phpModeler – an approach to Reverse Engineering legacy Web applications

Josip Maras
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Croatia
josip.maras@fesb.hr

Ana Petričić
Faculty of Electrical Engineering and Computing
University of Zagreb, Croatia
ana.petricic@fer.hr

Maja Štula
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Croatia
maja.stula@fesb.hr

ABSTRACT

Web applications are complex systems that are in the core of many businesses. However, their development is, contrary to other domains, not characterized by rigorous software engineering methods. The consequence is that many web applications are poorly structured and are not adequately documented, which leads to difficult maintenance. One way for dealing with web application complexity is their modeling on a higher level of abstraction. In this paper we present phpModeler, a tool for reverse engineering legacy PHP web applications. It generates UML diagrams showing resources that each web page is using, web page's functions, dependencies it has on other web pages, and the flow of execution. Once the models have been created, phpModeler can analyze them and generate dependency models for each entity in every web page model. phpModeler can also be used to highlight the difference between page models – a feature that, when combined with an SVN repository shows the way how a web page has evolved over time. Tool usability has been tested on a case study application – iForestFire.

Categories and Subject Descriptors

D.2.7 [**Software engineering**]: Distribution, Maintenance, and Enhancement – *reverse engineering tool.*

Keywords

Reverse engineering, application maintenance, architecture recovery, legacy web applications, application evolution

1. INTRODUCTION

In the last two decades, web applications have made a tremendous leap forward: from simple static web pages developed only in HTML (HyperText Markup Language) to complex dynamic web applications developed using server-side technologies, e.g. PHP, ASP.NET, Java that extensively use web services, databases and client-side technologies (e.g. JavaScript, Flash, Silverlight). Because of the very short time-to-market, these applications are often developed in an un-disciplined way. For example, in PHP this often leads to whole web pages developed as a mixture of SQL (Structured Query Language), PHP, HTML, CSS (Cascading Style Sheets) and JavaScript – five different languages in a single file, mixing business, presentation and data access logic. In turn, this often means that considerable effort is needed to maintain these applications [1].

According to surveys the process of software maintenance consumes somewhere between 50% and 80% of project budget [2]. During system maintenance, in the debugging phase approximately 47%, and in the system evolution phase approximately 62% of time is spent on activities related to better understanding of software systems [3]. Therefore, in order to decrease the general project cost it is necessary to develop methods and tools that will facilitate system understanding.

Complex systems can be understood easier if they are modeled on a higher level of abstraction. Web applications have many dependencies between web pages (e.g. via links, frames, forms) and resources (e.g. JavaScript libraries, database tables). As the size of the web application grows, so does the number of their interdependencies and soon they become hard to manage. In order to tackle their complexity and to improve development and maintenance efficiency they need to be modeled, preferably with standard design or modeling languages, such as UML (Unified Modeling Language).

PHP and JavaScript are dynamic script languages in which variable types are not specified, and the type of the variable is determined at runtime (dynamic typing). For that reason, when defining functions there is no need to specify argument and return value types. Both languages also support variadic function calls and anonymous functions. As with any other dynamic languages many errors and inconsistencies are discovered only at runtime, and current IDEs (Integrated Development Environment) offer little support for change tracking (e.g. function signature change), code completion, and error reporting. Because of this, maintenance is difficult. For a web developer, it would be useful to know where certain entities are used, so that if any change happens its propagation can easily be tracked.

Software systems are ever evolving and it is necessary to be able to track changes between different versions of the same artifact. Usually these source code changes are tracked via different text comparison tools, but if there are models that represent the system on a higher level of abstraction than source code, it would be beneficial to track changes among them too.

In this paper we focus on legacy web applications and ways of improving their maintainability and understandability by using a reverse engineering (RE) process. RE is used for information extraction from software artifacts (primarily source code) and their transformation to easily understandable abstract representations [4]. We have developed a RE tool – *phpModeler* that *i*) analyses web application source code (PHP scripts, HTML pages, and JavaScript libraries) and generates models that can be used as a basis for architecture recovery; *ii*) generates dependency models that show interdependencies between web application elements and *iii*) visualizes web page evolution.

The remainder of this paper is organized as follows: Section 2 gives an overview of the web application architecture and techniques that can be used for their modeling. Section 3 presents our tool – *phpModeler* in greater detail, while Section 4 describes a case study used for tool validation. Section 5 gives an overview of currently available web page modeling tools. Finally, the conclusions are presented in Section 6.

2. WEB APPLICATION MODELING

When deciding about the modeling approach, it is critical to determine the correct level of abstraction and detail in order to have a relevant representation of a system from a certain viewpoint. In this section we give an outline of common web application architecture and describe some of the techniques used for their modeling.

2.1 Web application architecture overview

The basic infrastructure necessary for a web application to run contains at least the following elements: a web browser used by the user to send web page requests, a web server that processes those requests and generates responses and a network over which request/response pairs are transmitted.

A web application can be viewed as a collection of web pages where each web page or a group of web pages is responsible for certain application functionality. Web pages are hosted on a web server that responds to user requests with a combination of HTML code, client side scripts and embedded objects (e.g. Flash or Silverlight content). The user can interact with the web application by navigating to different web pages in the system or by submitting information to the server, usually in that way changing the business state of the application.

Important web application entities are the following:

- *Server page* is a dynamic web page residing on the server whose content is built on every user request. It usually contains server executables that access various resources such as files, data bases or web services.
- *Client page* is a HTML web page sent as a response to the user that contains data, presentation and even logic (via client side scripting or embedded objects).
- *HTML form* is a client page element representing a collection of user input fields and is used by the user to input data that will be sent to the web server for further processing.
- *Client script* is a script executed in the web browser that enables dynamic behavior on the client side.
- *Database table*

2.2 Web application models

There are already several existing methodologies for web application analysis, some of them referring only to static web pages, and most of them putting emphasis on connections between web pages of a web site in order to create a web site map.

One of approaches for analysis and design of web applications which was entirely based on UML notation, was presented by Koch and Kraus [5]. They extended the language with an UML profile in order to achieve sufficient level of expressiveness. In their methodology, they use separation of concerns, and perform separate steps for conceptual, navigational and presentational modeling with an emphasis on navigational structure modeling. In each of these steps they create up to two models and use several diagram types, with UML class diagram as the basis for conceptual and navigational models, and UML composition diagram and sequence diagram for presentational model. In the mean time, web applications have evolved a lot, and currently most of the pages are highly dynamical via the use of client and server side scripting. Therefore, the approach mentioned above is not sufficient for our target applications, as it focuses on static web pages, and navigation between them.

Ricca and Tonella [6] present a reverse engineering tool – ReWeb, which deals with complex structures of web applications, their evolution and restructuring. They model the web site's structure as a directed graph where each node represents a single HTML page. An edge connects two nodes if a link exists between two pages represented by those nodes. Their model is primarily used to model the client side of the web application.

Web Application Extension (WAE) proposed by Conallen [7], provides mechanisms to comprise all important elements of a web applications stated in Section 2.1. It encompasses both server-side resources and client-side scripts. He proposes an extension to UML in a form of an UML profile which is designed so that all web-specific components can be modeled. At the same time it provides a proper level of abstraction and detail suitable for designers, implementers, and architects of web applications.

The starting point for Conallen's model is the class diagram, and web applications are built out of elements represented by following stereotypes extending the "Class" element:

- $\ll serverPage \gg$ represents a server page. Server page global variables become attributes and server pages functions become operations.
- $\ll clientPage \gg$ represents a Client page displayed in the user's web browser. Global variables and functions declared in the page's script tags map to attributes and operations, respectively.
- $\ll form \gg$ corresponds to the HTML form element. The input fields of the HTML form are mapped to attributes. This element contains no operations.
- $\ll script\ library \gg$ represents a JavaScript function library. Global variables and functions are mapped to attributes and operations.

These elements can be connected using the following connectors extending the association connection: *link, build, submit, redirect, forward, object, include, script.*

We consider that Conallen's model is at the appropriate level of abstraction in order to show the Web application architecture. In our opinion it encompasses all important Web application elements, except the database table element (many web applications are data centric). So we augment his model with one element stereotype: $\ll database\ Table \gg$ that enables us to model the usage of the database tables.

PHP and JavaScript are dynamic languages and important information, such as the actual number and types of function arguments, or the actual function calls, can only be found out at runtime. Since we are representing the runtime behavior of the web application, we also use sequence diagrams [8] since they offer an easy to understand way of visualizing the application execution and communication between different web application entities.

3. PHPMODELER

phpModeler is an Eclipse plugin that facilitates modeling and web application architecture recovery. It has three main features: page modeling, dependency modeling and model comparison. For the implementation of these functionalities the following modules are used: *dynamic analyzer*, *code processor*, *dependency analyzer*, *UML generator* and *difference analyzer*.

Dynamic analyzer is a module that tracks the execution of PHP scripts in order to gather data about control flow and used data types.

Code processor is a page parsing module that parses PHP, HTML and JavaScript code. Its main functionality is to generate a web page model based on the adapted Conallen model.

Dependency analyzer is a module that analyses all generated models. For every web page entity (JavaScript library, database table, file, PHP library, web page), it finds all other entities dependent on it. For example, for every database table these dependency models show all web pages that access them and for function libraries they show web pages that use those functions. These dependency models simplify the process of change, because for each change the developer can easily see where the change propagates.

UML generator is a module used to generate UML diagrams in a standard XMI [8] format used by the majority of UML tools. As an input it uses models generated by the dynamic analyzer, code processor, and the dependency analyzer modules.

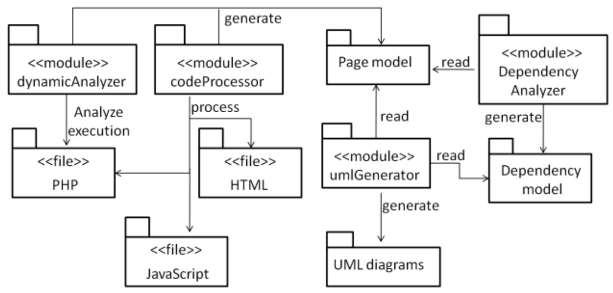

Figure 1 *phpModeler* – process of generating page models

The process of web page model generation (shown in Figure 1) goes as follows. In *phpModeler*, the user selects the target web page and executes it by using the *phpModeler* Eclipse plugin. The dynamic analyzer module then analyzes the execution of the web application by communicating with the Xdebug PHP debugger [10] and generates dynamic page models. When the execution flow enters a new PHP file, the *code processor* module statically analyzes the new page and generates static web page models. The *dependency analyzer* module then analyses the generated models and for each entity (server script, client page, database table) generates models showing other entities dependent on it. After the model generation phase, the *UML generator* module generates UML diagrams.

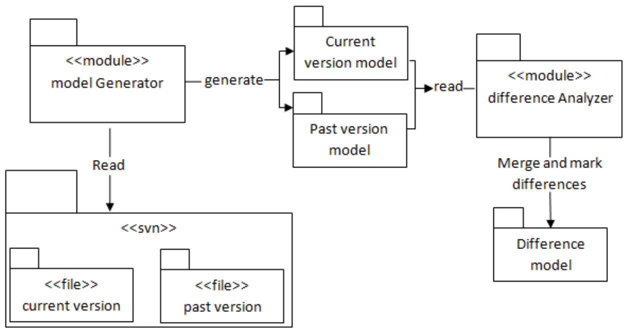

Figure 2 *phpModeler* – process of analyzing page evolution

phpModeler can also be used to highlight the differences between page models (shown in Figure 2). For that functionality it uses the *difference analyzer* module which is based on the algorithm presented in [10], which identifies common elements based on name and structure similarity. We have connected this module to a SVN (Subversion) repository and in that way gained the functionality of modeling page evolution. The differences between generated UML models are shown within the diagrams themselves, in a way similar to [11].

3.1 Generating page models

In *phpModeler*, model generation for individual pages is achieved with the *dynamic analyzer*, *code processor* and *UML Generator* modules. The process starts with the *dynamic analyzer* module which executes the selected page and tracks its execution while building a dynamic model of the web application execution. Based on gathered dynamic data the tool generates dynamic page models. For each page in which the execution flow enters, the *code processor* module separates the input file to three parts: server side PHP code, client side HTML code, and client side JavaScript code. Each of the parts is then separately analyzed and parsed using pattern matching and lexical analysis. Based on this analysis, augmented Conallen's models described in Section 2.2. are generated. Since all referenced pages do not have to be executed, the *code processor* analyzes all referenced PHP scripts, even the ones that will not be executed. The generated models are used as an input to the *UML generator* module which generates UML class diagrams in a standard XMI (XML Metadata Interchange) format that can easily be imported in most available UML tools.

3.1.1 Page model generation example

We illustrate the page model generation functionality with a simple login page example. Most web applications have a login form which enables user authentication to the system. Apart from great security issues (which are not in the focus of this example), these login forms are fairly simple. On the client side the user fills in his/her credentials which are then sent to the server page for processing. The server page checks the database for user data and if the data is valid, the user is granted access to the application. Otherwise the user is redirected to a new page where he/she can apply for registration.

```
<?php

require_once ("dal.php");

function checkLoginCredentials()
{
  if(isset($_REQUEST['username'])
  && isset($_REQUEST['password']))
  {
    $username = $_REQUEST['username'];
    $password = $_REQUEST['password'];

    if(userExists($username, $password))
    {
       header('Location: adminArea.php');
    }
    else
    {
     header('Location:loginInvalid.html');
    }
  }
}


checkLoginCredentials();
?>
```

Figure 3 PHP code for login page exaple

The difficult part of understanding web applications by browsing through their source code is that there is lot of “noise” in the source code. The content of the login.php web page is shown in Figure 3 (PHP part) and in Figure 4 (HTML part).

```
<html>
 <head>
  <link href="style.css" rel="stylesheet"
        type="text/css" />
  <title>phpModeler - login demo</title>
 </head>
 <body>
  <form action="login.php" method="post"
        class="loginForm" id="loginForm">
   <table>
    <tr><td>Username:</td>
     <td>
      <input type="text" id="username"/>
     </td></tr>
    <tr><td>Password:</td>
     <td>
      <input type="password" id="pass"/>
     </td></tr>
    <tr>
     <td colspan="2">
      <input type="submit" id="submit"
             value="Login"/>
     </td></tr>
   </table>
  </form>
 </body>
</html>
```

Figure 4 Login Web page example - HTML part

Layout sections, written in HTML, take up a lot of code, while providing little or no information about the web page functionality. It is important to emphasize that if the main focus of the application is the user interface, then it should be modeled separately.

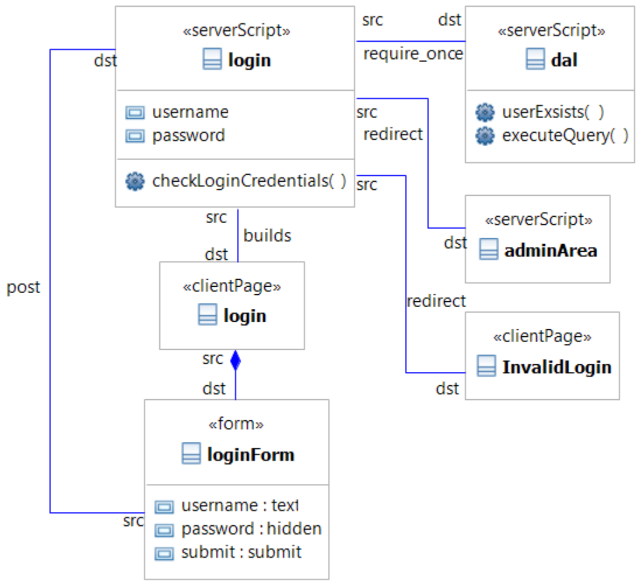

Figure 5 Page model generated from the login web page

Instead of dealing with complex code, we can use a model of that code. Figure 5 shows the login page model that was generated with *phpModeler* tool. From the architectural viewpoint, the important elements are the server script that checks the validity of submitted username and password and the client page containing the login form for data input. This web page also uses the functionality of a server script “dal.php” (Data Access Layer) and can redirect the user to either the “adminPanel.php” web page or “InvalidLogin.html”.

Figure 6 on the other hand shows a dynamic model of the page execution when the user has entered the correct login credentials.

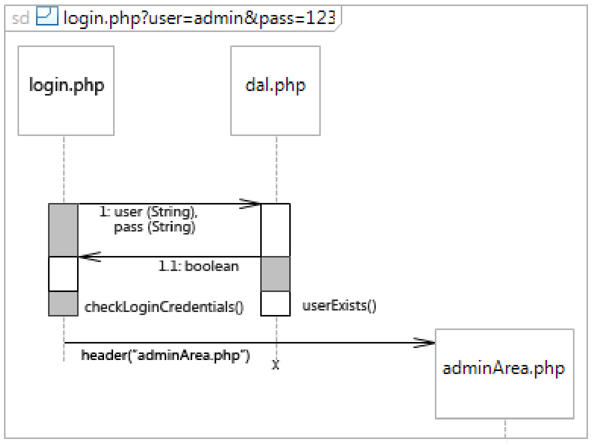

Figure 6 Sequence diagram representing the flow of application control when the user sends the login request

The dynamic model, shown on Figure 6 represents the code that was executed when the user has sent his/her user credentials when trying to login into the web application. The diagram shows the URL parameters accessed by the code, executed functions, and the types of function arguments and return values. The diagram is valid only for the analyzed execution flow.

3.2 Generating dependency models

One of the advantages of using PHP web applications is the ease of their deployment. In theory, it is necessary to transfer the web application files to the web server. Both PHP and JavaScript are script languages and there is no need to compile the application. However this has a drawback, since there is no compiler, many errors and inconsistencies are typically discovered at run time. This is especially inconvenient when making changes to the application (e.g. changing function signature, deleting functions, etc.) as it is hard to know where those changes propagate in case of large web applications. For that reason it is useful to have models that show dependencies between application elements.

In *phpModeler*, dependency models are created in the following way: the *code processor* module generates page models for every web page in the web application. Second, the *dependency analyzer* module goes through all generated models searching for unique entities (uniqueness is determined based on file URL for client scripts, server scripts and client pages and based on name for database tables). For each unique element the *dependency analyzer* determines its relationships to other entities and generates models showing those dependencies.

3.2.1 Dependency models example

We will illustrate the functionality of generating dependency models with an example that builds on the example described in Section 3.1.1. After the user has successfully logged in, he is redirected to the administration area where he has the ability to add new users. If we model this web application, using our *phpModeler* tool, we will get three models (the first two, representing the login page were already shown on Figure 5 and Figure 6). The third model, shown in Figure 7 represents the adminArea page.

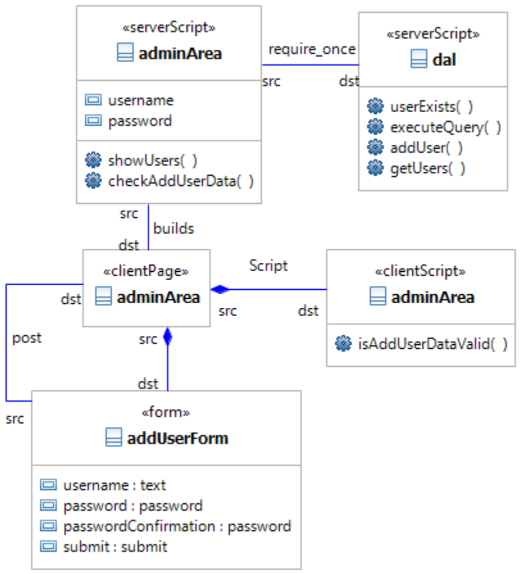

Figure 7 Page model of the adminArea web page

If we look closely at Figure 5 and Figure 7 we will notice that both models are using the same data access layer (server script "dal.php"). If any significant changes would be made to the server script "dal" those changes would most likely propagate to the "login.php" and "adminArea.php" server scripts. In this simple example it is easy to remember that function library "dal.php" is used in both server scripts, but imagine a complex web application with hundreds of scripts.

One of the dependency models generated by the *dependency analyzer* module is shown in Figure 8. The model clearly shows which server scripts use functionalities provided by the "dal.php" server script.

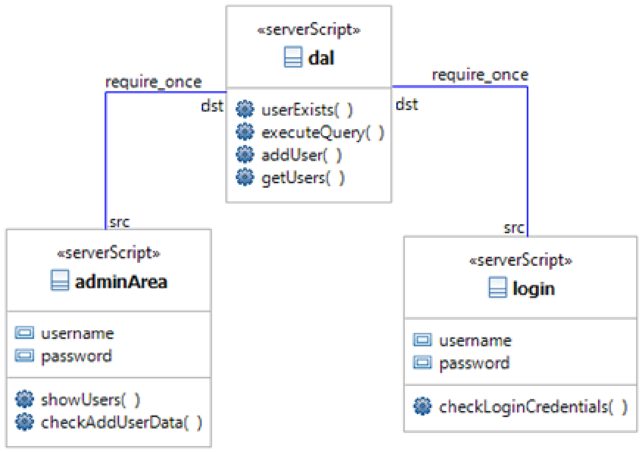

Figure 8 Page model representing dependency between server scripts and the server side function library dal.php

3.3 Modeling page evolution

Today, applications are evolving to fit customer needs. One of the problems that occur is tracking changes between various versions of the application.

As shown in Figure 2, in order to achieve this functionality, *phpModeler* uses the following modules:*i*) *svn Access* to access the source code from the SVN repository,*ii*) *code processor* and *dynamic analyzer* module that generate models based on the source code, and *iii*) *difference analyzer* that merges the input models (while clearly denoting elements belonging to both models, and the ones which are unique). Model differences are shown in a way similar to Fujaba [10] – parts detected as similar are shown in black, parts detected as belonging only to the older model are shown in green, while parts that can only be found in the newer model are shown in red.

The algorithm for merging models is based on type, name and structure similarity and is analogous to the one described in [9].

3.3.1 Modeling page evolution example

Let's build on the example shown in section 3.1.1. in the following way: the developers added a client script to stop the submission of empty username and password values (in order to relieve the server of unnecessary requests) and expand the functionalities of the "dal.php" and "admiArea.php" server scripts.

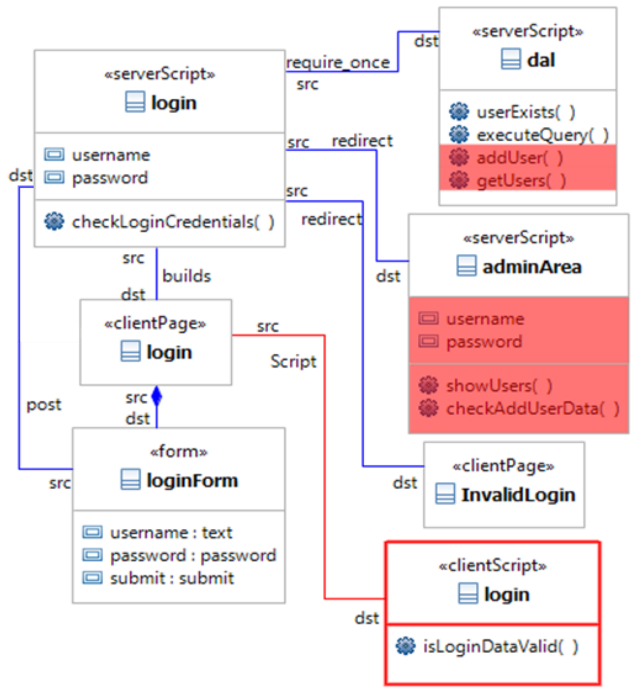

Figure 9 Example of a page evolution model

After the code has been checked in, a page evolution model can be generated. Figure 9 shows the evolution model of the "login.php" web page which clearly shows the newly added functionality (a new client script that checks the user data, and some changes in the "dal.php" and "adminArea.php" server scripts).

4. CASE STUDY

The usability of *phpModeler* is analyzed on a case study application – *iForestFire* [12]. *iForestFire* (web user interface is shown in Figure 10) is a web based system, developed on the University of Split, Croatia for early detection of forest fires. Forest fires are detected in incipient stage using advanced image processing and image analysis methods.

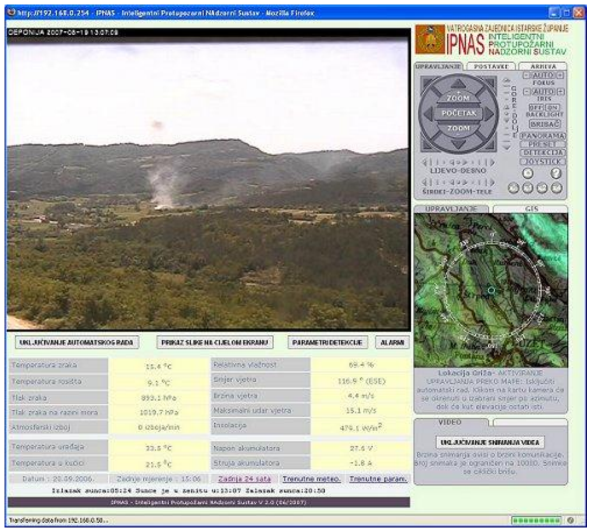

Figure 10 iForestFire user interface

iForestFire is composed of the following applications: (*i*)Fire detection application, which is an application running on the server. It is a standard multi-threaded C application that gathers data from distributed embedded devices (cameras and meteorological devices), analyzes that data and decides whether or not there is any sign of fire; (*ii*) Web application developed with PHP, JavaScript, HTML and CSS which acts as a user interface to the system. All of the systems functionalities and data access are done via a web browser; (*iii*) Database which is used for data storage and as means of communication between the fire detection application and the Web application.

In this case study we will concentrate on the Web application part which is composed of 402 scripts and pages in total. Source code analysis results are shown in Table 1 and Table 2.

	PHP	JavaScript	HTML
Total LOC	42534	11457	14255
Max LOC in one file	3725	1113	557
Average	105	28	35

Table 1 iForestFire code statistics

In order to gain a better understanding of *iForestFire,* Table 2 shows web application complexity in terms of SQL queries, function definitions, and function calls.

	SQL queries	Function definitions	Function calls
Total	348	1271	18773
Max	17	180	1561
Average	0.86	3	46

Table 2 iForestFire complexity statistics

We used *iForestFire* to benchmark *phpModeler* performance and usability. On the test computer (Intel Core2Duo at 2.16 GHz, 2 Gb RAM) the whole process of generating UML page models took 21 seconds, out of that 5 seconds have been used to parse individual pages and generate models, 3 seconds to generate dependency models and 13 seconds to generate UML diagrams from those models.

In the process, 402 page diagrams and 423 dependency diagrams (one for each server script, client script, and database table) have been generated. Figure 10 shows the complexity of generated page diagrams in terms of number of elements, their complexity (number of attributes and operations) and interconnections.

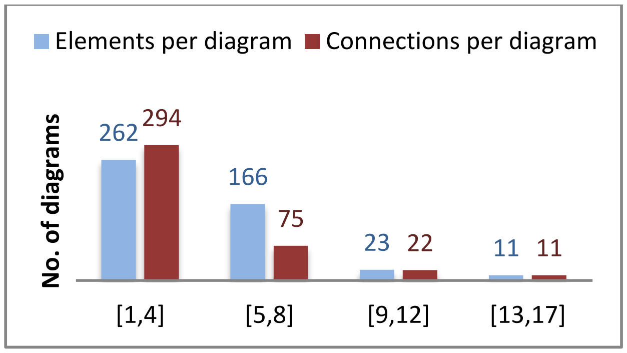

Figure 11 Complexity of page diagrams generated by *phpModeler*

As can be seen from Figure 10, on average, diagrams are of reasonable size (e.g. 262 diagrams have between 1 and 4 elements, and 294 diagrams have between 1 and 4 connections), although there are few diagrams that have too many elements and are too crowded. In that case we recommend manual partitioning of the diagram to a number of smaller, more specific ones.

Figure 11 shows that *iForestFire* is a complex system with a lot of interdependencies, but most of which are between several entities.

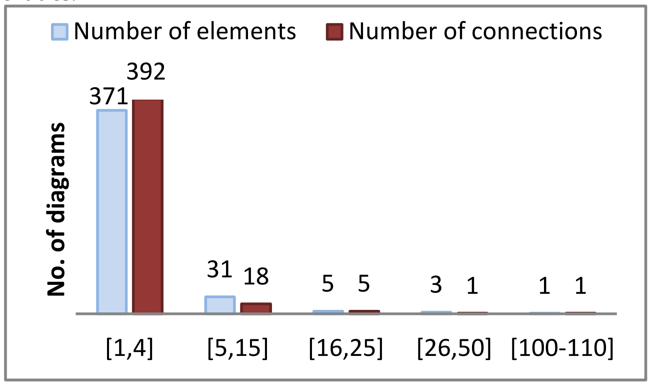

Figure 12 Complexity of dependency diagrams generated by phpModeler

We used *phpModeler* for the reverse engineering of *iForestFire* in order to recover the architecture and gain better understanding of the system. Currently, the system is being refactored in order to increase maintainability.

5. RELATED WORK

In the last decade several web application RE tools have been made, some of which are: WARE [13], WebUml [14], Enterprise Architect [15], Visual Paradigm for UML [16] and ReWeb [6].

Based on the type of generated models, these tools can be divided into three groups: (*i*) tools that generate standard UML class diagrams (i.e. Enterprise Architect and Visual Paradigm for UML), (*ii*) tools that generate web page models based on Conallen's web UML extensions (WARE and WebUml) and (*iii*) tools that generate models based on their own meta-models (e.g. ReWeb).

Enterprise Architect and Visual Paradigm for UML are commercial general modeling tools that perform web application RE, but in a sense that they generate standard UML diagrams from object-oriented PHP code. This is also the main difference, in comparison to phpModeler. Unlike these tools phpModeler uses Conallen's UML extensions, since models built using the extension describe web application architecture in much more expressive way. The second reason is that the great majority of legacy PHP web applications is written in procedural PHP code.

WARE tool is a RE tool developed as a part of the WARE approach (Web Application Reverse Engineering) that uses static, dynamic and behavioral analysis to generate class diagrams representing the architecture of web applications; sequence and collaboration diagrams to represent the dynamic model and use case diagrams to represent web application behavior.

WebUml is a RE tool that generates class diagrams, used to describe the structure and components of a web application and state diagrams that represent the behaviors and the navigational structure of the web application.

Both tools, WARE and WebUml, are missing analyses such as dependency analysis and page evolution analysis which exist in *phpModeler*. On the other hand, the WARE tool, except for static and dynamic analysis also provides the ability to perform the behavioral analysis of web applications.

Ricca and Tonella [6], developed ReWeb tool in order to deal with complex structures of web applications, their evolution and restructuring. ReTool uses reverse engineering approach to model web site's structure with a graph and allows applying several known analyses (including flow analyses, traversal algorithms, and pattern matching). The tool also provides analysis of site's history and evolution. They use their own modeling language and GUI to represent a model of the site. At the moment, this tool supports only analysis of static web pages and deals mostly with client-side of a web site, i.e. it lacks examination of resources available and used at server-side of the application.

6. CONCLUSION

Separation of concerns is a widely accepted practice that increases understandability and eases program maintenance. A disadvantage of the open-source web language chain (PHP, SQL, HTML, CSS, JavaScript) is that it does not enforce separation of concerns. It is possible to have up to five different languages in the same file, each dealing with separate concerns: HTML and CSS for presentation, JavaScript for client side logic, PHP mostly for business logic and SQL for data access. Apart from mixing concerns, there are parts of the files that deal with resources running on the server (PHP, SQL) and parts that deal with resources running on the client (HTML, CSS, JavaScript). It is understandable that this can, at times, be overwhelming. Web applications are characterized by a large number of interdependencies between web application entities that are often difficult to track. One way of coping with web application complexity is their modeling.

In this paper we have presented a tool *phpModeler* that we have developed and that facilitates reverse engineering and maintenance of legacy web applications. *phpModeler* analyses web application execution and source code and generates models that can be used: a) as a basis for architecture recovery and better system understanding; b) to facilitate system maintenance by showing interrelationships between web application elements; and c) as a way to track web page evolution.

We have tested the tool in a case study while reverse engineering an application – *iForestFire*. The analysis has shown that *phpModeler* is usable in reverse engineering legacy web applications. Model generation takes a reasonable amount of time (21 seconds) so models can be regenerated when necessary. The analysis has also shown that in a smaller number of cases (see Figures 10 and Figure 11) models are somewhat difficult to use due to their size.

For future work we plan to extend *phpModeler* with the ability to track dependencies between a client script and a server resource (AJAX). Also, the analysis has shown that the usability of the generated models drops when dealing with large models. It will be necessary to provide a way to automatically partition large diagrams into more manageable ones. There is also an issue with small diagrams that could probably be merged. Finally, the usability of the generated models will have to be tested.

7. ACKNOWLEDGMENTS

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and the Unity Through Knowledge Fund supported by Croatian Government and the World Bank via the DICES project.

8. REFERENCES

[1] **Mikkonen T., Taivalsaari A.** *Web Applications - Spaghetti Code for the 21st Century.*; Software Engineering Research, Management and Applications - SERA, 2008.

[2] **Boehm, Barry W.** *Software engineering economics.* 2002.

[3] **Nelson, Michael L.** A Survey of Reverse Engineering and Program Comprehension. *Software Engineering Survey.* 1996.

[4] **Chikofsky E, Cross J.** Reverse Engineering and Design Recovery - a Taxonomy. *IEEE Software.* January, 1990.

[5] **Koch, N., Kraus,A.** *The expressive power of UML-based engineering.* In Second International Workshop on Web Oriented Software Technology (CYTED), 2002.

[6] **Ricca F., Tonella P.** Understanding and Restructuring Web Sites with ReWeb. *IEEE MultiMedia.* Volume 8 Issue 2, 2001.

[7] **Conallen J.** Modeling Web application architecture with UML. *Communications of the ACM.* Volume 42, Issue 10, 1999.

[8] XMI Mapping Specification. *XMI.* [20. 9 2009.] http://www.omg.org/technology/documents/formal/xmi.htm.

[9] **Xing Z.. Stroulia E.** *UMLDiff: an algorithm for object-oriented design differencing.* Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering, 2005.

[10] **Niere J.** *Visualizing differences of UML diagrams with Fujaba.* In Proceedings of the 2nd Fujaba days, 2004.

[11] iForestFire [20.9.2009.] http://ipnas.fesb.hr/

[12] **Di Lucca G., Fasolino A.R., and Tramontana P.** Reverse engineering Web applications: the WARE approach. *Journal of Software Maintenance: Research and Practice.* Volume 16 Issue 1-1, 2004

[13] **Bellettini C., Marchetto A., Trentini A.** *WebUml: reverse engineering of web applications.* Proceedings of the 2004 ACM symposium on Applied computing, 2004.

[14] Enterprise Architect [20.9.2009.] www.sparxsystems.com.au/platforms/php_uml.html

[15] Visual Paradigm for UML *Visual Paradigm for UML* [20.9.2009.] www.visual-paradigm.com