Reusing Web Application User-Interface Controls

Josip Maras[1], Maja Štula[1] and Jan Carlson[2]

[1] University of Split, Croatia
[2] Mälardalen Real-Time Research Center, Mälardalen University, Västerås, Sweden
{josip.maras, maja.stula}@fesb.hr, jan.carlson@mdh.se

Abstract. Highly interactive web applications that offer user experience and responsiveness of desktop applications are becoming increasingly popular. They are often composed out of visually distinctive user-interface (UI) elements that encapsulate a certain behavior – the so called UI controls. Similar controls are often used in a large number of web pages, and facilitating their reuse could offer considerable benefits. Unfortunately, because of a very short time-to-market, and a fast pace of technology development, preparing controls for reuse is usually not a primary concern. In this paper we present a semi-automatic method, and the accompanying tool, for extracting and reusing web controls. The developer selects the control and performs a series of interactions that represent the behavior he/she wishes to reuse. In the background, the execution is analyzed and all code and resources necessary for the standalone functioning of the control are extracted. Optionally, the user can immediately reuse the extracted control by automatically embedding it in an already existing page.

1 Introduction

In the last two decades web applications have made a tremendous leap forward: from simple static web pages developed only in HTML to complex dynamic web applications developed using server-side technologies that extensively use web services, databases and client-side technologies. Initially the term “dynamic web application” was mostly used to describe that web page content was dynamically generated on the server side. More recently (2005 and onwards) with the wide spread adoption of AJAX and faster web browsers, web applications are also increasingly dynamic on the client side (e.g. applications such as Gmail, Facebook, etc.). Web developers now routinely use sophisticated scripting languages and other active client-side technologies to provide users with rich experiences that approximate the performance of desktop applications [23].

Web application user-interface (UI) is often composed of distinctive UI elements, the so called UI controls. Similar controls are often used in different web applications and facilitating their reuse could lead to faster development. Unfortunately, the web application development domain is exposed to a very fast pace of technology development and short time-to-market. This means that preparing code for reuse is often not a primary concern. So, when developers encounter problems that have already been solved in the past, rather then re-inventing the wheel, or spending time componentizing the already available solution (which is sometimes not preferable [8]) they perform reuse tasks [2]. Reusing source code that was not designed in a reusable fashion is known by different synonyms: copy-and-paste reuse [10], code scavenging [9] and, more recently, pragmatic-reuse [5]. Pragmatic reuse treats the system in a white-box fashion and involves extracting functionality from an existing system and reusing it within another system. The client-side web development domain is particularly pervious to white-box reuse, since code is transfered and executed in the browser. White-box reuse tasks are complex and error-prone, partly because the goal is to extract the minimum amount of code necessary for the desired functionality [5].

Reuse of client-side web UI controls is particularly difficult since there is no trivial mapping between source code and the page displayed in the browser; code is usually scattered between several files and code responsible for the desired functionality is often intermixed with code irrelevant for the reuse task. In order to reuse the chosen control, the developer has to locate the code and the resources defining the UI control. Next, the developer has to download the selected files, remove the unnecessary code and resources, and adjust for the now changed location. This is a time-consuming process.

The structure of a web page is defined by HTML code, the presentation by CSS (Cascading Style Sheets) code, and the behavior by JavaScript code. In addition, a web page usually contains various resources such as images or fonts. The interplay of these four basic elements produces the end result displayed in the user's web browser. Visually and behaviorally a web page can be viewed as a collection of UI controls, where each control is defined by a combination of HTML, CSS, JavaScript and resources (images, videos, fonts, etc.) that are intermixed with code and resources defining other parts of the web page.

In this paper we present a novel approach to semi-automatic extraction of reusable client-side controls. The developer selects the desired UI control on the web page and interacts with it, demonstrating the behavior that he/she wishes to reuse. In the background, the tool that we have developed – Firecrow [12] tracks all executed code, applied CSS styles and used resources, in order to locate the code and resources that are vital for the stand-alone functioning of the chosen UI control. In the end, all essential code and resources are extracted, all necessary adjustments are made and the control is packed as a reuse-friendly web page. Optionally, the developer can choose to embed the extracted UI control directly into a specific place of an already existing web page.

2 Extracting and reusing UI controls

In order to reuse a web UI control, we have to extract all that is necessary for the control to be visually and functionally autonomous. This means extracting all

HTML, CSS, JavaScript and resources that are used in the visual presentation and the desired behavior of the control.

The process can be separated into three phases: 1) Interaction recording, 2) Resource extraction, and 3) UI control reuse (Figure 1).

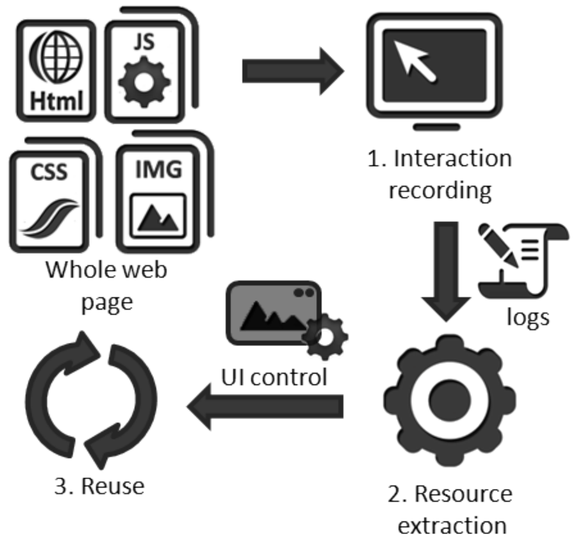


Fig. 1: Extracting and reusing UI controls in Web applications

The first step of the *Interaction recording* phase is to select the HTML node that defines the chosen UI control. Next, the user performs a series of interactions that represent the behavior of the control. The purpose of this phase is to gather a log of all resources required for replicating visual and behavioral aspects of the control.

The life-time of the web application client-side can be divided into two steps: *i)* page initialization – where the browser parses the web page code and builds the DOM (Document Object Model) [19] of the page, and *ii)* event-handling. All user interactions are handled in the event-handling phase by modifying the DOM built in the initialization phase. This means that in the beginning of the recording phase, before the user has started interacting with the UI control, a "snapshot" of the initial state of the control has to be made. This is done by logging all executed initialization code, all CSS styles and all resources used to initially define the control. Later, all code executed during the recorded interaction is also logged, together with all dynamically applied CSS styles and images.

When the user chooses to end the recording, the process enters the *Resource extraction* phase, where code models for all code files (HTML, CSS, and JavaScript) are build. Based on those models and logs gathered during the recording phase, the code necessary for replicating the visuals and the demonstrated behavior is extracted.

After the extraction phase is finished, the user can choose to enter the *Reuse* phase and automatically embed the extracted control in an existing web page, either by replacing, or by embedding it inside an already existing node. In this way a full cycle is completed: from seeing the potential for reuse, through extracting the desired control, all the way to actually reusing it and gaining new functionalities in the target web page. Each step of the process is described in more detail in the following sections.

The approach will be illustrated with an example of extracting and reusing a UI control from a web page. Figure 2 shows a web page developed in a previous project, and the control (marked with a dashed frame) that was selected for reuse. The control displays different images (marked with 1 in Figure 2) and captions (mark 2). Currently displayed items can be changed by clicking on bullets (mark 3), and the control replaces items with a fade-out, fade-in effect.

3 Interaction recording

The purpose of the interaction recording phase is to locate all code and resources necessary for stand-alone functioning of the target UI control. In order to do that, the user has to first select the chosen control. However, the control does not exist as a separate entity in the web page. So, in our approach the control is selected through the corresponding HTML node defining the UI control. This is done with Firebug's DOM (Document Object Model) explorer in which the user can go through the DOM of the page.

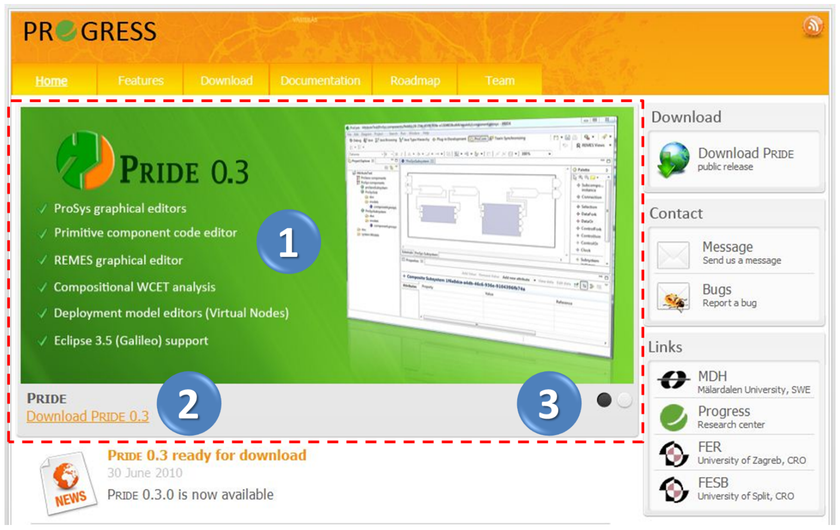


Fig. 2: The web page of the UI control chosen for extraction

When the user initiates the recording phase, the page is reloaded, subscriptions to the DOM mutation events [20] are registered, and the initial state of the UI control is logged. The initial state is composed of code executed while initializing the control, and styles and resources that initially define the control. Generally, all executed code is logged by communicating with the JavaScript debugger service, which provides hooks (or events) that activate on each execution and give information about the currently executed source code lines. In order to obtain the styles and resources that initially define the UI control, the DOM of the control is traversed and all CSS styles and resources applied and used in the control are logged. With this, a log of resources that initially define the control is obtained.

After the UI control is fully loaded, the modifications of the control are caused by user interactions and/or timing events. The executed code is again logged by communicating with the JavaScript debugger service, while any dynamic changes in styles and resources are logged by handling DOM mutation events. Using this approach we are able to locate all code and resources that define the control: *i)* HTML code, because the user directly selects the HTML node defining the control; *ii)* JavaScript code, because by communicating with the JavaScript debugger service we are able to log all executed lines; *iii)* CSS code; and *iv)* resources, because styles and resources applied to the control during the the whole course of the execution are logged.

4 Extraction

Once the recording of interactions is complete, the process goes into the extraction phase. As input, the extraction process receives all data gathered during the recording phase: the HTML code of the whole page; the xPath [21] expression uniquely defining the node designated for extraction; a collection of used CSS styles and resources; and for each JavaScript code file a list of executed lines. Based on this data, JavaScript files, CSS files, and resources are separately analyzed and cleansed of unnecessary elements. Since HTML documents can have JavaScript and CSS code embedded directly in them, these parts of the file are handled in the same way as the rest of the JavaScript and CSS code.

4.1 Extracting JavaScript code

The goal of JavaScript code extraction is to produce a minimal code that is syntactically correct, and semantically consistent with the recorded execution. A naive implementation, where one would simply split the file into lines, and then filter out the non-executed ones can only function in rare cases of specially formated code, but since real-world code can be arbitrarily formated this is not an option. Consider the example given in Listing 1.1.

```
/*1*/var a = getNum();
/*2*/if(a%2==0) {doEven();} else
/*3*/{
```

```
/*4*/    doOdd();
/*5*/}
/*6*/doOtherStuff();
```

Listing 1.1: Why naive line removal does not work

In this example, if the *getNum* function returns an even number, lines 1, 2 and 6 will be executed. If we do simple line removal, we would end up with code presented in Listing 1.2 which is not semantically equivalent to the original program. The *doOtherStuff* function would only be called if *a* is odd. And in the original code (Listing 1.1) it is called regardless if *a* is even or odd.

```
/*1*/var a = getNum();
/*2*/if(a%2==0) {doEven();} else
/*3*/doOtherStuff();
```

Listing 1.2: Naive line removal result

In order to tackle this problem, we build a model of the JavaScript source code. This model is produced by a JavaScript parser that we have developed. The parser is developed with ANTLR [7] according to the specification given in [6]. Listing 1.3 gives a code example and Listing 1.4 gives the model generated from the code example.

```
/*1*/function double(x)
/*2*/{
/*3*/    return 2*x;
/*4*/}
```

Listing 1.3: JavaScript code example

As can be seen in Listing 1.4, the model represents a simplified abstract syntax tree of the given source code. Although it is a lot more verbose then the source code from which it is derived from, the model provides all information about the position and type of used constructs.

```
{"srcElems":[{
  "type":"funcDecl", "strtLn":1, "strtCh":0,
  "name":"double", "params": ["x"],
  "body": { "type":"funcBody",
    "strtLn":2, "strtCh":0, "srcElems":[{
    "type":"rtrnStatemnt", "strtLn":3,"strtCh":1,
    "expr": {"type":"mulExpr", "strtLn":3,
      "strtCh":8, "exprs":[
        {"type":"numLit", "strtLn":3,
        "strtCh":8,"value":2,
        "endLn":3,"endCh":8},
        {"type":"mulExprItem", "strtLn":3,
        "strtCh":9, "operator":"*", "item":{
        "type":"ident", "strtLn":3,"strtCh":10,
        "id":"x","endLn":3,"endCh":10},
      "endLn":3,"endCh":10
    }], "endLn":3, "endCh":10
  }, "endLn":3,"endChar":11
  }], "endLn":"4","endCh":"0"
},  "endLn":4,"endCh":0
}]}
```

Listing 1.4: JavaScript model example

By traversing a source code model (e.g. Listing 1.4) we can remove all code constructs contained in the not-executed lines, while keeping the semantical correctness.

4.2 Extracting CSS code

Part of the data gathered during the recording phase are styles that get applied to the chosen HTML node. Similarly to the process of building the JavaScript code model, we have also developed a CSS parser that builds a model of the code. Listing 1.5 gives a code example, while Listing 1.6 presents the model of the code given in the example.

```
@import url('/css/style.css');
body { font-family: tahome; background: white; }
```

Listing 1.5: CSS code example

Based on the used CSS styles that were gathered during the recording phase, the code model of the whole web application CSS code is traversed and only code comprised of used styles is generated.

```
{ "imports": [ { "url": "/css/style.css" }],
  "body": { "items": [{
     "type": "ruleSet",
     "selectors": ["#mainContainer"],
     "declarations": [
       {"prop": "font-family", "val": "tahoma"},
       {"prop": "background","val": "white"}
     ]}]}}
```

Listing 1.6: CSS model example

The CSS code model is especially useful in the reuse phase, where the CSS code of the control is merged with the CSS code of the host web page. There we use both the CSS model of the control and the CSS model of the host page to detect naming conflicts, and in the case of conflicting styles offer the possibility of merging.

4.3 Extracting HTML code

In order to extract the HTML code of the chosen UI control, we have to be able to locate the code responsible for defining the control. In this case we build a standard model of the given web page – DOM [19] – with an open source HTML parser [14].

The visual layout of a certain HTML node is not only influenced by the HTML code of that node, but also by the type and presentation of its ancestors. For this reason, when traversing the DOM tree, the ancestors of the UI control are kept, but all siblings (both from the chosen HTML node, and from each of its ancestors) are removed. The resulting HTML DOM tree can contain image nodes and references to styles and scripts. Since the location of the HTML node will change (the code will be transfered from a web server to the user chosen location), the in-code references to those files also have to be changed. This is a time-consuming task, so it is handled automatically.

4.4 Extracting resources

During the recording phase, all resources that were at some point used by the UI control are tracked. In the final step of the extraction phase all resources are automatically downloaded to the target location.

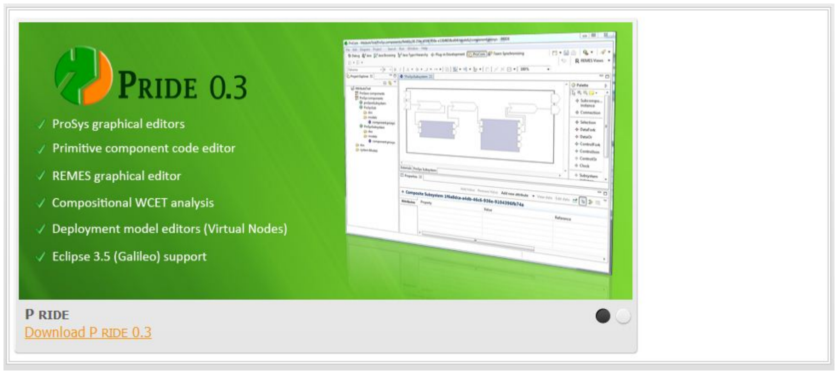

PRIDE 0.3
✓ ProSys graphical editors
✓ Primitive component code editor
✓ REMES graphical editor
✓ Compositional WCET analysis
✓ Deployment model editors (Virtual Nodes)
✓ Eclipse 3.5 (Galileo) support
P RIDE
Download P RIDE 0.3

Fig. 3: Extracted UI control from the web page shown in Figure 2

The result of the extraction process is an HTML document that contains the HTML code defining the selected node, and includes all necessary style sheets, scripts, and resources that were identified as necessary in the extraction phase. Figure 3 shows the results of extracting the UI control from the web page shown in Figure 2.

5 Reuse

Once the UI control has been extracted it can be embedded into an existing "host" web page. The user selects the host web page, a referent node in the host page, and the insertion type. The control can be inserted so that it replaces the referent node; or it can be inserted into, before, or after the referent node. In order to enable reuse, resources defining the extracted UI control and resources defining the host web page have to be merged. Naturally, this can lead to conflicts such as CSS style overriding, duplicate JavaScript libraries, name clashes, etc., that have to be tackled.

5.1 Detecting conflicts

Currently, the best reuse results are achieved if the extracted UI control is reused at an early stage of the new web page development – in a state where the host web page is not complex and does not include large CSS or JavaScript code bases. Even in that case, conflicts when merging HTML, CSS, JavaScript can occur.

Detecting HTML conflicts – When merging the HTML code of the control with the HTML code of the host web page the following conflicts can occur: inclusion of duplicate JavaScript libraries, occurrence of HTML nodes with the same id, and clashing HTML node classes. Before the merging is done, the control DOM and the host page DOM are analyzed, and if any conflicts are detected the user is notified.

Detecting CSS conflicts – In CSS, conflicts can arise from clashes based on HTML node IDs, node classes and node types. The first two cases (IDs and classes) are handled by detecting HTML conflicts, but the third case has to be handled separately. If there are CSS rules clashing because of node types, then we notify the user and offer the possibility to either accept one of the rules, or to merge them into one CSS rule.

Detecting JavaScript conflicts – Web applications often use JavaScript libraries (e.g. jQuery, Prototype, Mootools, etc.), and a situation might happen in which the host page uses a full library, and the control uses a subset of the code from the same library, or vice versa. In that case we handle the conflict by including the full library. Since JavaScript is a dynamic language, more advanced analysis (that is beyond the scope of this paper) is needed in order to detect conflicts on variable or function level.

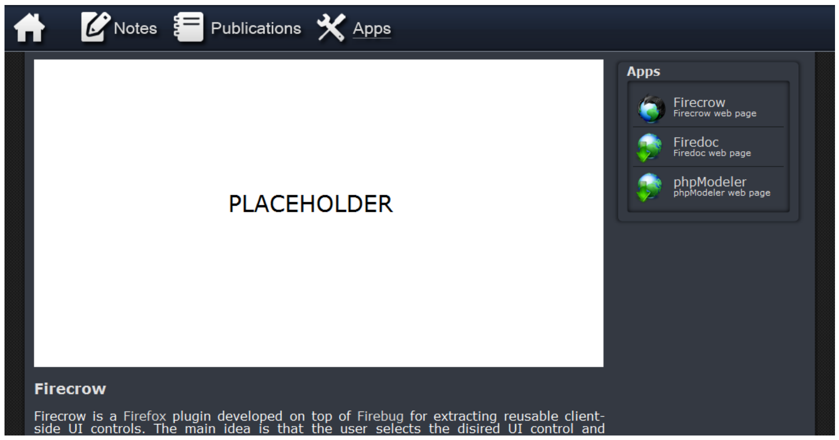
5.2 Example

We will demonstrate the process by reusing the UI control, described in Section II, in a test web page shown in Figure 4. When extracting the control we provide the path to the test host web page, and the xPath expression of the placeholder node which we want to replace. Then, when the control is extracted all control resources are merged with the already existing resources of the host page.

The result is shown in Figure 5. A video showing the whole process of reuse, can be found at the Firecrow web page[3].

In order to complete the reuse and adapt the extracted UI control to the new context, the developer has to manually replace some of the extracted resources.

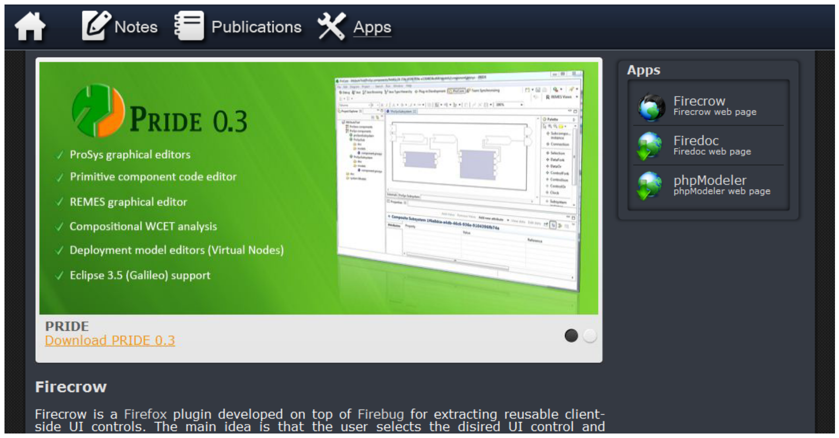
[3] http://www.fesb.hr/~jomaras/Firecrow



Firecrow

Firecrow is a Firefox plugin developed on top of Firebug for extracting reusable client-side UI controls. The main idea is that the user selects the disired UI control and

Fig. 4: The host web page with a placeholder for the extracted control



Firecrow

Firecrow is a Firefox plugin developed on top of Firebug for extracting reusable client-side UI controls. The main idea is that the user selects the disired UI control and

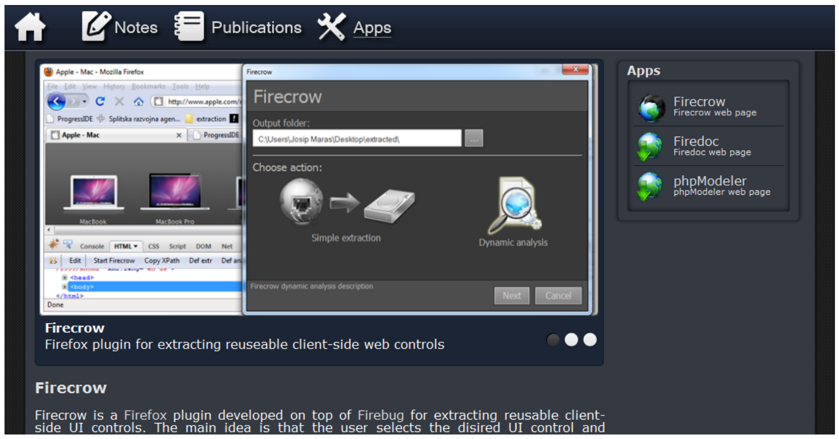
Fig. 5: The result of reusing UI control in the host web page

For example, in this case we have replaced background images, changed text captions, and added another option (the result is shown in Figure 6).

6 Tool

The whole process is currently supported by the Firecrow tool [12], which is an extension for the Firebug[4] web debugger. Currently, the tool can be used from the Firefox web browser, but it can be ported to any other web browser that provides communication with a JavaScript debugger, and a DOM explorer (e.g. IE, Chrome, Opera). Only the interaction recording phase is browser dependent;

[4] http://getfirebug.com



Firecrow
Firefox plugin for extracting reuseable client-side web controls

Firecrow

Firecrow is a Firefox plugin developed on top of Firebug for extracting reusable client-side UI controls. The main idea is that the user selects the disired UI control and

Fig. 6: The result of adapting the extracted UI control

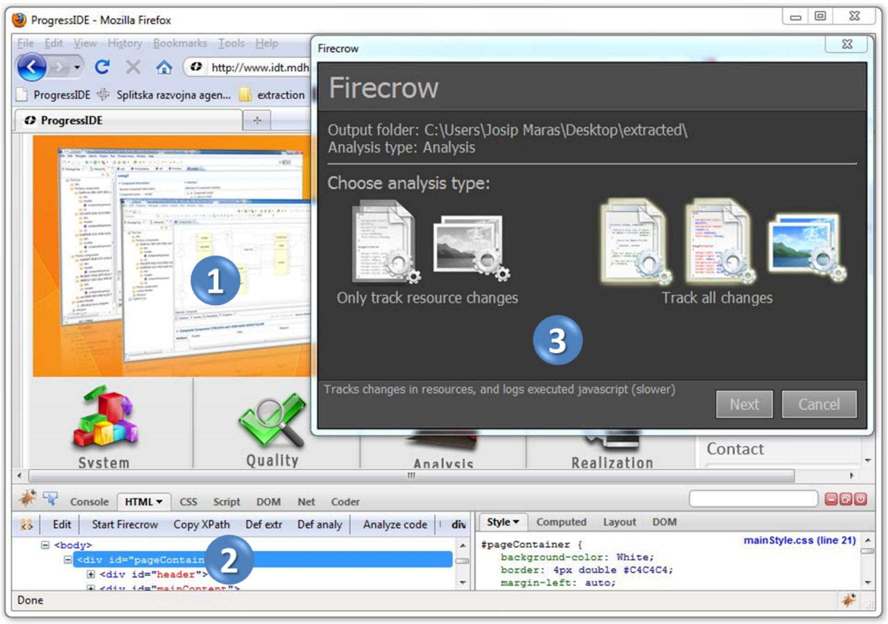
building source models, extracting code, downloading resources, merging code and resources are functionalities that are all encapsulated in a Java library that can be called from any browser on any operating system. The whole source of the program can be downloaded from http://www.fesb.hr/~jomaras/?id=Firecrow.

The Firecrow UI is shown in Figure 7. Mark 1 shows the web page chosen for extraction, mark 2 Firebug's HTML panel used for selecting controls, and mark 3 Firecrow's extraction and reuse wizard.

7 Evaluation and lessons learned

We have evaluated our approach by extracting UI controls from thirty-five web pages: twenty have been selected from the top 200 most visited web pages in the world [1] (including Google, Facebook, Twitter, and Apple), ten have been selected because they have visually interesting controls, and five from projects in which we have been involved earlier. The full list of tested web pages is available on the Firecrow web page and is updated regularly as more web pages are tested. Since the notion of what would be considered a user control can vary, each web page is accompanied with a screen-shot marking the extracted user controls.

In this evaluation we have learned more about the advantages and shortcomings of the approach and the accompanying tool Firecrow. From thirty-five web pages we were able to successfully extract 133 user controls. However, the extraction of eleven user controls failed. Mostly, the problems were with HTML parsing, heavily modified DOM, and JavaScript trying to access elements that were deleted in the extraction process. Problems with parsing HTML arise from the fact that browsers fix invalid HTML code, and since there is no standard way of handling HTML errors, each browser handles this problem in a specific way. We use an open source HTML parser, and the DOM produced by this parser does not always match the DOM built by a browser. Since we are identifying


ProgressIDE - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.idt.mdh
ProgressIDE
Splitska razvojna agen...
extraction
Firecrow
Choose analysis type:
Only track resource changes
Track all changes
Next
Cancel
1
2
3
System
Quality
Analysis
Realization
Contact
Console HTML CSS Script DOM Net Coder
Edit Start Firecrow Copy XPath Def extr Def analy Analyze code
Style Computed Layout DOM
mainStyle.css (line 21)
Done

Output folder: C:\Users\Josip Maras\Desktop\extracted\
Analysis type: Analysis

Tracks changes in resources, and logs executed javascript (slower)

```
<body>
  <div id="pageContain
    <div id="header">
```

```
#pageContainer {
    background-color: White;
    border: 4px double #C4C4C4;
    margin-left: auto;
```

Fig. 7: Firecrow user interface

nodes by xPath expressions which define the nodes' position in the DOM, in some rare cases, there can be a mismatch between the position of the node in the browser DOM and in the DOM built by the parser. Because of this, the extraction process can not locate the node, and the extraction fails. Also, a node can not be located if the chosen node was dynamically created and does not exist in the original HTML defining the web page.

In the current approach, we extract all code executed while loading the page and while executing control-specific behavior. This means that we will usually end up with more code than is actually needed for the user-control behavior (e.g. initialization code for other controls can be executed). Some of that code can try to access web page elements that are deleted in the extraction process, which in turn can cause JavaScript errors in the extracted web page. Detecting these errors with the Firebug web debugger, is however fairly straightforward.

8 Related work

There exist a number of approaches, environments and tools designed to support reuse. In the web application domain these include HunterGatherer [15], Internet Scrapbook [16], HTMLviewPad [17], and ReWeb [18]; while in the more general domain of reusing Java code there is G&P (Gilligan and Procrustes) [4].

HunterGatherer [15] and Internet Scrapbook [16] allow users to collect components from within Web pages, and to collect components from different Web pages into a newly created page. But since these approaches were developed in 1990's and early 2000, when web page development was not so dynamic on the client side, with the term "component" they refer to information components – most usually text paragraphs. These approaches are mostly used to create scrapbooks of data gathered from different web pages, and not to reuse certain functionality and visual elements of web pages.

Tanaka et al. [17] describe an interesting approach to clipping and reusing fragments of Web pages in order to compose new applications. They only target HTML elements, specifically HTML forms (no attention to CSS or JavaScript is given in their examples), and how to reroute data entered in the form to orginal servers that process the request. The applications created in this way are not deployable as standard web pages, but are executed within their tool – HTMLviewPad. This fact, along with not explicitly targeting the whole technology chain (HTML, CSS, and JavaScript) is the biggest difference between our approaches.

Our work is also related to program slicing [22], where by starting from a subset of a program's behavior, the program is reduced to a minimal form which still produces that behavior. In a sense our approach can be viewed as web page slicing with the goal of reducing the whole page (along with its code and resources) to a form in which only the visuals and the behavior of the selected user control are maintained. In the web engineering domain Tonella and Ricca [18] define web application slicing as a process which results in a portion of the web application which exhibits the same behavior as the initial web application in terms of information of interest displayed to the user. In the same work they present a technique for web application slicing in the presence of dynamic code generation where they show how to build a system dependency graph for web applications. This work is mostly dealing with reusing HTML and server-side code.

In the more general domain of Java applications, G&P [4] is a reuse environment composed of two tools: Gilligan and Procrustes, that facilitates pragmatic reuse tasks. Gilligan allows the developer to investigate dependencies from a desired functionality and to construct a plan about their reuse, while Procrustes automatically extracts the relevant code from the originating system, transforms it to minimize the compilation errors and inserts it into the developer's system. This work was further expanded [3] with the possibility to automatically recommend elements to be reused based on their structural relevance and cost-of-reuse.

There are also two tools that facilitate the understanding of dynamic web page behavior: Script InSight [11] and FireCrystal [13]. Script InSight helps to relate the elements in the browser with the lower-level JavaScript syntax. It uses the information gathered during the script's execution to build a dynamic, context-sensitive, control-flow model that provides feedback to the developers as a summary of tracing information. FireCrystal [13] is a standalone Firefox plug-in that facilitates the understanding of interactive behaviors in dynamic web pages. FireCrystal performs this functionality by recording interactions and logging information about DOM changes, user input events, and JavaScript executions.

After the recording phase is over, the user can use an execution time-line to see the code that is of interest for the particular behavior. Compared to our approach they make no attempts to extract the analyzed code.

9 Conclusion and future work

In this paper we have presented a novel approach and the accompanying tool for extracting and reusing client-side user interface controls in web applications. The process starts with the developer selecting the user control and demonstrating the behavior that he/she wishes to reuse. In the background, the executed code and used resources are analyzed. We have shown how, based on that analysis, a subset of the whole application code and resources necessary for the independent functioning of the control can be determined. We have evaluated the approach on thirty-five web applications and found that in a majority of cases the process is able to extract stand-alone UI controls.

During the evaluation we have noticed that some pages make extensive modifications of the original web page DOM, up to the point that the node defining the UI control does not exist in the original HTML code. Also, even though some statements get executed while interacting with the control, they are not necessarily required for the functioning of the web control. So, for future work, we plan to develop a method for tracking DOM changes, which should enable us to locate nodes in the original HTML code required for the creation of the node defining the UI control. We also plan to extend the analysis of executed code, so that only code statements that influence the behavior of the control are extracted.

The web page displayed in the browser is usually the result of server-side program execution, so we plan to extend the approach with server-side code analysis in order to facilitate code reuse on the server-side.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the strategic research center PROGRESS.

References

1. Alexa. Alexa top sites, October 2010. "http://www.alexa.com/topsites/".
2. Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *WEUSE '08: Workshop on End-user software engineering*, pages 1–5. ACM, 2008.
3. R. Holmes, T. Ratchford, M.P Robillard, and R. J. Walker. Automatically Recommending Triage Decisions for Pragmatic Reuse Tasks. In *ASE '09: Proceedings of the 2009 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009.

4. R. Holmes and R. J. Walker. Semi-Automating Pragmatic Reuse Tasks. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 481–482. IEEE Computer Society, 2008.
5. Reid Holmes. *Pragmatic Software Reuse*. PhD thesis, University of Calgary, Canada, 2008.
6. ECMA international. ECMAScript language specification. "http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf".
7. Jean Bovet. Antlr web site, February 2011. "http://www.antlr.org/".
8. Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28. IEEE Computer Society, 2006.
9. Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
10. B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. *SIGCHI Bull.*, 20(SI):69–73.
11. Peng Li and Eric Wohlstadter. Script Insight: Using Models to Explore JavaScript Code from the Browser View. In *Web Engineering, ICWE 2009*, pages 260–274, 2009.
12. Josip Maras, Maja Štula, and Jan Carlson. Extracting Client-side Web User Interface Controls. In *ICWE 2010, International Conference on Web Engineering*, pages 502–505, 2010.
13. Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–108. IEEE Computer Society, 2009.
14. Open source Tagsoup. Tagsoup, Sept 2010. "http://home.ccil.org/ cowan/XML/-tagsoup/".
15. M.C. Schraefel, Yuxiang Zhu, David Modjeska, Daniel Wigdor, and Shengdong Zhao. Hunter Gatherer: Interaction Support for the Creation and Management of Within-Web-Page Collections. In *11th international conference on World Wide Web*, pages 172–181, 2002.
16. Atsushi Sugiura and Yoshiyuki Koseki. Internet scrapbook: creating personalized world wide web pages. In *CHI '97: Extended abstracts on Human factors in computing systems*, pages 343–344. ACM, 1997.
17. Yuzuru Tanaka, Kimihito Ito, and Jun Fujima. Meme Media for Clipping and Combining Web Resources. *World Wide Web*, 9:117–142, 2006.
18. Paolo Tonella and Filippo Ricca. Web Application Slicing in Presence of Dynamic Code Generation. *Automated Software Engg.*, 12(2):259–288, 2005.
19. World Wide Web Consortium (W3C). Document Object Model (DOM), Sept 2010. "http://www.w3.org/DOM/".
20. World Wide Web Consortium (W3C). Document Object Model Events, Sept 2010. "http://www.w3.org/TR/DOM-Level-2-Events/events.html".
21. World Wide Web Consortium (W3C). Xml path language (xpath), Sept 2010. "http://www.w3.org/TR/xpath/".
22. Mark Weiser. Program slicing. In *ICSE '81: 5th International Conference on Software engineering*, pages 439–449. IEEE Press, 1981.
23. Alex Wright. Ready for a Web OS? *Commun. ACM*, 52(12):16–17, 2009.