have increased the need for methodologies, tools and models to improve Web applications (e.g., applications design and development methodologies, documenting tools, and development process and testing tools). Several proposed methodologies to model and test Web applications are based on existing Object-Oriented ones. For example, [8] and [7] model Web applications from development point of view using OO; [13] and [2] use OO model to represent reverse engineered information extracted from existing Web applications; [14], [9], and [11] introduce OO testing models; HTTPunit[1], PHPUnit[2] and Javascript Assertion Unit[3] are XUnit tools for functional Web testing inspired to OO ones; and so on. Thus, the scientific community studies new ad-hoc techniques or how to adapt existing OO techniques to use them on Web software to improve the quality and dependability of these software system.

Software testing is one the most important and effective approach to verify software systems. Often, Web testing is performed traversing the Web site to simulate navigation and user gestures in order to verify possible executions (e.g., see [4], [14], [10]). Instead, the use of OO approaches to design and describe (or implement) Web applications let us reuse the knowledge developed in the field of OO testing in order to improve the quality of the implemented Web software. Object-Oriented software systems are composed of a set of objects collaborating through messages, and every object has fields and methods thus, it has a set of states defining its evolution. Moreover, an OO language (e.g., Java) may support information hiding, abstraction, inheritance, polymorphic calls, dynamic binding, exception calls, and concurrence, and so on. These specific assets of OO software let the testers use some ad-hoc techniques to test OO software. Often, in OO software the testing unit is the class (or a group of strictly related classes) and the main testing levels are: *basic unit testing* (the intra-method testing focused on methods behaviours); *unit testing* (the intra-class testing focused on the test of isolated modules composing a software system); *integration testing* (the inter-class testing focused on the test the correctness of the interaction between software modules); *system testing* (the testing of the entire system, for example, a system may be view as

Web Maintenance and Reengineering 2006 (WMR 06).
March 24, 2006, Bari, Italy.

[1]http://httpunit.sourceforge.net
[2]http://www.phpunit.de
[3]http://jsassertunit.sourceforge.net

black-box to test its functionalities). More generally, to test a class (or a group of classes) we need to isolate it (them) from the software system and we build the environment (scaffolding) needed to perform the test for the class (or the group) and composed of test cases, specific objects used in every test case, and oracles. In particular, we need to study its interactions with other classes (or groups) and then, we need to build a set of *stub* and *driver* modules. Stub is a (fictitious) module simulating the part of software called from the object under test. While, a driver is a (guide) module simulating the pieces of program that invoke the object under test, and it is used to prepare the environment needed to call the object under test in order to execute a test case for it (a driver may instance new objects, call methods, may define parameters and variables, and so on). Therefore, a minimal test case for OO software is a set of constructor calls, methods calls, parameters settings, inputs values configurations, and so on.

This paper proposes a gray-box and OO-derived approach to test existing Web software. The proposed approach named OTMW is based on unit, integration and system testing. The starting point of this approach is the use of reverse engineering techniques to analyze applications and describe them using a predefined OO model composed of UML class and state diagrams. Thus, OTMW proposes and approach to identify the set of units to test through a method inspired by the conventional category partition method. Moreover, to perform integration testing a testing order (i.e., sequence of units) is defined and then the clusters (i.e., group of units of the order) are tested using the same partitions-based method. Finally, system testing is performed (in terms of traditional Web testing) traversing the Web site through sequences of URLs. However, this paper introduces the approach and shows how to apply it on existing application through a detailed case study.

2. WEB MODELING

In literature several works suggest the use of OO models to design Web Applications in order to increase their dependability and quality. Every technique (e.g., see Conallen [8]) maps OO and Web concepts in order to define an OO-based logical point of view to design, describe and analyze Web systems. In our WAAT project an OO model inspired to [8] has been developed using UML in order to represent existing Web applications and in particular, legacy applications[4]. The main difference between the WAAT model and the [8] is that the Conallen's model aims at describing an application from a logical point of view, as required when it is being designed. On the other hand, the WAAT model focuses on the software implementation, which is the starting point for the software analysis. The WAAT model is based on UML class and state diagrams to represent Web software. The class diagrams are used to describe the structure and components of a Web application. E.g., forms, frames, Java applets, HTML input fields, session elements, cookies, scripts, and embedded objects. A specific asset of our WAAT model is the definition of a fictitious function in a class representing a given Web page or object and containing code not wrapped in functions or classes defined in the original source code. For example, a fictitious method (e.g., "Main") is added in a UML class representing an HTML page to model the source code of the entire HTML page. Furthermore, for a PHP4 page containing code without the definition of functions, the page source code is wrapped in a "Main" fictitious method. From a logical point of view, this ("Main") method may be viewed as an implicit constructor of the same class. Figure 1 shows the class diagram meta-model used in the WAAT project. Every Web application model is an instance of this meta model. Instead, state diagrams are used to represent behaviors and navigational structures of the elements described in the applications class diagram. A navigational structure may be composed of client and/or server pages, navigation links, frames sets, form inputs, scripting code flow control, and other static and dynamic contents. The use of state diagrams let us model relevant assets, such as an active document (i.e., composed by HTML and client side scripting code). In particular, the state diagram of an active document can define the function calls flow of the scripting code, and some relevant behaviors/navigation dynamic information (e.g., dynamic links, frames, and so on).In our model, a Web application is associated to a state diagram and Web documents are associated to substates (subdiagrams). A static document is represented by a simple state, while an active document is represented by a composed state that may be concurrent if the page contains client-side scripting code. Dynamic documents are modeled by simple or composed state. If the document does not contain some relevant navigation element, it is described with simple state, with composed state otherwise. E.g., a dynamic page that builds many client side HTML pages is modeled with a composed state with many substates, one for every HTML page generated. In general, the transitions are defined by links, function calls, and various HTML form inputs. An HTML frame set is modeled via composed concurrent state where every frame corresponds to a substate. See

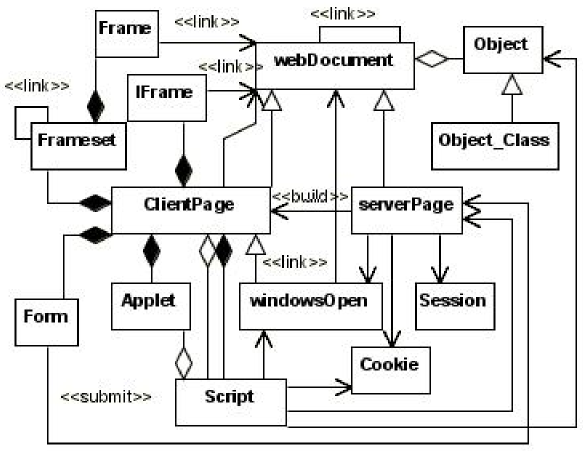


Figure 1: Web Applications UML Meta-Model

[3] and [2] for more details and samples about the OO model used in the WAAT to describe applications. To the aims of this paper, we recall here that we introduce an approach to test Web software. Nevertheless, the OO-based model used to represent Web applications is not really the focus of this paper because some existing OO-based modeling techniques may be useful with the testing approach presented in this paper. Moreover, we use a set of reverse engineering techniques ([2]) to recovery UML models from existing applications but OO-based models are often defined in a design phase of the development life-cycle and the proposed approach may be used too. The reverse-engineered model is based on static and dynamic analysis. The technique uses static methods derived from traditional source code analysis adapted to extract static and dy-

[4]Legacy applications are the kind of Web software where the business logic is embedded into the Web pages, instead of more recent and layered Web applications where the business logic is implemented through server-side components. The analyzed applications are composed of Web documents (static, active or dynamic) and Web objects.

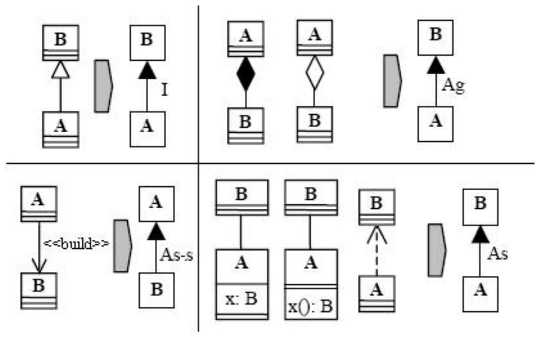


Figure 2: wTDG main rules

namic information from Web. Moreover a combined method based on static and dynamic analysis is used to define navigational structure and application behavior. We have paid particular attention to the server side dynamic aspects of Web applications, we analyzed it with a dynamic method based on application execution and on mutational analysis applied to source code [2]. This dynamic analysis is performed with the generation of a set of source code mutants, used into navigation simulation. Then, procedure results are analyzed with static traditional source code techniques. The use of mutation lowers user interactions in the reverse engineering phase and let us defines a more detailed description.

To test an application we use its UML model composed of UML class and state diagrams to extract several kinds of information usable to identify the units to test and/or to guide the test cases definition and/or to calculate the code coverage reached with a set of test cases. In particular, using the class diagram we may build a graph of the system components dependencies. While, using a UML state diagram we may build an "extended function calls graph" (eFCG, "extended" for the presence of the fictitious methods) for the entire system (named class eFCG) and/or for every components (i.e., we may build a graph of the function calls for every system component). In this case, for every software component that it is represented in the state diagram using a complex state (i.e., a state grouping a set of sub-states) we build a eFCG where every node is a sub-state and every edge is a state transition that exist among states. In the case of our OO model, transitions between states may be function calls (for both fictitious and not methods), link clicks, specific user gestures needed to evolve the software, and so on. Thus, a path in the eFCG represents a possible execution of the software under analysis.

Several works studying the integration-orders problem use application models (i.e., the class diagrams) as a basis to build a graph representing dependencies among components. Then, this graph is used in order to search the best integration order. In case of Web software, we suggest to use the "Web Test Dependence Graph" (wTDG). [12] presents an extended version of the original TDG adapted for working with specific OO assets (polymorphic dependencies, as well as the nature of the dependence such as aggregation, association, inheritance and so on). wTDG is a simplified version of this TDG at classes-level. wTDG is a directed graph whose vertices represent UML classes and directed edges represent dependencies among them. A wTDG may contain loops because a class may be directly or indirectly dependent from each other ones. In the TDG, an arrow from B to A means that "B is test dependent on A" thus, we need to test A before B. Figure 2 show four main rules of the wTDG construction used to map UML class diagrams in wTDG. Given two classes A and B:

1. If B extends A then A is test-dependent on B through an inheritance dependence and in wTGD the edge that connects A to B is labeled "I".

2. If A is a composition (or aggregation) of B, A is test-dependent on B through an aggregation dependence and the A to B edge is labeled "Ag".

3. If A is associated or depends on B, A is test-dependent on B through an association dependence and the A to B edge is labeled "As".

4. If A is associated to B through specific WAAT-model relationships defined using the UML stereotype "<<build>>" (e.g., relationship existing between a server-side page and the built one or more client-side pages) thus, B is test-dependent on A through a specific association dependence and the A to B edge is labeled "As-s".

Through this set of rules we may build a wTDG from a reverse engineered UML class diagram of Web applications.

3. OTMW-BASED WEB TESTING

3.1 Rationale

The testing performed in the OTMW model is inspired by the category partition method (see [15] and [5]). This method is a specific sub-type of the functional testing method known as "equivalence classes"-based testing (EC). The EC method, for every testing layer (unit, integration, system) defines subdivisions of the application input domain in equivalence classes which are used to derive test cases. The main ideas are that a failure found by one value in a class will be found by all values in the same class and that all components of a class are treated in analogous mode by the software (i.e., producing correlated results). The main goal of this type of test is to define test data that may reveal possible classes of errors/bugs. An equivalence class is represented using a set of (valid and/or not valid) input data and a set of software states for the output data produced through the class inputs. Thus, the category partition approach may be viewed as composed of the following steps: software specification analysis to identify the functional unit to test (and for every one, identification of its parameters and the needed environments); classification of the identified units in categories; subdivision of the categories in choices; definition of constraints among the choices; definition and documentation of tests. In more details, the OTMW model may be used to test an application through a gray-box approach (i.e., a functional approach that considers some interesting structural information to perform the test) inspired by the category partition testing method and applied in six main steps to perform unit, integration and system testing. These steps are the following:

- We need to build the OO model for the existing application under testing.

- When the application under test is described through UML class and state diagrams we use the class diagram to identify software units to test in isolation.

- Then, we perform the unit testing and thus, for the current unit under test, we use its state diagram to build its eFCG (graph of function calls and actions) and we use eFCG as a basis to define the test cases through the expected unit behavior shown in this eFCG (and using the idea of the "equivalence class" to subdivide the input domain and to define the set of representatives test cases). Then, we build the scaffolding (i.e., stubs, drivers, oracles) needed to test unit through the defined set of functional test cases. In particular, the scaffolding may be expressed in terms of fragments of code (i.e., scripting code) written using a set of XUnit tools. Thus, we may execute every test case using these XUnit-based code.

- Then we need to identify the integration order of system units needed to test the software components interactions (i.e., the definition of the best user-adequate unit sequences). In this phase, we use the wTDG graph, built from UML class diagram, to extract information about the components dependencies and we use a genetic-based algorithm that analyzes some coupling measurements among components in order to devise the best set of sequences usable to test the components integration.

- Then we test every cluster identified in the previous step (a cluster is a group of software units collaborating among them). In this case, we treat a cluster as a "big-unit" and we use the merge of eFCGs for units in the same cluster to define the functional test cases and then to write scripting code using the XUnit tools. Thus, we may test clusters using these written classes of test.

- In the last step of OTMW we need to perform system testing. In particular, we use the UML models to build a graph considering only high level information in order to describe the application as a graph composed of nodes representing pages (considering client or server side and static or dynamically generated pages) and edges representing links existing among pages. Through this graph, we perform some random walks paths to traverse the graph and to simulate user navigations using a set of sequences of URLs randomly generated based on the graph coverage (i.e., nodes/edges/ paths coverage).

The OTMW layered model lets us perform different kinds of test, for example, in the unit testing we test every component of the software architecture. A Web application may be written in several languages and may be composed of some different components collaborating among them. A component may be a client-side page (e.g., composed of HTML and Javascript code), a server page (e.g., composed of PHP 4/5 code), and Web object and/or other component (e.g., written in PHP, or ActiveX, or other server/client-side scripting code, XML files and database, and so on). In particular, for a complex page/object (such as written in PHP) we may test its functionalities, or its main execution paths stressing several sequences of methods defined in the same page/object. On the other hand, through the integration testing we test the integration (i.e., collaborations) of the software components. Thus, we treat a cluster (i.e., group of software unit) as a unique unit with an interface composed of the sum of the units interfaces, this let us test more and more invoked sequences to stress methods of every unit in order to analyze every state of the evolution and/or execution of every unit. Finally, the system testing let us perform the conventional pages-based testing in order to focus the test in the navigational system and the structure of the application (i.e., sequences of pages). In the following sub-sections we analyze and describe every step with several details in order to guide the user (i.e., Web testers) to use our OTMW to test Web applications.

3.2 Unit Testing

The main steps of the OTMW unit test are: identification of units to test in isolation; test cases definition (using a *testing table* defined through the analysis of the inputs and the eFCG for the current unit under test); identification of the needed drivers and stubs; and test script description using XUnit tools.

To test the elements composing the software we need to use the UML class diagram used to describe it. We use this diagram to identify the units that we may test in isolation. These kinds of units may be:

- static HTML pages (with or without scripting codes)

- client side objects such as the scripting codes (e.g., fragments of Javascript code)

- server side objects (e.g., objects written in PHP 4/5)

- server pages written in PHP 4/5 and their set of dynamically generated HTML client-side pages (we consider a server side page and its dynamically generated pages such as a unique unit)

- client-side scripting code (e.g., Javascript) generating a set of HTML pages (we consider as unique unit)

- Web objects and components (such as txt file, xml, database, and so on)

- other components not previously classified.

However, the analysis of the dependencies (and their types) described in the UML class diagram may be used to define components representing units that may be tested in isolation and to identify they needed stubs. A Stub is a fictitious module simulating the part of software called from the object under test. In particular, dependencies such as: inheritances, compositions, <<build>> are traditionally considered as "not breakable" while other such as associations, aggregations, <<submit>>, and so on, may be breakable. This information and the different kinds of elements listed before may help us to identify units and stubs. For example, a PHP server page that uses a PHP object to build a set of three dynamically generated HTML pages may be viewed as composed of two units. The first is the PHP object used by the server page while the other is the server page with its three generated HTML pages. Moreover, this last unit uses the PHP object and thus, this unit needs a stub to execute it in the testing phase.

For every identified unit we build its *testing table* (inspired by the decision table defined in [5] for OO software and then refined in [9] for Web software). This table is used to define a set of test cases through our method inspired by the traditional category partition approach. For the definition of a testing table for a unit: we need to identify the input parameters of the unit (from our UML class diagram); and then we need to describe the eFCG (the graph that describes the unit executions in terms of function calls and actions performed) for the modules composing the unit under test. We use eFCGs to extract several paths (i.e., a path represents a possible software execution at level of function calls and actions sequence) through the traditional coverage criteria (such as: nodes, edges, n-cycles path, couples of def-use, and so on coverage) applied to the same graph. These paths are the basic information to define test cases. For example, the eFCG for a PHP object may be a function calls graph. Thus, traversing the graph through the coverage graph criteria, we may extract several paths where every path is composed of a sequence of methods (i.e., defined/used in/by PHP object) calls and it represents a possible execution of the PHP object. Then, we use these information as a basis to fill the *testing table* for the unit under test. Figure 1 shows the skeleton of a testing table composed

Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
...			...		

Table 1: skeleton of Testing Table

of two sections: the first for the input data and the other for output (expected) data.

A table is composed of six sections as following:

- *Variables*: listing of the unit input. For example, for client-side page the input fields of its HTML forms, for client-side scripting functions the HTML DOM-tags, for server pages the needed GET/POST variables, and so on.
- *Actions*: listing the actions (i.e., user gestures, function calls, class instantiations, and so on) needed to perform the testing. In particular, it represents a sequence of actions needed to realize the unit execution extracted from the eFCG. For example: link clicks, button press, method calls, class instantiations, and so on.
- *State Before Testing*: containing the values assumed before the test from specific application elements such as client-side pages, cookies, tags, the state of used Web objects, session variables, server objects, and so on.
- *Expected Results*: listing the expected output results when the test is executed
- *Expected Output Actions*: describing the actions performed by the pages/objects under test when the test case is executed (i.e., the functionality performed with the test case, e.g., login action, sending data, write files, and so on)
- *State After Testing*: describing the expected values assumed after the test execution from specific application elements such as the same described in the “State Before Testing” section.

Notice that to fill the *Actions* section, the user (i.e., tester) needs to identify the steps (user actions, method calls, objects instantiation, etc.) useful to implement the execution path extracted from the eFCG of the unit under test. Thus, the user may define more than one sequence of steps usable to perform a single eFCG path. In our *testing table*, the combination of *Input Variables* and *Actions* is used to identify the equivalence classes (EC) and thus, it is used to subdivide the input domain of the application unit (composed of input variables and states of the components under test) in classes usable to derive several test cases of the same classes (e.g., changing the input values). When the testing table is filled we may proceed writing the scripting to execute test cases through specific XUnit tools. For example, to test HTML-based pages we may use tools such as: HTTPunit or HTMLUnit[5] but also HTML Tidy[6], HTML validators[7]; to test server side objects written in PHP 4/5 we may use tools such as PHPUnit or PHP Assertion [1]; to test objects based on Javascripts code we may use HTMLUnit or JSUnit[8] or Javascript Assertion Unit; to test components based on more than one language (e.g., client side pages that send data to server side pages we may use combination of this tools). In particular, every row of the table may represent a class of test cases and may be converted in a testing script using these XUnit tools. Then we may proceed with the test cases execution using the ad-hoc written script code and repeating its execution using several different input values.

[5]http://htmlunit.sourceforge.net

[6]http://www.w3.org/People/Raggett/tidy

[7]http://validator.w3.org

[8]http://www.edwardh.com/jsunit

Drivers and stubs modules are needed to test a given unit. A driver is a (guide) module simulating the pieces of program that invokes the object under test, and it is used to prepare the environment needed to call the object under test in order to execute a test case (a driver may instance new objects, call methods, may define parameters and variables, and so on). Typical Web driver may be composed of fragments of code derived from pages or objects that interacts with the unit under test, filling HTML forms, generating events (e.g., to simulate user gestures), and so on. This type of code may include scripting fragments (client/server side), Web objects, DOM objects, and so on. Instead, a stub may be a client/server page/object that it is used by the unit under analysis in order to perform its task.

The main goals of this unit testing phase may be to test the loading, in some different context, of every elements composing the applications (e.g., pages, objects, page components such as forms, scripts, tables, server components, and so on); the structure and navigational system of every component (e.g., elements compositions, self links, submitting operations, etc.); the evolution (in terms of states reached) of every complex components such as client/server side code (e.g., using JSUnit and PHPUnit we may test several different function calls sequences representing different software executions); and the construction of the dynamically generated pages (e.g., the HTML code generated by a server page).

3.3 Integration Testing

In the integration testing phase we test the interactions among software components (i.e., units identified in the previous testing step). In particular, we may test data or messages exchanged among units. For example, we may test the following cases:

- the data/messages exchanged between an HTML page and its Javascript code, such as function defined a Javascript fragment and called in an HTML tag with mouse events, or HTML tags filled with the returned value of a Javascript function.
- data/messages exchanged among PHP objects, such as functions or data variables in a PHP5 class but defined in another class.
- data/messages exchanged between an HTML page and a PHP page that elaborates these data to generate outputs (e.g., HTML form data submitted to PHP page, or PHP function called from HTML code)
- the use of several kinds of files (e.g., TXT, database, XML) to write, read, modify data from PHP or Javascript code
- the use of scripting code or server-side applications by a PHP object

Therefore, in a Web application we need to test some different types of interactions because every application may be written in more than one software language (e.g., HTML, Javascript, PHP, and so on). Thus, we need to test interactions such as among the following elements: HTML code and Javascript; Javascript and Javascript; Javascript and PHP; HTML and PHP; PHP and PHP;

Javascript, HTML and PHP; Javascript, HTML, PHP and other elements (TXT, database, XML); and so on. To verify the interactions among components we need to identify the sequence of units to test. Then, we need to treat every integration cluster (group of units in the sequence) as a single "unit" in order to fill its *testing table* and to write its set of test cases.

Given a software system, to test its components and their relationships, the first problem is to decide the integration order, because different orders may define some different complexity in terms of effort. For OO-modelled software it may be very difficult to choose the testing order because the system has specific assets (i.e., information hiding and abstraction, inheritance, and so on) and because the architectures may be very complex and several components may be strongly connected (i.e., cyclical dependencies). Thus, to define the best integration order we need to use a method studying the components dependencies and the scaffolding complexity. For example, we may consider a small system composed of four classes (A B C and D), and where "B uses C", "C uses A", and "D uses A". In this case, some possible integration orders may be found defining a topological order[9] of the class-usage graph, for example A D C B or A C B D may be orders usable into integration testing. Instead, if this system contains another *use* relationship such as "A uses B" the system contains a dependencies loop (composed of the classes A B C), thus it is impossible to define a topological order, but we may define a partial order such as A D C B (where A needs B as stub), or A C B D (where A needs B as stub). In literature, there are several works that use class diagrams representing OO systems (and defined during the analysis and design phase or extracted from code using reverse engineering techniques) as a basis to build graphs representing relationships among components then analyzed through deterministic or random approaches to find optimal integration orders. Most of the proposed strategies are focused on the analysis of this dependencies-graph derived in order to minimize the effort needed to test the application and (often) the effort is expressed in terms of stubs number (or complexity) needed to test using a specific integration order. In particular, the proposed solutions "break" some dependencies in cycles contained in graph to obtain an acyclic graph representing the dependencies of the entire system. This approach implies that the modules related to the broken relationships need to be stubbed in the integration testing. However, there are several deterministic and random approaches usable to define how to break cycles and devise orders, see [12] for a review of existing techniques and for some empirical comparisons. These approaches may be grouped in four categories as following:

- finding of the strictly connected components (SCC) of the system (dependencies cycles); and to break some (one or more) randomly-selected dependencies in SCC
- finding the strictly connected components (SCC) of the system (dependencies cycles); to weigh every dependency in SCC counting the parameters passing through this dependency; and to break the dependency with the smaller weight
- finding the strictly connected components (SCC) of the system (dependencies cycles); for every component in SCC counting the number of cycles it belongs to; and to break the component that is part of the highest cycles number
- finding the best orders using a genetic algorithm (i.e., a semi-random approach, see [6]) that uses the permutation encoding where every chromosome is a string of class labels and that defines a chromosome as an integration order (i.e., a sequence of system classes). Then to evolve the population using a set of genetic operators (selection, mutation and crossover) and a fitness function based on the stubbing complexity (in terms of coupling measure between the current test order and its needed stubs).

[9]A topological order is a node ordering for a direct graph such that each predecessor node of a given node is listed before the same node in the topological ordering

When the units order has been defined we may start the test of every cluster composing this sequence. We test all clusters using the same methods and tools used in the unit testing, because we consider every cluster as a unique "unit" with an interface described as the sum of all units-interfaces composing the cluster. This let us define several invocation sequences to stress methods of all units in cluster, in order to verify the cluster in every state it may reach. Thus, for a cluster we need to identify its input variables, to define its needed stubs and drivers and to fill its testing table. We use eFCGs of units in cluster to define relationships existing among components, and then we extract several paths from these graphs through coverage criteria. Finally, we may filled testing table and we may use it to extract a set of test cases and to write the testing scripts using XUnit tools. Finally, we may execute them more than ones time using several different values for input variables.

3.4 System Testing

The system testing for a Web application may be essentially based on high level representation where the application is described through a graph composed of nodes corresponding to Web pages and edges corresponding to links. In our modeling approach, we may extract this graph from the UML class diagram. Then, the test consists in sequences of URLs requested to Web server with their inputs values (if needed). This test let us verify the navigational system and the structure of Web application by traversing the graph. [4] shows the approach used in our WAAT project and implemented in TestUml tool. While, [14] describes another similar approach. We recall here that the main goal of this paper is to describe an approach to perform unit and integration testing of Web software. However, to perform system testing we traverse a given Web application simulating user (random) navigations and gestures. In particular, we use the application graph to extract several paths (sequences of URLs) selected through conventional coverage measures such as nodes or edges, n-cycles paths, def-use couples coverage, and so on. Then, these sequences (test cases) are completed with the needed input values and executed (more time) performing requests to the Web server. This testing method is semi-automatic due to the fact that the user (tester) must complete the inputs not randomly identified or extracted from log-files analysis.

4. CASE STUDY

MiniLogin is a simple Web application we use to show how to apply the OTMW approach to existing software systems. This application is composed of some PHP5 and HTML files with Javascript, and its main functionality is to control the access to a reserved Web area through login and password. Through WebUml (see [3]), the tool that implements our reverse engineering techniques, we perform static and dynamic analysis on this Web application in order to extract information needed to build MiniLogin UML model composed of UML class (Figure 3) and state diagrams. In the following sub-sections we show how to apply our Web testing approach to MiniLogin in case of unit and integration testing. Instead, for the system testing (we would like to recall here that it is not the main goal of this paper) see [4] for more information about techniques

Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
Javascript_1					
	(1) 1.load script in HTML 2.call controlData() 3.read returned value	def(formMain.user) def(formMain.psw) def(ptagUsername) def(ptagPassword)	true or false	login and password verification	ptagUsername, ptagPassword == “is Number” or “is String (with Number)” or “is String”
	(2) 1.load script in HTML 2.call controlUsername() 3.read returned value	def(formMain.user) def(ptagPassword)	true or false	login verification	ptagUsername == “is Number” or “is String (with Number)” or “is String”
control_php					
$user, $psw	(3) 1.class instantion 2.call setCombine() 3.read returned value		new String equal to $user.$psw	$user.$psw strings concatenation	
$user, $psw	(4) 1.class instantion 2.call setCombine() 3.call verify() using 2. result 4.read returned value		“one” or “two” or” “theree” or “error”	$user.$psw strings concatenation and verification	
member_php					
$user, $psw	(5) 1.class instantion 2.call counter() 3.read returned value	def($count)		to control the $count session variable	$count+1 or $count=0
...					

Table 2: MiniLogin samples of testing table

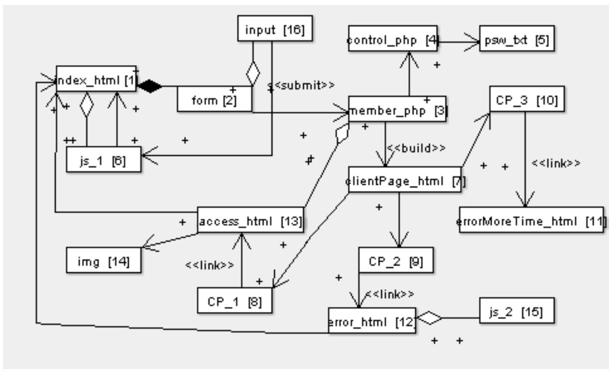


Figure 3: MiniLogin UML Class Diagram

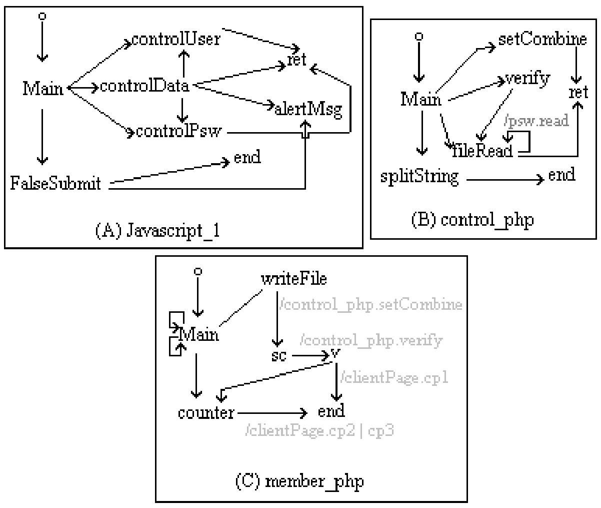


Figure 4: MiniLogin samples of eFCGs

used in the WAAT project to perform system testing as briefly introduced in the previous sections.

4.1 Unit Testing

We identify the unit to test considering different types of elements composing the application UML class diagram (HTML static page, PHP page/object, HTML dynamically generated pages, etc.) and the types of the relationships existing among elements (associations, aggregations, compositions, etc.). For example, there are two Javascript objects (Javascript_1/2), one PHP page (member_php), one PHP object (control_php), several HTML pages, and so on. In the case of unit testing, for every unit we need to define stubs and drivers usable in the test phase. We identify these elements using the dependencies described in the class diagram. For example, the PHP page named member_php has (“use”) relationship with: PHP object named control_php and with an HTML page (named access_html). Instead, the PHP object control_php has no relationship with other PHP elements but only one with a TXT file. Thus, control_php may be tested as a unit stubbing the TXT file, while member_php needs three stubs to be tested. Furthermore, to test a Javascript code usually we need to stub the fragment of the HTML page that interacts with this scripting code (and its DOM too). In our case, the object Javascript_1 needs to stub the index_html page.

For every unit we need to fill its testing table as described in the previous sections. In particular, we identify every method (considering also the fictitious methods “Main”), we identify the input variables and the variables used by current unit but defined in other ones. For example, Javascript_1 contains a method named *alertMsg* that has four parameters as input. Furthermore, the same unit has a method named *controlPsw* that does not have input parameters but that uses two variables defined in another unit such as an HTML form field (i.e., *formMain.password*) and a HTML page tag (i.e., named *ptagPassword* in index_html).

Then through the state diagram of every unit we build its eFCG (see Figure 4 for some Minilogin samples[10]). From this eFCG we

[10]In this figure, to semplify the readability, we have omitted some

```
<head>
 <script src='jsUnitCore.js'>
 </script>
</head>
...
<body>
...
<script name='test' language=JavaScript>
function testcontrolData() {
  document.all.ptagUsername.innerText=''
  document.all.ptagPassword.innerText=''
  document.formMain.username.innerText='prova'
  document.formMain.password.innerText='prova'
  assertTrue(controlData())
 assertEquals(document.all.ptagUsername.innerText,
 'Username is String');
}
function testcontrolUsername() {
  document.all.ptagUsername.innerText='inizio'
  document.formMain.username.innerText='prova'
  debug(document.all.ptagUsername.innerText);
  assertTrue(controlUsername());
  debug(document.all.ptagUsername.innerText);
assertEquals(document.all.ptagUsername.innerText,
'Username is String');
}
</script>
</body>
```

Figure 5: Javascritp_1 test cases: (1),(2)

extract some paths using the conventional coverage criteria (i.e., nodes or edges coverage) in order to select several possible executions of the current unit (at funcion calls and actions level). For example, considering the eFCGs in Figure 4 we may extract the following execution paths:

- Javascript_1: Main, controlData, ret; Main, controlData, controlUser, ret; Main, falseSubmit, alertMsg, end; (1)Main, controlData, alertMsg, ret; (2)Main, controlUser, ret; and so on
- control_php: (3)Main, setCombine, ret; Main, verify, fileRead, ret; (4)Main, setCombine, verify; ret; Main, fileRead, fopen,ret, and so on
- member_php: Main, get, get, writefile, control_php.setCombine, sc, control_php.verify, v; (5)Main, counter; Main, writeFile, counter; and so on

Through this set of information and using our knowledge about the MiniLogin application we may fill the testing table for every unit. Table 2 show fragments of testing tables for three units of the Minilogin application (i.e., Javascript_1, control_php, and member_php). Now, we may use the testing table with the XUnit tools to write test cases, every row of the table may represent a class of test cases that may be implemented in a testing script. For example, the rows (1) and (2) of Table 2 may be implemented with JSUnit tool as shown in Figure 5. Instead, the rows (3) and (4) may be implemented using PHPUnit2 as shown in Figure 6. Moreover, the row (5) may be also implemented through PHPUnit2 as shown in Figure 7. In this last case, to treat the member_php page as a unit testable with XUnit tools we need to wrap the entire page code in a fictitious "Class member ..." and the main code of the same page on a fictitious method "function Main()...". This lets us treat the PHP page as a conventional OO class. In the set of written testing cases, the only case that found a "bug" is for the control_php object and it is named "testsetCombineVerify_ErrorInTest()". In particular, for this test case we expect the string "one" as result but after the execution we obtain "error", this is due to the fact that the fragments of PHP code written for this test case contains a mistake (i.e., not the application).

information such as the label of the edges corresponding to actions to perform

```
<?php
require_once('PHPUnit2/Framework/TestCase.php');
require_once('control.php');
class controlTest extends
        PHPUnit2_Framework_TestCase {
public function testsetCombine_withStringValues(){
 $control=new control();
 $user="primo';
 $psw='secondo';
 $expectedOutput='primosecondo';
 $output=$control->setCombine($user,$psw);
 $this->assertEquals($expectedOutput, $output);
}
public function testsetCombineVerify_error(){
 $control=new control();
 $user='primo';
 $psw='secondo';
 $expectedOutput='error';
 $output1=$control->setCombine($user,$psw);
 $output2=$control->verify($output1);
 $this->assertEquals($expectedOutput, $output2);
}
public function testsetCombineVerify_ErrorInTest(){
 $control=new control();
 $user='user';
 $psw='one';
 $expectedOutput='one';
 $output1=$control->setCombine($user,$psw);
 $output2=$control->verify($output1);
 $this->assertEquals($expectedOutput, $output2);
}
public function testsetCombineVerify_correct(){
 $control=new control();
 $user='User1';
 $psw='One';
 $expectedOutput='one';
 $output1=$control->setCombine($user,$psw);
 $output2=$control->verify($output1);
 $this->assertEquals($expectedOutput, $output2);
}
}
?>
```

Figure 6: control_php test cases: (3),(4)

4.2 Integration Testing

To do integration test among software units, identified in the previous step, we need to define an integration order among them (i.e., a sequence of units that helps us to define the order in which to test the units). Thus, from the MiniLogin class diagram we extract the wTDG and we use it to calculate some coupling measures among units. Then, we use this measures in wJenInt, that is our ad-hoc written tool implementing a genetic algorithm [6] usable to devise an optimal testing order through a fitness function based on coupling measure used to calculate the stubbing complexity (that is expressed in terms of coupling measures between the current order and its needed stubs). In the case of MiniLogin we have defined the following integration order: Javascript_1; psw_txt; control_php; img; member_php; input; form; Javascript_2; index_html; errorMoreTime_html; error_html; access_html; clientPage_php; clientPage_1; clientPage_2; clientPage_3. Through this order we need to cut only one dependency associating Javascript_1 and index_html elements. Therefore, we need to test every cluster defined in this order and that contains units collaborating among them, for example, we test: psw_txt -control_php; (6)member_php, control_php (7)Javascript_1, input-form; Javascript_1, member_php, input, form, index_html; (8)member_php, control_php, clientPage_php, client-

Input			Output		
Variables	Actions	State Before Testing	Expected Output	[Expected Output Actions]	State After Testing
Cluster 1					

	(7.2)
	1.load index_html
	2.put user string
	3.onMouseOut activation
	4.call controlUsername()
	5.click submit
	6.onMouseClick activation
	7.call controlData()
		def(formMain.user)
		def(formMain.psw)
		def(index_html.ptagUsername)
		def(index_html.ptagPassword)
			ptagUsername==
			'is String'
			+alert(Number Sedning)
				login and password verification	ptagUsername,
					ptagUsername==
					“is String” and
					'ptagPassword==
					undef

Cluster 2					

$username
$password
	(8)
	1.load index_html
	2.put real username
	3.put real password
	4.click submit
	5.load access_html
	6.it contains gif and link
	7.click link
		def(formMain.user)
		def(ptagPassword)
			access OK +
			load access_html +
				username and password	

...					

Table 3: MiniLogin samples of Integration testing table

```
import com.gargoylesoftware.htmlunit.*;
import java.net.URL;
import com.gargoylesoftware.htmlunit.html.*;
import junit.framework.TestCase;
import java.util.*;
public class SimpleHtmlUnitTest extends junit.framework.TestCase {
public void testHomePage1() throws Exception {
 WebClient webClient = new WebClient();
 java.net.URL url = new  java.net.URL('http://localhost:8080/ãlex/logred2/index.html');
 HtmlPage page = (HtmlPage) webClient.getPage(url);
 assertEquals('Home Page', page.getTitleText());
 HtmlForm form = page.getFormByName('formMain');
 HtmlTextInput textField=(HtmlTextInput)form.getInputByName('username');
 textField.setValueAttribute('prova');
  HtmlPage appWindow=(HtmlPage) page.executeJavaScriptIfPossible(
          textField.getOnMouseOutAttribute(),'testCU',false,textField).getNewPage();
  assertEquals('Home Page', appWindow.getTitleText());
  assertEquals('Username is String',
          appWindow.getHtmlElementById('ptagUsername').getFirstChild().asText() );
  HtmlSubmitInput button = (HtmlSubmitInput)form.getInputByName('Submit');
  List collectedAlerts = new ArrayList();
  webClient.setAlertHandler( new CollectingAlertHandler(collectedAlerts) );
  HtmlPage newPage = (HtmlPage)button.click();
        List expectedAlerts = Collections.singletonList('Number sending'); [or 'Numbero sending']
  assertEquals( expectedAlerts, collectedAlerts );
}
}
```

Figure 8: cluste1 test case (7.2)

```
<?php
require_once('PHPUnit2/Framework/TestCase.php');
require_once('member.php');
class memberTest extends
        PHPUnit2_Framework_TestCase {
public function testcounter_1(){
 $mem=new member();
 $mem->counter1();
 $this->assertEquals(0, $_SESSION['count']);
}
public function testcounter_2(){
 $mem=new member();
 for($contatore=1;$contatore<=10;$contatore++){
  $mem->counter1();}
 $this->assertEquals(10, $_SESSION['count']);
}
}
?>
```

Figure 7: member_php test case (5)

Page_1, access_html; and so on.

For every cluster we use the eFCGs of its units identifying the invocation sequences of methods/variables used among units to collaborate. For example, for (6) may be: (member_php.Main & control_php.Main), member_php.wF, control_php.setCombine, control_php.verify. While, for (7) possible sequences may be: (7.1) (index_html.Main & Javacript_1.Main & input.Main & form.Main), formMain.user, input.onMouseOver, Javacript_1.controlUser; (7.2) (index_html.Main & Javacript_1.Main & input.Main & form.Main), formMain.user, input.onMouseOut, Javacript_1.controlUser, formMain.submit, input.onMouseClick, Javacript_1.controlUser; and so on. While for (8) a possible sequence may be: (member_php.Main & control_php.Main), member.writeFile, control_php.setCombine, control_php.verify, clientPage.Main, clientPage.cp1, cp1.Main, cp1.Main, cp1.gif, cp1.link, index_html.Main;

Then, we may fill the testing tables (see Table 3 for samples) for clusters and, using XUnit tools we may write the scripting code to test clusters. Tables 8 and 9 show the testing classes written in

```
import com.gargoylesoftware.htmlunit.*;
import java.net.URL;
import com.gargoylesoftware.htmlunit.html.*;
import junit.framework.TestCase;
import java.util.*;
public class SimpleHtmlUnitTest extends junit.framework.TestCase {
 public void testHomePage1() throws Exception {
 WebClient webClient = new WebClient();
 java.net.URL url = new  java.net.URL('http://localhost:8080/ãlex/logred2/index.html');
 HtmlPage page = (HtmlPage) webClient.getPage(url);
 assertEquals('Home Page', page.getTitleText());
 HtmlForm form = page.getFormByName('formMain');
 HtmlTextInput textField=(HtmlTextInput)form.getInputByName('username');
 textField.setValueAttribute('User1');
 HtmlTextInput textField2=(HtmlTextInput)form.getInputByName('password');
 textField2.setValueAttribute('One');
 HtmlPage appWindow1=(HtmlPage) page.executeJavaScriptIfPossible(textField.getOnMouseOutAttribute(),
         'testCU',false,textField).getNewPage();
 assertEquals('Home Page', appWindow1.getTitleText());
 assertEquals('Username is String',
         appWindow1.getHtmlElementById('ptagUsername').getFirstChild().asText() );
 HtmlPage appWindow2=(HtmlPage) page.executeJavaScriptIfPossible(textField2.getOnMouseOutAttribute(),
         'testCP',false,textField2).getNewPage();
 assertEquals('Home Page', appWindow2.getTitleText());
 assertEquals('Password is String',
         appWindow2.getHtmlElementById('ptagPassword').getFirstChild().asText() );
 HtmlSubmitInput button = (HtmlSubmitInput)form.getInputByName('Submit');
 List collectedAlerts = new ArrayList();
 webClient.setAlertHandler( new CollectingAlertHandler(collectedAlerts) );
 HtmlPage newPage = (HtmlPage)button.click();
 List expectedAlerts = Collections.singletonList('<username>User1</username><password>One<password>');
 assertEquals( expectedAlerts, collectedAlerts );
 assertEquals('ACCESS', newPage.getTitleText());
 HtmlElement root=newPage.getDocumentElement();
 List imgs=root.getHtmlElementsByTagName('img');
 assertEquals(1,imgs.size());
 assertNotNull(newPage.getAnchorByHref('index.html'));
 HtmlAnchor link = newPage.getAnchorByHref('index.html');
 HtmlPage page3 = (HtmlPage) link.click();
 assertEquals('Home Page', page3.getTitleText());
 }
}
```

Figure 9: cluste2 test case (8)

HTMLUnit and related to test cases (7.2) and (8).

When testing classes are written using the filled testing tables we may perform the test by repeating the test cases execution changing input values.

4.3 System Testing

We recall here that it is not the main goal of this paper, see [4] for details about the system testing performed in our WAAT project. Generally speaking, using TestUml tool from the class diagram we build an high level graph of the application under test where nodes are Minilogin Web pages and edges are links. Then we use coverage criteria and random walks analysis to extract some paths that helps us to traverse the application graph and to simulate user navigations and gestures. In this case of system testing, a test case is a sequence of URLs (of the pages composing the defined sequence) and its input values. For example, in case of Minilogin application the following sequences of URLs and inputs (expressed in the form <page to load, [list of parameters values]>) may be a set of test cases: (index.html), (member.php); (index.html), (member.php, “username”, “password”), (access.html); (member.php, “user1”, “psw1”), (errorTime.html); (member.php, “user2”, “psw2”), (error.html), (index.html); and so on.

5. CONCLUSIONS

In this paper we have presented our OTMW framework usable to test Web applications through an OO approach. In OTMW we use an OO model to describe applications from a logical point of view and then we identify software units testable in isolation (such as client and/or server Web pages, scripting code, Web objects, and so on) and, for every one, we perform a category-partition derived technique to test it at function call level. Then, we use an existing technique to derive an integration order (i.e., sequence of units) and we use it to select clusters (i.e., group of units of the order) to test using the same functional-derived approach. Finally, we perform a system testing using traditional Web testing in terms of sequences of URLs. Through this OTMW framework we treat (i.e., design and test) Web software as traditional OO software in order to test several aspects such as navigational system, functionalities, structure and, in order to test every component in every state, in different execution contexts, in isolation and in collaboration with other components exchanging data or messages. Moreover, OTMW uses tools developed in our laboratory for the WAAT project (such as WebUml, TestUml, wJenInt) but to execute the test cases it uses traditional XUnit testing for Web applications.

6. REFERENCES

[1] Php assertion.
http://jsassertunit.sourceforge.net/docs/phpassertunit.html.

[2] C. Bellettini, A. Marchetto, and A. Trentini. Dynamic Extraction of Web Applications Models via Mutation

Analysis. *Journal of Information -An International Interdisciplinary Journal- Special Issue on Software Engineering*, 2005.

[3] C. Bellettini, A. Marchetto, and A. Trentini. WebUml: Reverse Engineering of Web Applications. *19th ACM Symposium on Applied Computing (SAC 2004)*, Nicosia, Cyprus. March 2004.

[4] C. Bellettini, A. Marchetto, and A. Trentini. TestUml: User-Metrics Driver Web Applications Testing. *20th ACM Symposium on Applied Computing (SAC 2005)*, Santa Fe, New Mexico, USA. March 2005.

[5] R. Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 1999.

[6] L. Briand, J. Feng, and L. Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. *14th international conference on Software engineering and knowledge engineering*, Italy. 2002.

[7] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Ninth International World Wide Web Conference (WWW9)*, Amsterdam, Netherlands. May, 2000.

[8] J. Conallen. *Building Web Applications with UML*. Addison-Wesley, 2000.

[9] G. A. Di Lucca, A. Fasolino, F. Faralli, and U. De Carlini. Testing Web Applications. *International Conference on Software Maintenance (ICSM'02)*, Montreal, Canada. October 2002.

[10] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *Ieee Transactions on Software Engineering*, November 2001.

[11] D. C. Kung, C. H. Liu, and P. Hsia. An Object Oriented Web Test Model for Testing Web Applications. *24th International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan. October 2000.

[12] V. Le Hanh, K. Akif, Y. Le Traon, and J. Jézéquel. Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. *15th European Conference on Object-Oriented Programming (ECOOP2001)*, 2001.

[13] F. Ricca and P. Tonella. Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'200)*, Genova, Italy. April 2001.

[14] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. *23th International Conference on Software Engineering (ICSE'2001)*, Toronto, Canada. May 2001.

[15] M. Young and M. Pezzè. Software Testing and Analysis: Process, Principles and Techniques. *John Wiley and Sons (WIE)*, 2004.

Supporting the Evolution of Service Oriented Web Applications using Design Patterns

Manolis Tzagarakis
Computer Technology Institute
26500 Rion
Greece
+30 2610 960482
tzagara@cti.gr

Michalis Vaitis
University of the Aegean
University Hill, GR-811 00 Mytilene
Greece
+30 22510 36433
vaitis@aegean.gr

Nikos Karousos
University of Patras
26500 Rion
Greece
+30 2610 960482
karousos@cti.gr

ABSTRACT

Web applications make increasingly use of services that are provided by external information systems to deliver advanced functionalities to end users. However, many issues regarding how these services are integrated into web applications and how service oriented web applications evolve, are reengineered and refactored are still addressed in an ad hoc manner. In this paper, we present how design patterns can lessen the efforts required to integrate hypermedia services into web applications. In particular we present how evolution and maintenance issues are addressed within Callimachus, a CB-OHS that web applications need to integrate in order to provide hypertext functionality to end users.

.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]:

General Terms

.

Keywords

Web application, service oriented architectures, hypertext.

1. INTRODUCTION

The term "web application" characterizes a particular class of applications that make use of internet technology to deliver content and services such as HTTP, HTML, XML and Web Services [1]. One particular class of Web applications deals with integrating information systems into web applications.

Web applications are still developed in an ad hoc manner, resulting in applications that fail to fulfill several importand requirements including:

1. User needs, meaning that the web application is not what the user wanted
2. Easy maintenace and evolution
3. Long useful life
4. Performance and security.

As already pointed out in [13]

"*Web systems that are kept running via continual stream of Patches or upgrades developed without systematic approaches*"

The problems are even more complicated, when web applications are built upon service oriented architectures (SOA) that differ greatly fom traditional client server architectures. SOA exhibit great flexibility with respect to services and require new approaches to service integration. Within the hypermedia field, Component-Based Hypermedia Systems (CB-OHS)[15] have emerged, consisting of an underlying set of infrastructure services that support the development and operation of an open set of components (called structure servers), providing structure services for specific application domains. The theoretical and practical aspects of this promotion of structure from implicit relationship among data-items to a first-class entity constitute the subject of the field of structural computing [5]. Attempts to integrate services provided by CB-OHS with web applications are already underway [14].

CB-OHS are among the forerunners of a trend for service-oriented computing (SOC) [8]; the computing paradigm that utilizes services as fundamental elements for developing applications [2] and relies on a layered SOA. A SOA combines the ability to invoke remote objects and functions (called "services") with tools for dynamic service discovery, placing emphasis on interoperability issues [3]. As both hypermedia applications and the class of web applications categorized as informational [1] are content-intensive, the employment of structure services (following the SOC paradigm) would improve efficiency and convenience [9].

Unfortunately, today's developers of hypermedia and web applications face various problems when attempting to integrate services offered by CB-OHS into web applications. This is in particular true when considering evolution and maintenance issues. Currently, such concerns are addressed by developing structure services from scratch [4] redesigning appropriately the provided services. We argue that one of the reasons for this situation is the lack of both an adequate software engineering framework for CB-OHS construction, integration, and maintenance and the appropriate tools to support it. This results in ad-hoc integration methodologies which produce systems missing certain essential characteristics including difficulty to evolve and maintain.

In this paper, we present how design patterns can lessen the efforts required to integrate hypermedia services provided by service oriented systems into web applications. In particular we present how evolution and maintenance issues are addressed within Callimachus, a CB-OHS that web applications need to integrate in order to provide hypertext functionality to end users.

The paper is structured as follows: first we outline aspsects of SOA that makes integration into web applications difficult and error prone. We then present Callimachus and how the services provided are integrated into web applications. Next, we present and analyse the design patterns that are used to address evolution and maintenance concerns. Finally, future work concludes the paper.

2. Service Oriented Architectures (SOA)

Traditionally, hypermedia systems have been built according to client server (or point-to-point) architectures that provided an adequate framework for bringing hypertext functionality to web applications. However, the design and development of these hypermedia systems were based on assumptions that reflect the architecture upon which they were developed. Moving hypermedia systems to service oriented architecture requires these assumptions to be re-examined and adjusted. This is because service oriented archtectures differ greatly from client server architectures. Table 1 summarizes the main differences between service oriented and client server architectures.

In service oriented architectures, bindings to services (i.e. references to operations provided by services) are established dynamically and during runtime which is completely incompatible with client server based hypermedia systems where such binding of clients to services happen very early in the development process (in particular during design or compile time). At run time, changing bindings is impossible.

Table 1. Comparison of Client Server vs Service-Oriented Architectures

Client Server Architectures	Service Oriented Architectures
Early binding (compile/development time)	Late binding (run time)
Domestic (evolve smoothly and planned)	Feral (evolve abrupt and uncontrolled)
Location dependent	Location independent and transparent
Single interface (protocol)	Set of interfaces (protocols)
Development oriented	Integration oriented
Tightly coupled	Loosely coupled
Monolithic	Composable
Stable	Unstable due to ad hoc nature

While client server architecture evolves in a controlled and disciplined fashion, service oriented evolves in a rather feral way. This is mainly due to the autonomous nature of services that implies an autonomous evolution path as well. As a result client-side bindings to hypermedia services can easily be invalidated. In addition, it is evident that while client server architectures exhibit location dependence thus forbidding changes in location information (e.g. in terms of host and port) service oriented architectures are location independent making conventional clients unable to operate in such an environment. With respect to the supported interfaces, in client servers systems only a small, bound number of interfaces are supported whereas in service oriented systems an unbound number of interfaces exists. Thus while in client server systems it is enough for all software entities (e.g. client application) to be reactive when considering interfaces to hypermedia services, in service oriented architectures all software entities need to be proactive. Furthermore, client server systems are tightly coupled systems, meaning that design changes in the service are followed by design changes on the client side. This is not the case in service oriented characterizing this architecture as loosely coupled. Finally, while in client server systems the main task during development is to extend the client and the server respectively, in service oriented architectures the main task of a developer is to integrate services.

From the above discussion it is clear that service oriented architectures represent an environment where all software entities need to exhibit flexibility, autonomy, and adaptability in order to function correctly and take advantage of the plethora of services that are presented. Within such an agile environment, web developers require new tools and infrastructures that will enable smooth evolution as well as seamless integration of the provided services into web applications.

2.1 The Callimachus Component Based Hypermedia System.

Callimachus is an open hypermedia system [6, 7] that aims at providing hypertext functionality to an open set of applications. It provides support for wide range of domain specific abstractions thus addressing a broad range of hypertext domains [5]. Such domains include *navigation*, allowing the interlinking of information and *taxonomic reasoning* to develop for example directory services on the world wide web [14].

Callimachus follows a component-based architecture as depicted in figure 1. Each component provides a number of services through which clients can request domain specific hypertext functionality. Its primary architectural elements are client applications, structure servers and infrastructure. Client applications can be either native or third-party applications, such the MS Office Suite and Emacs, or even web servers and entire web applications. Client applications (clients for short) request services from structure servers using a well defined protocol. Structure servers provide the domain specific abstractions of a particular hypermedia domain by offering a consistent set of services. The infrastructure provides services across hypermedia domains such as storage, naming and notification.

The on-the-wire messages sent between clients and structure servers are encoded using XML and transferred using HTTP tunneling. The adoption of this technique has been imposed mainly by the need to overcome the access restrictions to non WWW services enforced by firewalls. HTTP is used as a transport protocol to tunnel client requests. The Content-Type parameter specifies the protocol that is being used.

All client-side aspects of the protocol come in the form of a library that implements an API. Different structure servers require different protocols to communicate with client applications. The construction of the client-side API takes place during the development of the structure server. In Callumachus, all structure servers have the form of a TCP/IP daemon listening on a specific port for incomming requsts. Each structure server can serve concurrently many clients that can be of different types (e.g. web application, Emacs etc).

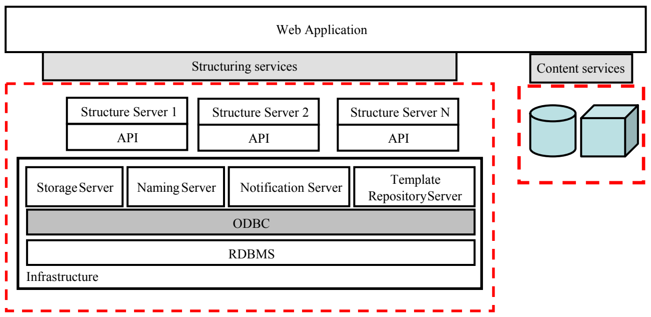


Figure 1: The conceptual architecture of Callimachus and how it is integrated with web applications

Being clients, web applications request structuring services from Callimachus and content services form other information systems. For example, in case the web application provides directory services, it invokes structuring services such as openCategory or getPathOfCategory from Callimachus, and it resolves the returned content identifiers using the content services. At the web application layer, the outcome of both service invokations are merged and transformed to the appropriate format (e.g. html or XML). The result is then sent back to be displayed to end users.

The development of structure servers and the integration mechanisms (i.e. APIs) follows an evolutionary rapid prototyping approach with short iterations and many releases. This means that there is a constant evolution of services with which the entire framework has to cope with.

Design and development is split into tasks, each one dealing with a particular aspect of the structure server. Three main tasks are carried out, each producing a prototype subsystem. The integration of developed subsystems results in a working structure server. The specification, design and implementation of each subsystem does not follow a particular process model, because of their tightly coupled nature and their "small" size as software artifacts. These tasks are described more detailed below.

Server shell development: During server shell development, the structure server's interface is built. In this task, the emphasis is on the design of the exact procedure the structure services are invoked. More precicely, all aspects of the structure server when viewed as the receivers of client requests are addressed. Such aspects include listening to, parsing and validating incoming requests, as well as preparing and passing these requests to the domain model for execution.

Domain model development: During this task, the syntactic and behavioral aspects of the domain-specific abstractions (including their relationships) are designed and developed. The syntactic and behavioral specifications originate from the scenario and are defined in terms of the Callimachus Abstract Structural Element.

Integrator development: The aim of this task is the development of the necessary software modules that will enable integration of clients with the structure server. These modules come in the form of a client-side API. Specifically, a wrapper container and a communicator are developed [52] so that client applications are able to request structure services.

The prototyping phase starts with the development of an initial domain model prototype. Consequently, the server shell and the integrator prototypes are developed. After an initial cycle, each prototype is refined by constantly iterating through the tasks until an acceptable structure server prototype has been completed. The prototype structure server is tested by end-users aiming to assess its accordance to the scenario.

These challenges include both non-functional and functional aspects of structure servers:

Incremental service (and operation) formalization: During prototyping, the set of the provided services (and operations that clients can request) is initially unknown, with their name, behavior and parameters slowly emerging, as prototypes become available for testing. By having services emerging and evolving while development is progressing, the emphasis is on ways to easily integrate new or modify existing services, without requiring changes in functionally unrelated modules of the structure server (which cause major concerns to developers). In particular, the goal here is to achieve localization of the effects during the evolution of services.

Smooth evolution of protocol implementations: Although the design of multi-protocol support ensures easy integration of new protocols developed entirely from scratch, it does not address evolution of existing protocols. During protocol evolution, new methods might be added to existing protocol implementations; existing methods might change their signature or might even be associated with different operations at the domain model layer. Such tasks need to be carried out quickly to ensure short iteration cycles.

3. Design Patterns

Within the Callimachus project, design patterns [11] have been proven a valuable mechanism to support smooth evolution of hypermedia services and their seamless integration with Web applications.

In particular, design patterns are utilized to address changes at the hypermedia services layer due to new web application needs as well as changes at the web applications layer due to changes at hypermedia services. Consequently, two types of design patterns can be identified: patterns that address concerns at the hypermedia services layer and patterns that address concerns at the web application layer.

Next, for each layer, we briefly present the design patterns used. Although the design patterns discussed are already well known, the focus is mainly on what benefits can be gained when using them in service oriented environments.

3.1 Design Patterns at the Hypermedia Service Layer

3.1.1 Protocol Handlers

Everytime a connection with a client is establised, all received requests for structure services need to be parsed in order to be checked for validity and prepared for execution. Validity checking includes the examination of the conformance of the requesting message to the syntax of the domain protocol specifications, as well as to the semantics of the domain model functions (i.e., the indicated operations along with the type of parameters supplied). Preparing a request for execution refers to the necessary actions dealing with determining the appropriate operation in the domain model that has to be executed. Such tasks are the responsibility of the protocol handler [12]. Since different structure servers require different protocols, development of protocol handler is performed every time a new structure server is developed. The situation gets more perplexed when considering that the same structure server can be accessed using different protocols meaning that the same structure server needs to provide support for a number of protocols that need to be activated at runtime. The question thus is how to make the same set of operations provided by structure servers available through different protocols.

To achieve smooth evolution of protocol issues within structure servers, parsing of incomming requests must be decoupled from invocation of the operation that requests designate. For this reason, the strategy design pattern is used [11]. This permits also, the parsing algorithm to vary according to the incomming request.

How the strategy design pattern is utilized is depicted in figure 2. Within each structure server, the ServerContext class deals with all low level aspects of receiving a request from the TCP/IP socket, as well as parsing the HTTP headers of the tunneled request. The class also maintains a reference to an instantiation of the HypertextProtocol, an abstract class that is used to parse the received request and supports only the public virtual methods `Parse` and `Clone`. While the `Parse` method encapsulates the suitable algorithm for parsing and preparing incoming requests, the `Clone` method returns a copy of the HypertextProtocol instance, used in the context of the prototype design pattern. All protocols supported by a particular structure server, are derived from the HypertextProtocol class. Every derived class (that constitutes a protocol handler) implements the method parse, where the appropriate code for parsing, validating and preparing the request is placed by the developer. The appropriate protocol is determined and instantiated during runtime based on HTTP's Content-Type parameter. For this task, the prototype design pattern is utilized, determining how the appropriate available protocol implementations are declared and instantiated during runtime. The hypertext protocol factory is part of the ServerContext class and is instantiated during initialization of the structure server. There is exactly one hypertextProtocolFactory for every structure server.

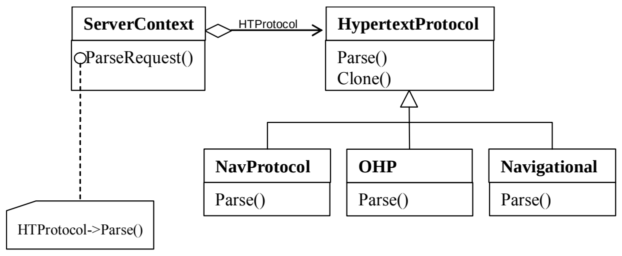


Figure 2 : Protocol Handler

Adding support for new protocols is fairly trivial, allowing developers to focus only on parsing and preparing without spending time about how to integrate the new protocol into the structure server. During design time, developers have to create a class that resembles their protocol implementation (derived from the `HypertextProtocol` class) and to provide the implementation for the `Parse` method. Furthermore, they have to register the new class in the factory's `registerProtocol` method that takes place in the factory's constructor. During runtime, correct deployment of the new protocol handler is ensured by the prototype design pattern [11], since the mechanism of how to determine which class to instantiate is independent of protocol handlers.

3.1.2 Service Execution

Different web applications may require different set of operations from the same structure server. For example some web application providing directory services might require complex editing of entire subtrees such as deleting directories or moving and copying them to different locations while others don't. Moreover, for all available operations, undo, redo, logging and queuing options should be available. The question here is how to systematically extend the available operations (and thus services). The goal is to provide domain specific operations in a plug-and-play fashion. To support such development tasks, the invocation of an operation needs to be seperated from its execution. Within Callimachus, this is achieved using a variation the active object and command processor design patterns [10].

In the design pattern of figure 3, all client requests (denoting operations, such as openNode, traverseLink in case of a navigational structure server and deleteDirectory in case of a taxonomic structure server) are instantiated as separate objects. There exists one class for each operation available to clients, elevating operations to first class entities, thus allowing them to be stored, scheduled and even undone. Such treatment of operations also allows the support of transactions. All available operations are derived from the DomainOperation class, an abstract class with two methods: `Execute` and `Undo` (implemented by the concrete derived classes). The Execute method of each concrete class executes the operation by calling the appropriate method of the class HMDomain that represents the interface to the domain model subsystem. For example, the openNode class would call the openNode method of class HMDomain.

The appropriate concrete operation instances are created by the HypertextProtocol class, after having parsed and validated incoming client requests. The HypertextProtocol class decides which operation to instantiate in order to be flexible with respect to which method of HMDomain class to invoke. There might be cases where a matching method might not be available in the HMDomain class, so an equivalent method (or set of methods) in that class should be invoked. For example, a getNode operation (that would be modeled as a separate class) has to invoke the available openNode method (i.e., an equivalent method) of the HMDomain class, when a getNode method is not available. Such choice is conveniently done in the HypertextProtocol class after parsing and before the execution phase of client requests.

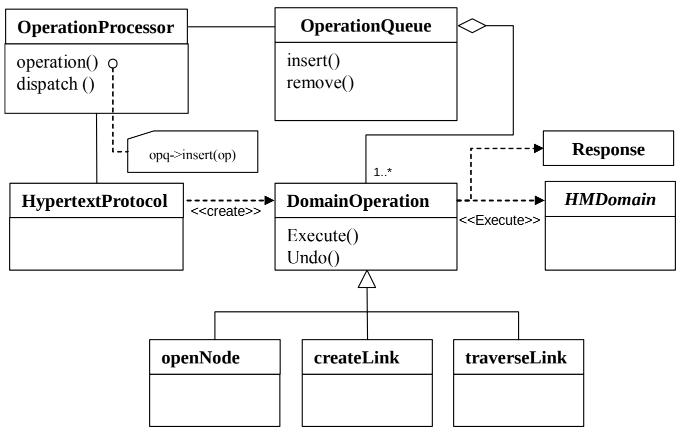


Figure 3: Service execution

The HypertextProtocol class enqueues all operation instances by calling the Operation method of the OperationProcessor class. There is exactly one OperationProcessor instance for every structure server. Thus, an OperationProcessor constitutes a singleton [12]. The OperationProcessor class maintains the operation objects in the OperationQueue, and schedules their execution. The OperationQueue class may arrange the operations by priority and decide which operation is ready to be executed by calling the operation's canExecute method. Operations are dequeued and executed concurrently by calling the appropriate methods of the HMDomain class. Each operation executes in a separate thread of control. The output of each operation is available through a specific class (see Response class in Fig. 3) that is used to send replies back to clients.

During structure server evolution, developers can systematically approach the problem of constant change in the domain operations, in the protocol specifications and in their bridging. New operations can be added during design time by extending the DomainOperation class and delegating execution to the appropriate domain specific interface method. Since identification and invocation of the operation are provided by the framework at run-time, developers can focus only on semantic aspects of the operations. In addition, the framework provides the foundation for supporting a number of advanced (but necessary) capabilities, such as the undo/redo operations, as well as transaction management for all structure servers in a uniform manner, thereby reducing maintenance efforts.

3.2 Design Patterns at the Web Application Layer

While the previous sections presented design patterns that facilitate the evolution of structure server when new web applications requirements emerge, the following design patterns address concerns at the web application layer and in particular attempt to address issues that deal with hypermedia service invocation.

With respect to invocation, the patterns aim at providing mechanisms to achieve the following:

1. provide a single point from which requests to the hypermedia services originate.
2. Offer templating mechanism for re-occuring invocation schemes.

3.2.1 Single Invocation Point: Dispatching requests

Everytime developers need to issue requests from the web application layer to Callimachus they place code (e.g. that uses the API for accessing hypermedia services) in different web application modules. Such approach to hypermedia service provision results in code that is unstructured and thus unmaintainable. The question here is how can be hypermedia service invocations be systematically integrated into web applications reducing thereby maintainance efforts.

Systematic integration is achived by using the action dispatcher design pattern. The action dispatcher design pattern provides a single access point for communication with hypermedia services, selecting the appropriate action by dispatching centrally all incomming requests. Firgure 4 depincts the action dispatcher design pattern.

In Figure 4, all requests for hypermedia services are dispatched by the Dispatcher class that creates the appropriate operation that needs to be requested from structure server. Thus, every operation that is available by a particullar structure server is represented as a separate class. Each such class, in turn, extends a generic Action Handler class.

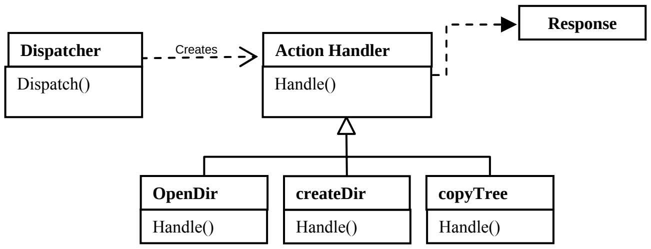


Figure 4: Dispatchign requests

Selection of the specific operation (or action) is done using a creational pattern (e.g. factory method).

3.2.2 Request chaining

At the web application layer and specifically during the handling of a particular user request, a number of hypermedia services need to be invoked sequentially – passing responses from one invocation to the other - to complete a user transaction. Moreover, situations arise where hypermedia and content services need to be invoked sequentially to produce the final response that will be sent back to the user. Similar invocations schemes are used within the web application layer (and not only in relation with Callimachus) such as validating user request using filters before invoking hypermedia services.

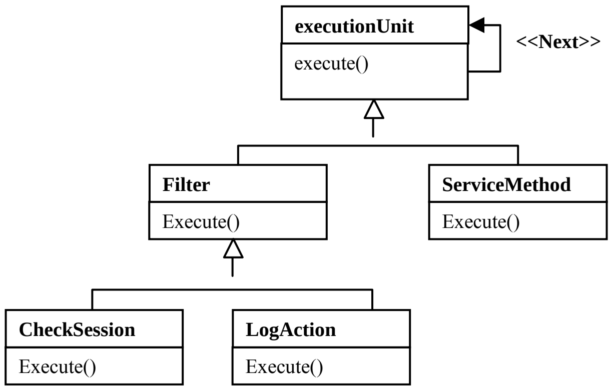


Figure 5: Sequential invokation of operations

Figure 5 depicts the design pattern to support sequential invocation schemes. Currently at the web application layer, two types of operations can be chained: filter and hypermedia service invocations. Actions that need to be invoked within such "chain" need to extend the appropriate class providing developers a convenient way to specify sequential invokation of services and operations in general.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented design patterns that address evolution concerns in web application that are based on SOA. In particular we described what design pattterns have been implemented within the Callimachus project – a CB-OHS- that provides hypermedia services to a broad range of clients including web applications. In Callimachus, design patterns are used to address evolution and maintenance concerns at the web application and hypermedia service layer. Although the design patterns mentioned are already known, we have discussed them in a service oriented context.

Future work includes identifying additional design patterns to address even more elaborate evlolution scenarios. We believe that design patterns have a particular role to play when building web applications on SOA.

5. REFERENCES

1. Ginige, A., Murugesan, S., Web Engineering: An Introduction, *IEEE MultiMedia*, 8(1), Jan.–Mar. 2001, pp. 14–18. *(CHI '00)* (The Hague, The Netherlands, April 1-6, 2000). ACM Press, New York, NY, 2000, 526-531.
2. Papazoglou, M. P., Georgakopoulos, D. (eds.), Service-Oriented Computing, *Communications of the ACM*, 46(10), 2003.
3. Agrawal, R., Bayardo, R. Jr., Gruhl, D., Papadimitriou, S., *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, in Proceedings of the 10th Int'l Conference on World Wide Web (WWW '01, Hong Kong, Hong Kong), 2001, pp. 355–365.
4. Wiil, U. K., Nürnberg, P. J., Hicks, D. L., Reich, S., *A Development Environment for Building Component-Based Open Hypermedia Systems*, in Proceedings of 11th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '00, San Antonio, Texas, USA), 2000, pp. 266–267.
5. Nürnberg, P. J., Leggett, J. J., Schneider, E. R., *As We Should Have Thought*, in Proceedings of the 8th ACM Int'l Conference on Hypertext and Hypermedia (Hypertext '97, Southampton, UK), 1997, pp. 96–101.
6. Tzagarakis, M., Avramidis, D., Kyriakopoulou, M., Schraefel, M., Vaitis, M., Christodoulakis, D., Structuring Primitives in the Callimachus Component-Based Open Hypermedia System, *Journal of Network and Computer Applications*, 26(1), January 2003, pp. 139–162.
7. Vaitis, M., Papadopoulos, A., Tzagarakis, M., Christodoulakis, D., *Towards Structure Specification for Open Hypermedia Systems*, in Proceedings of the 2nd Int'l Workshop on Structural Computing, Springer-Verlag LNCS 1903, 2000, pp. 160–169.
8. Wiil, U. K., *Multiple Open Services in a Structural Computing Environment*, in Proceedings of the 1st Int'l Workshop on Structural Computing (SC1, Darmstadt, Germany), Technical Report AUE-CS-99-04, Aalborg University Esbjerg, Computer Science Department, Denmark, 1999, pp. 34–39.
9. Beringer, D., Melloul, L., Wiederhold, G., *A Reuse and Composition Protocol for Services*, in Proceedings of Symposium on Software Reusability (SSR'99, Los Angeles, California, USA), 1999, pp. 54–61.
10. Buschmann, F., Meunir, R., Rohnert, H., Sommerland, P., Stal, M., *Pattern Oriented Software Architectures: A System of Patterns*, John Wiley & Sons, 1996.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
12. Hu, J., Schmidt, D. C., JAWS: A Framework for High-performance Web Servers, in Fayad, M., Johnson, R. (eds.), *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, 1999.
13. Dart, S.: Configuration Management: the missing link in Web engineering. Artech House, 2000.
14. Karousos, N., Pandis, I., Reich, S., and Tzagarakis, M. (2003). Offering Open Hypermedia Services to the WWW: A Step-by-Step Approach for the Developers. In Proceedingss of Twelfth International World Wide Web Conference WWW2003, (Budapest, Hungary), pp. 482-489.
15. Wiil, U., Nurnberg, P., Evolving hypermedia middleware services: Lessons and observations. Proceedings of the Thirteenth ACM Symposium on Applied Computing (SAC 99), San Antonio,TX, US, Mar.,1999

Towards Empirical Validation of Design Notations for Web Applications: An Experimental Framework

Paolo Tonella[1], Filippo Ricca[1], Massimiliano Di Penta[2], Marco Torchiano[3]
[1]ITC-irst, Trento, Italy
[2]University of Sannio, Benevento, Italy
[3]Politecnico di Torino, Italy
tonella@itc.it ,ricca@itc.it, dipenta@unisannio.it, marco.torchiano@polito.it

ABSTRACT

Web application design involves at least one additional dimension over traditional software design: navigation, as supported by hyperlinks. Available design notations for Web applications offer enhanced separation of different design concerns (among which, navigation) and promise increased understandability and maintainability. However, such claims have not yet been tested in the field.

In this paper, we propose a framework for the execution of empirical studies aimed at assessing the cost-effectiveness of Web design notations. The context of the empirical studies is a typical maintenance and evolution scenario, involving activities such as program comprehension, impact analysis and change implementation. The most important obstacles and challenges in the design of such studies will be considered in this paper. We will propose counter-measures and possible mitigations for them. Finally, we will instantiate the framework into a specific empirical study that we plan to conduct in the next few months.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.7 [**Distribution, Maintenance, and Enhancement**]:

Keywords

Empirical Studies, Web Applications, Design Notations.

1. INTRODUCTION

Web application design is a complex activity which requires the ability to deal with multiple and different kinds of concerns. A Web application is typically composed of various parts that need to be modeled at design time. Among them, the most important ones are persistent data, business logic, navigation structure, user interface. Other relevant concerns include security, transaction management, authentication. All these dimensions of a Web application must be addressed properly in the design documents.

Several design notations and methodologies have been proposed in the literature, in an attempt to provide solutions to the problems mentioned above. Among the most referenced approaches are WebML [2], UWE [6], WSDM [10], OOHDM [9], Conallen [3]. Many of these notations are extensions of UML [8]. Their most distinctive feature is typically the ability to model explicitly the navigation structure of a Web application through a dedicated model. Such a model is often accompanied by "more traditional" entity-relationship (or similar) models (for the data), static and behavioral models (e.g., class, interaction and activity diagrams) for the business logic, etc.

Separation of concerns during Web application design is clearly important during the initial development. However, it poses many problems during the maintenance and evolution phase, which actually accounts for the vast majority of an application's life cycle[4]. In fact, it is hard to keep the different views up to date and aligned. Traceability towards the implementation may be also problematic. The overlaps and interferences between different models may be hard to detect. Overall, it might be not so obvious that the benefits encountered during the initial development are kept during the evolution phase, if assessed against the associated costs (updates, alignment, traceability, etc.).

In such a context, it is extremely important to precisely understand the relative merit of the various models that have been proposed in the literature, once considered during the maintenance and evolution of an existing Web application. It might be the case that some design notations are useful mainly during the initial development, while becoming only marginally useful later, with a negative cost-benefit trade off. Others might on the contrary reveal themselves as powerful tools that can be used to tackle the typical maintenance and evolution scenarios. Gathering such knowledge is fundamental for the final user, who would be able to make an informed decision. However, no empirical study was conducted so far in this direction. In the literature, Web design methodologies are evaluated only on small examples constructed ad-hoc by the proponents or through isolated case studies, whose results cannot be usually generalized and do not provide any comparative information.

Goal	Analyze the support given by Web design notations to the comprehension and modification activities during evolution.
Null hypothesis	No significant effect on effectiveness of task execution and quality of the result.
Main factor	Design notations being validated.
Other factors	Systems, tasks, subjects and subject skills, training, tools.
Dependent variables	Knowledge acquired, capability to locate changes precisely, quality of the result.

Table 1: Template for the empirical studies.

In this paper, we propose a framework for the execution of empirical studies aimed at comparing different design notations in order to assess the support they provide to the maintenance and evolution of Web applications. The aim of the framework is to support systematic and controlled execution of experiments for the empirical validation of Web design notations. The framework specifies the high level goal and research questions of the studies, identifies the relevant factors and proposes ways to deal with the main challenges. An instance of the framework is a specific empirical study, for the assessment of specific notations in a specific evolution scenario. In this paper, an instance related to the validation of the stereotyped class diagrams (following the Conallen's notation) is presented.

Section 2 describes the framework, while Section 3 presents one example of instantiation of the framework, which is the empirical study we are going to conduct in the next few months. Conclusions and directions for future work are drawn in Section 4.

2. TEMPLATE FOR THE EXPERIMENTAL DESIGN

Table 1 summarizes the main elements of the experimental framework. Such a template follows the guidelines from well–known experimental software engineering books by Wohlin *et al.* [11] or Juristo and Moreno [5].

The general goal is quite clear: assessing Web design notations in the maintenance phase. When instantiating the framework, the general goal takes the form of a specific validation objective that addresses a specific question about the relative merit of specific notations, selected among those available in the literature.

2.1 Hypothesis

The null hypothesis is that the treatments being compared (e.g., two design notations) exhibit no significant difference. When the null hypothesis can be rejected with relatively high confidence, it is possible to formulate an alternative hypothesis, which typically admits a positive effect of one design notation in the execution of maintenance tasks. The alternative hypothesis can be further specialized according to the specific context in which it holds (see Table 3). In turn, this is characterized by the independent variables (see discussion of *other factors* below). The alternative hypothesis is formulated in terms of the main independent variable controlled in the experiment, i.e., the design notations being used.

2.2 Treatments

The treatments compared can be either two alternative web-specific design notation or a general purpose notation and a web-specific notation.

2.3 Objects

In order to support maximal internal and external validity of the studies, all the other independent variables that may affect the outcome of the study must be taken into account and possibly controlled. These include the software systems that are the object of the maintenance tasks and the tasks themselves. To mitigate the effect of these factors on the experiment's validity, the subject systems should be selected with features (size, complexity, functionality) that are typical of real Web applications. The tasks should be representative of the activities carried out by Web developers in their daily work.

2.4 Subjects

The subjects executing the maintenance tasks are another crucial factor affecting the possibility to generalize the outcome of the study. To mitigate the effects of this factor, proper training should be given to the involved subjects, so as to ensure a common, basic knowledge of the technologies involved in the experiment, as well as of the design notations being validated. Moreover, questionnaires can be used to assess the actual skills of the participating subjects and to (possibly) include them among the factors (independent variables) being considered. Such an assessment allow to properly design the experiment, ensuring a uniform distribution of subjects with high and low ability across all experiment groups. Moreover, the awareness of the subjects' ability permits to use blocking [11] when analyzing the results.

The tools provided to the subjects and the associated programming environment must be also selected carefully, so as to mimic, as much as possible, the working environment used for Web development.

2.5 Procedure and design

As discussed above, several factors affect the internal and external validity of an empirical study such as the one we are proposing. We already described ways to mitigate their effect on the generality of the results. For some of them, an additional method is counter-balancing, which can be achieved through careful design of the experimental sessions.

Table 2 shows an experimental design which balances the effects of the software system under maintenance, of the order of the treatments and of the learning curve of the involved subjects. This is achieved by dividing the subjects into four groups and involving them in at least two experimental sessions (laboratories). The order in which the systems under

Goal	Analyze the use of stereotyped UML diagrams reverse engineered from the code.
Null hypothesis 1	No significant effect on comprehension level.
Null hypothesis 2	No significant effect on impact analysis.
Null hypothesis 3	No significant effect on maintenance result.
Main factor	Stereotyped (Conallen's) UML diagrams vs. traditional UML diagrams.
Other factors	Systems (TuDu and DMS), tasks (comprehension, impact analysis and maintenance), subjects (students), training, tools.
Dependent variables	Comprehension level, accuracy of impact analysis, quality of modified code.

Table 3: Template instance for validating the use of stereotyped (Conallen) UML class diagrams in software maintenance tasks.

	Group 1	Group 2	Group 3	Group 4
Lab 1	Sys1-Treat1	Sys1-Treat2	Sys2-Treat1	Sys2-Treat2
Lab 2	Sys2-Treat2	Sys2-Treat1	Sys1-Treat2	Sys1-Treat1

Table 2: Experimental design.

study are presented to the subjects is reversed when considering groups 1, 2 with respect to groups 3, 4. The order of the treatments is also reversed between groups 1, 3 and 2, 4. The combination of system and treatment is completely counter-balanced, by covering every possible sequence of system and treatment.

Overall, this experimental design requires the execution of at least two experimental sessions with at least four groups of subjects. When these constraints are met, complete balancing of the order in which systems are considered and treatments are subministered is obtained.

2.6 Variables

In order to measure the effects of a treatment (design notation), metrics must be defined that allow evaluating the experimental hypotheses. For example, such metrics could capture the comprehension level reached, the ability to locate the requested change and the quality of the modified system. Questionnaires, code inspections and change impact estimates are examples of techniques that can be used to derive metrics that map to the effects to be measured.

3. INSTANTIATING THE TEMPLATE

We are planning the execution of a first empirical study that instantiates the framework described in the previous section. The *goal* of the study is to analyze the use of stereotyped UML diagrams (following the approach by Conallen [3]), with the purpose of evaluating their usefulness in Web application comprehension, impact analysis and maintenance. The *quality focus* is ensuring high comprehensibility and maintainability, while the *perspective* is multiple:

- **Researcher**: evaluating how effective are the stereotyped reverse engineered diagrams during maintenance.
- **Project manager**: evaluating the possibility of adopting a Web application design and reverse engineering tool in her/his organization.

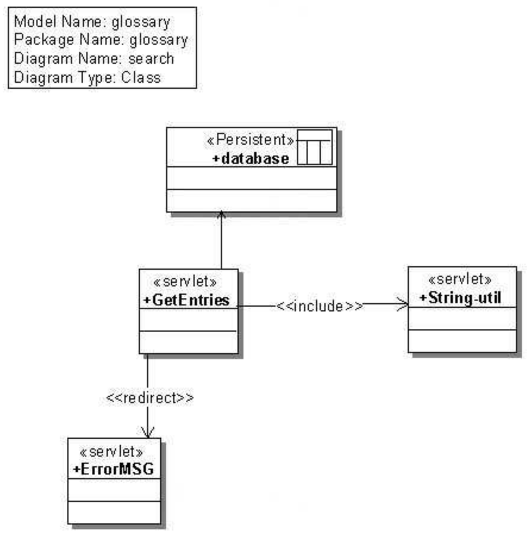
3.1 Hypotheses

Since we are interested in how stereotypes affect comprehension level, impact analysis and maintenance, we formulate three different null hypotheses (and the related alternative hypotheses):

- H_{01}: When doing a comprehension task the use of stereotyped reverse engineered class diagrams (versus non-stereotyped reverse engineered class diagrams) does not significantly affect the comprehension level.
  H_{a1}: When doing a comprehension task the use of stereotyped reverse engineered class diagrams (versus non-stereotyped reverse engineered class diagrams) significantly affects the comprehension level.
- H_{02}: When doing an impact analysis task, the use of stereotyped reverse engineered class diagrams (versus non-stereotyped reverse engineered class diagrams) does not significantly affect the accuracy and the effectiveness in the execution of the task.
  H_{a2}: When doing an impact analysis task, the use of stereotyped reverse engineered class diagrams (versus non-stereotyped reverse engineered class diagrams) significantly affects the accuracy and the effectiveness in the execution of the task.
- H_{03}: When doing a maintenance task, the use of stereotyped reverse engineered class diagrams does not significantly affect the effectiveness in the execution of the task.
  H_{a3}: When doing a maintenance task, the use of stereotyped reverse engineered class diagrams significantly affects the effectiveness in the execution of the task.

3.2 Treatments

The treatment considered in this experiment is the design notation proposed by Conallen [3]. Since this notation extends UML through a set of stereotypes, the notation used for comparison (second treatment) is basic UML, with no Web-specific stereotype. The aim is to determine the amount of improvement (if any) that can be obtained by means of Conallen's stereotypes in the maintenance and evolution phase. A similar, prelimilary study, focused on the use of stereotypes for comprehending applications related to the communication domain has been conducted by Kuzniarz *et al.* [7]. The authors showed that the use of stereotypes helped to improve the comprehension. Diagrams are reverse engineered from the code and then adjusted manually, so as to reproduce a situation where diagrams are aligned with the code and at the same time represent a meaningful and compact abstraction of the implementation.



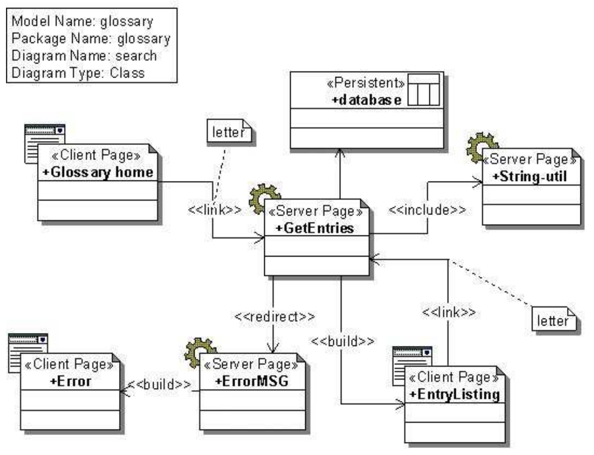


Figure 1: Basic UML class diagram (left) compared to Conallen's diagram (right).

TuDu		
	Files	**LOC**
Java	62	2929
JSP	19	1232
Total	81	4161

DMS		
	Files	**LOC**
Java	40	3731
JSP	11	1125
Total	51	4856

Table 4: Characteristics of the systems under study.

Figure 1 gives an example of the extra information provided by Conallen's diagrams, compared to that usually represented in standard UML class diagrams. The modeled Web application implements a glossary. On the left is the basic UML diagram, showing the Servlet (*GetEntries*) and the database. On the right, the same diagram is enriched with Conallen's notation. It includes the client pages generated by the Servlets (e.g., *EntryListing*), the static pages (*Glossary home*) and the hyperlinks (notation: `<<link>>`).

3.3 Objects

Two Web applications were selected for this study: *DMS* and *TuDu*. Both are small/medium size applications (see Table 4) based on the Servlet/JSP technology and downloaded from sourceforge.net. Although commercial or institutional Web applications may be larger, given the time constraints of the experiment and the involved subjects (students), it was not feasible to consider larger examples. The application domains of the selected system is pretty typical of existing Web applications. The same holds for their organization and overall functioning. *TuDu*[1] is an on-line application for managing todo lists supporting cooperative work of distributed teams. It can be accessed via RSS feed. *DMS*[2] is a document management system, providing a Web centric interface to manage, access and distribute documents which are kept under version control.

[1]http://app.ess.ch/tudu

[2]http://docmgmtsys.sourceforge.net/

3.4 Subjects

The subjects participating in the study are University students. The study will be replicated at three different sites: University of Trento, University of Sannio (Benevento) and Politecnico di Torino, in Italy. The participating students are at different levels of their course of studies, ranging from undergraduate students, to graduate and master students. Replication with students having different levels of expertise will give us the opportunity to investigate this further dimension, by comparing the results obtained at the different sites.

3.5 Procedure and design

Students will be trained on Conallen's notation, as well as all the technologies used in the target applications (e.g., Servlets/JSP). They will be involved in four experimental sessions (laboratories), each lasting approximately 2 hours. The assignment given to each group of students in each laboratory follows the experimental design in Table 5, which is an instance of the counter-balanced scheme described in the previous section.

Each laboratory in the original scheme (see Table 5) is split into two (*Lab N-a, Lab N-b*, with $N = 1, 2$). The first laboratory (*Lab N-a*) consists of the execution of a comprehension task followed by impact analysis. Comprehension is driven by a request for change. Impact analysis consists

	Group 1	Group 2	Group 3	Group 4
Lab 1-a	TuDu-Con	TuDu-UML	DMS-Con	DMS-UML
Lab 1-b	TuDu-Con	TuDu-UML	DMS-Con	DMS-UML
Lab 2-a	DMS-UML	DMS-Con	TuDu-UML	TuDu-Con
Lab 2-b	DMS-UML	DMS-Con	TuDu-UML	TuDu-Con

Table 5: Instantiation of the experimental design.

of an estimate of the portions of the Web application affected by the requested change, The second laboratory (*Lab N-b*) is the implementation of the change. The programming environment will be the one students are familiar with (Eclipse), with plugins supporting the design notation being validated (Conallen). The treatments indicated in Table 5 are Conallen (Con) vs. basic UML (UML).

Finally, we will ask students to fill-in a survey questionnaire (both after *Lab N-a* and *Lab N-b*) regarding the task and system complexity, the adequacy of the time allowed to complete the tasks and the usefulness of the provided diagrams.

3.6 Variables

The dependent variables of the study are:

- Comprehension level (hypothesis H_{01}).
- Capability of doing impact analysis (hypothesis H_{02}).
- Quality of the maintained code (hypothesis H_{03}).

In order to assess the effects of the treatments on the dependent variables, we will use questionnaires, test case execution and design/code inspections, and we will measure:

1. Number of correctly answered questions and time needed to answer them (both for the comprehension and for the impact analysis questionnaire).
2. Functional behavior of changed code (passed test cases).
3. Time required to implement the changes.
4. Flaws in new design (determined through inspections).
5. Code quality (determined through inspections).

4. CONCLUSIONS

The research in Software Engineering (SE) is expected to produce scientific knowledge. However, this is difficult to achieve since humans are typically in the loop of any novel SE technology. This is especially true for design notations, such as those proposed for the development of Web applications.

This work represents a first step in the direction of gathering systematic knowledge [1] about the cost-effectiveness of Web design notations. We have defined a common framework for the empirical studies focused on this topic. Then, we have instantiated the general template, obtaining the design of the first experiment that will be executed in this area. We tried to control as much as possible the factors possibly affecting the outcome of the experiment. Replication at three different sites, with different subjects, will further strengthen the results. In the design of the experiment, particular care was devoted to the balancing of the main independent variables.

A lot of future work remains to be done. First, we will actually conduct the planned experiment and replicate it at three distinct sites. We will also encourage further replications by other researchers outside the initial project team. We expect that the results of the study will provide feedback on the usefulness of different design views, according to the tasks at hand and depending on the features of the application under study. Data on the (possibly) different behaviors of subjects with different skills will be also gathered. Overall, we aim at putting the design notations proposed for Web applications in the context of a maintenance and evolution scenario, in order to assess their cost effectiveness. Replication with notations different from the one considered initially will be also fundamental to corroborate our initial findings.

5. REFERENCES

[1] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, July/August 1999.

[2] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2002.

[3] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.

[4] T. C. Jones. *Estimating Software Costs*. McGraw Hill, 1998.

[5] N. Juristo and A. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, Englewood Cliffs, NJ, 2001.

[6] A. Knapp, N. Koch, and G. Zhang. Modeling the structure of web applications with argouwe. In *Proc. Fourth Int. Conference on Web Engineering*. Springer Verlag, July 2004.

[7] C. W. L. Kuzniarz, M. Staron. An empirical study on using stereotypes to improve understanding of uml models. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 14–23, Bari, Italy, 2004.

[8] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 2004.

[9] D. Schwabe and G. Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems*, 4(4):207–225, 1998.

[10] O. M. F. D. Troyer and C. J. Leune. Wsdm: a user centered design method for web sites. In *Proceedings of the seventh international conference on World Wide Web 7*, pages 85–94. ACM Press, 1998.

[11] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction.* Kluwer Academic Publishers, 2000.

User-Centered reverse engineering: Genesis-D project

Luca Mainetti
Dipartimento Ingegneria dell'Innovazione
Università di Lecce
Via per Arnesano, 73100 Lecce, Italy
Tel: +39 0832 297216
Fax: : +39 0832 297279
luca.mainetti@unile.it

Roberto Paiano
Dipartimento Ingegneria dell'Innovazione
Università di Lecce
Via per Arnesano, 73100 Lecce, Italy
Tel: +39 0832 297296
Fax: : +39 0832 297279
roberto.paiano@unile.it

Andrea Pandurino
Dipartimento di Ingegneria dell'Innovazione
Università di Lecce
Via per Arnesano, 73100 Lecce Italy
Tel.: +39 0832 297229
Fax: +39 0832 297279
andrea.pandurino@unile.it

ABSTRACT

In the last years, the requirements of the end-users are notably evolved. A good software must not have only a good functionality cover but it also must have good usability features. From this point of view, the most recent design methodologies focus on the interaction between the end-user and its user experience; in this way, the design focus there is not on the data element (represented as objects or relational entity) but on the end-user and its perception of the information anymore. According to growing needs, it is more and more frequent the requests of reengineering of existing products that, developed in many years, have a good coverage of the application domain; these existing products result completely unsuitable to the modern paradigms of interaction. In this paper, we introduce an experience of reengineering (in web perspective) of a legacy application on the environment monitoring. This experience has been performed into the industrial research project (funded by Italian Government) called "Genesis-D"[1] (Global Environmental Network System of Information for Sustainable Development) that understanding the importance of the new user-centred approach wants to reengineering its own products.

Categories and Subject Descriptors

D.2.0 [**Software**]: Software engineering – *General.*

General Terms

Management, Documentation, Design, Standardization, Languages, Theory, Legal Aspects.

Keywords

User experience, Web application design, Environmental domain, reverse engineering process

1. INTRODUCTION AND BACKGROUND

The rapid growth of the web and the on-line services has made the complexity of the design and the following development manageable only through structured and engineered approaches. In order to improve the web application (WA) quality, the designer must not only manage the information and navigation aspects but, also all the multi-user and multi-device requirements combined with the customization needs; although, these aspects have been subject of study [6][3], their cohesion and mutual implications have brought to the birth of new methodological approaches.

In the last years, several design solutions have been proposed; their main goal is to adapt the well know methodology to the new WA requirements; thus, these methodology would be able to introduce complex contents and to manage the requirements of user-accessibility.

The methodologies so adapted are founded on the idea that to model any page type (and so to describe a WA), is enough to represent its fundamental elements (pages, form and link) and to relate themselves through the classical object-oriented relationships.

Starting from this perspective, we have to take in consideration Jim Conallen approach [4]: WAE (Web Application Extension), an UML profile that adding new stereotypes, simplifies the Web page representation. WAE is strongly confirmed in the industrial environment and UML community. WAE is very useful and powerful when is used to describe the logical design of the software modules that composes a Web system; but, on the contrary, it has some weakness (due to the lack expressive ability and the inadequacy) to represent the User Experience aspect.

These considerations are helpful both when the designer is planning an application ex-novo and when he/she is making an application reengineering; rather, in a phase of reengineering the semantics of the user experience must drive the designer: the designer must not be influenced by the object-oriented paradigm. After all, supposing to have a pure object-oriented design, if WAE was applied the result would be only the same application translated in the web domain but with the same politics of interaction of the first one getting a "porting" and not a reengineering in the user-centred point of view. Starting from these considerations, we introduce a reengineering experience in which different design techniques, both traditional and user-centred, are combined.

[1] The leading company is Edinform SpA and is located in Lecce

2. THE APPLICATION DOMAIN AND THE APPLICATION

The approach to the environmental protection (understood as habitat of all the organisms and as organic structures of systems and subsystems), is evolved considerably. If, in the sixties, the public administration attitude was finalized to control to the law prescription, today great importance is given to the knowledge acquisition of the factors that heavily affect the environment quality; this choice is determined by the high growth rate of the population and by the evolution of the productive system that make pressure on the environment.

Today the monitoring activity and environmental control are made not only by the public administrations such as Municipality, Provinces, Regions but also by several associations and organizations and by protection environment agencies in the national and international territory.

The number of involved actors and the need to acquire the quality status of the environment and territory, lead to create a consistent, coherent and reliable informative exchange. To achieve this goal organization and institution nets (with the target to improve the collaboration for a common environmental politics) were born. The efforts to collect and to delivery the environmental knowledge in Italian and European area does not match in regional institutional level, because the technological infrastructures does not support informative exchange. In this context take places the research project GENESIS-D[1] sponsored by Edinform S.p.A in collaboration with the University of Lecce and Polytechnic of Milan. The project goal is the creation of a "framework" for the modelling and the development of software systems about the environmental management at regional or sub-regional institutional level. The Web applications, obtained starting from the framework Genesis-D, have to improve the interface and the interchange of environmental information among different institutional subjects such as Regions, Provinces, Municipality, ARPA (Regional Agencies for the environmental protection), etc.

The application domain is characterized by a considerable number of actors (public administrations, authorities, corporate bodies, local health services, experts of domain, etc), of administrative documents (norms and national laws, regional, directives of the European Community, etc.), of studies (international standard, studies of sector, models, etc.). It is clear that the creation of a framework based on the reverse reengineering of existing product (developed in about ten years), that considers many laws, studies, is a good starting point. In accord with Edinform S.p.a., it was established to perform the reverse engineering of the application SIRA (Environmental Regional Information System).

2.1 SIRA

SIRA supports the environment management and control activities in a regional context.

In accord with the standard SINAnet (Cognitive National System and of the Environmental Controls), SIRA split the environmental subject in the thematic area base of the National Thematic Centres: Waters inside and sea coastlines (EKB-AIM), Wastes (EKB-RIF), Soil and contaminated sites (EKB-SSC), Nature Preservation (EKB-CON), Air climate and emissions in atmosphere (EKB-ACE), Physical Agents (EKB-AGF). This segmentation strategy is the base of EKB (Environmental Knowledge Base). Its reality is the Environmental Reality composed by environmental facts and phenomena.

Starting from this point of view, SIRA is structured in the following subsystems:

- *General registry*: it manages all the registry data of the firms, of the subjects and of the operational structures with an impact on the environment or that they have involved in the control and in the environmental prevention.
- *Management procedures*: it allows the administrative management of the documents produced by the activity of environment monitoring.
- *Soil*: it allows the management of data coming from the monitoring of the environment risk areas (polluted sites and plants at risk of accidents with dangerous substances) present on the regional territory.
- *Water*: it allows the management of data coming from the monitoring of hydrographical basins, the water bodies, the waterworks, the withdrawal work, the unloading and presence of mud on the regional territory.
- *Nature*: it allows the management of data coming from the monitoring of the protected areas, and of the relative areas of protection, on the regional territory.
- *Wastes*: it allows the administrative management of the unique form of environmental declaration, annually introduced by the firms and by the municipalities that they participate in the cycle of waste management.
- *Security*: it allows the definition of the access profiles of the system.

3. THE REVERSE ENGGINEERING PROCESS

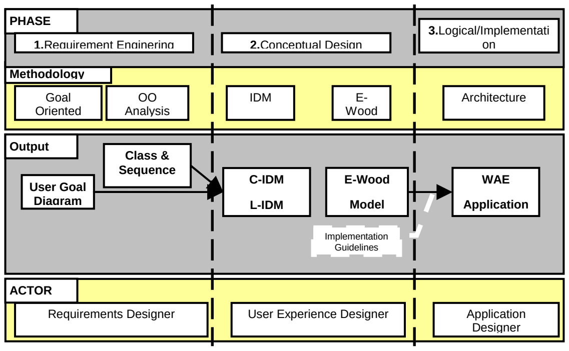
As written above, in the first project phase the main goal is the reengineering (with user centred approach) of SIRA. In order to manage the complexity of the application domain and taking into consideration the kind of application, the creation of a process in order to correctly drive the designer is needed. The main process is divided in three macro-phases:

1. *Requirements elicitation and analysis*. This phase can be divided in two parts: the first one uses the "user centred" approach focused on the stakeholders and on their goals; the second one aims to represent the application domain knowledge. In the first sub phase, we recommend the use of requirement engineering approach based on "goal oriented" techniques; in our case study, we used the methodology AWARE based on the KAOS theory of Lamsweerde [5]. This technique traces the meaning of each requirement that is related to its specific goal. The "stakeholder" is whoever (end-user, developer, manager, buyer, financier, etc.) has a specific interest in the system and so it is able to express his goals. The goals of a specific stakeholder can eventually be shared with other one. A single goal not related to a specific stakeholder, it is not a goal for the web system and it must be therefore deleted. The output of the analysis is a user-centred vision of the application requirements and will be the base for the following process phases. The second sub phase, instead, uses object-oriented technique to perform the

application reverse engineering. Output of this sub phase is the complete diagrams (class diagram and sequence diagram) that could be defined still part of the requirement engineering because they aim to describe the application domain. Our case study describes the application SIRA using OO paradigm and this is a good starting point for the informative object study of the domain and the relationships among them.

2. *User experience design*. This is the first phase of the reengineering and must have performed using WA design methodologies based on user centred approach. In the case study, we used two methodologies IDM [2] (used to describe the interactive and navigational essential aspects of multi-channel applications, focusing on the dynamics of dialogue end-user / application) and E-Wood (Edinform Web Object-oriented Design) that, refining the IDM analysis, uses the object-oriented techniques integrated with the necessary semantics for the web applications. Both the two designs have kept in mind of the informative objects derived by the SIRA reverse engineering combined with the goal-oriented analysis; in other words, the two methodologies allow to filter the OO analysis with the goal-oriented vision of the domain for the specific stakeholder. The E-Wood design methodology has been created by the Polytechnic of Milan, and inherits the notation from UML. E-Wood allows the conceptual design of the application with the WAE profile. Thus, its output could be adapted to the specific implementation technology.
3. *Implementation design*. The output of the phase is intended to the developer and provides the implementation model of the system; in other words, it describes through WAE the pages and the software components that the developer must implement using a specific implementation technology such as Micorsoft .Net, J2EE model 1, J2EE model 2 etc. It is called also "logical design" and it allows adding the implementation details directly connected with the system and the selected architecture.

In figure 1, it is possible to see the process scheme of reverse engineering. The transition from the conceptual modelling E-Wood to the implementation design is made easier thank to the guidelines provided. The guidelines provide several advantages to create the final product; in fact, applied in a systematic way after having established the architecture type to use, they allow to conform the implementation of specific E-Wood structures and accordingly to get an uniformity in the code; in other words, the guidelines limit the freedom of the developer to translate the methodology objects in code.



Figures 1: Scheme of the process of reverse engineering

3.1 The E-Wood methodology

Following UML community approach, in order to model the page features such as layout, contained, navigation in E-wood several views are used. The goal is to separate the different aspects into different design in order to improve the quality of the analysis of the aspects that are correlated each other. The required views are:

- *Structural navigation view:* it specifies the pages used to represent the information content related to a conceptual entity. In this view the navigation among these pages is defined too.
- *Association view*: it allows to specify how create the navigation between pages that describe different entity linked by a semantic association.
- *Navigation Path view*: it allows to specify as the navigation among pages created for supporting the end-user interaction with driven path of navigation.
- *Operation Views*: it allows to specify the pages that support the execution of operations.

The E-Wood methodology provides also these general views:

- *Page Template View*: it defines the general structure of the pages and the aspects of layout specifying general contents and links of landmark shared with various pages.
- *Navigational Map View*: it provides a view of whole application, or related to the screens belonging to a single package of pages, showing the main Screens and the possible navigation among them.

4. REVERSE ENGINEERING OF THE SIRA APPLICATION

According to the process described above, the SIRA application reverse engineering was performed. In the stakeholder analysis, it must be highlighted that different authorities and corporate bodies share the responsibilities for protecting and preserving the environment, that operate at different institutional levels (municipality, intercity, provincial, inter-provincial, regional, inter-regional, national, EU, etc). Among these subjects, at national level we remember the APAT (Agency for the Protection of the environment and Technical services), the Minister of the environment, the Forest Body of the State, the Italian Red Cross, the Civil Protection, etc. At regional level it is opportune to mention the various ARPA (Regional Agencies for the Protection of the environment), the Basin Authorities and the Park Authorities, the Provinces, the Regions. At provincial level we remember the APPA (Provincial Agencies for the Protection of the environment), the metropolitan cities, the prefectures, the provinces, the offices responsible for police force and public order, while at town level there are the municipalities. Studying the competences of these corporate bodies, it is possible to identify not detailed professional figures (that would be hundreds considering that each organization has an inside structure and own rules) therefore we focused on the roles that the figures assume in an environmental monitoring system; in detail, three different typologies of roles are been founded:

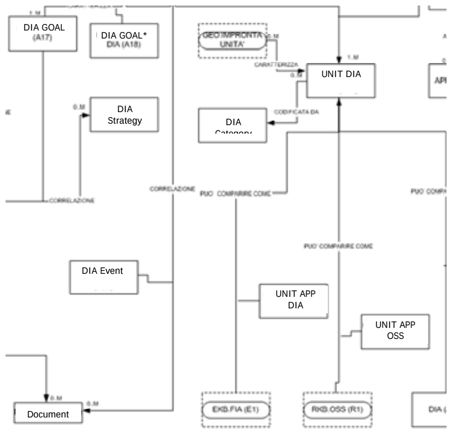
- *Government role*: who adopts the opportune tools of government for the protection and preservation of the environment and cooperating with the government end-users of other corporate body and authorities in order to perform an integrated territory management. The first level goal are: Optimal management of the territory, Reduction of the level of acoustic pollution, reduction of the level of atmospheric pollution, the waters' preservation, reduction of the wastes and reclamation of the polluted sites, preservation of the human health.

- *Coordination role:* supervise the job of the operational end-user; it provides all the necessary information to the government end-user to adopt the opportune measures. Its goals are the same of high-level government role but with different assignments and functionality; for instance, in order to perform an optimal management of the territory, the coordination end-user takes in care the promotion and planning of the use services of the local areas and parks (as the creation of cycle routes); thus, it performs studies and projects preparatory to the environmental activities and territorial planning, it finds the development opportunities of the territory compatible with the environment, it promotes initiatives to enhance the naturalistic patrimony and to protect the biodiversity and the environmental quality, and it deals with the management of the censuses of the wildlife and of the surveys of the habitats in the natural reserves.
- *Operational role* deals with to perform the surveys for the environmental monitoring and to point out to the coordination end-user about particular anomalous values emerged by the analyses performed so that to be able to adopt the opportune measures, effect the plans of management of the reserves, deals with to perform the inquiry of environmental impact evaluation for the realization of new works, and to perform the environment monitoring, that is to periodically perform the censuses of wildlife and the survey of the habitat in the parks, in the reserves and in the other areas of interest.

At the end of the AWARE analysis, the reverse engineering of the application SIRA was performed with Object-oriented paradigm. The application SIRA from the end-user point of view is very bind to the information managed; in fact, the user interface in its structure and navigability mirrors the relational model and, therefore, it is limited to a set of forms of insert/view.

The environment business logic is directly contained instead in a set of objects related to the insert forms. The application allows the end-user profiling preventing the access to particular information to the end-users not authorized.

The OO analysis identified about 190 classes with the relative methods and objects (in figure 2 a part of the class diagram is showed).

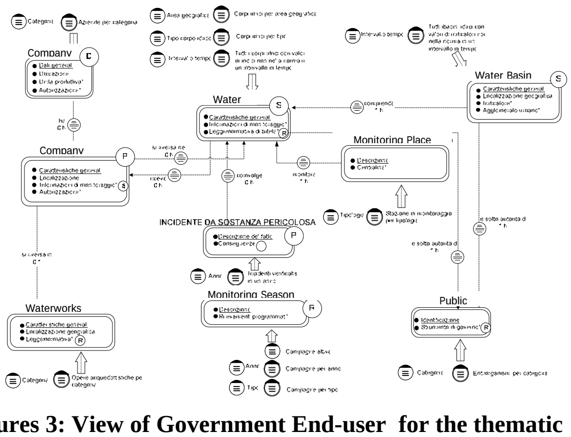


Figures 2: Part of the class diagram of SIRA

In the figure 2, the identified objects are tightly bind to the information that represent and cannot directly be used in a user-centric application, since they mirror the data and do not keep in mind as they are perceived by the end-user. Using the goal analysis and the detailed information (attributes and methods) derived from the OO analysis, the IDM methodology is applied.

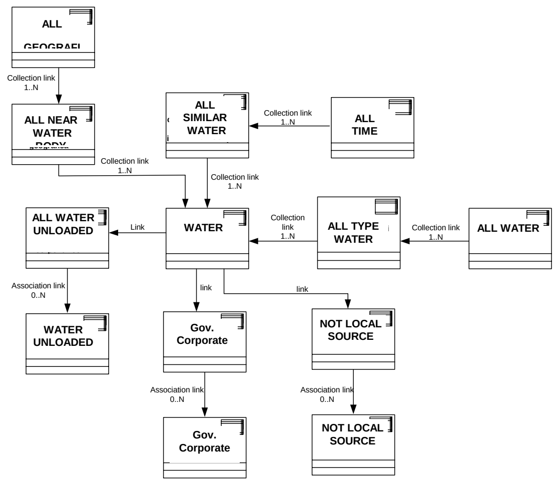
The conceptual model of the new system of environmental monitoring has been realized keeping in mind the thematic of the environmental domain: Water, Soil, Air, Nature, and Waste. Keeping in mind therefore the aforesaid thematic environmental and the typologies of stakeholders, have been realized for each end-user five IDM views, one for each thematic. In figure 3 the IDM scheme of the thematic Water for the Government is showed.

The founded topics contain all the information derived by the objects (OO analysis) modified with the end-user perception of them: for instance, the topic "waterworks" contains inside the dialogs act: General Features (Description, Type of work, Type of waterworks net, Manager, Year of realization, flow in, flow out, Pressure in, Pressure out, K, Quota), Geographical Location (imprint, geo-code, geo-references), Law / normative (Denomination, Text, Category tool, Absorbed Tool). It is clear that all the dialogs acts derives from different objects; in fact, in the class diagram there is the object RKB.OSS of which the waterworks is an instance that is related with the class "DIA unit of application" to which the geo-references (GeoImprint Unit) is related.



Figures 3: View of Government End-user for the thematic Water

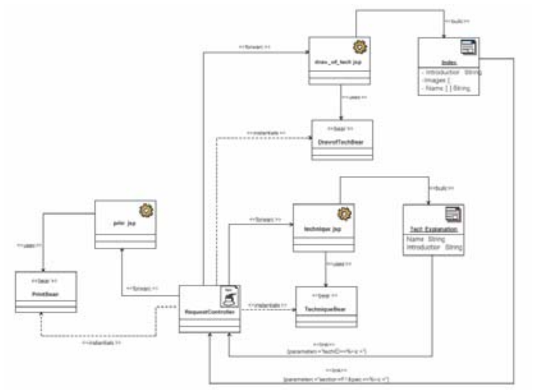
At the end of the IDM design (in which the information is modified in terms of user experience), the E-Wood analysis is performed; thus, all the E-Wood views for specific end-user and environmental thematic are produced: Structural Navigation View, Association View, Navigation Paths View, Operations View, Page Template View, Navigational Map View.



Figures 4: View of the navigational map of the water Body for the government end-user

The figure 4 shows the navigational map of the "water body" object.

At the end of the E-Wood modelling, established the software architecture, the implementation view could be produced. This task is not excessively complex because E-Wood uses a similar notation of implementation views and thus, it is possible to establish a mapping between the objects of the conceptual modelling and those of the "implementation view"; furthermore, Polytechnic of Milan has established the guidelines that allow an easy translation of the E-wood diagrams in the specific architecture.



Figures 5: "implementation View" of the navigational map of the water Body for the government end-user

The Figure 5 shows the implementation view of the navigational map of the water Body for the government end-user in the case was chosen as implementation architecture MVC model 2: the request controller is present and each JSP page invokes the bean of the corresponding entity.

5. CONCLUSIONS

The growing demand of new services and the continuous interest for the web is forcing a lot of company to evolve their applications. This transition is heavy: all the application logic has to change from a system vision to user centric vision. The information is not fundamental while the perception and the interaction that the user has with it is the design core. It is clear that whether to resolve the problem of the reengineering is not enough a methodology but it is necessary a *process* that leads the designer to understand the domain, the stakeholders and the following phase of analysis and design. This paper presents a reengineering process that, integrate well known methodologies as Aware, Object-oriented, IDM and E-wood, applied to a real case. The output is good: a logical model effectiveness and uniform ready to be implemented. The effort to perform the complete design with user centred approach has required just 4 months of a designer (a very small effort for a domain very extended). It is sure, that the introduction of the guidelines for the implementation level, constitutes a great facilities for the designer and it allows to get a design more uniform and correct. Since the guidelines are tightly connected with the selected implementation architecture, a very interesting future development is to create new guidelines toward new technologies.

6. REFERENCES

[1] Balconi T., Mainetti L., Paolini P., Perrone V., *GENESIS-D: Formal description of the Conceptual Model*, Polytechnic of Milan, deliverable D2.2, project GENESIS-D, October 2004.

[2] Bolchini, D., Piccinotti, N., Randazzo, G., Gobbetti, D., IDM To User-Centred Model Shaping User Interaction as to Dialogue, *In Proceeding of the HCII 2005 International Conference on Human-computer Interaction* (Las Vegas, USA, 2005).

[3] Brusilovsky, P. Methods and tecnique of adaptive hypermedia. *User modeling and user adaptive interaction*, vol 6, nos. 2-3, (1996) 87-129.

[4] Conallen, J., Building *Web Applications with UML, Second edition.* Addison-Wesley, 2003.

[5] Dardenne, T., Van Lamsweerde, T., Fickas, S. Goal-directed Requirements Acquisition. Science of Computer Programming, (1993) Vol. 20.

[6] Oreizy, P., Gorlick, M.M., Taylor, M.M. et al. An Architecture-Based Approach to Self-Adaptive Software.