FUNCTION CLONE DETECTION IN WEB APPLICATIONS:

A SEMIAUTOMATED APPROACH

FABIO CALEFATO, FILIPPO LANUBILE, TERESA MALLARDO

Dipartimento di Informatica, University of Bari, Italy
{calefato,lanubile,mallardo}@di.uniba.it

Many web applications use a mixture of HTML and scripting language code as the front-end to business services, where scripts can run on both the client and server side.

Analogously to traditional applications, code duplication occurs frequently during the development and evolution of web applications. This ad-hoc but pathological form of reuse consists in copying, and eventually modifying, a block of existing code that implements a piece of required functionality. Duplicated blocks are named clones and the act of copying, including slight modifications, is called cloning. When entire functions are copied rather than fragments, duplicated functions are called function clones.

This paper describes how a semiautomated approach can be used to identify cloned functions within scripting code of web applications. The approach is based on the automatic selection of potential function clones and the visual inspection of selected script functions. The results obtained from the clone analysis of four web applications show that the semiautomated approach is both effective and efficient at identifying function clones in web applications, and can be applied to prevent clone from spreading or to remove redundant scripting code.

Key words: web applications, refactoring, code duplication, clone detection, function clones

1 Introduction

Code duplication occurs frequently during the development and evolution of large software systems. This ad-hoc form of reuse consists in copying, and eventually modifying, a block of existing code that implements a piece of required functionality. Duplicated blocks are named *clones* and the act of copying, including slight modifications, is called *cloning*. When entire functions are copied rather than fragments, duplicated functions are called *function clones*.

Cloning often occurs because programmers find it cheaper and quicker using the copy-and-paste feature of their editors than writing instructions from scratch or applying correct reuse mechanisms, based on invocation, inclusion or inheritance. Cloning can also be unintentionally encouraged whenever individual performance are assessed by measuring developers' productivity as code size over effort.

Cloning can be considered a pathological form of code reuse because of its negative effects on software maintenance and evolution. When a failure is discovered during testing or normal operations, the underlying fault might be duplicated together with clones, thus multiplying the cost of repair. In general, when clones are affected by a modification, applying a change is more expensive and error-prone because of the larger impact and possible side effects. Another undesirable consequence is that redundancy increases the size of a software system, thus complicating program comprehension.

Identifying software clones for the purpose of prevention or removal can help to defend against "software aging" [27], where even small changes become very difficult to apply.

Researchers have extensively studied cloning detection for being applied to procedural programs [1, 2, 3, 6, 12, 21, 25] as well as object-oriented programs [4, 5, 12, 14, 28]. Analogously to procedural and object-oriented programs, code duplication occurs frequently during the development and evolution of web applications, which use web technologies as the front-end to business services for the ease of deployment and minimal client configuration [10]. Since web applications evolved from web sites by adding business functionality, many people have erroneously thought that software engineering principles and methods did not apply to web development [29]. As a consequence, web applications are often developed incrementally but without a disciplined approach, and the phenomenon of duplicated code is even worse, because more pervasive, for this class of applications.

Clone identification has been proposed for static web documents [8, 11, 30], written in HTML. However, modern web applications are a mixture of HTML and scripting language code, where scripts can run as event handlers on the client-side or perform HTTP processing on the server-side. Scripts come embedded in *client pages* (i.e., documents available to browsers) as the content of the *script* HTML element, or they are retrieved from include files (containing pure scripting code with no HTML) using the *src* attribute (of the *script* HTML element) in every client page that needs them. The vast majority of client scripts are written with JavaScript. On the server side, the enabling technologies are more varied depending more on the vendors than on standards. A major approach to server-side processing of HTTP requests is using *server pages*, i.e., documents containing HTML annotated with server-side interpreted scripts. Many scripting languages are supported and representative examples are Java Server Pages (JSP), Microsoft's Active Server Pages (ASP), and PHP.

The goal of our research is to investigate how to improve poorly designed web applications. Finding function clones in scripted web pages for the purpose of eliminating duplicated code can be seen as a first step to introduce refactoring [15]. This paper describes how a semiautomated approach can be used to identify cloned functions within scripting code of web applications. The approach is based on the automatic selection of potential function clones and the visual inspection of selected script functions. In [23] we reported a preliminary evaluation of our approach. In this paper, we extend the experimentation with four significant web applications which are currently being maintained and evolved.

This paper is organized as follows. Section 2 presents our approach for function clone identification in scripted web pages. Section 3 introduces the case studies and reports the experimental results. Section 4 summarizes related work. Finally, Section 5 provides conclusions and directions for future work.

2 The Clone Detection Approach

Our approach to detect cloned script functions is based on two main stages: (1) automatic selection of potential function clones, and (2) visual inspection of selected script functions. In the following we use as examples some code excerpts from a web application within the commercial domain. The application, called Conference Management Service, is part of the MS Exchange 2000 Conferencing Server [26] and keeps track of scheduled conferences and provides administrators with control of attendee access to conferences.

2.1 Automatic selection of potential function clones

Most of the times script functions are copied without any change or just slight modifications such as changes to indentation, comments, blank lines, local variables or literals. Often these differences are negligible with respect to duplication removal, other times parameters should be added to functions in order to account for variability. Sometimes, the body of script functions is extended to add new functionality or changed to modify its logic.

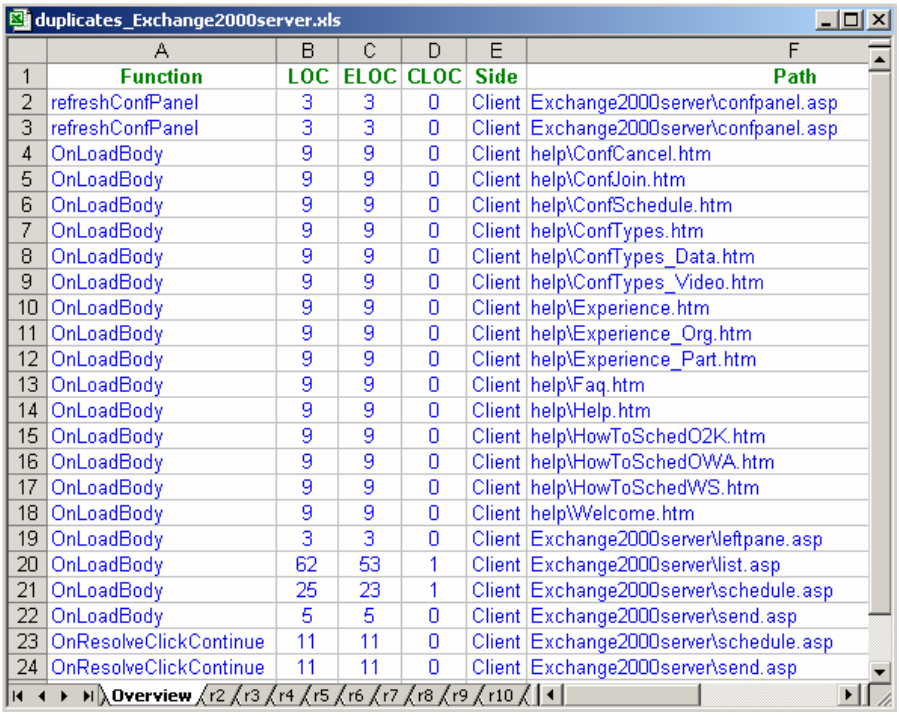
Based on the observation of web programmers' behavior and the investigation of many existing web applications, the selection of candidate function clones is based on the following assumption: whenever web programmers find a script function which fits their required functionality, they duplicate code without changing the function name.

The selection of candidate function clones is automatically performed by a tool, called *eMetrics*, which we developed at the University of Bari. The tool analyzes web applications based on Microsoft's ASP technology and outputs results in HTML and Excel format. Besides measuring the size of a web application to different granularity levels (including function-grain level), the tool selects homonym programmer-defined functions (written in JavaScript or VBScript) as potential cloned functions. Homonymy is defined by the equality of identifiers that are used as function names in the declaration of script functions. The comparison of function names is case insensitive (JavaScript is a case-sensitive language but VBScript is not) and does not take into account the list of parameters.

For each homonym function, selected as a potential cloned function, the tool reports the processing side (client or server), the extended file name (i.e., including complete path) and the following three measures of code length:

- number of lines of code (LOC)
- number of effective lines of code (ELOC), excluding comments and blank lines
- number of comment lines of code (CLOC)

The tool provides both an overview sheet including all the potential cloned script functions (see Figure 1), as well as a sheet for each group of homonym functions (Figure 2).

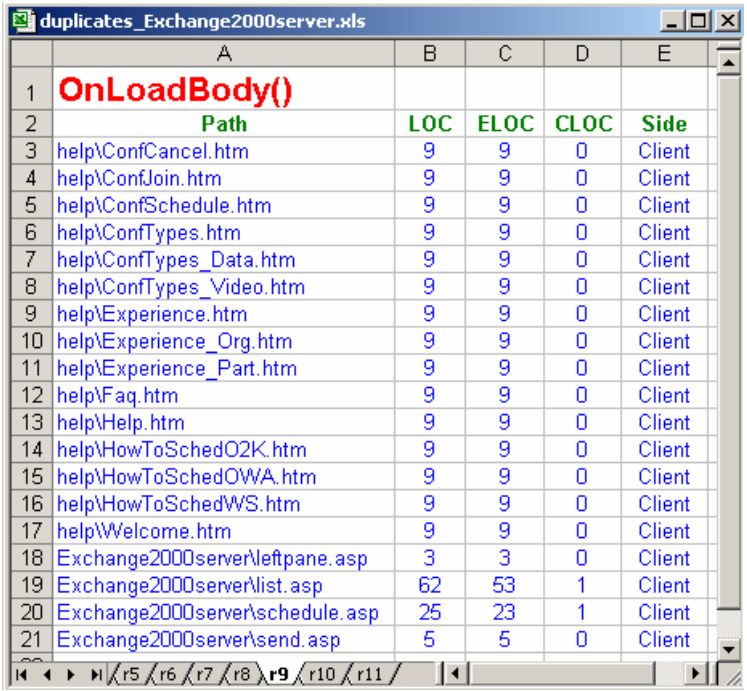


duplicates_Exchange2000server.xls

	A	B	C	D	E	F
1	Function	LOC	ELOC	CLOC	Side	Path
2	refreshConfPanel	3	3	0	Client	Exchange2000server\confpanel.asp
3	refreshConfPanel	3	3	0	Client	Exchange2000server\confpanel.asp
4	OnLoadBody	9	9	0	Client	help\ConfCancel.htm
5	OnLoadBody	9	9	0	Client	help\ConfJoin.htm
6	OnLoadBody	9	9	0	Client	help\ConfSchedule.htm
7	OnLoadBody	9	9	0	Client	help\ConfTypes.htm
8	OnLoadBody	9	9	0	Client	help\ConfTypes_Data.htm
9	OnLoadBody	9	9	0	Client	help\ConfTypes_Video.htm
10	OnLoadBody	9	9	0	Client	help\Experience.htm
11	OnLoadBody	9	9	0	Client	help\Experience_Org.htm
12	OnLoadBody	9	9	0	Client	help\Experience_Part.htm
13	OnLoadBody	9	9	0	Client	help\Faq.htm
14	OnLoadBody	9	9	0	Client	help\Help.htm
15	OnLoadBody	9	9	0	Client	help\HowToSchedO2K.htm
16	OnLoadBody	9	9	0	Client	help\HowToSchedOWA.htm
17	OnLoadBody	9	9	0	Client	help\HowToSchedWS.htm
18	OnLoadBody	9	9	0	Client	help\Welcome.htm
19	OnLoadBody	3	3	0	Client	Exchange2000server\leftpane.asp
20	OnLoadBody	62	53	1	Client	Exchange2000server\list.asp
21	OnLoadBody	25	23	1	Client	Exchange2000server\schedule.asp
22	OnLoadBody	5	5	0	Client	Exchange2000server\send.asp
23	OnResolveClickContinue	11	11	0	Client	Exchange2000server\schedule.asp
24	OnResolveClickContinue	11	11	0	Client	Exchange2000server\send.asp

Overview | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10

Figure 1. Overview sheet reported by the tool



duplicates_Exchange2000server.xls

	A	B	C	D	E
1	OnLoadBody()				
2	Path	LOC	ELOC	CLOC	Side
3	help\ConfCancel.htm	9	9	0	Client
4	help\ConfJoin.htm	9	9	0	Client
5	help\ConfSchedule.htm	9	9	0	Client
6	help\ConfTypes.htm	9	9	0	Client
7	help\ConfTypes_Data.htm	9	9	0	Client
8	help\ConfTypes_Video.htm	9	9	0	Client
9	help\Experience.htm	9	9	0	Client
10	help\Experience_Org.htm	9	9	0	Client
11	help\Experience_Part.htm	9	9	0	Client
12	help\Faq.htm	9	9	0	Client
13	help\Help.htm	9	9	0	Client
14	help\HowToSchedO2K.htm	9	9	0	Client
15	help\HowToSchedOWA.htm	9	9	0	Client
16	help\HowToSchedWS.htm	9	9	0	Client
17	help\Welcome.htm	9	9	0	Client
18	Exchange2000server\leftpane.asp	3	3	0	Client
19	Exchange2000server\list.asp	62	53	1	Client
20	Exchange2000server\schedule.asp	25	23	1	Client
21	Exchange2000server\send.asp	5	5	0	Client

r5 | r6 | r7 | r8 | r9 | r10 | r11

Figure 2. Function-specific sheet reported by the tool

2.2 Visual inspection of selected script functions

The second stage of our approach uses as input the report of potential cloned functions as a guide to the visual inspection of code. The goal of this stage is to check whether homonym script functions can be actually considered clones, and to identify the opportunities of refactoring.

As a first step, pairs of homonym script functions are classified according to the clone classification scheme shown in Table 1.

Table 1. Classification for potential function clones

Level	Name	Description
1	Identical	No differences
2	Nearly-identical	Differences are negligible and do not affect output or state
3	Similar	Despite differences, there are common characteristics that can be factored out
4	Distinct	Functions share the name but what they do is so dissimilar that refactoring does not make sense

Classification follows an ordinal scale based on the degree of equivalence between a pair of script functions. Checking proceeds from level 1 to level 4 and stops when a level can be recognized. Checking is helped by looking at size measures in the report of suspect clones: functions with similar size metrics are inspected first.

The first level, *Identical*, holds when the two functions are exactly equal, because no changes have been applied after the copy. This is the simplest occurrence of function cloning: it does not matter which of the copies will be taken off to eliminate duplication.

The second level, *Nearly-identical*, occurs when the two functions differ for modifications which have no effect on output or application state. Analogously to the first level, any of the cloned functions can be removed during refactoring, but you need to choose which copy will be discarded. Changes to indentation, comments, and blank lines surely fall in this class. Also differences in local variables or constants, if not used for output, can be classified at Level 2. Figure 3 shows two client-side script functions, both named *refreshConfPanel*, that are located in the same ASP file: the former in the *head* element and the latter in the *html* element. They differ because the former (Figure 3.a) uses a constant to represent the frequency of screen-refresh while the latter (Figure 3.b) uses the same value but in a numerical form and includes one comment line. The second function might be removed, and then the two functions can be classified as *Nearly-identical* clones.

```
function refreshConfPanel(secDelay)
{
  window.setTimeout("document.location.reload(true);", secDelay*SECOND);
}
```

(a)

```
function refreshConfPanel(secDelay)
{
  // Timeout uses milliseconds
  window.setTimeout("document.location.reload(true);", secDelay*1000);
}
```

(b)

Figure 3. A pair of script functions classified as Nearly-identical

The third level, *Similar*, takes place when two script functions have common characteristics, such as same code structure and same expressions, but refactoring will require changes to unify the functions. This may happen when the two function clones have different output statements or work on different inputs. A simple refactoring can be the addition of parameters to take into account variability. For example, Figure 4 shows two client-side script functions, both named *OnResolveClickContinue*, that differ for a string assigned to the *action* field of the form. To invoke the same function exemplar, a parameter should be added to the script function and callers should pass a string as argument. The resulting script function should be placed in a client-side include file (with a 'js' filename extension) and retrieved using the *src* attribute of the *script* tag. Then we can classify the couple of selected script functions as clones at level 3 (*Similar*).

```
function OnResolveClickContinue()
{
  var bRet;
  bRet = OnResolveClickContinueWork();
  document.formSchedule.action = "schedule.asp"
  document.formSchedule.hiddenResolveStatus.value = "Resolved"
  if(bRet)
  {
    document.formSchedule.submit();
  }
  return bRet;
}
```

(a)

```
function OnResolveClickContinue()
{
  var bRet;
  bRet = OnResolveClickContinueWork();
  document.formSchedule.action = "send.asp"
  document.formSchedule.hiddenResolveStatus.value = "Resolved"
  if(bRet)
  {
    document.formSchedule.submit();
  }
  return bRet;
}
```

(b)

Figure 4. A pair of script functions classified as Similar

Another example is provided in Figure 5 that shows two script functions, named *OnLoadBody*, which differ because of a numeric value passed as argument to the same called function. These two homonym functions might be replaced by a new script function with a parameter added to its signature, and then they can be classified as function clones at level 3 (Similar).

```
function OnLoadBody()
{
  if (null != parent.frames["panel"])
  {
    if (null != parent.panel.ChangeToMenuItem)
    {
      parent.panel.ChangeToMenuItem(14);
    }
  }
}
```

(a)

```
function OnLoadBody()
{
  if (null != parent.frames["panel"])
  {
    if (null != parent.panel.ChangeToMenuItem)
    {
      parent.panel.ChangeToMenuItem(10);
    }
  }
}
```

(b)

Figure 5. Another pair of script functions classified as Similar

Sometimes, a function is copied and augmented with new statements (see Figure 6). This duplication can be eliminated by invoking the original function from the extended function. However, duplication removal might be time consuming and you might decide whether refactoring is worth the effort.

```
sub AddSubscription(SubLevel, MemberID, CatID, ForumID, TopicID)
  '--- Insert the appropriate sublevel subscription
  StrSql = "INSERT INTO " & strTablePrefix & "SUBSCRIPTIONS"
  StrSql = StrSql &
           "(MEMBER ID, CAT ID, FORUM ID, TOPIC ID) VALUES ("
           & MemberID & ", "
  if sublevel = "BOARD" then
    StrSql = StrSql & "0, 0, 0)"
    elseif sublevel = "CAT" then
      StrSql = StrSql & CatID & ", 0, 0)"
    elseif sublevel = "FORUM" then
      StrSql = StrSql & CatID & ", " & ForumID & ", 0)"
    else
      StrSql = StrSql & CatID & ", " & ForumID & "," & TopicID &")"
  end if
  my Conn.Execute(strSql),,adCmdText + adExecuteNoRecords
end sub
```

(a)

```
sub AddSubscription(SubLevel, Member ID, CatID, ForumID, TopicID)
  ' --- Insert the appropriate sublevel subscritpion
  strSql = "INSERT INTO " & strTablePrefix & "SUBSCRIPTIONS"
  strSql = strSql & "(MEMBER ID, CAT ID, FORUM ID, TOPIC ID) VALUES (" &
         Member ID & ", "
  if sublevel = "BOARD" then
           strSql = strSql & "0, 0, 0)"
       elseif sublevel = "CAT" then
           strSql = strSql & CatID & ", 0, 0)"
       elseif sublevel = "FORUM" then
           strSql = strSql & CatID & ", " & ForumID & ", 0)"
       else
           strSql = strSql & CatID & ", " & ForumID & ", " & TopicID & ")"
       endif
       my Conn.Execute(strSql),,adCmdText + adExecuteNoRecords

       Response.Write "You are subscribed to "
       if sublevel = "BOARD" then
             Response.Write "<br /> all posts in the "
             Response.Write "<br />" & strForumTitle & "forums "
       elseif sublevel = "CAT" then
             strSql="SELECT "& strTablePrefix &"CATEGORY.CAT NAME "
             strSql= strSql & "FROM" & strTablePrefix & "CATEGORY "
       ...
```

(b)

Figure 6. Yet another pair of script functions classified as Similar

The fourth level, *Distinct*, occurs when two script functions, albeit homonym, differ so much in what they do (and then in how they do it) that any refactoring for eliminating duplication would not make sense. Function names might be equal by chance or because of a common triggering event. For example, Figure 7 shows two script functions, again named *OnLoadBody*, which differ with respect to the accomplished functionality. These functions are homonym just because they are triggered by the same *onLoad* event in the *body* element. Then the cloning relation is classified at level 4 (*Distinct*), and it is excluded from refactoring for duplication removal.

```
function OnLoadBody()
{
 <% If((Request.Form("hiddenResolveStatus")="Send") AND
    (bNamesToResolve)) Then %>
       CopyFields();
  <% EndIf %>
}
```

(a)

```
function OnLoadBody()
{
  ChangeToMenuItem(<%=inSelection%>);
}
```

(b)

Figure 7. A pair of script functions classified as Distinct

After having classified potential function clones, the script functions are grouped according to the refactoring opportunity. A refactoring opportunity is a simple but useful change that merges multiple function clones without modifying the external behavior of the application.

If a set of homonym functions shares the same level-*n* cloning relation, then the set is classified as a group at level-*n*. For example, if there are 4 homonym script functions and all 6 pairs can be classified as *Identical* (level 1), then we have one group of 4 script functions at level 1 for doing refactoring. However, if one of the script functions is *Similar* (rather than *Identical or Nearly-identical*) with respect to the other 3 homonyms, then there is a group of 4 script functions at level 3, because we want to take the most from a unique refactoring opportunity. On the contrary, if one of the script functions is *Distinct* (rather than *Identical*, *Nearly-identical*, or *Similar*), then there is one group of just 3 script functions at level 1 and the different (albeit homonym) function is considered out of the refactoring scope. This is because a level-4 cloning relation should be considered as a false positive since the tool has erroneously identified homonym functions as suspect clones. Figure 8 shows how a function-specific sheet is edited to take notes of groups of clones corresponding to refactoring opportunities.

	A	B	C	D	E	F	G	H	I	J
1	**DetectCaller()**									
2	**Path**	**LOC**	**ELOC**	**CLOC**	**Side**	**Level**		**Comment**		
3	administration\checklist.asp	4	4	0	Server	Identical	Similar	The first three functions use as input a global variable named "child". The second group of identical functions use as input a variable global named "caller". The last function uses as input another global variable named "childBack". All the clones can be replaced by a unique copy of the function but global variable names should be unified too.		
4	administration\helpCustomize.asp	4	4	0	Server					
5	administration\saveHelpCustomize.asp	4	4	0	Server					
6	administration\administration.asp	4	4	0	Server	Identical				
7	collection\collection.asp	4	4	0	Server					
8	discovery\discovery.asp	4	4	0	Server					
9	followUp\followUp.asp	4	4	0	Server					
10	logOn\logOn.asp	4	4	0	Server					
11	overview\overview.asp	4	4	0	Server					
12	planning\planning.asp	4	4	0	Server					
13	planning\setPassword.asp	4	4	0	Server					
14	rework\rework.asp	4	4	0	Server					
15	administration\delHelpCustomize.asp	4	4	0	Server					

Figure 8. Function-specific sheet annotated with respect to refactoring opportunities

3 Case Studies

We applied our approach for function clone detection to four web applications with the main goal to assess the effectiveness and efficiency of the approach, and measure the extent of refactoring opportunities. We used three web applications from the public domain, for which we did not have expectations about how much duplication exists, and one web application from the research domain for which it was known that there were many duplicated script functions.

The first web application, QuickAuction [35], is a basic auction application that can be integrated into other web sites to add simple auctions features. The second and third applications, respectively Web Wiz Forums [34] and Snitz Forums 2000 [32], are both web-based bulletin board engines. The fourth web application, IBIS [22], provides groupware support for distributed software inspections. All four applications use MS ASP technology to implement web server pages. They range in size from tens to hundreds of web pages, and are currently being maintained and evolved. Table 2 shows a characterization of the web applications that we selected as case studies, including size-related statistics.

We first used the eMetrics tool to retrieve potential function clones in the four web applications. Then, we used the reports from the tool to visually inspect the code of the selected script functions, classify suspect clones, and group discovered function clones according to refactoring opportunities.

Table 2. The web applications used for case studies

	QuickAuction	Web Wiz Forums	Snitz Forums 2000	IBIS
Version	2.0.0	7.01	3.4.3	1.3.2
Client-side scripting language	none	JavaScript	JavaScript	JavaScript
Server-side scripting language	VBScript	VBScript	VBScript	JavaScript
No. of files (total)	65	403	217	257
No. of ASP files	48	159	93	124
No. of HTML files	2	0	1	0
No. of client-side include files	0	1	3	10
No. of server-side include files	11	29	29	7
No. of client-side script functions	0	18	47	103
No. of server-side script functions	63	50	222	299

31. Effectiveness and Efficiency

In this section we present the overall performance of our approach with respect to effectiveness and efficiency. Table 3 reports the following measures:

- number of existing function clones: script functions that provide identical behavior or share same functionality;
- number of candidate function clones: homonym script functions that have been automatically selected by the tool as potential clones;
- number of discovered function clones: homonym script functions that have been classified as Identical, Nearly-Identical or Similar, and then can be considered as true function clones;
- number of false negatives: existing function clones that have not been discovered; in our approach function clones with different names remain undiscovered;
- number of false positives: homonym script functions that have been classified as Distinct, and then have been erroneously selected as candidate clones;
- recall: percentage of discovered function clones over existing function clones;
- precision: percentage of discovered function clones over candidate function clones;
- inspection effort: effort (measured as person/hours) spent for the visual inspection of code to classify candidate function clones and identify the refactoring opportunities.

Table 3. Effects of function clone detection

	QuickAuction	Web Wiz Forums	Snitz Forums 2000	IBIS
No. of existing function clones	40	47	69	166
No. of candidate function clones	40	58	75	186
No. of discovered function clones	36	45	46	143
No. of false negatives	4	2	23	23
No. of false positives	4	13	29	43
Recall	90%	96%	67%	86%
Precision	90%	78%	61%	77%
Inspection effort (person/hours)	1h 40min	2h 45min	2h 20min	7h 5min

Recall and precision were initially developed for evaluating Information Retrieval systems [33] and now are being used in assessing clone detection techniques [19, 20]. We use the recall metric for evaluating the effectiveness of the approach to clone detection, while precision and inspection effort

contribute to assess its efficiency. The other metrics provide the basis for the computation of recall and precision.

Recall reflects the completeness of the results produced by a clone detection technique by comparing discovered clones with clones that really exist and could have been retrieved with a faultless clone detector (for practical reasons, the faultless clone detector is replaced by human analysis of source code). Thus, the higher the amount of false negatives (undiscovered clones), the lower the recall. In our multiple case studies, values of recall were high in three cases (between 86% and 96%) but not for the Snitz Forums application (67%). This can be explained by the presence on the client side of many analogous functions that were named differently depending on a variable which characterized the differences among functions. In this case, the assumption behind function clone selection (programmers copy and paste script functions without changing function names) resulted partially invalid.

Precision reflects the accuracy of a clone detection technique by comparing discovered clones with those that were selected as candidate clones. When the amount of false positives is high (and then precision is low), more time will be wasted for inspecting irrelevant components. While precision was high for the QuickAuction case study (90%), values of precision for the remaining three case studies varied between 61% and 78%. This is partly explained by extensive changes that have occurred after initial copy-and-paste with the consequence that it would be easier to redesign from the beginning rather than applying refactoring.

Although the tool selected some irrelevant function clones, the effort to visually inspect of code varied between 1 hour 40min and 2 hours 45min for three case studies. The inspection of candidate function clones in IBIS took about 7 hours because there were much more function clones in the application. Thus, our approach detects a significant amount of function duplication with a reasonable effort.

3.2 Refactoring opportunities

In this section we show how clone detection helped to identify refactoring opportunities for duplication removal.

Results for the four cases studies are reported respectively in Table 4-7. For each level, from level 1 (*Identical*) to level 3 (*Similar*), the following values are reported:

- number of refactoring opportunities: they correspond to groups of discovered function clones that share the same level of clone relationship and can be merged into a single function;
- function clones involved in refactoring: the number of functions that are affected by refactoring
- % of functions involved in refactoring: the number of function clones involved in refactoring over the total number of script functions in the application;
- ELOC involved in refactoring: the amount of scripting code affected by change, measured as effective lines of code;
- % ELOC involved in refactoring: the amount of scripting code affected by change over the amount of code of all script functions in the application.

The total columns in the tables contain the cumulative frequencies and cumulative percentages of the three levels (*Identical, Nearly-identical, Similar*). The totals represent the number and the impact of refactoring opportunities that are worth taking advantage of.

Results are reported separately for client-side and server-side script functions, except for the QuickAuction application that does not use scripting code on the client side.

Refactoring of QuickAuction and Web Wiz Forums would involve most of the server-side script functions (respectively 57% and 80%) within the applications. We discovered that Web Wiz Forums included two identical ASP files in different subsystems. Then for each of the twenty script functions contained in the duplicated file there is a refactoring opportunity corresponding to a pair of identical functions.

While refactoring affected 21% of server-side script functions of Snitz Forums 2000, none of the 12 existing client-side function clones was selected for refactoring because cloned script functions had been renamed after duplication.

Table 4. Refactoring opportunities in QuickAuction

	Server-side			
	Level 1	Level 2	Level 3	**Total**
Refactoring opportunities	4	0	4	**8**
Function clones involved in refactoring	16	0	20	**36**
% Functions involved in refactoring	25%	0%	32%	**57%**
ELOC involved in refactoring	224	0	504	**728**
% ELOC involved in refactoring	19%	0%	42%	**61%**

Table 5. Refactoring opportunities in Web Wiz Forums

	Client-side				***Server-side***			
	Level 1	Level 2	Level 3	**Total**	Level 1	Level 2	Level 3	**Total**
Refactoring opportunities	1	0	1	**2**	20	0	0	**20**
Function clones involved in refactoring	2	0	3	**5**	40	0	0	**40**
% Functions involved in refactoring	11%	0%	17%	**28%**	80%	0%	0%	**80%**
ELOC involved in refactoring	6	0	54	**60**	1120	0	0	**1120**
% ELOC involved in refactoring	2%	0%	19%	**21%**	73%	0%	0%	**73%**

Table 6. Refactoring opportunities in Snitz Forums 2000

	Client-side				***Server-side***			
	Level 1	Level 2	Level 3	**Total**	Level 1	Level 2	Level 3	**Total**
Refactoring opportunities	0	0	0	**0**	12	1	9	**22**
Functions involved in refactoring	0	0	0	**0**	28	4	14	**46**
% Functions involved in refactoring	0%	0%	0%	**0%**	13%	2%	6%	**21%**
ELOC involved in refactoring	0	0	0	**0**	663	73	476	**1212**
% ELOC involved in refactoring	0%	0%	0%	**0%**	10%	1%	7%	**18%**

Table 7. Refactoring opportunities in IBIS

	Client-side				***Server-side***			
	Level 1	Level 2	Level 3	**Total**	Level 1	Level 2	Level 3	**Total**
Refactoring opportunities	7	1	2	**10**	4	5	6	**15**
Functions involved in refactoring	24	2	9	**35**	48	35	25	**108**
% Functions involved in refactoring	23%	2%	9%	**34%**	16%	12%	8%	**36%**
ELOC involved in refactoring	389	7	77	**473**	281	736	829	**1846**
% ELOC involved in refactoring	12%	0%	2%	**15%**	3%	8%	9%	**21%**

Finally, more than one third of script functions in the IBIS application could be improved by means of refactoring. Most of the refactoring opportunities were at level 1 (*Identical*) and then simple to apply. We then applied refactoring on the IBIS application, and spent less than one person/day to eliminate more than one hundred of duplicated script functions.

4 Related Work

Various techniques are used to detect clones in software systems: string and token matching [3, 12, 16, 17, 18, 24, 31], subtrees-subgraphs comparison [6, 19, 20], and metric-based characterization [2, 5, 19, 21, 25]. Also, numerous tools have been built to find clones in source code written in a variety of programming languages, such as C, C++, COBOL, Smalltalk, Java, and Python.

The *Dup* tool [3] uses a line-by-line parameterized match to identify code portions that differ in semantic substitution of variable and constant names. A string matching algorithm is also applied by the *Duploc* tool [12, 31]. It offers a clickable matrix display that allows users to visually inspect the source code that produced the match. Like our semiautomated approach, automatic clone detection and visual exploration of code are combined to guide refactoring. In [16, 17], a viewing tool is presented that identifies exact repetitions of text using fingerprints, which are short strings used in place of larger data objects for more efficient comparisons. *Sif* [24] is based on the same approach. *CCFinder* [18] concatenates the tokens of a single file into a single token sequence, skipping whitespaces and applying transformation rules, such as replacement of variables with special tokens. From all the substrings in the transformed token sequence, equivalent pairs are detected as clone pairs.

Baxter et al. [6] propose a tool that identifies cloned fragments in C source code and also produces macro bodies for doing refactoring. The tool parses source code to build an Abstract Syntax Tree (AST), and then compares subtrees for similarity, using a hashing function. An analogous approach is proposed in [20], where PDGs (Program dependence graphs) are used to represent both the structure and the data flow within the program.

The above clone detection techniques do not easily scale up because they are computationally expensive [5]. In order to improve efficiency, Mayrand et al. [25] use metrics as a signature for code functions, thus allowing for a fast search of code duplication at the function-grain level. The strategy for identifying function clones is based on four points of comparison: function name, layout metrics, expression metrics, and control flow metrics. Any pair of functions is then compared with respect to these characteristics to define an ordinal scale of cloning, based on degree of similarity between function clone pairs. The first class of this taxonomy, *ExactCopy*, requires that the function names are identical as well as metric values. The second class, *DistinctName*, has the same requirements of the first class except for the names of functions which are different; it assumes that function cloning occurs by renaming the function to avoid name conflicts in a module. An empirical validation of the proposed classification, shows that the first classification level, *ExactCopy*, is predominant over the second one, *DistinctName*, which accounts for less than 1% of the overall function clones. The other classification levels are not reliable because they result in a too high number of false positives (i.e., low precision). This same approach, but limited to the first two classes, has been used in [1, 2] to study cloning evolution across multiple releases. However, no empirical data are provided to distinguish between the two classes.

Analogously to [25], our approach to clone analysis focus on whole functions rather than code fragments. However, because the metric-based approach appear to be effective only for the *ExactCopy* class, rather than using measures to detect and automatically classify clones, we select homonym functions as potential function clones, and use size measures as a guideline for the visual inspection.

Kontogiannis [19] proposes another metric-based approach to make clone detection faster. The approach uses an AST for program representation and computes the Euclidean distance between code fragments on a 5-dimensional space defined by data-flow and control-flow metrics. Unlike previous studies, the experimental results include recall and precision values. Results show that a recall of 60% can be achieved with a precision of 41%, while at higher recall values (e.g., 70%) precision goes down (e.g., 19%). Because a fast clone detection is obtained at the expense of accuracy, Balazinska et al. [5] use a hybrid approach of metric-based characterization and dynamic pattern matching: measurement is limited to a pre-processing stage for reducing the search space, and then a pattern matching algorithm is applied to compare code fragments for object-oriented programs written in Java.

Although the automatic detection of clone candidates, in our approach, is based on the search for homonym functions, we achieve better results of recall (between 67% and 96%) without drops in precision (between 61% and 90%). In the case of the largest web application, for which we removed more than one hundred of duplicated script functions, recall is 86% and precision is 77%. With such good results in the domain of web applications more sophisticated solutions are not worth being implemented and applied. We hypothesize that the naïve approach to web application development in the early years have resulted in so many clones that even simple clone detection solutions are extremely effective.

The identification of clones in web documents has been proposed by Di Lucca et al. in [11]. They address the detection of similar static HTML pages by computing the distance between items in web pages and evaluating their degree of similarity. Cloned static web pages are also identified in [8, 30] with the aim of restructuring them into dynamic web pages that retrieve extracted information from a database. However, the progressive reduction of purely static web sites suggests that looking for HTML duplication in static web pages is not enough. Then, our approach, focusing on scripted code within client and server pages in web applications, can be seen as a complement of clone identification in static web documents.

5 Conclusions

Clone detection techniques based on string and token matching or subtrees-subgraphs comparison have been proposed for programs written in C/C++ or Java. They have never been implemented and experimented for scripted web applications. Because they are computationally expensive, metric-based approaches have been introduced for identification of clones. However, metric-driven clone detection have failed to achieve high levels of recall and precision. When recall is kept high, precision dramatically drops down. In other words, using structural metrics for automatically detecting candidate clones has not shown to work well, unless their use is limited to a preprocessing stage.

This paper presented a semiautomated approach for identifying function clones embedded in HTML scripting of web applications. A list of potential cloned script functions is automatically produced by a tool, while classification of suspect clones is performed through visual inspection of source code. The contribution of this approach is directed to support both verification tasks, before a new file is put under configuration management, and reengineering activities, after that the lack of method in web development has driven applications towards code decay.

An empirical validation of the approach has been performed on four non-trivial case studies. We measured recall (there were few false negatives except in one case), precision (there were various false positives but noise did not bothered too much), effort (visual inspection of potential clones did not take more than one workday), and the refactoring opportunities (there were many simple changes that could be applied to eliminate duplication). We also took the chance to remove function clones in the web application for which there were more duplicates: refactoring took one person/day with very few cloned script functions left out in the code.

The results obtained from the case studies show the potential of the semiautomated approach for identifying function clones in web applications, and its usefulness for the goal of verification, to prevent clone from spreading, or for the goal of refactoring, to remove redundancy and shrink application size.

However, that we found a high number of function clones does not mean that we were able to identify most of code duplication. In fact, our approach looks for clones at the function level but it does not detect duplicated code at lower granularity levels. Thus, our approach would not provide so much help to those web programmers who fail to employ user-defined functions for performing specific tasks. Nevertheless, the proposed approach has the merit to keep clone detection for web applications simple but effective. Simplicity is a key design principle which has inspired the development of the World Wide Web [7] as well as of the whole Internet [9].

Our approach might be refined, analogously to [4, 13, 15], by developing specific redesign patterns for web applications to provide a guide to duplication removal. The automatic selection of potential clones could also be extended to other web enabling technologies, such as Java Server Pages or PHP.

Our future plans also include to use the proposed approach as an instrument to study the evolution and decay of web applications through time.

Acknowledgements

We would like to thank Vincenzo Fiorentino and Biagio Taccogna for having helped to implement a preliminary version of the eMetrics tool. We are also grateful to the anonymous reviewers for their helpful improvement suggestions.

References

1. Antoniol, G., Villano, U., Merlo, E. and Di Penta, M. Analyzing cloning evolution in the Linux kernel. Information and Software Technology 2002, 44(13). 755–765.
2. Antoniol, G., Casazza, G., Di Penta, M. and Merlo, E. Modeling Clones Evolution Through Time Series. in International Conference on Software Maintenance, (Florence, Italy, 2001). 273-280.
3. Baker, B.S. On Finding Duplication and Near-Duplication in Large Software Systems. in Second Working Conference on Reverse Engineering, (Toronto, Canada, 1995). 86-95.
4. Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K. Partial Redesign of Java Software Systems Based on Clone Analysis. in Sixth Working Conference on Reverse Engineering, (Atlanta, USA, 1999). 326-336.
5. Balazinska, M., Merlo, E., Dagenais, M., Lague, B and Kontogiannis, K. Measuring Clone Based Reengineering Opportunities. in Sixth IEEE International Symposium on Software Metrics, (Boca Raton, USA, 1999). 292-303.
6. Baxter, I.D., Yahin, A., Moura, L., Santa Anna, M. and Bier, L. Clone Detection Using Abstract Syntax Trees. in International Conference on Software Maintenance, (Washington DC, USA, 1998). 368-377.

7. Berners-Lee, T. Principles of Design. 1998, last change: January 2002. http://www.w3.org/DesignIssues/Principles.html
8. Boldyreff, C. and Kewish, R. Reverse Engineering to Achieve Maintainable WWW Sites. in Eight Working Conference on Reverse Engineering (WCRE'01), (Stuttgart, Germany, 2001). 249-257.
9. Carpenter, B (Ed.). Architectural Principles of the Internet. RFC 1958, June 1996. http://www.ietf.org/rfc/rfc1958.txt
10. Conallen, J. Building Web Applications with UML. Addison-Wesley: Reading, MA, 2000.
11. Di Lucca, G.A., Di Penta, M., Fasolino, AR. and Granato, P. Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages. in Seventh IEEE Workshop on Empirical Studies of Software Maintenance. (Florence, Italy, 2001). 107-113.
12. Ducasse, S., Rieger, M. and Demeyer, S. A Language Independent Approach for Detecting Duplicated Code. in International Conference on Software Maintenance, (Oxford, U.K., 1999). 109-118.
13. Fanta, R. and Rajlich, V. Removing Clones from the Code. Journal of Software Maintenance: Research and Practice 1999, 11(4). 223–243.
14. Fioravanti, F., Migliarase, G. and Nesi, P. Reengineering Analysis of Object-Oriented Systems via Duplication Analysis. in International Conference on Software Engineering, (Florence, Italy, 2001); 577-590.
15. Fowler, M. Refactoring: Improving the design of existing code. Addison-Wesley: Reading, MA, 1999.
16. Johnson, J.H. Identifying Redundancy in Source Code using Fingerprints. in CAS Conference. (Toronto, Canada, 1993). 171–183.
17. Johnson, J.H. Substring Matching for Clone Detection and Change Tracking. in Proceedings International Conference on Software Maintenance, (Victoria, Canada, 1994). 120-126.
18. Kamiya, T., Kusumoto, S. and Inoue, K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions On Software Engineering 2002, 28(7). 654–670.
19. Kontogiannis, K. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. in Fourth Working Conference on Reverse Engineering, (Amsterdam, The Netherlands, 1997); 44-54.
20. Krinke, J. Identifying Similar Code with Program Dependence Graphs. in Eighth Working Conference on Reverse Engineering, (Stuttgart, Germany, 2001). 301-309.
21. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M. and Hudepohl, J. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. in International Conference on Software Maintenance, (Bari, Italy, 1997). 314-321.
22. Lanubile, F. and Mallardo, T. Tool Support for Distributed Inspection. in Proceedings 26th Annual International Computer Software & Applications Conference, (Oxford, U.K., 2002). 1071-1076.
23. Lanubile, F. and Mallardo, T. Finding Function Clones in Web Applications. in Seventh European Conference on Software Maintenance and Reengineering, (Benevento, Italy, 2003). 379-388.
24. Manber, U. Finding Similar Files in a Large File System. in USENIX Winter 1994 Technical Conference. (San Francisco, USA, 1994). 1-10.
25. Mayrand, J., Leblanc, C., Merlo, E.M. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. in International Conference on Software Maintenance, (Monterey, USA, 1996). 244-254.
26. Microsoft, Exchange 2000 Conferencing Server. Available: http://www.microsoft.com/exchange/techinfo/conferencing/ [Accessed January 2004].
27. Parnas, D.L. Software Aging. in 16th International Conference on Software Engineering, (Sorrento, Italy, 1994). 279-287.
28. Patenaude, J.F., Merlo, E.M., Dagenais, M. and Lague, B. Extending Software Quality Assessment Techniques to Java Systems. in Seventh International Workshop on Program Comprehension, (Pittsburgh, USA, 1999). 49-57.

29. Pressman, R.S., Lewis, T., Adida, B., Ullman, E., DeMarco, T., Gilb, T., Gorda, B., Humphrey, W. and Johnson, R. Can Internet-Based Applications Be Engineered? IEEE Software 1998, 15(5). 104–110.
30. Ricca, F. and Tonella, P. Using Clustering to Support the Migration from Static to Dynamic Web Pages. in of the 11th International Workshop on Program Comprehension, (Portland, USA, 2003). 207-216.
31. Rieger, M. and Ducasse, S. Visual Detection of Duplicated Code. in Workshop on Experiences in Object-Oriented Re-Engineering. (Brussels, Belgium, 1998). 75-76.
32. Snitz Forums 2000, Product Specifications and Downloads. Available: http://forum.snitz.com/spec.asp [Accessed January 2004].
33. van Rijsbergen C. Information Retrieval. Butterworths: London, UK. 1979.
34. Web Wiz Guide, Web Wiz Forums Downloads. Available: http://www.webwizguide.info/web_wiz_forums/forum_download.asp [Accessed January 2004].
35. Xcent, QuickAuction. Available: http://www.xcent.com/products/QuickAuction.htm [Accessed January 2004].