AUTOMATIC WEB WIDGETS PREDICTION FOR WEB 2.0 ACCESS TECHNOLOGIES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2013

By
Alex Qiang Chen
School of Computer Science

Contents

Abstract 16

Declaration 17

Copyright 18

Acknowledgements 19

1 Introduction 20

1.1 Motivation 25

1.1.1 Supporting Older Users of Dynamic Web Content 32

1.1.2 Supporting Visually Impaired Users of Dynamic Web Content 34

1.2 Technological Freeze 36

1.3 Research Questions 37

1.4 Thesis Structure 38

1.5 Publications 41

2 Background and Related Work 43

2.1 Web Development Technologies and Recommendations 44

2.1.1 Web Accessibility 45

2.1.2 Web Design 50

2.2 Reverse Engineering 56

2.2.1 Purposes for Identifying Web Widgets 56

2.2.2 Coding Format 57

2.2.3 Code Comprehension 60

2.2.4 Attempts to Identify Graphical Objects and Web Widgets . . . 72

2.3 Using Ontology to Classify Web Widgets 74

2.4 Summary 76

3 Classifying Web Widgets **77**
3.1 Developers & Users Perception 78
3.2 Widget's Taxonomy 79
3.2.1 Widgets Layer 82
3.2.2 Components Layer 82
3.2.3 Tell-Signs Layer 87
3.2.4 Code Constructs Layer 90
3.3 Widget Identification Ontology (WIO) 92
3.3.1 Widget's Granularity 92
3.3.2 Reusable Widget's Objects 93
3.3.3 Auto Suggest List Widget 94
3.3.4 Carousel Widget 96
3.3.5 Collapsible Panel Widget 99
3.3.6 Customisable Content Widget 100
3.3.7 Popup Content Widget 101
3.3.8 Popup Window Widget 103
3.3.9 Slide Show Widget 103
3.3.10 Tabs Widget 104
3.3.11 Ticker Widget 106
3.4 Popularity of Widgets 107
3.5 Summary 111

4 Feasibility Investigation for Using Tell-Signs **112**
4.1 Experiment Setup 112
4.1.1 Scripting Language 113
4.1.2 Experimental Data Set 113
4.2 Auto Suggest List (ASL) Widget 114
4.2.1 The Tell-Signs 114
4.2.2 Assumptions and Rules 120
4.2.3 Limitations 122
4.2.4 Results and Discussions 122
4.3 Carousel Widget 124
4.3.1 Slide Show Widget 125
4.3.2 Tabs Widget 126
4.3.3 The Tell-Signs 127
4.3.4 Assumptions and Rules 130

4.3.5 Limitations 131
4.3.6 Results and Discussions 132
4.4 Suggested Refinements 135
4.4.1 Linking Tell-Signs User Interface Objects 135
4.4.2 Annotating the Source Code 137
4.5 Summary 137

5 Predicting Widgets On A Web Page 139
5.1 Issues With Analysing the Web Page Source Code 139
5.1.1 Multiple External Resources 140
5.1.2 Missing Code 140
5.1.3 Web Browsers Monitoring Management 140
5.1.4 Weakly-Typed Scripting Languages 141
5.1.5 Non-Standardisation Interpretation 142
5.1.6 Just-In-Time (JIT) Interpreters 142
5.2 Profiling The Web Page Source Code 143
5.2.1 Handling Events 145
5.2.2 Dynamic Code Generation 145
5.2.3 Redundant Code 145
5.3 Approaches to Identify Web Widgets 146
5.3.1 Bottom-up Approach 147
5.3.2 Top-down Approach 147
5.3.3 Fusion of Top-down and Bottom-up Approach 148
5.3.4 Implementing a Tracer 149
5.3.5 Implementing a Profiler 150
5.3.6 Tailor Designed Profiler 150
5.4 Analysing The Code 150
5.4.1 Synthesising the Code 152
5.4.2 Inside the Profiler 152
5.4.3 Widget's Inference 162
5.4.4 Expanding the System 162
5.5 Evaluating the Profiler 165
5.5.1 Profiler's Evaluation Setup 168
5.5.2 Profiler Evaluation's Results and Discussions 168
5.6 Techniques to Predict Widgets 175
5.6.1 Ticker Widget Tell-Signs 176

5.6.2 Popup Content Widget Tell-Signs 177
5.6.3 Collapsible Panel Widget Tell-Signs 180
5.6.4 Auto Suggest List (ASL) Widget Tell-Signs 181
5.6.5 Tabs Widget Tell-Signs 182
5.6.6 Carousel and Slide Show Widgets Tell-Signs 184
5.6.7 Limitations 189
5.7 Summary 189

6 Widget Prediction Evaluation 190
6.1 Evaluation Setup 191
6.1.1 Data Collection 191
6.1.2 WPS Setup 192
6.1.3 Manual Analysis 193
6.2 WPS Evaluation Results 194
6.2.1 Ticker Widget Evaluation 194
6.2.2 Auto Suggest List (ASL) Widget Evaluation 196
6.2.3 Popup Content Widget Evaluation 197
6.2.4 Collapsible Panel Widget Evaluation 198
6.2.5 Tabs Widget Evaluation 200
6.2.6 Carousel Widget Evaluation 201
6.2.7 Slide Show Widget Evaluation 202
6.3 Analysis and Discussion 204
6.4 Summary 213

7 Conclusions and Future Work 214
7.1 Contribution of the Thesis 215
7.1.1 Widgets Can Be Formally Modelled and Classified 216
7.1.2 Tell-Signs Can Be Discovered From the Web Page Source Code 217
7.1.3 Widgets Prediction Can Be Done Automatically From the Web Page Source Code 217
7.2 Insights to WPS 218
7.3 Outstanding Issues 220
7.4 Future Research 221
7.5 Summary 224

Nomenclature 225

A Code Constructs Objects **238**

B Results for Profiler's 10K Character Size Evaluation **243**

C Results for Profiler's 15K Character Size Evaluation **259**

D Results for Profiler's 20K Character Size Evaluation **281**

E Identification Reference of Tell-Signs **308**

Word Count: 79373

List of Tables

3.1 Popularity of widgets for the top 50 websites 109

3.2 Ranking of the types of widget based on their probability of occurrence in the top 50 Websites. 110

4.1 Feasibility investigation results for ASL widget detection. 123

4.2 Feasibility investigation results for the Carousel widget detection. . . 133

5.1 Code patterns for functions that will and will not be identified by M4's regular expression 155

5.2 Character sizes of the default page for the top 30 most popular Websites ranked by Alexa Top 500 Global Sites selected on 09 March 2011. 166

5.3 Compiled evaluation results breakdown for 10, 15 and 20 thousand characters comparison between the Code Profiler results (Auto) and manual detection results (Manual). The full extent of the results can be found in Appendix B, C, D. 171

5.4 Overall evaluation results for 10, 15 and 20 thousand characters comparison between the Code Profiler results (Auto) and manual detection results (Manual), as well as the similarity scoring between both sets of results. 174

6.1 Comparison of results with and without `digg.com` for Popup Content widget. 208

6.2 Comparison of results with and without `digg.com` for the combination of Popup Content and Collapsible widget. 210

B.1 The Functions results extracted from the profiler for google.com when evaluating over 10K character size. 245

B.2 The Objects results extracted from the profiler for google.com when evaluating over 10K character size. 245

B.3 The Listener results extracted from the profiler for google.com when evaluating over 10K character size. 246
B.4 The Functions results extracted from the profiler for facebook.com when evaluating over 10K character size. 247
B.5 The Objects results extracted from the profiler for facebook.com when evaluating over 10K character size. 247
B.6 The Functions results extracted from the profiler for youtube.com when evaluating over 10K character size. 248
B.7 The Functions results extracted from the profiler for yahoo.com when evaluating over 10K character size. 250
B.8 The Objects results extracted from the profiler for yahoo.com when evaluating over 10K character size. 250
B.9 The Objects results extracted from the profiler for live.com when evaluating over 10K character size. 250
B.10 The Functions results extracted from the profiler for blogger.com when evaluating over 10K character size. 251
B.11 The Functions results extracted from the profiler for baidu.com when evaluating over 10K character size. 253
B.12 The Objects results extracted from the profiler for baidu.com when evaluating over 10K character size. 253
B.13 The Listener results extracted from the profiler for baidu.com when evaluating over 10K character size. 254
B.14 The Functions results extracted from the profiler for wikipedia.org when evaluating over 10K character size. 254
B.15 The Listener results extracted from the profiler for wikipedia.org when evaluating over 10K character size. 255
B.16 The Functions results extracted from the profiler for twitter.com when evaluating over 10K character size. 256
B.17 The Objects results extracted from the profiler for twitter.com when evaluating over 10K character size. 256
B.18 The Functions results extracted from the profiler for qq.com when evaluating over 10K character size. 257
B.19 The Objects results extracted from the profiler for qq.com when evaluating over 10K character size. 258

C.1 The Functions results extracted from the profiler for google.com when evaluating over 15K character size. 262
C.2 The Objects results extracted from the profiler for google.com when evaluating over 15K character size. 263
C.3 The Listener results extracted from the profiler for google.com when evaluating over 15K character size. 263
C.4 The Functions results extracted from the profiler for facebook.com when evaluating over 15K character size. 265
C.5 The Objects results extracted from the profiler for facebook.com when evaluating over 15K character size. 266
C.6 The Functions results extracted from the profiler for youtube.com when evaluating over 15K character size. 267
C.7 The Objects results extracted from the profiler for youtube.com when evaluating over 15K character size. 267
C.8 The Functions results extracted from the profiler for yahoo.com when evaluating over 15K character size. 270
C.9 The Objects results extracted from the profiler for yahoo.com when evaluating over 15K character size. 270
C.10 The Objects results extracted from the profiler for live.com when evaluating over 15K character size. 271
C.11 The Functions results extracted from the profiler for blogger.com when evaluating over 15K character size. 271
C.12 The Functions results extracted from the profiler for baidu.com when evaluating over 15K character size. 274
C.13 The Objects results extracted from the profiler for baidu.com when evaluating over 15K character size. 274
C.14 The Listener results extracted from the profiler for baidu.com when evaluating over 15K character size. 275
C.15 The Functions results extracted from the profiler for wikipedia.org when evaluating over 15K character size. 275
C.16 The Listener results extracted from the profiler for wikipedia.org when evaluating over 15K character size. 275
C.17 The Functions results extracted from the profiler for twitter.com when evaluating over 15K character size. 277

C.18 The Objects results extracted from the profiler for twitter.com when evaluating over 15K character size. 278
C.19 The Listener results extracted from the profiler for twitter.com when evaluating over 15K character size. 278
C.20 The Functions results extracted from the profiler for qq.com when evaluating over 15K character size. 280
C.21 The Objects results extracted from the profiler for qq.com when evaluating over 15K character size. 280

D.1 The Functions results extracted from the profiler for google.com when evaluating over 20K character size. 285
D.2 The Objects results extracted from the profiler for google.com when evaluating over 20K character size. 286
D.3 The Listener results extracted from the profiler for google.com when evaluating over 20K character size. 286
D.4 The Functions results extracted from the profiler for facebook.com when evaluating over 20K character size. 289
D.5 The Objects results extracted from the profiler for facebook.com when evaluating over 20K character size. 290
D.6 The Functions results extracted from the profiler for youtube.com when evaluating over 20K character size. 292
D.7 The Objects results extracted from the profiler for youtube.com when evaluating over 20K character size. 292
D.8 The Functions results extracted from the profiler for yahoo.com when evaluating over 20K character size. 295
D.9 The Objects results extracted from the profiler for yahoo.com when evaluating over 20K character size. 296
D.10 The Objects results extracted from the profiler for live.com when evaluating over 20K character size. 296
D.11 The Functions results extracted from the profiler for blogger.com when evaluating over 20K character size. 297
D.12 The Functions results extracted from the profiler for baidu.com when evaluating over 20K character size. 299
D.13 The Objects results extracted from the profiler for baidu.com when evaluating over 20K character size. 300

D.14 The Listener results extracted from the profiler for baidu.com when evaluating over 20K character size. 300
D.15 The Functions results extracted from the profiler for wikipedia.org when evaluating over 20K character size. 301
D.16 The Listener results extracted from the profiler for wikipedia.org when evaluating over 20K character size. 301
D.17 The Functions results extracted from the profiler for twitter.com when evaluating over 20K character size. 303
D.18 The Objects results extracted from the profiler for twitter.com when evaluating over 20K character size. 304
D.19 The Listener results extracted from the profiler for twitter.com when evaluating over 20K character size. 304
D.20 The Functions results extracted from the profiler for qq.com when evaluating over 20K character size. 306
D.21 The Objects results extracted from the profiler for qq.com when evaluating over 20K character size. 307

E.1 List of common tell-signs that can be shared among different types of widgets. 309
E.2 List of tell-signs private to Auto Suggest List (ASL) widget. 309
E.3 List of tell-signs private to Ticker widget. 309
E.4 List of tell-signs private to Popup Content widget and Collapsible Panel widget. 310
E.5 List of tell-signs private to Tabs widget. 310
E.6 List of tell-signs private to Carousel widget and Slide Show widget. . 310

List of Figures

1.1 The transition process of a Tabs widget when a panel is loaded in http://uk.yahoo.com. 21
1.2 The components that form WIMWAT project. 23
1.3 An example of the Auto Suggest List widget by Google.com 26
1.4 Difference between SASWAT and WPS lifetime. 32
1.5 Interface for the Older User Widget Assistant. 34
1.6 Tabs on the Yahoo! home page. 35

2.1 The iterative process to develop and maintain an application. 51
2.2 Reverse and forward engineering cycle 56
2.3 The architecture used in Di Lucca et al. [2005a] for the Web Interaction Design Pattern Recovery System 61
2.4 The overall architecture of the approach for DP-Miner 63
2.5 The overall architecture of the automated process that transforms the UML model of a design pattern application into its evolved form . . . 65
2.6 The identification process of UWA entities and semantic associations types in [Bernardi et al., 2008] 66
2.7 Architecture for the Hyperbase model abstractor module in Bernardi et al. [2008] for the Re-UWA environment. 67

3.1 Carousel widget from Sky.com 79
3.2 Slide Show widget from Yahoo.com 80
3.3 Widget Identification Ontology (WIO) paradigm 81
3.4 Taxonomy of the Widgets layer. 83
3.5 Taxonomy of the Components layer. 84
3.6 Taxonomy of Visible components objects in the Components layer. . . 85
3.7 Taxonomy of Visible Buttons components objects in the Components layer. 86

3.8 Taxonomy of Code components objects in the Components layer. . . . 86
3.9 The subset taxonomy of manipulated DOM objects under the Code components objects in the Components layer. 87
3.10 Taxonomy of the Tell-Signs layer. 89
3.11 Taxonomy of the Code Constructs layer. 91
3.12 Visualisation of a Carousel widget process. 97
3.13 The use of Collapsible Panel widget in `kayak.co.uk`. 99
3.14 The use of Popup Content widget in `AOL.co.uk` website. 102
3.15 Visualisation of a Slide Show widget process. 104
3.16 The use of Tabs widget in `NYTimes.com`. 105

4.1 An example of ASL widget (non-shade region) in action on Google.co.uk 115
4.2 ASL widget selection moving up 116
4.3 ASL widget selection moving down 117
4.4 ASL widget detection's tell-sign process flow chart 121
4.5 An example of the Carousel widget (non-shaded region) in action on blogger.com. The arrows are used to move to the previous or next item. 125
4.6 An example of the Slide Show widget (non-shaded region) in action on aol.com 126
4.7 An example of the Tabs widget (non-shaded region) in action on yahoo.com 127
4.8 Carousel widget increment tell-sign (ts1) detection flow 128
4.9 Carousel widget detection's tell-sign process flow chart 132
4.10 Static graph examples for Carousel widget?s UI components 136

5.1 Overview of the browser interpretation process. 144
5.2 Widget Prediction System (WPS) overall architecture. 151
5.3 Widget Prediction System (WPS) Code Profiler block's internal process flow. 153
5.4 Data structure to store the functions found in the JavaScript code of a Web page. 156
5.5 Data structure to store the objects found in the JavaScript code. 158
5.6 Data structure to store the event listeners found in the JavaScript code. 160
5.7 Data structure to store the event listeners factory method found in the JavaScript code. 162

5.8 The architecture and connections of the Tell-sign's Object Block that is found in the overall WPS architecture as seen in figure 5.2. 164
5.9 Data structure to store the tell-signs found in the JavaScript code. . . . 164
5.10 Website's default page character size distribution for the top 30 most popular Websites ranked by Alexa Top 500 Global Sites selected on 09 March 2011. 167
5.11 The final WPS Popup Content and Collapsible Panel widget prediction process flow for each candidate. 178
5.12 The final WPS Auto Suggest List (ASL) widget prediction process flow for each candidate. 181
5.13 The final WPS Tabs widget prediction process flow for each candidate. 183
5.14 The final WPS Carousel and Slide Show widget overall prediction process flow for each candidate. 185
5.15 The final WPS Carousel and SlideShow widget 'Next' button prediction process flow for each candidate. 186
5.16 The final WPS Carousel and SlideShow widget 'Back' or 'Previous' button prediction process flow for each candidate. 188

6.1 The setup of WPS as an Add-On in Mozilla's FireFox Web browser. . 192
6.2 Prediction results for Ticker widget 195
6.3 Prediction results for Auto Suggest List (ASL) widget. 196
6.4 Prediction results for Popup Content widget. 198
6.5 Prediction results for Collapsible Panel widget. 199
6.6 Prediction results for Tabs widget. 201
6.7 Prediction results for Carousel widget. 202
6.8 Prediction results for Slide Show widget. 203
6.9 Simplified 'Next' and 'Previous' buttons concepts. 205
6.10 Prediction results when Carousel widget and Slide Show widget are combined. 205
6.11 Prediction results when Popup Content widget and Collapsible Panel widget are combined. 207
6.12 Comparison of Popup Content widget results with and without `digg.com` 208
6.13 Prediction results when Popup Content widget and Collapsible Panel widget are combined without `digg.com`. 209
6.14 The usage of common tell-signs analysed from our evaluation results. 210

6.15 The usage of specific types of widget tell-signs analysed from our evaluation results. 212

7.1 WPS interface as a service. 218

Abstract

AUTOMATIC WEB WIDGETS PREDICTION FOR WEB 2.0 ACCESS TECHNOLOGIES
Alex Qiang Chen
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2013

The World Wide Web (Web) has evolved from a collection of static pages that need reloading every time the content changes, into dynamic pages where parts of the page updates independently, without reloading it. As such, users are required to work with dynamic pages with components that react to events either from human interaction or machine automation. Often elderly and visually impaired users are the most disadvantaged when dealing with this form of interaction. Operating widgets require the user to have the conceptual design knowledge of the widget to complete the task. Users must have prior experience with the widget or have to learn to operate it independently, because often no user documentation is available.

An automated Widget Prediction Framework (WPF) is proposed to address the issues discussed. It is a pre-emptive approach that predicts different types of widget and their locations in the page. Widgets with similar characteristics and functionalities are categorised based on a definition provided by widget design pattern libraries. Some design patterns are more loosely defined than others, and this causes confusion and ambiguity when identifying them. A formal method to model widgets based on a Widget Ontology was developed. The paradigm of the ontology provides a framework for developers to communicate their ideas, while reducing ambiguity between different types of widget. A Widget Prediction System (WPS) was developed using the concepts of the WPF. To select the types of widget for WPS evaluation, a widget popularity investigation was conducted. Seven of the most popular widgets from the investigation, done across fifty Websites, were selected. To demonstrate how WPF can be applied to predict widgets, fifty websites were used to evaluate the feasibility of the approach using WPS. On average, WPS achieved 61.98% prediction accuracy with two of the widgets $> 84\%$ accuracy. These results demonstrated the feasibility of the framework as the backend for tools that support elderly or visually impaired users.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

Acknowledgements

Most of all, I would like to thank my supervisors Simon Harper and Andrew Brown for their guidance, advice and support. During the course of this research, they have fostered an environment that has engaged me in interesting and enthusiastic discussions, that has often been a source of great inspiration. I am also grateful to Dmitry Tsarkov for verifying the Ontologies and providing his invaluable advice.

I would like to extend my gratitude to the members of the Web Ergonomics Lab, especially Darren Lunn for his advice when formulating my concepts, and many inspirational discussions.

Most importantly: my family – particularly my parents for their constant encouragement and support, which I am indebted.

Chapter 1

Introduction

There is a myth that providing accessible content equates with delivering content that can be reached and interpreted by people with all types of physical, sensory or cognitive capabilities. As the World Wide Web (Web) becomes overly a visual and interactive medium, we will explore how this myth of accessible content has affected the Web, the existing guidelines, and tools to improve Web Accessibility. We believe more needs to be done to narrow the gap, so that the myth of providing accessible content will be inline with the constantly evolving Web technologies and trends. A novel approach is presented that uses a framework to predict the regions in the Web page where Web components with a specific purpose react to events from humans or machines (widgets). By providing users with the knowledge to operate widgets, we aim to improve the user experience when interacting with the page indirectly.

The technological advancement of the Web is evolving at a fast pace due to its popularity and necessity in our everyday lives. A collection of static content pages was used to convey content to the users during its infant years. This form of content delivery requires the page to be reloaded every time the content has changed, so that the user will be up to date with the current information or task. Due to technological advancement and demand, Web pages have become more interactive and have evolved into dynamic pages, where parts of the page can be updated independently without reloading the page.

Internet applications such as e-mail, social networking services and instant messaging, which we interact with on a daily basis, highlight the growing importance of the Web. Furthermore, the demand for quick real-time information has been seen across

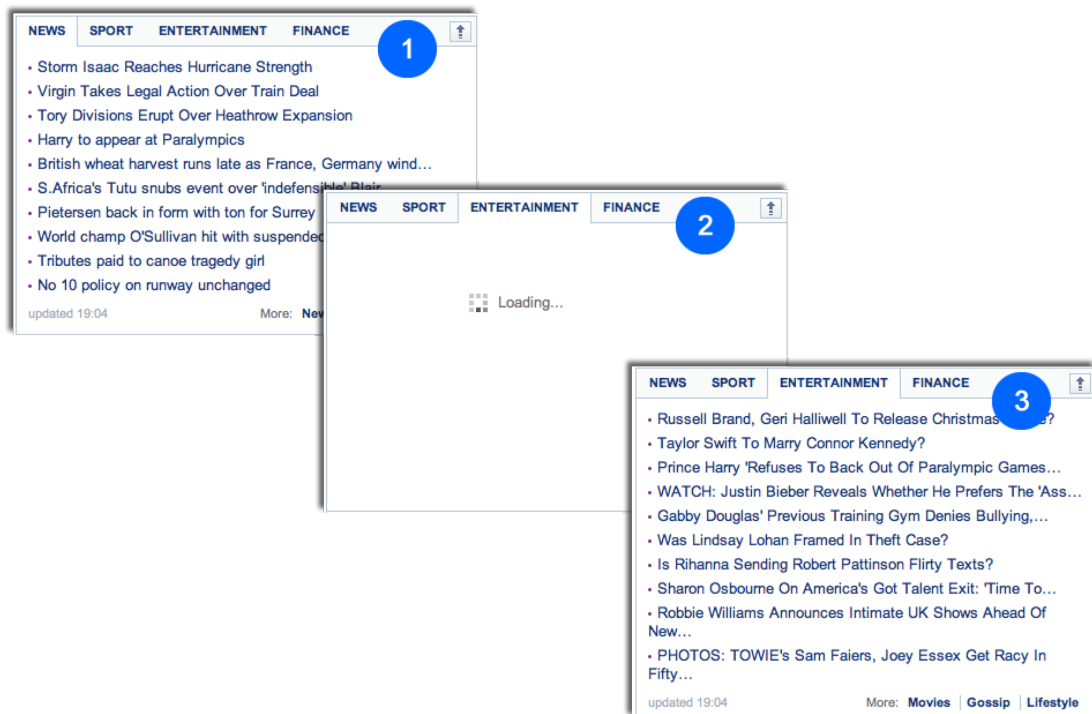


Figure 1.1: The transition process of a Tabs widget when a panel is loaded in http://uk.yahoo.com/. ① When Web page is first loaded and by default the 'News' tab is selected. ② When the user clicks on the 'Entertainment' tab, the content of this tab is loaded using remote scripting. ③ When 'Entertainment' tab content is fully loaded.

many news channels (such as the BBC[1], Reuters[2], CNN[3]) and social networking (such as Facebook[4], LinkedIn[5], Orkut[6]) Websites.

Developers often use Remote Scripting techniques to control and update small sections of content within the page to deliver the latest content to the user. This technique is normally deployed using Asynchronous JavaScript and eXtended Markup Language (collectively known as "AJAX"), JavaScript Object Notation (JSON), and client-side

[1]URL: `http://www.bbc.co.uk/news/`. Last accessed on 17th August 2012
[2]URL: `http://www.reuters.com/`. Last accessed on 17th August 2012
[3]URL: `http://edition.cnn.com/`. Last accessed on 17th August 2012
[4]URL: `http://www.facebook.com/`. Last accessed on 17th August 2012
[5]URL: `http://www.linkedin.com/`. Last accessed on 17th August 2012
[6]URL: `http://www.orkut.com/`. Last accessed on 17th August 2012

scripting languages interchangeably. Using these approaches, developers can overcome the heterogeneous nature of the Web and the constant flow of updates to be presented, while freeing the user from constantly reloading the entire page. On the other hand, this form of content delivery requires the user to be able to notice the changes to the sections of content in the page (see figure 1.1). Often it even requires the user to be able to operate the widget to extract the relevant content or accomplish a task. This implies that users have to be conversant with Web technological trends. Often Assistive Technologies struggle to cope to be on a par with the changes. Thus, features Web developers have created are not known to the users of these technologies and mobile devices [Yesilada et al., 2010; Hardesty, 2011].

The evolution of the way content is delivered over the Web requires users to be able to find their way around the page independently. This form of interaction commonly requires the user to learn how to operate the widgets independently, if no prior experience with the page is established. Frequently, users approach the widgets using a trial-and-error method to learn how to operate the widgets. This is because no user documentation relating to the widget is normally available. Furthermore, widgets are often developed specifically for a model of content delivery or part of a bigger process for which the page was created. A type of widget can deviate in appearance and behaviour from page-to-page or task-to-task; however, the overall concepts remain the same. It is left to the users to figure out the perceivable design concepts of the widgets. Depending on the user's cognition ability, the user has to be able to work out how to operate the widget using its components. If these conceptual connections cannot be made, then the user experience the developer wishes to deliver will be lost.

An automated widget prediction approach to assist users to identify widgets on the page is proposed to improve the issues faced when interacting with widgets. This is a pre-emptive approach that will inform users about the regions in a page where the widgets are found, as well as where dynamic content may occur. By providing this information, Assistive Technologies can inform users how to operate the widgets.

Existing Web widget design pattern libraries provide a general idea of how a type of widget should behave and the necessary components that form it. However, the definitions are left ambiguous to accommodate the wide variety of approaches available. Often this ambiguity translates into weakly defined widgets, and when interpreting the design patterns for these widgets it becomes even more loosely coupled because it depends mainly on the developer's perspective and application. Furthermore, the loose method of describing widgets adds another layer of confusion on top of the ambiguous

definition for users, who are normally less informed about the technical aspects.

We propose an approach to modelling widgets formally and assisting developers when developing the widget identification process. This approach provides both documentation as well as a medium for the developers/users to develop a cognitive paradigm of the different types of widgets' definitions. A widget ontology is created from the widget taxonomy introduced, so that widget designs can be modelled as a result. The widget ontology uses a paradigm that models widget designs in four layers. From the top layer, it describes the abstract concepts of the design, then the components that form the widgets in the second layer. To search for the components of the widget from the source code, traces and clues of the components called tell-signs model the components in the third layer. Lastly, the bottom layer describes the actual code constructs that form the metaphor of the tell-signs. We believe the widget ontology will provide the key to a rounded solution for both users and developers, so that concepts of the different types of widget can be shared.

A novel framework called the Widget Prediction Framework (WPF) is an umbrella framework that provides recommendations to guide the development of an automated widget prediction system. WPF covers everything from the development of the definition of widgets all the way to materialising the shared knowledge of the different widgets in the prediction system. Two main phases are suggested: 1) modelling the

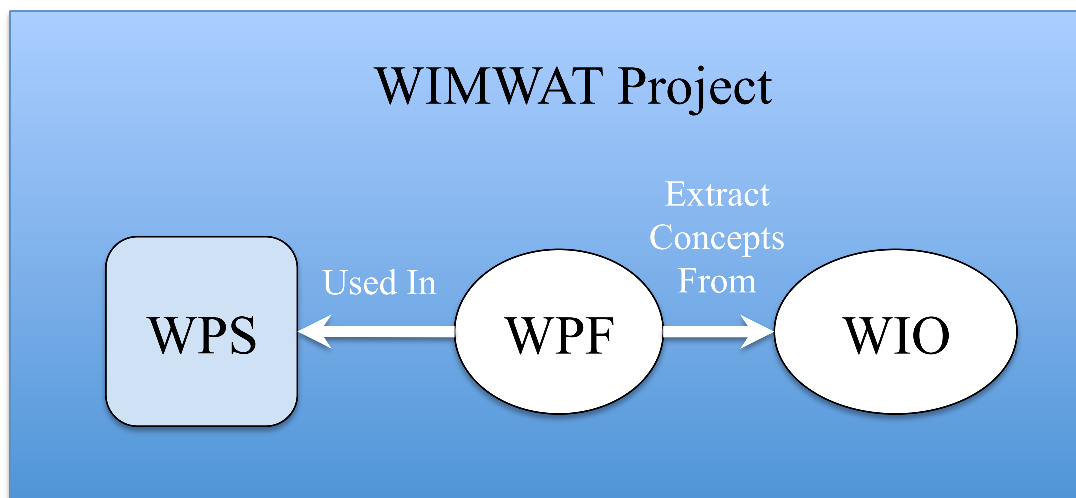


Figure 1.2: The components that form the Widget Identification and Modification for Web 2.0 Access Technologies (WIMWAT) project. Where WPS is Widget Prediction System, WPF is Widget Prediction Framework, and WIO is Widget Identification Ontology.

widgets' definition and 2) implementation of the automated Widget Prediction System (WPS).

In the first main phase, the widget ontology paradigm is used to assist in modelling the different types of widget to form the Widget Identification Ontology (WIO). A strongly typed language[7] is used to define the conceptual models of the widgets. It provides a platform to share knowledge with designers and developers. The ontology is not directly used by the system; instead, the shared knowledge informs the developer/designer about the rules to apply in WPS. These rules are then developed in WPS by manually coding them in main phase two, and the system does not query the ontology when trying to predict widgets. A discussion of how to implement WPS to predict widgets is presented and evaluated to demonstrate the feasibility of the approach in the wild Web. WPS, WIO and WPF are outcomes of the Widget Identification and Modification for the Web 2.0 Access Technologies (WIMWAT) project, to research for solutions to improve Web widget accessibility for Web 2.0 content. Figure 1.2 illustrates the different components resulting from the WIMWAT project. It shows how these components are associated with one another to predict widgets. To select the types of widget to be included in WPS, a study of widget popularity over fifty popular Websites was conducted. Later, seven of the most popular widgets from the popularity study are included in WPS, to evaluate the feasibility of the approach over fifty Websites. The evaluation exposed that the prediction techniques are more accurate for some types of widget than others, depending on the nature of the widget. These results demonstrate that widgets can be identified from the source code, and that detection at the granularity of the widget's components is also feasible. This form of detection granularity will enable tools using the WPS or the WPF, to provide users with the awareness of the location, the components of the widget, and the type of widget in a page. With this information, one can provide insights into the widgets in the page to help Assistive Technologies, and assist the insertion of Accessible Rich Internet Applications Suite (WAI-ARIA) syntax for reverse engineering processes, development, maintenance and adaptation processes. Finally, issues raised by the evaluation are described and suggestions for overcoming them are also presented for future work. Enhancement to the work presented in this thesis and some application of the WPF are also discussed.

The WPF can reside in Assistive Technology, or as an add-on to these technologies and user agents, as well as a RESTful Web service to analyse the set of source code

[7]The Web Ontology Language (OWL) is used to model WIO before the shared knowledge is implemented in WPS.

provided. It can be seen as a backend engine to provide Assistive Technologies with the region in the page where widgets are employed by the developer, as well as predict possible areas in the page where dynamic content will mutate. The success of the proposed research has been applied in two prototype tools (SCWeb2.0 and SASWAT), as a service to assist them in establishing whether the page contains dynamic content. When WPF is applied to an Integrated Development Environment (IDE) for development, it can help developers to predict regions in the code which require WAI-ARIA syntax injection. It also assists developers in checking for a widget's design conformance, as well as assisting in reverse engineering processes for Website maintenance. The range of applications for WPF demonstrates the importance of the research, as well as its significant contributions to the Web community.

1.1 Motivation

Web accessibility guidelines such as the Web Content Accessibility Guidelines (WCAG) and WAI-ARIA, and Section 508 of the Rehabilitation Act in the United States of America, provide recommendations to Web developers and authors for building accessible Web pages. However, before a Web page can be truly accessible, Web browsers, Assistive Technologies and Web pages must align themselves with the guidelines, along with the different corresponding versions.

Developing an accessible Web page is difficult, time consuming and often, developers do not see the benefits of it. Thus, compliance to accessibility guidelines can be hard to achieve. Furthermore, developing widgets that are of an accessible form would be even more difficult and time consuming. This is because the components that form the widget are based around conceptual models employed by developers. Besides these issues, commonly little or no user documentation is available for users. Hence, this makes it difficult for users to interpret the intention of the task, and frequently they are unable to fully utilise the widget.

As an example, we will have a closer look at a type of widget called Auto Suggest List (ASL) to illustrate the importance of being able to operate a widget, and how it will impact the page's accessibility, usability and user experience. The ASL widget is often used to assist users when filling up a form or a search query by suggesting a list of related content that the user may intend to input into the text field. Figure 1.3 illustrates the example of an ASL widget in action when the user has entered a few characters as part of his/her query. Shortly after the user has entered part of the query,

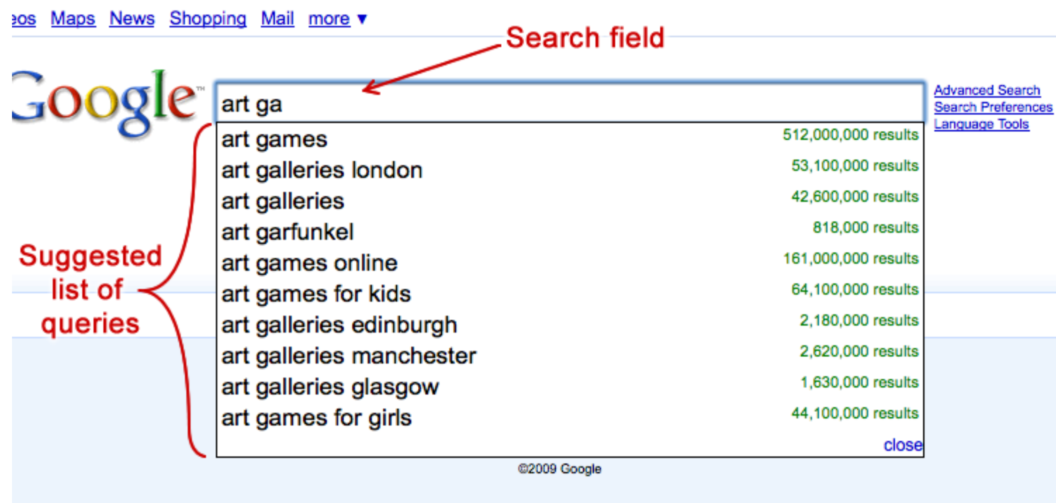


Figure 1.3: An example of the Auto Suggest List widget by Google.com

a list of suggested possible inputs beginning with the same few characters, or which have close relations to the partly entered query, is presented to the user. At this point, the user can either continue to type the remaining query or choose from one of the suggestions presented in the list to complete the search process.

The ASL widget is frequently used to reduce typing/spelling errors, speeding up the querying process, and assisting the user when filling up the search field. However, the interface of this widget requires the user to be able to visually notice the change of content on the page so that they will be able to utilise the widget. Often, users of Assistive Technologies, especially screen readers, are unaware of these changes because a lot of these technologies are unable to keep up with the evolution of the Web [Chen and Harper, 2008]. Developing a widget to be accessible not only makes the content accessible to users of Assistive Technologies, but it also should assist in making all types of users aware of its features.

In order to achieve these concepts, when developing Web pages, considerations of the content accessibility and usability should be taken into account. Since the Web can be accessed by many different kinds of devices (i.e. personal computers and mobile devices). To ensure the widget is usable, developers should also consider the size of the screen and its resolution, and whether the targeted devices are capable of utilising the widget. When these considerations are met, users will be more aware of the widget's content and components, and this will help them to utilise the widget more effectively, thus reducing the initial learning time.

In the past, attempts were made by Brown and Jay [2008b] and Borodin et al. [2008] to help users become aware of micro-content updates, especially for visually disabled users. The Single Structured Accessibility Stream for Web 2.0 Access Technologies (SASWAT) project done by Brown and Jay [2008b] and the Dynamo project done by Borodin et al. [2008] provide awareness to their users that the content has changed but do not assist them in identifying or using the widget. In order to assist users to use the dynamic Webpages, analysing the code within the page is required so that widgets can be identified. If a common definition for different types of widgets can be established, then after the widgets are detected, users can be informed about them so that the widgets can be utilised effectively. These projects, together with the Senior Citizens On The Web 2.0 (SCWeb2.0) project[8] are the main motivators of the WIMWAT project. They helped to scope the initial concepts, and lay out the importance of the proposed research.

During the course of searching for solutions for the proposed research, techniques employed from the immediate fields, along with the surroundings fields, were explored. The matrix approach is used in Dong et al. [2007] study to reverse engineer the design patterns applied during the development phase for desktop programmes, and Stencel and Wegrzynowicz [2008] applied this method to automatically detect occurrences of design patterns in desktop applications. Concepts to reverse engineer widgets from the source code, using instances in the source code as clues to identify the patterns of a widget's design, were inspired. The concepts to model patterns like reusable objects were inspired initially after reading Fowler [1996]. However, the concepts to model widgets based on reusable objects were further formalised after reading the demonstration by Presutti and Gangemi [2008], where the composition of ontology design patterns for Semantic Web technologies was described. Using the idea of developing ontology to encode concepts rather than logical design patterns, a combinatory notion to apply the concepts of the ontology, to assist widgets classification during the prediction process was further developed. These concepts led to a conceptual model to provide a paradigm for developers and users to base their work on, so that the concepts of a widget could converge.

Before Assistive Technologies can advise their users about objects and content in the Web page, these technologies need to know what is contained in the page. Identifying the widgets that orchestrate different sets of micro-content is crucial when assisting

[8]SCWeb2.0 – `http://wel.cs.manchester.ac.uk/research/scweb2/`. Last accessed on 3rd September 2012

users in understanding how to use and access dynamic micro-content. This information will allow the user or other tool to have a clue about the components in the page and how the different components operate. Widgets are conceptual models to organise presented information, thus reverse engineering the code to derive the concepts of the widget will be challenging.

Reverse engineering the design concepts of different widgets is difficult. This is because design patterns often get lost within the code when applying them during development [Gamma et al., 1995]. More recently, Dong et al. [2008b] reiterated the difficulties of tracing the design's information from the source code, and compared different existing design pattern mining techniques including their DP-Miner [Dong et al., 2007] which uses matrices to discover design patterns. The DP-Miner approach is suited to desktop applications. It does not cover the same degree for Web applications, let alone Web widgets.

Web applications are heterogeneous combinations of technologies to enable it to work. They can be broken down into smaller processes, located at different locations (Web services), and every Web service can be created using a different technology. More recently, Web applications are appearing in smaller forms, commonly known as widgets, which deal with small tasks, and often more than one of these widgets may coexist in a page. Breaking down the tasks into smaller parts allows the developers to provide real time interaction with the user. Thus, the complex nature of incorporating widgets into a page requires a different approach to reverse engineer them.

Widgets enable better interaction but, like all applications, users need to go through an initial training or trial-and-error phase before they are capable of utilising the widget effectively. In the process of making the Websites more interactive and coping with the pace of Web trends, commonly developers do not provide any form of user documentation. This negligence often becomes a barrier for their users. Quite often inadequate or outdated design documentation causes all sorts of problems for assistive technologies, and as highlighted by Di Lucca et al. [2006], the issues with documentation also affects the maintenance and evolution of the system.

Attempts were made by Bernardi et al. [2008]; Rossi et al. [2008] to recover conceptual models from Web applications. Rossi et al. [2008] suggested that, if it was possible to identify the model of a Rich Internet Application (RIA), then this information would be able to assist refactoring RIAs. They discussed their anecdotal concepts to recover the model of the Carousel widget only. However, Rossi et al. [2008] did not continue to demonstrate, in practice, how their approach could be applied in a

system to automatically detect the Carousel widget. Neither did they do an extensive evaluation to prove the feasibility of their concepts. Bernardi et al. [2008] has shown the possibility of reverse engineering the Web applications, to derive the conceptual design expressed in conceptual modelling languages such as Unified Modelling Language (UML) and Web Modelling Language (WebML) [Ceri et al., 2000]. This type of modelling language is often very expressive, and it requires a lot of computation power to compute and form these models. Thus, they are more suited for Ubiquitous Web Application (UWA) [Finkelstein et al., 2002] or non-realtime processes. On the other hand, identifying widgets does not necessarily need so much information to be processed as demonstrated by Chen and Harper [2009]. This is because widgets are specific purpose Web components that process a limited amount of tasks for users to interact with the page.

Identifying widgets and components in the Web page is not the only approach to assisting users with interpreting and accessing content on the page. Earlier, projects like the Structural-Semantics for Accessibility and Device Independence (SADIe) also inspired WIMWAT with the approach of transcoding the page content so that the inaccessibility of Web pages can be overcome without the need to annotate them [Harper et al., 2006]. The applicability of SADIe was demonstrated in Lunn et al. [2009a] where CSS annotations were transcoded to generate Access-Enabling AJAX (AxsJAX) code, and later in Lunn [2009] towards Behaviour-Driven transcoding of Web content by analysing the coping strategies employed by users. These approaches only 'tame' the static content presented on the page, while WPF deals with dynamic content and widgets that are widely used to incorporate Web 2.0 concepts. Studies like SADIe exposed the gaps that methodically devised part of the widget prediction framework concepts. However, to deal with the dynamic nature of the content and widgets, a deeper analysis of the code is required. The research extends the analysis to the behaviour and content components of the page as well. The applications for WPF are also partly fuelled by SADIe. This includes assisting developers and Assistive Technologies to inject WAI-ARIA syntax or AxsJAX code into the components that interact with the user, as well as the content manipulated by the widget. Unlike SADIe, WPF can locate the components of the widget to increase the granularity of the injection, hence providing users with the conceptual application of the widgets and improving their experience.

The SASWAT project provides inspirational techniques and approaches for the WIMWAT project. SASWAT aims to provide a framework for mapping the areas on a

Web page where competing dynamic micro content is produced by Web 2.0 technologies to audio Brown and Jay [2008b]. These aims can be broken down into five major objectives[9]:

- Determine the visual experiences of sighted users when attempting to assimilate the information of interactions presented in the micro content.
- Research the nature and evolution of the Web infrastructure in its transition from static to dynamic Websites.
- Using the results from the previous goals, develop a model of the Web interaction with which a mapping from sighted to visually impaired can then be constructed.
- Create a framework to mediate between the competing demands of the micro content present on the Website.
- Evaluate this system in a repeatable experiment to validate the findings.

Presently, the approach for the SASWAT project monitors and tracks the Document Object Model (DOM) for content updates in a Web page [Brown and Jay, 2008b]. The changes in the Web page are characterised into four categories for further analysis. User's focus and user's preferences are also accounted for. This information will help the system decide whether the user has interest in a piece of updated information. This is a reactive approach to the dynamic content problem. Often users have to wait until the content has mutated before they can be notified about the changes.

From a recent evaluation by Brown [2012], it can be noticed from the evaluation transcripts: providing users with knowledge of the particular type of widget were deemed to be helpful when they interacted with the page. These results highlight the practical aspects of the proposed research and an avenue for its application. Due to the nature of the proposed research, it suggests that collaborations between WIMWAT and SASWAT should be done to investigate the feasibility of providing information about the widgets to the user.

The objectives of the SCWeb2.0 project is to investigate how ageing users perceive and interact with Web pages that use Web 2.0 technologies, then suggest conclusive interventions that will improve the interactivity issues faced by these users[10]. As reported

[9]Project Aims of SASWAT — `http://wel.cs.manchester.ac.uk/research/saswat/`. Last accessed on 3rd September 2012

[10]Project objectives of SCWeb2.0 — `http://wel.cs.manchester.ac.uk/research/scweb2/`. Last accessed on 3rd September 2012

in Lunn et al. [2009b], the older Web users evaluated faced orientation and scrolling issues on static Web pages. The study went on to suggest that, due to the interface complexity such as changes in content and context, additional problems might arise in Web 2.0 pages. Efforts by the World Wide Web Consortium (W3C) WAI-ARIA guidelines, and the fifth major revision of the HyperText Markup Language (HTML5) specifications, attempted to improve the situation. However, the fruits of these efforts will not be immediate, as discovered in a study by Harper and Chen [2012]. It was also reported that some of these standards/recommendations, especially those relating to making Web content accessible, never get taken up much by the Web community. Studies such as Lunn and Harper [2011] attempted to address the issues pertaining to older users' interactions and perceptions of Web 2.0 technologies with the SCWeb2.0 Assistance Tool in their study. These tools can benefit from WPF to inform their users about Web 2.0 technologies employed in the Web page.

The insights into the interactivity problems and content accessibility raised by Web 2.0 technologies were highlighted in the SASWAT project. It demonstrated the feasibility of identifying changes in dynamic content by monitoring the DOM of the Web page. It was shown in the SASWAT project how accessibility and usability for Web pages with Web 2.0 technologies can be improved by providing users with the areas in the page where dynamic content updates. However, it also reveals the insights of users' experience when interacting with dynamic content and widgets.

Unlike SASWAT, WPF is a pre-emptive approach that not only identifies the regions where dynamic content may occur, but also the widgets that orchestrate the content. The WIMWAT project takes SASWAT ideas to another level. Now, User Interface (UI) objects and areas in the page where dynamic content is found are analysed to deduce the expected process of the components in the widgets. WPF is dependent on the shared knowledge provided by the Web widget ontology, to code WPS, so that the conceptual models of widgets' components can assist the widget prediction process. The Web widget ontology was created based on textual information within the source code and DOM, and concepts of processes are derived from past textual information found during each analysis. The textual information analysed includes the hypertext content, styling code and behaviour scripts. Finally, after the widget is determined, the location of the widget in the page is deduced.

Indeed, WPS is not an iterative process: it is a pre-emptive approach to establish if the page contains any dynamic content by predicting widgets in it. The prediction process is normally done at the first instance after the Web page is visually loaded, but

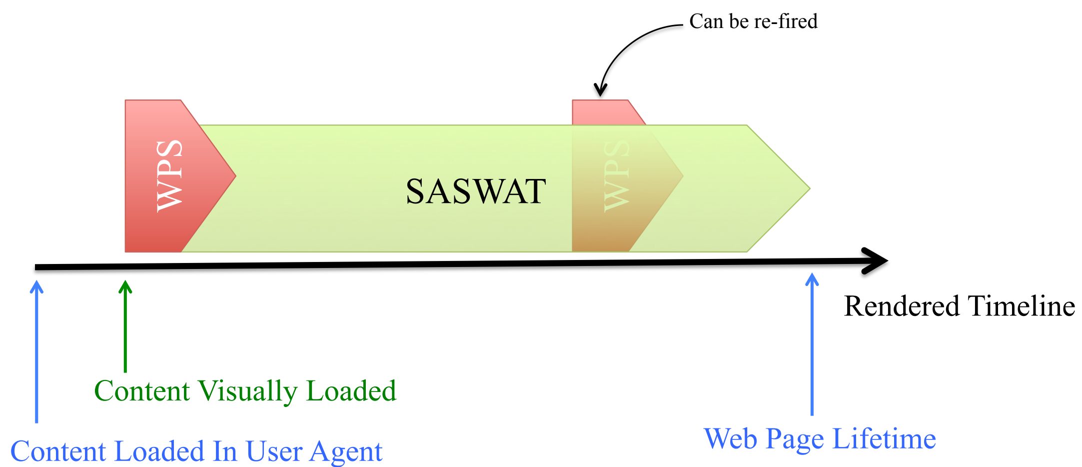


Figure 1.4: Difference between SASWAT and WPS lifetime.

it can be called again at a later stage in the page's lifetime for each load whenever it is required. Both projects approach dynamic content accessibility issues from different ends. Another difference is that SASWAT executes parallel to the page's lifetime as depicted in Figure 1.4. The rationale behind WPF is to predict the types of widget and their location within the page. As the lifetime of the page increases and after the user interacts with the page, more clues of the widget may become available, thus the prediction rate can be improved if WPS is re-fired at a later stage.

The possibilities of applications for WPF put forward a strong case for the proposed research discussed above. These ideas have led to two prototype tools [Chen et al., 2013], to assist users as they interact with dynamic content, to incorporate WPF as part of their processes. These tools, discussed below as use cases, provide older users and visually impaired users with improved access to Web 2.0 content.

1.1.1 Supporting Older Users of Dynamic Web Content

When users are faced with differing types of content, they have the difficulty of dividing their attention between the elements to complete tasks effectively [Sàenz et al., 2003]. This additional load is a major problem for an ageing population of knowledge workers expected to work longer into old age. The general effects of ageing include changes in attention, cognition and behaviour, all of which affect how people use the Web [Hanson, 2009]. Studies have shown that elderly Web users experience

a heightened cautiousness and a hesitancy about making responses that may be incorrect [Kurniawan et al., 2006; Memmert, 2006]. In addition, elderly users show difficulty in maintaining attention, focus, and concentration on tasks where there is a lot of distracting information [Hartley, 1992].

During Galvanic Skin Response (GSR) studies, to identify the stress levels of older users, Lunn and Harper [2010] observed that, unlike younger participants, there was a large variance within the results for the older user groups and that there were signs from some users of hesitancy and uncertainty when completing the tasks. Based on these results, a prototype tool was developed in SCWeb2.0 project to assist older users as they interact with dynamic content. The tool is implemented as a Web Browser extension to allow for rapid prototype development and also to allow users to feel more comfortable. As Hanson and Richards [2005] note, *"users tend to prefer a standard browser with the accessibility transformations added rather than a specialised browser offering only a limited set of features (which would also tend to mark them as being disabled)."*

As the page loads, the tool applies WPF to establish if the page contains any dynamic content. If widgets are present in the page, then an information icon is displayed on the browser to allow users to receive help if they require assistance, ① in Figure 1.5. If users are comfortable with interacting with dynamic content, then they can ignore the information icon.

The area ② shows the panel that is displayed when the user clicks on the assistance icon. A list of the widgets that have been detected is displayed. In this case, three widgets were identified – Auto Suggest List, Carousel and Tab Box. The user clicks on those buttons to receive help if they do not understand what content is present. In the example shown, the user has clicked on the "Tab Box" button. This results in a short paragraph appearing explaining what a tab box does along with a demonstration video. As the video is being played, the widget on the page is highlighted in a pink circle, ③. This is to ensure that users are aware of what part of the page is being discussed in the demonstration video. While widgets have similar functionality, they may have slightly difference appearance. The video is designed to be as generic as possible and the pink highlight draws the user's attention to the widget on the page that is being talked about.

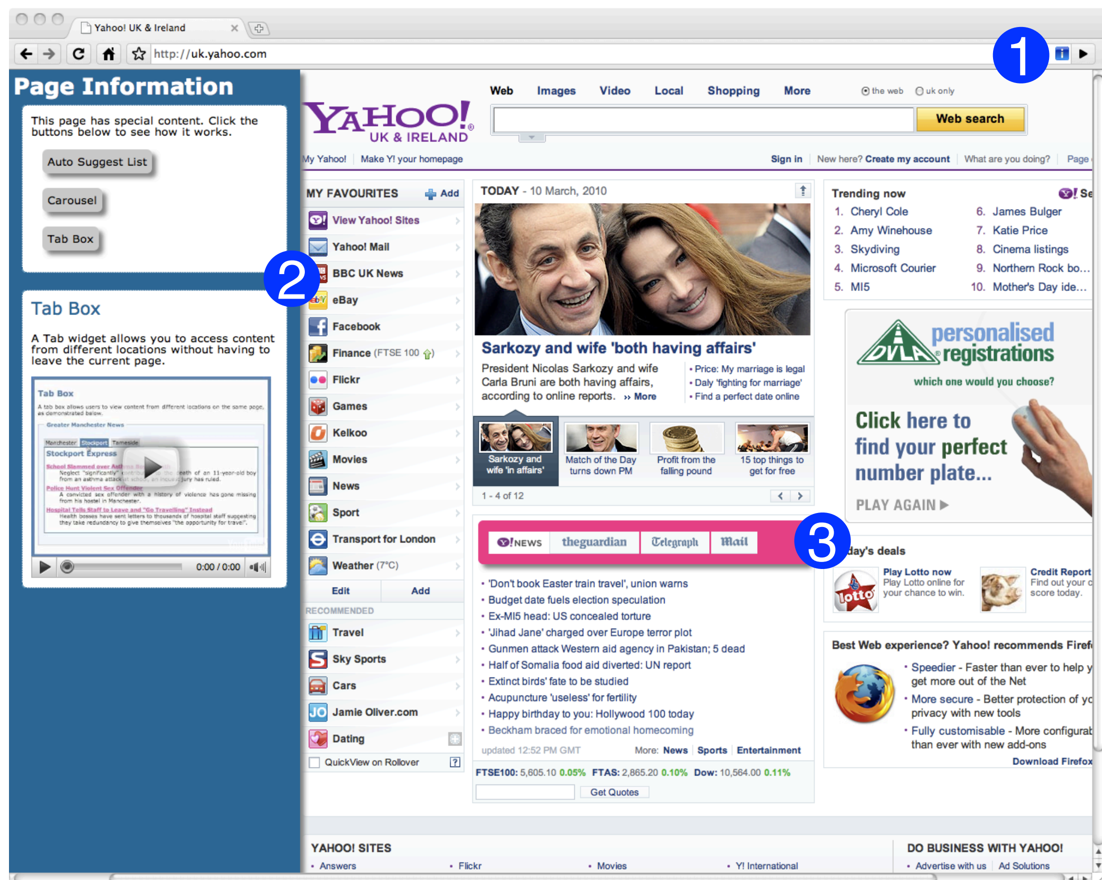


Figure 1.5: Interface for the Older User Widget Assistant.

1.1.2 Supporting Visually Impaired Users of Dynamic Web Content

The second use-case for widget prediction as an aid to accessibility explores how identifying widgets as coherent high-level units, rather than low-level controls, could benefit screen reader users. Eye-tracking studies of sighted users have given insight into how people interact with, and benefit from, different types of dynamic content [Brown et al., 2010, 2009], allowing development of rules for presenting updates according to how they were initiated and their effect on the page [Jay et al., 2010]. During evaluation of these rules [Jay et al., 2010] the nature of each of the test pages, and their dynamic content, was explained to the blind and visually impaired participants. This was noted to be beneficial to the users, as it gave them a more accurate mental model of how the interactions were likely to work, and thus enabled them to use the widgets more effectively.

Sighted users are able to scan a page, or a section of a page, rapidly, perceiving the

Figure 1.6: Tabs on the Yahoo! home page.

content at a glance. Visual clues, such as borders and spacing, enable them to identify units within the page, and layout conventions help them to recognise the type of widgets present. For example, the Yahoo! home page contains an area with news content, organised into 4 sections, and presented as tabs; clicking on the tab title changes the content (Figure 1.6). This section is clearly separated from the surrounding content by spacing and a border, and the tab titles are styled in such a way that most sighted users will recognise the metaphor and understand how to interact with it.

The clues are, however, mostly implicit, so those people who wish to browse these pages using a screen reader miss many or all of them, and are thus unable to recognise the content. The visual differentiation is not available to them, so it is not clear that they are navigating into a new region (that may be a widget). The ability to glance over an area is also much diminished, so relating components to each other is more difficult, particularly as the 2-dimensional visual layout is translated into a 1-dimensional audio one. For these people, therefore, who encounter a linear sequence of content and controls, un-grouped and undifferentiated from surrounding content, it is very difficult to recognise the form and function of a widget. Understanding its use is thus more difficult, as is forming an expectation of the result of any action. Coupled with the lack of information current screen readers provide when content changes, this makes using

these widgets a demanding experience.

A prototype application is thus underway that will utilise the tell-sign approach to widget recognition and apply it to making the presence of widgets explicit to screen-reader users. This is being done in two ways: by inserting text into the DOM so that the user is informed when entering or leaving a widget, and by modifying the widget controls (for example AxsJAX [Chen and Raman, 2008]) so that their effect will be more explicit. Although development is underway and user-testing is yet to be performed, it is expected that these techniques will mean that screen reader users will no longer have to interact with unrelated low-level controls, but that they will encounter widgets at a high-level, thereby understanding better the function and effects of the low-level controls within. This should lead to more confident and efficient use of dynamic content.

1.2 Technological Freeze

The constant evolution of the Web, both technologically and conceptually, makes it exciting as well as difficult to manage. Often this is driven by the desire to make Web pages more interactive and attractive to users. An example would be the instant search feature introduced on 08 September 2010 by `Google`. This feature allows the user to retrieve related results of queries as he/she types. However, this feature not only updates the search results returned, but also changes the landscape of the page, both visually and structurally.

The instant search feature is an example of how widget design constantly evolves. In order to conduct a sound investigation, and to ensure that the experiments are repeatable, a technological freeze is introduced. It was decided that the technological freeze for this thesis would begin from the beginning of the third year of this PhD. This means only Web technologies before September 2010 and after September 2008 will be considered. The technological freeze will enable us to deal with a consistent set of widget concepts within the specified time frame for close examination. Since WPF is expandable, the evolution of widget designs can be added, modified or removed in future, and the technological freeze will not affect the concepts of the framework.

1.3 Research Questions

This thesis attempts to answer these research questions:

1. **Can widgets be formally modelled using a set of well-defined patterns?**
   To help classify the widgets, widget's interaction design patterns from YUI[11] and Welie.com[12] were employed to identify the widget for further investigation. From our widget popularity study, we found that many patterns were not well defined. Axtell et al. [2007] attempted to model Web applications and Mikkonen [1998] attempted to formalise design patterns, were essential but too idealistic to make useful semantics of the widget process, or else the widget definition would be ambiguous in many scenarios for the diverse possibilities of the Web. We introduce the concept to include the processes and the behaviour of the widget in the widget design pattern. Using these additional parameters, we believe we will provide a more accurate definition for classifying widgets. It will also provide developers with a framework to communicate more effectively and reduce false expectations.

2. **Is it possible to discover tell-signs of different components from a widget in the Web page source code?**
   An initial investigation was conducted to test the feasibility of identifying the tell-signs of a widget's component from the source code. By manually coding the knowledge from the widget ontology in WPS, widgets can be predicted from rules applied when discovering the components. Answering this research question will give insights and sufficient evidence to expand and refine the approach to take on research question 3. To achieve this, a basic demonstration to model and classify a widget must be satisfied by providing a preliminary evaluation to show that the concept of automatically identifying the tell-signs of a component from the source code is feasible. The demonstration will hold the key to further expansion of the widget classification approach and provide insights into the automatic detection methods, along with incorporation of classifying the widget with the detection methods.

[11] Yahoo! User Interface Library - `http://developer.yahoo.com/yui/2/`. Last accessed on 18th August 2012

[12] Welie.com Patterns in Interaction Design - `http://www.welie.com/patterns/`. Last accessed on 18th August 2012

3. **Can we build a framework to automatically identify the widget?s tell-signs and predict the widgets in the Web page?**
   The evaluation section of this thesis discusses the finer aspects of the automated detection possibility. Although some widgets can be more difficult to discover than others, the prediction rate shows the positive aspects of the approach and encourages further research to refine certain tell-signs detection approaches, so that the prediction rate can be improved. To answer this question, evaluation of different types of widget over the popular Websites will demonstrate the proof-of-concepts reflective of the wider Web, and expose the insights of our approach. If questions 2 and 3 are answered, we believe this will prove that widgets can be identified from the Web page source code using our tell-signs method, and the framework that is symmetrical for both ends of the technical abilities (abstract and in-depth) to reach a common understanding of a type of widget, reducing the ambiguity between concepts to the minimum.

1.4 Thesis Structure

More detailed descriptions relating to the research, experiments and key findings are presented in the rest of the thesis. The organisation of the remainder of the thesis is as follows:

Chapter 2 - Background and Related Work: A critical exploration of the existing literature reveals that Web Accessibility still has much room for improvement from the dynamic content perspective. This is mainly due to the fast pace at which the Web evolves and its heterogeneous nature. We covered how Web pages are developed and the steps taken to quality control the page's accessibility. Up till now, very little research has been conducted to predict widgets from a Web page source code. Existing techniques can only partially detect the components of the widget in an ideal case and they do not take into account the diverse approaches one can take to developing the widget and component. We explored different techniques to reverse engineer design patterns and design concepts from the source code, and to search for more ideas. Many studies have shown that recovering the design patterns from the source code is difficult because it is difficult to reconstruct the intention or perception of the developer from the code that will lead to design patterns that can be discovered. These insights are important for our research work since other techniques besides using

the source code can be applied to aid the prediction of widgets. Further exploration of modern techniques will also be investigated to assist us in modelling and classifying widgets.

Chapter 3 - Classifying Web Widgets: In this chapter, a widget taxonomy and an ontology of different types of widget to provide a framework to formally model and communicate widget designs are introduced. A manual investigation was conducted to understand the popularity of the different kinds of widget, so that the types of widget to be included in our investigation can be identified. The study consists of a tabulation of the population of different types of widget found in the top 50 Websites. At this stage, we used the definition described by popular Web widget design pattern libraries and JavaScript libraries to aid the investigation. Through this study, we found that commonly definitions provided by these libraries are ambiguous and this often makes it difficult to distinguish between the widgets. Among the types of widget analysed in this study, we have noticed that seven types of widget are more popular than the rest. An ontology that describes each of the seven widgets suggested is discussed in this chapter. Using the tell-signs as predicates in the ontology, if all predicates are met, a widget can be modelled and distinguished from another. Using this framework, both developers and users will be able to communicate more effectively and reduce false expectations.

Chapter 4 - Feasibility Investigation for Using Tell-Signs: Two widgets of the pool of widgets selected in Chapter 3, which we perceive to be more difficult to identify, were used to investigate the possibility of detecting their tell-signs from the Web page source code. In this chapter, we present the methodology applied in the feasibility investigation and the results of this pilot study. Issues with false positives and false negatives were reported, demonstrating the positive feasibility of the approach. From the evaluation results, it is suggested that, due to the complexity of reverse engineering the widget for prediction and the heterogeneous nature of the Web, no one technique is sufficient. A fusion of techniques was suggested and described to overcome the issues raised.

Chapter 5 - Predicting Widgets On A Web Page: Here we describe the technical aspects of developing a predictive tool based on the WPF. The framework is a build up from concepts introduced in Chapters 3 and 4, along with the recommendations and results from their evaluations suggested. The ontology is applied as a concept in the code to infer different types of widget. Using the concepts from

the ontology, we can distinguish one widget from another if all required components exist in the code or all the predicates of the widget are met. Since only the first instance of the Document Object Model (DOM) is analysed after the page is visually loaded, a confidence percentage is given for all candidates for a type of widget. This is because not all the candidates will exhibit all the required tell-signs. However, these clues may appear later during the page's lifespan. Therefore, a confidence percentage method is presented to predict widgets from the available candidates. Finally, we present how the concepts of WPF are implemented, the issues faced during implementation, and the techniques applied to improve or overcome the issues suggested in Chapter 4.

Chapter 6 - Widget Prediction Evaluation: This chapter reports the predictive capabilities and accuracy of the Widget Prediction System (WPS) which is based on the WPF. The quantitative and qualitative evaluations conducted show that the tool is able to predict the widgets on Web pages, and of predicting the location/element within the page where the widget lies. We noticed that some tell-signs of widgets could be missing from the DOM in the first instance after the page is visually loaded. These candidates often have a lower confidence level and they will need further clues to detect them. However, due to the confidence percentage methods, these widgets are also captured. From the results suggested, the threshold confidence percentage for each type of widget is presented. The issues reported suggest future research to improve our prediction methods, but the evaluation results showed evidence that our approach is feasible and future research should be pursued.

Chapter 7 - Conclusions and Future Work: In this chapter, we recap on the key contributions of our research. We envision that the findings discussed in this concluding chapter will help to address misconception issues faced by Web developers, designers, authors and users, and provide an alternative to improve the accessibility issues pertaining to the research proposed. We acknowledge that the WPF is not perfect and more research is required to resolve the predictive accuracy issues which arose. This suggests possible future directions for the work described in this thesis and we discuss them accordingly in this final chapter.

1.5 Publications

The research described in this thesis led to 5 technical reports and 4 peer-reviewed publications. The code for WPS as a Firefox add-on can be downloaded from `https://bitbucket.org/webergonomicslab/wimwat` and WPS evaluation results can be found in `http://wel-data.cs.manchester.ac.uk/data_files/8`, while the full ontology is available for download at `http://wel-data.cs.manchester.ac.uk/data_files/9`.

i. Simon Harper, Alex Q. Chen. "Web Accessibility Guidelines: A Lesson From The Evolving Web", World Wide Web, 15(1) pp.61–88, 2012
There are many aspects to Web accessibility that can affect the accessibility of a Web page. A broad study to understand the Web's technological trends and problems was conducted over a ten year longitudinal study. This study provided important insights into the current Web trends and helped us to understand the current technological barriers and issues, especially the uptake of WAI-ARIA and validation of individual pages for accessibility. The contribution to this paper includes the analysed information that provided the Web User Interface (UI) technological and recommendations trends, and uncovered the extent of Web 2.0 technologies adaptation. However, the analysed data also highlighted that Websites do not always conform to Web accessibility guidelines (See §1.1). This paper provided an opportunity to solidify the scope and broad direction for the thesis by highlighting the specific problem areas in writings, which some of these contributed to the Background and Related Work chapter (see Chapter 2).

ii. Alex Q. Chen, "Widget identification and modification for web 2.0 access technologies (WIMWAT)", SIGACCESS Accessible Computing, Issue 96, pp.11–18, 2010
This paper provides an overview to the WIMWAT project. It describes a pilot investigation (see Chapter 4) which was conducted to answer the research problem, provide more insights into the issues by investigating whether our tell-sign identification concepts are feasible, and clarify the area of interest. The results for the two types of widgets were evaluated over the top twenty Websites were reported, and suggestions for addressing the issues when using tell-signs as the identification method were shown. An overview of an earlier version of the paper was presented at the ASSETS' 09 Doctoral Consortium, Pittsburgh, USA. This event gave the opportunity to discuss about WIMWAT with experts in the field, and in

the process provide valuable insights to refine our methodologies (see Chapter 5), and suggestions relating to future research directions (see Chapter 7) were gained.

iii. Andy Brown, Caroline Jay, Alex Q. Chen, Simon Harper. “The Uptake of Web 2.0 Technologies, and Its Impact On Visually Disabled Users”, Universal Access in the Information Society, 11(2) pp.185–199, 2012
Web 2.0 technologies have lots of perks, but they also introduce problems for the Web community. This paper highlights the impact of Web 2.0 concepts on the visually disabled user, and addresses some of the issues by suggesting important aspects to developers to make the dynamic content updates more accessible. It supports the purpose of this thesis and provides an updated view of the uptake of Web 2.0 technologies and the related accessibility issues for visually disabled users (see Chapter 2). The study contributed suggestions of possible avenues for investigating possible applications (See §1.1.2), which was one of the applicable approaches discussed in this thesis §7.1.3. The contribution to this study includes an updated view of Web 2.0 UI technological trends from the data analysed over a ten year longitudinal study.

iv. Alex Q. Chen, Simon Harper, Darren Lunn, Andrew Brown. “Widget Identification: A High Level Approach”, World Wide Web, 16(1) pp.73–89, 2013
This paper reports a further empirical investigation using the tell-sign technique (see Chapter 4) coded in the system, to apply the concepts from the ontology (see Chapter 3), to identify/predict widgets. Although the possible tell-signs of all seven widgets were discussed in this study, only detailed investigation of the two widgets – Slideshows and Carousels – were presented to aid description of how our widget prediction process is feasible and can improve Web accessibility. This paper created the opportunity to formally update the status of WIMWAT, and explore possible tools (i.e., SCWeb2.0) and applications (injecting WAI-ARIA syntax) that can use WPS to assist their users to achieve the full experience the developer intended them to have. This thesis further developed the concepts discussed in this paper, to implement WPS and WPF.

Chapter 2

Background and Related Work

A review of studies related to accessibility issues surrounding rich content Web pages, design pattern discovery and reverse engineering, along with related studies to Web widgets detection, is presented here. The literature review provides insights into the current issues relating to the research, the necessary theoretical background for the work described, and justification for our proposed research. This chapter is divided into three main sections. In §2.1 (Web Development Technologies and Recommendations), an overview of the issues faced when interacting with Web pages that use widgets is discussed. We outline the importance of being able to interact with Websites that have widgets and dynamic content, and how not being able to operate these widgets can obstruct us from communicating, and prevent us from accessing information on the page. Further exploration of the challenges faced during the design and development phase are also discussed in §2.1.1 and §2.1.2.

Previous work undertaken to reverse engineer design patterns and conceptual designs from the application source code are presented in §2.2 (Reverse Engineering). The investigations to identify components and objects on a Web page are also discussed to provide the necessary knowledge of existing and related work, and the issues that surround them. The final section (Using Ontology to Classify Web Widgets) reviews how ontologies can assist the WPF to provide a rounded approach to define widgets, so that it will be simple enough for a broad range of Web users to understand, while sufficiently in-depth for predicting the widgets on a Web page.

2.1 Web Development Technologies and Recommendations

The Web is a medium that provides an environment where files are interlinked, and can be accessed publicly via the Internet. It is a heterogeneous combination of technologies, documents, recommendations, and guidelines. Since the proposal of the Web 2.0 concepts in 2004 [O'reilly, 2007], the Web has flourished with Rich Internet Applications (RIA). More recent methods of developing RIAs do not present content using the conventional 'Page Model' concept [Maurer, 2006]. Instead they source and remix data from multiple sources, and present them in small sections within the content called micro-content. Rather than reloading the entire page, information is updated independently in micro-content.

Micro-content can be updated with preloaded content using a client-side scripting language to manipulate the presented content. However, content can also be loaded on the fly when requested after the page is loaded via remote scripting methods. One of the leading set of technologies of Web 2.0 is AJAX [Deitel and Deitel, 2007]. This technology uses remote scripting to allow the Web page to have ad hoc communication with Web servers, so that the page can request for Web services or content, while rendering the page in parallel.

Much Web data stored in databases or embedded within the Web document is invisible to the user. It would require the Application Programming Interface (API) or Web widgets to extract this information [Cooper, 2007]. Think of flight schedules, movie timings or news content for example; a lot of this information will be available on the Website, but Web developers employ widgets to manage the desired information requested by the user. Hence, this information is invisible to the user until the specific data is requested. This form of data is known as the 'Deep Web' [Bergman, 2001]. To retrieve content that lies in the 'Deep Web' greatly depends on whether users are able to operate the RIAs or widgets. After extracting the desired content, only the accessible content is delivered. Discussed by Borodin et al. [2008], Assistive Technologies like screen readers often face problems when attempting to cope with the evolution of Web technologies. The lag behind the new technologies makes it difficult, or sometimes even impossible, for visually impaired users to access the information [Brown and Jay, 2008a; Cooper, 2007]. At least both Web content authors/developers and user agents must comply with the guideline recommendations such as Web Content Accessibility Guidelines (WCAG) or the Accessible Rich Internet Applications (WAI-ARIA) suite

in order for users to benefit from the advancement. However, in practice, this coordination is often not achievable.

The Web is evolving constantly and at an expeditious rate due to its popularity. This rapid change and little economic benefits, make guidelines difficult to obey, especially those relating to accessibility [Power and Petrie, 2007; Harper, 2008; Richards and Hanson, 2004]. Another important factor that causes this issue to arise is that much of the Web content is created by non-professionals, and they lack the knowledge of the importance of accessibility. With Web applications being used more and more as real applications, and replacing some traditional desktop applications, the role of the user agent is increasingly becoming like operating systems [Anttonen et al., 2011]. The shift in the application delivery paradigm demonstrates the increasing importance of guidelines conformance. These richly interactive and intensive applications will require more than just merely delivering accessible content, but operationally accessible applications that deliver accessible content.

The WIMWAT project requires a broad understanding of areas such as code comprehension, so that the patterns of a Web widget can be recognised, and techniques used to distinguish the different widgets are required. In the next section, Web development guidelines and recommendations will be explored to assist us with identifying instances of the components of the widgets, as well as their design patterns. Consequently, it will help us separate codes that are well developed and that conform to the guidelines from those that do not. Using this knowledge, we will discuss how they play a part during development by exploring the approaches developers take to create the widgets in §2.1.2.

2.1.1 Web Accessibility

The practice to make Web pages accessible to all users, especially to those with disability, refers to Web accessibility [Thatcher et al., 2003]. Harper and Yesilada [2008] described the Web accessibility field as a mechanism to provide equal opportunity to everyone. Their review of Web accessibility and guidelines suggested that disabled people still face difficulties when accessing the Web. All recommendations by W3C take into account Web accessibility when designing them. Hence, covering the generic knowledge of Web accessibility will provide a rounded perspective on the general Web standards and recommendations.

To provide true Web accessibility, a number of guidelines have been developed to provide recommendations for Web developers/authors, user-agents, authoring tools,

and Assistive Technologies developers. However, very few developers are willing to rework old content so that they conform with the guidelines, mainly because the benefits of Web Accessibility affect only a small population of the Web users [Richards and Hanson, 2004] and have few economic returns. Often very little documentation about the older Web pages is available, thus it is time consuming to rework them, and the guidelines/recommendations constantly change due to the technological advancement of the Web. Furthermore, much of the Web content is developed by non-professionals or people with little awareness of Web Accessibility [Power and Petrie, 2007] or different development standards, while the Web accessibility domain requires more professionalisation [Cooper, 2007].

To encourage more people to adapt their Website to be accessible, Richards and Hanson [2004] highlighted the benefits of making Websites accessible to a wide range of audiences with different physical capabilities. They also discussed the methods that will cost developers less to adapt to these changes. More importantly, they highlighted that accessibility will not only benefit people with disabilities, but also older adults, and suggested how semi-automatic tools can help the adaptation process for Websites to conform to the Web accessibility guidelines.

Web accessibility guidelines such as section 508 in the United States of America, and the commonly referred to international guidelines to follow, published by the Web Accessibility Initiative (WAI) [Shawn, 2009] from the World Wide Web Consortium (W3C), provide a framework to produce accessible Web content. WAI covers an extensive range of guidelines to cover most aspects of Web accessibility. The more notable guidelines for Web page developments are Web Content Accessibility Guidelines (WCAG) and Accessible Rich Internet Applications (WAI-ARIA), while User Agent Accessibility Guidelines (UAAG) provide recommendations for user agent developments.

A series of guidelines is layout by WCAG for Web content developers, authors, and developers of Web accessibility evaluation tools. The WCAG working group, which is part of W3C's WAI, has developed these guidelines. WCAG 2.0 [Caldwell et al., 2008] has taken over from WCAG 1.0 [Chisholm et al., 1999] since December 2008 due to the advancement of technologies and ease of understanding, and suggests a more precise automated testing and human evaluation. WCAG 2.0 suggests techniques for general Web development and technology-specific development, such as the WAI-ARIA suite [Craig et al., 2009], HTML/XHTML, CSS, and scripting.

The advancement of Web technologies introduced by the Web 2.0 concepts have

many perks but have also brought about new challenges relating to dynamic content updating and the overwhelming of users with interactivity. WCAG 2.0 attempts to provide an answer to these issues by including WAI-ARIA. A study by Thiessen and Chen [2007] investigates AJAX live regions using a chat example to illustrate the robustness of WAI-ARIA. They have reported that developing accessible Rich Internet Applications (RIA) is possible through WAI-ARIA. This investigation has given an indication of how WCAG will perform when obeyed.

The WAI-ARIA recommendation provides a mechanism for Web content and Web applications, especially those developed with HTML, JavaScript, AJAX and related technologies, so that they are more accessible to people with disabilities [Craig et al., 2009]. This recommendation is developed by Protocols and Formats Working Group (PFWG) and the Education and Outreach Working Group (EOWG), which are part of the World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI). Currently, WAI-ARIA 1.0 Candidate Recommendation has been published on 18 January 2011, but it is not without its flaws. Issues surrounding alternative text and `longdesc` often surface, and WAI-ARIA 1.0 is not equipped to support accessibility for mobile [Schwerdtfeger, 2012]. These issues are planned to be addressed in WAI-ARIA 1.1 and later 2.0, HTML5, and efforts by the User Interface (Indie UI) Working Group have begun to address issues relating to browser independence for gestures and other user interactions for a wide range of devices [Cooper, 2012].

These guidelines and specifications provide semantic information for the user to interact with the components on the page. However, to operate widgets, users must be able to perceive the design concepts of the widget. Only when this information is achievable will the user be able to operate the components of the widget in the page independently. These guidelines and specifications lack coverage of this type of information. Instead, it is left to the developers to include descriptions of elements in the page via `longdesc` or `aria-describedby`. However, when using this method, there is no way of verifying the credibility of the content supplied. Furthermore, even if the semantic relationship between elements can be established, it will only cover the components level of the widget. The information will not be sufficient to recover the perceived design concepts of the widget. In the WIMWAT project, WPF bridge the gap for Assistive technologies and tools, such as SCWeb2.0, to deliver this type of information to the user.

Web authors or non-technically related people provide content to most of the Web. Often these people use some form of authoring tools to assist them with their Web

page development. Authoring Tool Accessibility Guidelines (ATAG) provide a set of guidelines that describe to developers of authoring tools how to create tools that produce accessible Web content [Henry and May, 2008]. These tools include those that transform documents into Web formats, and content layout management like CSS formatting tools.

A cornerstone component of the Web experience is the user agent. Often these applications are the gateway to users of the Web, and they are increasingly playing the role of an operating system as the importance and application of the Web documents are progressively being used like a real application platform Anttonen et al. [2011]. The User Agent Accessibility Guidelines (UAAG)] explain how to develop user agents that are accessible to people with disabilities [Henry and May, 2012]. Most of the Web content is delivered through a user agent such as Web browsers, media players, and at times Assistive Technologies. Thus, by providing a set of guidelines to develop this medium, Web content can indirectly be made more accessible. UAAG 1.0 is currently the recommendation. However, UAAG 2.0 is anticipated to be completed in 2013. In UAAG 2.0, it aims to lay the path for future generations of Web browsers, by suggesting alternative information about technologies and platforms the user may operate with. Furthermore, the guidelines in UAAG 2.0 also updated UAAG 1.0 guidelines, to align them with WCAG 2.0 and ATAG 2.0.

A regular practice by developers is to evaluate their Web pages by running them through some industry recognised Web accessibility evaluation tools. These tools provide developers with a method to evaluate their Web pages against some industrial standards, provide feedback to developers about their work, and recommend suggestions to make their content conform to the recommended guidelines. Notably a few of these evaluation tools are WAVE, HiSoftware Cynthia Says, RAVEn and Web Accessibility Inspector. We will explore these tools to investigate the range of guidelines covered and the extent of compliance evaluated.

WAVE is a free online Web accessibility evaluation tool, developed and maintained by WebAIM [WebAIM, 2009]. It checks the Web pages or content submitted to it for the compliance issues with many of the Section 508 and WCAG guidelines. Then it shows the areas of the original Web pages using icons and indicators, to reveal the accessibility of it. However, it does not check all the issues in these guidelines. HiSoftware Cynthia Says is developed and maintained by HiSoftware Inc., to identify problems in the Web page content submitted to it based on Section 508, and WCAG

1.0 guidelines [HiSoftware Inc., 2009]. The tool comes in three versions: the enterprise, desktop, and free Web service. The free Web service version is meant for educational purposes, and restricts its users to submitting only one page from a site per minute. The IBM Rule-based Accessibility Validation Environment (RAVEN) inspects and validates Java rich-client, and Web based Graphical User Interface (GUI) oriented applications for accessibility [IBM, 2009]. This tool is now part of the Eclipse-based[1], validation component in the Accessibility Tools Framework (ACTF). Fujitsu[2] developed a multilingual evaluator called Web Accessibility Inspector. Like most evaluators or validators, this software will highlight the parts that require modifications. This software provides the ability to evaluate Web accessibility to not only English literate users, but Chinese and Korean literate users too. The importance of this type of tool has a growing importance in the Web community. As investigated by O'Neill et al. [2003], as the Web gains in popularity, the distribution of non-English content Websites is also growing. However, the WI was developed to evaluate Websites against the WAI WCAG 1.0 guidelines only. Hence, this tool will not be able to fully examine Web pages created using the Web 2.0 concepts.

From the evaluation tools covered, none of the tools can evaluate the Web page fully against all guidelines and specifications. Often issues pertaining to guidelines and recommendations conformance are never noticed and dealt with at the development phase. On some occasions, this is not the fault of either the evaluation tools or the developers, but that of the ambiguity of the recommendations. The Web Accessibility Initiative (WAI) guidelines are widely accepted as the international standards for Web accessibility such as WCAG [Shawn, 2009]. However, WAI guidelines are constantly undergoing changes, to keep up to speed with the technologies; from WCAG 1.0 [Chisholm et al., 1999] to WCAG 2.0 [Caldwell et al., 2008]. These constant updates add to the difficulties that many designers and developers face when trying to design their Web pages to comply with these guidelines.

Abascal et al. [2004] discussed a repair tool called EvalIris, developed to automatically evaluate Website accessibility using sets of guidelines, and the guidelines can be easily updated or replaced. This tool was created to check the Web page's accessibility based on WCAG 1.0 guidelines. The EvalIris system was evaluated using the W3C's Techniques For Accessibility Evaluation and Repair Tools [Ridpath and Chisholm, 2000] working draft. It was reported that this tool successfully fulfilled the

[1] http://www.eclipse.org
[2] http://www.fujitsu.com

objectives of the study, and demonstrated the importance of having an evaluating tool that is capable of accepting new sets of guidelines. Since the EvalIris system is an ongoing project, further enhancements were also proposed, so that a crawler could be integrated. A user interface was also suggested to this Web service, and a XML translation helps to assist the translation for new guidelines into the XML schema used by EvalIris [Abascal et al., 2004]. The EvalIris system is great for static content, but evaluating dynamic content manipulated by widgets has to deal with a different degree of complication. Now, conceptual designs need to be assessed as well, before the content present can be extracted for accessibility evaluation.

A framework called AxsJAX was suggested by Chen and Raman [2008] to allow developers to bootstrap it with WAI-ARIA, so that better accessible Web applications or widgets could be provided. However, this framework can only be put into place either during development or maintenance, or it will require additional tools to assist it to identify the widget in the page - such as the proposed research - before syntax of this framework can be inserted.

Our review on Web guidelines and recommendations, in particular with regard to Web accessibility, highlights the challenges faced by those creating Web pages. The ill compliance of guidelines and recommendations highlighted in Harper and Chen [2012], suggests that much Web content is not accessible to all. Moreover, Web pages with dynamic content add another layer of conceptual complexity to accessing this type of content [Chen et al., 2013]. This demonstrates the importance of research such as the proposed work in this thesis, to improve the current situation and provide an alternative for users to overcome the existing issues.

2.1.2 Web Design

Designing a good Website involves a few key components; these are the appearance, usability, maintainability and organisation. Figure 2.1 illustrates the iterative process visited by the management when designing and developing, or maintaining a Web application. Understanding the sequence and components required when designing a Website will indirectly give a clearer concept of how a specific widget is formed when attempting to reverse engineer it. The key components that are used for designing most Websites include the content, the styling and the behaviour aspects, along with the platform on which the Web page will be rendered. These components will be discussed and covered in greater depth in §2.2.2.

Designing Websites has a lot of differences from and similarities to engineering

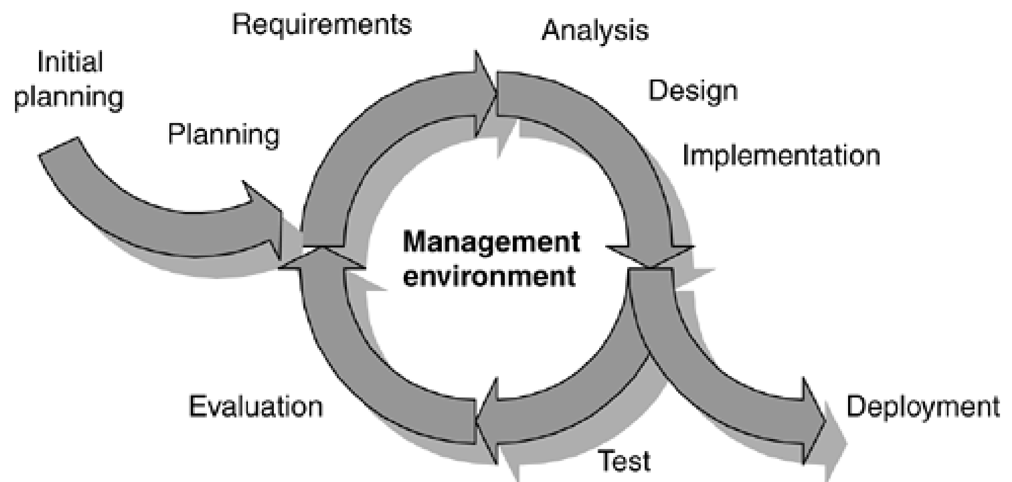


Figure 2.1: The iterative process to develop and maintain an application. [Kruchten, 2000]

desktop software applications. As described by Overmyer [2000], three differences between the domains were highlighted, particularly usability and shorter life cycles. Later, O'reilly [2007] went on to describe the different types of evolving design models in software development, and business over the internet. He highlighted that the Web 2.0 is a fuller realisation of the true potential of the Web and how it can help to achieve richer user experiences. With the Web 2.0 concepts, software is no longer delivered as a product, but a service, thus ending the release cycle of software. It encourages programs to be lightweight, so that they will be good for reusability.

Designing for Usability

A usable Website allows its users to access the Web page, and find their way around the Website quickly and efficiently. This means Web pages must be accessible to all, including people with disability.

As discussed by Maurer [2006], designing Web pages with RIAs that are usable and accessible is difficult, and she goes on to elaborate the key challenges that designers will face when ensuring their applications are usable. However, accessibility problems with RIAs seem to be an increasing issue as the Web evolves with the Web 2.0 concepts [Gwardak and Påhlstorp, 2007]. Gwardak and Påhlstorp [2007] then explored the usage of guidelines when designing Web pages with RIAs. From the investigations, they suggested a set of guidelines that are a hybrid between the desktop user interface

and the Web Content Accessibility Guidelines (WCAG) 1.0.

In order to help Assistive Technologies to adapt to Web 2.0 technologies, studies such as Borodin et al. [2008] and Brown and Jay [2008b] have attempted to improve the issues through audio browsing and provide further insights into issues surrounding Web 2.0 technologies. However, both studies monitor for changes in content and inform users about the updates. They do not go further to interpret the content and code so that the type of widget can be derived and presented to the user.

Maintainability & Organisation

Understanding the overall development cycle will expose the challenges of widget designs, so that decisions chosen for developing the widget can be understood on an abstract level when developing the WPF. Then, steps can be taken to overcome the perceivable issues. The life cycle of a Web page goes through a number of stages, and some of these stages loop around endlessly during the life span of the Website. Hence, good organisation and structure to the Website are required to ensure the quality, efficiency, and ease of maintaining it. Although good documentation of the design is crucial, often this is lacking, and it affects future maintenance processes. Understanding the development cycles will expose the usability and accessibility issues of the Websites that arise from the development phase. This exploration will strengthen the knowledge relating to these issues for surrounding Website development and maintenance.

During the design phase of the Website, keeping unambiguous records of the Web pages designed is an essential key to providing good documentation. Fortunately, there are useful tools such as Unified Modeling Language (UML) [Conallen, 2003; Koch and Kraus, 2002], Web Modeling Language (WebML) [Ceri et al., 2000], and Hypertext Design Model (HDM) [Garzotto et al., 1993] to provide a means of communicating the most difficult to explain Web application concepts to designers, developers, and users [Distante et al., 2004]. This documentation will prove to be useful during maintenance when developers, or new developers, need to revisit the problem again.

As highlighted by some studies, the maintenance task for a Website is not easy and often requires tools described in studies such as Di Lucca et al. [2005a]; Distante et al. [2004]; Stencel and Wegrzynowicz [2008] to help ease this work. This is because quite often, due to the competitiveness, advancement of the Web, and pressure imposed by their competitors, documentation is lacking, and there is little time for developers to turn the Web applications around.

Design Patterns

When developing a Web page, different components come together to form the user interface, and the engine that controls its behaviour. Commonly, to describe the different parts of an application, design patterns are used. Design patterns are described by Alexander [1979], as “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”. He illustrated why one pattern will differ from another, and how it is reliant on the surroundings and the people that created it. This issue relates to the granularity issues of the design patterns, and varied interpretations by [Gamma et al., 1995; Buschmann et al., 1996; Arvola, 2006; Vlissides et al., 1996; Larman, 1998].

Gamma et al., also known as the Gang of Four, described design patterns as reusable elements for an object-oriented software approach in [Gamma et al., 1995]. The book catalogued 23 design patterns that are widely referred to by many in this field [Mikkonen, 1998; Dong et al., 2008b; Stencel and Wegrzynowicz, 2008; Hendrix et al., 2002; Tsantalis et al., 2006]. It gives a full description for each pattern, including the names and intentions of the patterns. Additional design patterns for interaction are suggested in Arvola [2006], so that users can control the visibility of information. Gamma et al. emphasised the importance of determining the object's granularity, which can vary tremendously from one person to another, and the implementation of the object. This is the key to making a flexible design and ensuring the design patterns can adapt to changes later in their life cycle. A closer look at patterns reveals that some patterns provide the structure of an application into subsystems, and others support the refinements of the subsystems and components [Buschmann et al., 1996].

The documentation of design patterns will evolve over time, thus incorporating flexibility into the design is vital. Commonly the design patterns are not properly documented, and this may result in the violation of the constraints and properties of the design patterns. Furthermore, Bosch discussed the issues with breaking down the design patterns, and reorganising them to suit the structure of an application [Bosch, 1998]. This could lead to traceability problems in the latter part of the application life cycle. Investigations by Dong et al. [2009], aimed to tackle the evolution problems face when adding or removing design elements from existing design patterns during the transformation process. They proposed an automated process for the evolution of design patterns. Such a system can reduce errors and missing parts. However, if the pattern overlaps other pattern instances in the design, errors and inconsistencies may

arise from it. This study highlighted the importance of designing our methodology for identifying Web widgets, so that it will be able to handle the evolution of widget patterns.

Benatallah et al. [2002] suggested four additional patterns in 2002 for architecture and managing composite Web services. They proposed these patterns because they believe "Web services are loosely coupled Internet-accessible software entities delivering functionalities provided by business applications and processes". These patterns provide a solution to the recurrent problems of integrating and providing Web services. However, the Web has evolved since, and with Web 2.0 concepts, and many new business models, the definitions of these patterns need to be revisited.

As described by Mikkonen [1998], formalising design patterns will allow rigorous reasoning, and it also support a software engineering view for the development. He explains how to formalise the temporal behaviours of design patterns to create the specifications for complex systems. According to Mikkonen [1998], formalisation of a pattern can remove ambiguity, so that the temporal behaviours of patterns can be rigorously attended to. This technique can be useful to ensure the soundness of the design concept when designing a system. However, it does not bridge a bidirectional relationship between the designing and implementation stage.

Web Widgets Design Pattern Libraries

Defining different types of widgets can be a confusing process due to the different perspective the individual widget design pattern libraries acquire. Widgets design patterns and JavaScript libraries such as Dojo[3], jQuery[4], YUI[5] and Welie.com[6] were examined to aid the process of defining the types of widget. It is noticed that deriving with a common definition for each type of widget is difficult because the definition varies from library to library; in some cases, they can mean different concepts. Reported in Vora [2009], currently no consensus has been reached on how widget pattern libraries should document and maintain their designs so that these patterns can be shared with others. Thus, there is a gap that the methods used in WPF can fill when classifying widgets.

[3] The Dojo Toolkit - `http://dojotoolkit.org/api/`. Last accessed on 12th February 2009

[4] jQuery JavaScript Library - `http://docs.jquery.com/Main_Page`. Last accessed on 10th November 2011

[5] Yahoo! User Interface Library - `http://developer.yahoo.com/yui/2/`. Last accessed on 18th August 2012

[6] Welie.com Patterns in Interaction Design - `http://www.welie.com/patterns/`. Last accessed on 18th August 2012

The principle concepts of the Web evolve around reusing available data, and these concepts can be applied when designing Web widgets. As observed and reported by many studies, different types of core user interface components and objects exist within different types of widgets [Karanam et al., 2011; Brown et al., 2009; Vora, 2009; Rossi et al., 2008]. These core components and objects are widely used to distinguish one type of widget from another, and these concepts are also used as part of the classifying methodology in WPF [Chen et al., 2013]. However, the main difference between the design pattern libraries and studies are the conceptual objects. These concepts are difficult to model and are often left unexamined. The widget classifying method employed by WPF attempts to include the conceptual objects of the different types of widgets in the classifying process. Through this approach we aim to reduce widget definition ambiguity as well as attempting to predict the type of widget from the Web page source code.

Often Web widget design pattern libraries have the tendency to borrow patterns from each other. Libraries that have a high rate of borrowing patterns from other libraries are not included here. It was found that the YUI and Welie.com libraries have more widgets that have similar perspectives with defining widgets, and they also provide more documentation for their widget design patterns.

Ubiquitous Web Application

Web pages are no longer just accessed by desktop applications but also non-conventional methods, for example, applications from mobile devices. The demand for accessing the Web using these devices draws developers to create Web pages and applications that are ubiquitous. Finkelstein et al. [2002] proposed a framework for developing ubiquitous Web applications because this form of Web application suffers from the "*anytime/anywhere/anymedia syndrome*". This syndrome means that you can access Web application via any medium at any time and in any place. Although this framework provides the mechanism for the flexibility of context-aware computing and personalisation, it does not address the accessibility issues raised by Yesilada et al. [2010] and Hardesty [2011]. Unlike the framework proposed by Finkelstein et al. [2002], WPF provides a mechanism to predict the widgets in the Web page, so that Assistive Technologies can use this information to alert their users about the widgets for both desktop and mobile users.

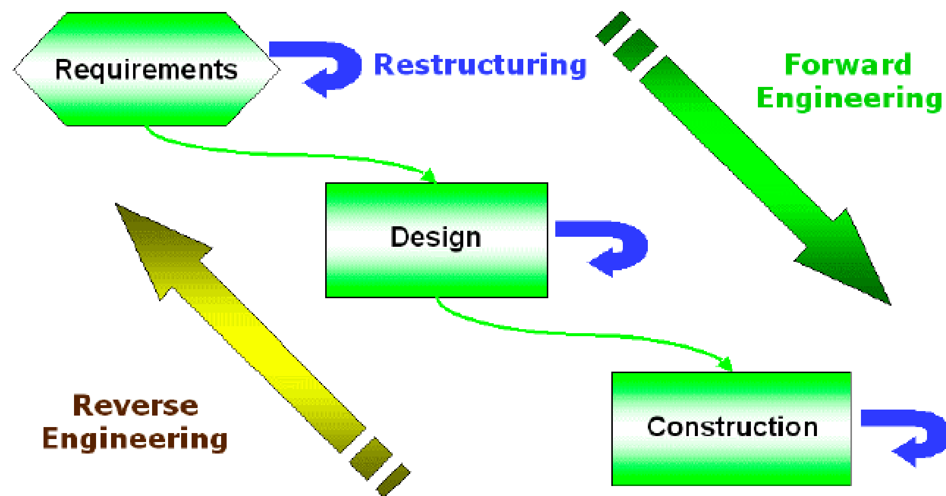


Figure 2.2: Reverse and forward engineering cycle [Tilley and Huang, 2001]

2.2 Reverse Engineering

The process of reverse engineering a Web widget is an analytical study of the widget's source code, so that the technological principles can be discovered. Through reverse engineering, design information – such as the decisions made earlier by developers – can help to understand the conceptual designs of the widget. An illustration of the flow of processes for forward engineering and reverse engineering is presented in figure 2.2. The purpose of reverse engineering a widget will allow the production of content by the widget to be evaluated, so that any inaccessible content produced can be modified. The proposed work requires different reverse engineering techniques for its analysis. To understand the depth of the investigation, selecting the correct coding format to conduct the evaluation, being aware of the types of coding formats and their different versions when attempting to comprehend it, and understanding the different techniques available for the task are crucial.

2.2.1 Purposes for Identifying Web Widgets

The technique employed to identify Web widgets is used for many reasons, such as to assist the user, to evaluate a Web page, to assist a development process, to identify or fix security flaws in a Web page, or to maintain existing Web pages. Guha et al.

[2009] detects AJAX Web applications (widgets) so that the possibility of AJAX intrusion detection can be investigated. However, some attempt is made to identify Web applications to make maintenance of the Website easier [Di Lucca et al., 2005a; Dong et al., 2009].

Web pages that want to provide interactivity to their users so that the targeted content can be delivered, or Web applications that want to assist the users to complete part of a task, will often include some widgets in their pages. Some studies, such as those done by Brown and Jay [2008a] and Hardesty [2011] , highlight the importance of identifying widgets, especially those that use AJAX as part of their process because they introduce accessibility issues to Assistive Technologies and mobile devices.

Before WCAG 2.0 became a set of accessibility guidelines, Di Lucca et al. [2005b] attempted to identify and rectify accessibility issues in a client's page code. They used conceptual models to model the Web accessibility guidelines (WCAG 1.0 and WCAG 2.0), and then they proposed an accessibility model to identify Web accessibility problems on a Web page. Di Lucca et al. [2005b] conducted a case study to determine the conformability of ten Websites to evaluate the concept. Twenty Web pages from ten Websites were examined. It was reported that most of the Websites had Web accessibility problems except for two of them. This study reconfirms the need and demand for a rectification tool at the developer's end. However, it only examines superficial code, and it does not attempt to understand the semantics behind the scripting code and the hypertext code. Furthermore, it also does not relate the different objects in the Web application.

Without understanding these elements, the multifarious Web applications that extensively reuse objects that are created within a code and APIs, it will not be possible to modify the Web applications, so that they are in accessible form. Furthermore, the Web has evolved to be much more dynamic since then, and WCAG 2.0 has now become a guideline.

2.2.2 Coding Format

The Web allows a combination of various Web document formats to form together to make the Website work. These formats include markup hypertext, styling and client-side scripting languages. Understanding the different scripting languages and their different versions is vital when comprehending them.

Web documents rendered by user agents to produce the DOM mainly consist of the content layer, the styling layer and the behaviour layer. Surgical analysis of the DOM

will only give an account for what is deemed by the user agent engine suitable for presentation to the user at that instant in time. In order to identify and predict the type of widget, information provided by the DOM alone is not sufficient. Only inferences derived from all three layers combined with the DOM information will allow most widgets to be identified.

All content intended to be presented to the user is structured in some form of Markup Language in the content layer. The most commonly used is the Hypertext Markup Language (HTML) and W3C covers a few versions of specifications for this language. The newer versions can fluctuate by quite a lot from their predecessors, as seen in HTML5 [Hickson and Hyatt, 2009]. In many ways, the different types of HTML will affect the way developers deliver their content, where the more visually expressive types will be loaded with more content compared with the less expressive ones. Hence, when reverse engineering a Web page, identifying the type of HTML is crucial to spotting the correct pattern when trying to identify tell-signs from the source code.

The HTML Document Type Declaration (DTD) has evolved into a number of versions together with evolution of the Web. From a recent study conducted to understand the evolution of the Web over a ten year time frame, Chen [2008] presented that the popularity of the latest version of DTDs means that it tends to be preferred and adopted by Web developers. As reported, Web pages currently still use the DTD of HTML 4.01, Extensible Hypertext Markup Language (XHTML) 1.0 and 1.1 mainly. This is because XHTML 2.0 [Axelsson et al., 2006] and HTML5 are both currently still in their working draft state.

HTML5 attempts to be backwards compatible, however, as suggested by xhtml.com [2008]. The backward compatibility of different versions of HTML/XHTML should be dealt with by the user-agent. Making HTML5 backward compatible may make it more confusing for users, and adds complexity to the validators and work surrounding Web accessibility. Another proposal to make HTML5 an object-model approach rather than the element format, will foreseeably add complexity when machines attempt to comprehend hypertext documents. If this form of HTML5 is rolled out, it will change the way widgets are created and designed if HTML5 is adopted by developers.

The concept of splitting the styling component of the page from the content allows these two components to be dealt with separately. This idea has given raise to the style layer where Cascading Style Sheets (CSS) were introduced to take over the job of adding styles to the Web page. CSS is a language to include style in Web documents,

and controls the appearance of the text, i.e. the size, colour, font family, etc., the Web page's layout. It comes in a number of levels and profiles to deal with different devices; namely, levels 1, 2 and 3, mobile profile 1.0 etc. [Bos, 2009]. CSS allows the document structure to be separated from the presentation, so that developers/designers can have precise control of the Web page layout and styling without interfering with the markup content [Jacobs and Brewer, 1999]. Through this method, Web content authors can concentrate on the content itself, and the accessibility aspect of it. However, like all Web technologies, CSS is also ever changing to keep up with technological advances, so it will be interesting to follow up the advancement of CSS when HTML5 becomes a recommendation, and the changes that will affect the development of widgets.

A key feature of manipulating content uses scripting languages to do the task via the behaviour layer. Since only widgets that use JavaScript are covered, only the extent of this language will be analysed. JavaScript is a client-side scripting language to enhance the functionality of Web browsers. It is an interpreted programming language that has object-oriented capabilities embedded in most popular Web browsers. Although JavaScript is not Java, the syntactic core of the language resembles C, C++, and Java [Flanagan, 2002]. JavaScript as a client-side scripting language manipulates the Web browser's functionality from the behaviour layer. However, the language may perform slightly differently when used in different Web browsers. Patents such as Dencker et al. [2008] attempt to create a JavaScript client framework, so that it will provide object-oriented features, and enable cross-window and cross-frame communications. This framework wants to standardise the scripting language, so that it will be independent from the different types of browsers and the different versions.

Web widgets often use client-side scripting languages, such as JavaScript and Visual Basic Scripting (VBScript), to assist them in manipulating the style and the content of the page. Web pages that employ the Web 2.0 concepts may even use the scripting language to employ remote scripting techniques, so that they can communicate and retrieve additional information from the Web server without reloading the Web page. Due to the popularity of JavaScript, which dominates most of Web [Chen, 2008], our proposed work will focus on widgets that are developed by this type of client-side scripting language.

2.2.3 Code Comprehension

Described by many [Liu and Wilde, 1990; Rajlich and Wilde, 2002; Di Lucca et al., 2005a] code/program comprehension plays a vital role in software evolution and software maintenance. These techniques are often used to identify parts of the code/program that resemble a conceptual design for maintenance purposes and for detecting security flaws. The proposed work comprehends the source code to identify the regions that will manipulate the content layer of the Web page. Often, to manipulate the content, the styling properties of an element can be changed to cause a visual effect where the content is hidden or shown. Alternatively, the content within the elements can be updated via the Behaviour layer.

The bottom-up theory for program comprehension is based on the parts of a code that the programmer recognises [Rajlich and Wilde, 2002]. Commonly, smaller sections of the code are nested within a larger set of code. This is a similar concept of pattern granularity to that discussed in [Alexander, 1979; Gamma et al., 1995]. However, Rajlich and Wilde [2002] described how most programmers tends to used the "as needed" strategy as programs become larger, and usually it is not possible to completely comprehend the source code due to the time available. This method of development provides a case for researching whether comprehending the Web page's source code, to search for instances of Web widgets, will be a feasible approach.

Documentation of an application has been commonly found lacking for both software applications [Dong et al., 2008b], and Web applications [Di Lucca et al., 2005a]. Accessing the documentations for Web applications is even more difficult, because applications can reuse APIs from different sources, and sometimes documentation for these APIs is not available [Dong et al., 2008b].

Di Lucca et al. [2005a] investigate an approach to recovering user interaction design patterns for Web application, which will make the maintenance task of the Website easier. They proposed a three-phase process to search for patterns in Web applications: the training phase, candidature phase, and a validation phase. The architecture used in the study for recovering Web interaction design patterns is depicted in figure 2.3.

Six Web interaction design patterns were chosen to evaluate the approach on a small set of training samples in Di Lucca et al. [2005a]. These patterns include Guestbook, Login, Poll, Registration, Search and Sitemap. The experiment obtained 79% precision. Then a wider set of 208 Web pages was conducted on another experiment for comparison using the same approach and achieved 66% precision. This study demonstrated the diversity of the Web. However, the application of this study is different

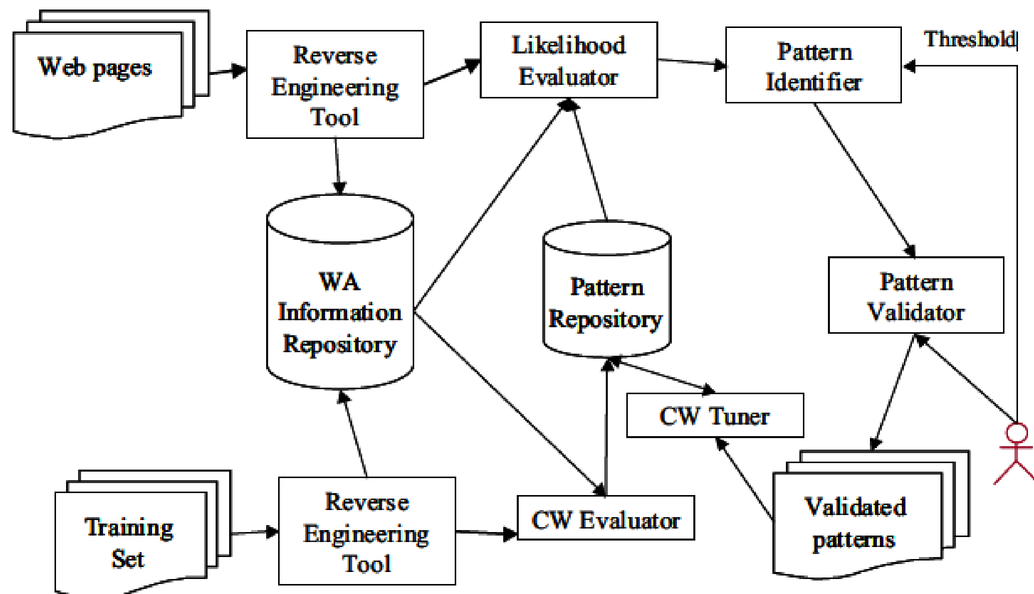


Figure 2.3: The architecture used in Di Lucca et al. [2005a] for the Web Interaction Design Pattern Recovery System

from ours. It does not examine the scripting code, semantics and the relationships between these elements. The results reported in Di Lucca et al. [2005a] can provide us with a benchmark to base our methodology upon when attempting to identify the user interaction design pattern of the Web widgets.

In the early 1990s, studies such as Liu and Wilde [1990] were conducted to assist reengineering and maintenance of old code, so that object-like features from a non object oriented language could be recovered, and the code could be transformed into the object-oriented concept. This kind of study provides a useful referential source for general concepts when identifying objects from the source code, while applying it for a different purpose.

Liu and Wilde [1990] suggested an approach that uses a Global Based Object Finder and a Types Based Object Finder to answer the object oriented concept transformation. The Global Based Object Finder searches for global and persistent data within the code, and creates the relationships of these data with the routines that manipulate them. The Type Based Object Finder relies on the data types to establish the relationships between the data types, and the routines that use them as a parameter or to return values. However, this proposal is a semi-automatic approach, and it requires human input.

These two studies gave an overview of the code comprehension techniques used for investigating Web applications Di Lucca et al. [2005a], as well as to assist the transformation of one concept into another [Liu and Wilde, 1990]. However, a review in Dong et al. [2008b], demonstrated a number of different techniques that can be employed to cater for different purposes. A comparison between the techniques was also presented. Based on this review and additional literature, we grouped the different code comprehension techniques into five general categories that are presented in the follow sections.

Matrices

Matrices are used in processes when comprehending the source code. Dong et al. [2007] proposed a novel approach to discover design patterns directly from the source code. They developed a tool called DP-Miner that uses matrix and weight to discover design patterns from the source code. A pattern matrix is used to represent the system structure, where the columns and rows are all the classes in the system. In each cell, the value represents the relationships between the classes. A system matrix is used for the structure of each design pattern. Using this method, design patterns can be discovered by matching the two matrices; if they match, an instance of the pattern is found.

Three analyses were conducted to efficiently discover the instances of a design pattern. These analyses included a structural analysis of the system and design patterns to be discovered, a behavioural analysis to understand the dynamic relationships among the participating patterns, and a semantic analysis that searches for the design documentation of the source code, in-line comments in the source code, and clues from the naming convention of the classes. Finally, the tool was tested on the `Java.awt` package[7] in JDK 1.4 to search for instances of the Adapter pattern. It is reported in Dong et al. [2007] that it took 2.44 seconds to discover 21 Adapter pattern instances.

The approach proposed by Dong et al. [2007] is a good demonstration that studies surrounding identifying patterns from the source code exist. However, this study focuses only on desktop applications or object-oriented programs. It does not cover the same magnitude as recovering Web applications design patterns. Furthermore, no further work has been done to understand the conceptual designs for applying the patterns and the rationale behind the application processes.

The DP-Miner is a novel approach that is based on matrix and weight, to discover instances of design patterns from object-oriented software systems [Dong and Zhao,

[7] The AWT package consists of 346 files, a total of 485 classes, and 111 interfaces.

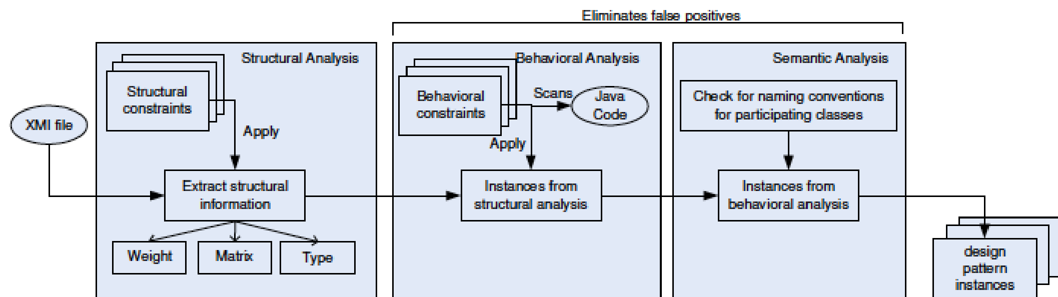


Figure 2.4: The overall architecture of the approach for DP-Miner [Dong and Zhao, 2007]

2007]. This tool was developed to provide an alternative approach to search for design patterns directly from the source code. It uses XMI-based intermediate representations for structural analysis, while the behavioural and semantic analyses are extracted from the system's source code directly, so that false positives can be reduced. The overall architecture of the approach is illustrated in figure 2.4. This work seems to be at its preliminary stages, and the authors have reported some inconsistency in their approach. Furthermore, the study only comprehends object-oriented software systems, and not the same level as Web widget detection. Dong and Zhao [2007] only attempted to discover the design patterns from the source code. They did not extend the process to reconstruct the design concepts for using the design patterns. Discovering design patterns can be difficult, but understanding and reconstructing the design objectives of the code is much more challenging, and it requires an even more intensive computation process.

In a recent study by Dong et al. [2008a], there was an attempt to discover design patterns by using template matching. The motivation behind this investigation is due to the poor design documentation. Template matching is done by matching a pattern matrix with a system matrix to determine whether a design pattern exists. This paper extends this approach by calculating the normalised cross correlation (CCn) between the pattern and system matrix, so that the degree of similarity between the design pattern and the part of a system can be computed.

$$CCn = \sum \frac{f(x) \cdot g(x)}{|f(x)| \cdot |g(x)|}, \tag{2.1}$$

where $f(x)$ and $g(x)$ are two vectors, and $x = 1, \ldots, n$.

It was attempted to encode eight design features from four large open-source systems in [Dong et al., 2008a]. The results presented are convincing, and they claimed that their approach could avoid false positive detection caused by individual matches.

Tsantalis et al. [2006] conducted a study with a similar purpose to the one done by Dong et al. [2008a]. The method proposed uses the similarity scoring between the graph vertices, instead of template matching. It calculates the similarity scores between the vertices of the system and the pattern graph. This method allows patterns that are both in this basic form and modified versions to be detected. An evaluation of Tsantalis et al. [2006] approach was done across three popular open-source projects that employ design patterns extensively and systemically. The evaluation demonstrated that the approach is precise and accurate, but a few false negatives surfaced, and no false positive was detected.

Studies by Dong et al. [2007]; Dong and Zhao [2007]; Dong et al. [2008a]; Tsantalis et al. [2006] provide evidence that design patterns can be discovered from the source code, either by template matching, similarity scoring, or using matrix and weight. However, these investigations focused around desktop systems and do not have the same degree of complexity. Nonetheless, some of the methods suggested can be reused, but they have to be modified or applied from a different prospective, so that the concepts will work with the vast combinations of technologies that make the Web work.

Conceptual Models

Conceptual models allow their users to express the meaning of terms and concepts, and to find the correct relationships between the different concepts. This form of modelling approach removes ambiguous terms and meanings, so that projects are more robust and reliable [Fowler, 1996]. This is a technique mainly used to formalise and model their concepts during the planning and development phase. Hence, investigating techniques others have used to recover the conceptual models from the source code during reverse engineering processes will provide further insights into different techniques for reverse engineering the widget designs.

Modelling languages, such as Unified Modeling Language (UML), are often employed to provide a conceptual model of the system developed during the design process. However, design patterns will evolve over time, and due to the lack of good documentation, problems arise when trying to recover them. Dong, Zhao and Sun

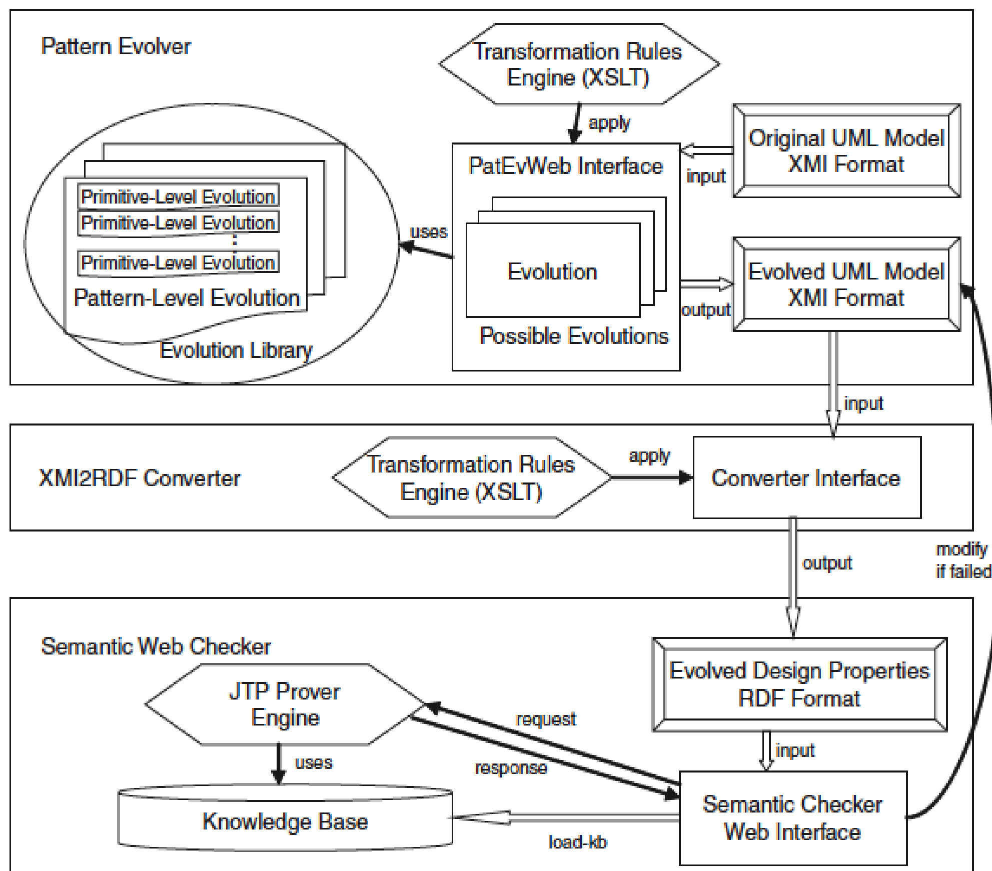


Figure 2.5: The overall architecture of the automated process that transforms the UML model of a design pattern application into its evolved form [Dong et al., 2009]

proposed an automated process that transforms the UML model of a design pattern application into its evolved form [Dong et al., 2009]. This is a three phase approach (see figure 2.5), where the first phase defines the evolution of the design pattern process using XML Metadata Interchange (XMI) in the primitive level and the pattern level. Then, the XMI format is translated using the XSLT transformation rules to convert the interface in the second phase. In the third phase, a semantic Web checker based on the Java Theorem Prover (JTP) was developed to ensure the consistency of the evolution transformation process. Dong, Zhao and Sun envisioned that automating the evolution process of design patterns could reduce errors, inconstancy and missing parts for the evolved design patterns.

A number of studies were done to recover conceptual models for reverse engineering a Web application. Most of these studies surround Web applications that use the

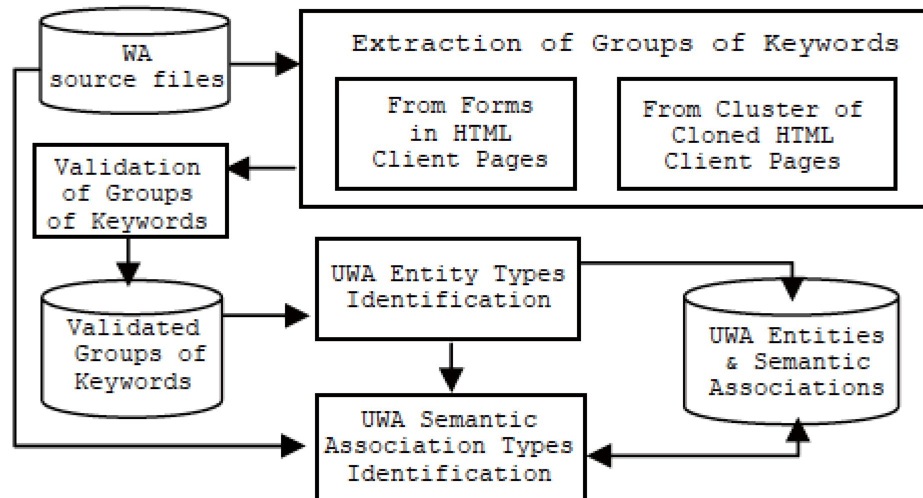


Figure 2.6: The identification process of UWA entities and semantic associations types in [Bernardi et al., 2008]

Ubiquitous Web Application (UWA) framework. This framework comprises a methodology and set of meta-models for user-centred designs for Web applications [Bernardi et al., 2008]. In these studies, Web applications are defined to deal with processes of the same scale as desktop programs. Often these applications cover more complex processes than a widget. Although the magnitudes of the investigations are very different and can be specific to a task, the techniques applied for this form of reverse engineering can be scaled to make it generic for identifying Web widgets.

Bernardi et al. [2008] proposed a semi-automatic recovery of user-centred conceptual models from Web applications. This approach is defined according to the UWA design framework. The UWA Hyperbase model is a user-centred conceptual model that represents the Web application's content, organisation (entities and components) and semantic associations between the entities where navigation paths are derived. The process taken by Bernardi et al. [2008] to identify the UWA entities and semantic association types is depicted in figure 2.6. Agreed by most of the Web application design methods proposed, including Ceri et al. [2000]; Bernardi et al. [2008], three models can be used to define the development of Web applications: the Contents model, the Navigation model, and the Presentation model. To support the recovery process of the UWA conceptual models from a Web application, the architecture designed to carry out this task is described in figure 2.7.

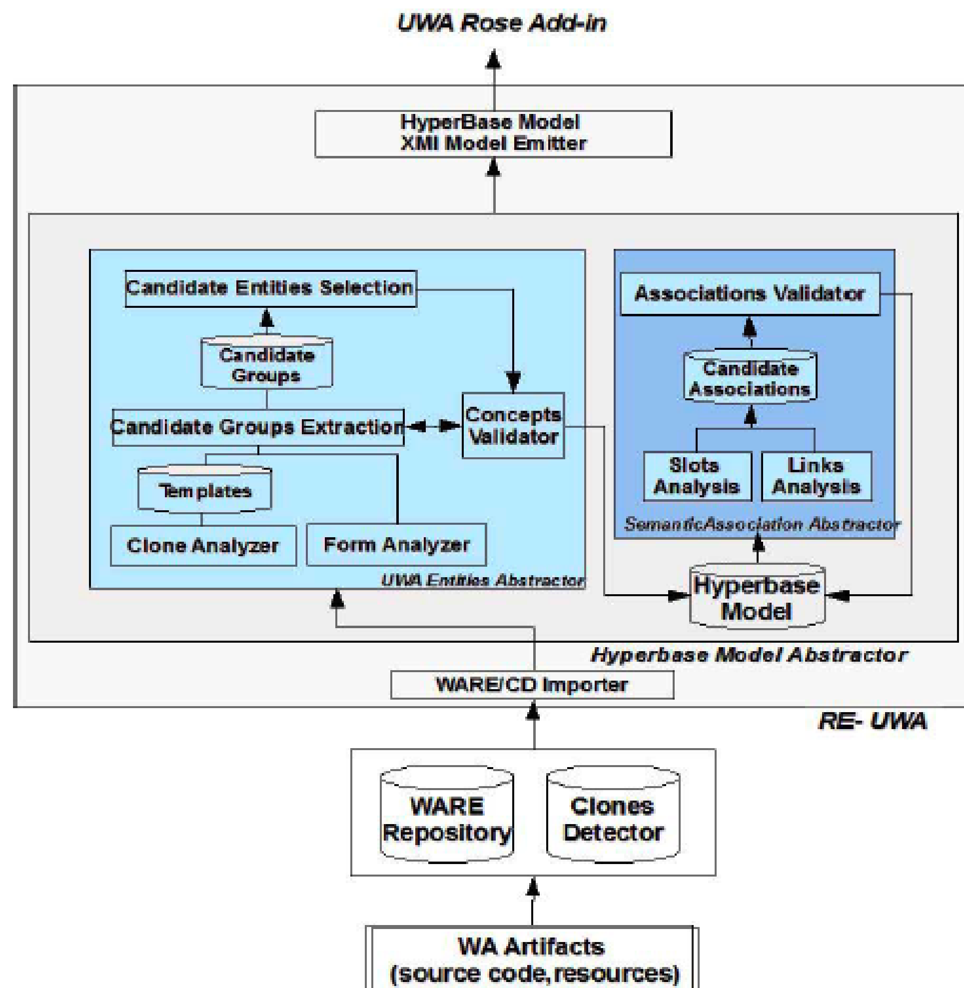


Figure 2.7: Architecture for the Hyperbase model abstractor module in Bernardi et al. [2008] for the Re-UWA environment.

A case study was conducted on the semi-automatic recovery of user-centred conceptual models to verify (1) the group of keywords extracted, (2) the candidate's UWA entities, (3) to ensure no UWA entity was undetected, (4) the semantic association of the candidate's UWA was identified, and (5) no UWA semantic association was undetected [Bernardi et al., 2008]. The case study was based on four Web applications. The authors' definition of a Web application in this study seems to refer to a Website. The reported results suggest that the group of keywords identified in HTML templates do not represent the application domain's concept. The evaluation results of this approach do not tell us much. In fact, they rely heavily on keywords for their recovery process

and depend a lot on the developers to conform to the UWA framework. Conceptually, this approach seems viable, but it requires evaluation over the Web in the wild to examine the practicability of it. Furthermore, many Websites like Google, CNN and Facebook are made up of micro-applications within it. In fact, most Web pages consist of more than one micro-application. The concept of a Web application in Bernardi et al. [2008] seems to be a reuse of the conventional desktop applications, while this is not the case for the Web medium. We believe that, by redefining this concept, a different set of results for the investigation will be produced.

Studies were conducted to reverse engineer the conceptual user-centred models from Web applications. Di Lucca et al. [2006] proposed a reverse engineering approach to recover the conceptual model, so that structural information and UWA models can be extracted from the Web application. To recover UWA models, a static analysis process was conducted to Web applications to identify the entity types, semantic association types, collection types, node types and cluster types. A tool named RE-UWA was developed, and evaluated using three experiments. The first experiment consists of simple Web pages developed by undergraduate students, each having only one of the UWA concepts implemented. The second experiment was conducted on a small size Web application developed by graduate students. This application allows users to make predictions on football matches. Finally, the third experiment consists of a large set of client-side pages downloaded. The first two experiments achieved precise identification of entities, collection, and associations. However, the third experiment showed a lower precision when identifying entities and associations. Di Lucca et al. [2006] demonstrated that this approach is feasible and has good effectiveness, but refinements to the approach are expected to improve its effectiveness and precision.

Distante et al. [2004] demonstrated a technique for reverse modelling, so that it will be able to comprehend Web application transaction design and implementation, in the absence of original program documentation. The study attempts to recover information from an existing Web application, and feed it as design input to an extended version of UWA transaction model. It is a demonstration of the group's concept, and it was reported that the inspection of the main steps of the reverse modelling technique was carried out by a human. Thus, they suggested an automated tool as their future work for this approach.

Conceptual models such as Di Lucca et al. [2006], employ the visual representation of an application, such as UML, and their associated automated environments to comprehend the source code claims to provide potential improvements. However,

Hendrix et al. [2002] highlighted that these studies are generally limited in scope, and do not necessarily desig to scale up for industrial practice. The team investigated the effectiveness of visualisation representation when performing source code comprehension. The investigation compared the Control Structure Diagram (CSD) results with 2 groups of participant results, one involving a senior-level class of software engineering students, and a few graduate students, while another group was made up of undergraduate students with little programming experience. It was reported that the CSD responded positively to shortening the response time, and have better correctness in the experiments. Thus, it was suggested by Hendrix et al. [2002] to include the visualisation representation approach, as it can improve productivity and reliability.

Vectors

Vectors can represent the pattern structure or process of a source code, so that further analysis can be conducted to comprehend the code. Studies such as Tsantalis et al. [2006]; Dong et al. [2008a] use this method to model the relationships between classes for the matrices, so that the matrices can be manipulated easily, and they can clearly convey the matrix's concept to engineers and computer scientists.

In an investigation by Canfora et al. [1996] to improve the techniques to identify objects from the code, a statistical approach was introduced. Although the objective of the study was to provide a solution to make software comprehension easier, the concept from this approach can be deployed to assist code comprehension for Web applications. It was reported that a major problem in software comprehension is to comprehend the relationships between the system's components. The technique suggested by Canfora et al. [1996] exploits a graphical method, and applies a statistical technique to the interconnections of a bipartite graph. Using an iterative clustering algorithm that terminates when the graph forms a candidate object the graph was plotted. An object in this study is referred to as a collection of data items and routines. An experiment conducted to evaluate the quality of the proposed algorithm demonstrated that it identifies the routines that are likely to introduce undesired connections, and the case studies demonstrated that some degree of human interaction is required.

Searching for Instances

A common method to search for a pattern when comprehending the source code is to look for its instances. Quite often, as seen in Tsantalis et al. [2006]; Dong et al.

[2008a], this method is combined with other techniques to further comprehend the source code.

Stencel and Wegrzynowicz [2008] proposed an automatic pattern recognition method that could detect nonstandard implementations of design patterns. The motivation for this proposal is because the existing approaches are not perfect, and sometimes fail to capture the source code intended. The aim of this proposal is to capture the intent of created patterns, as well as to discover as many variants of implementation method as possible. A tool was trained to recognise four design patterns - the Singleton, Factory Method, Abstract Factory and Builder - and tested against two other state-of-the-art tools. The demo source of "Applied Java Patterns" (AJP), which provides an exemplary Java implementation of Gang of Four patterns, was used to test the three tools. The results proved that this method was flexible and efficient.

The techniques used in Stencel and Wegrzynowicz [2008] have proved a valuable approach for our purposes. However, this study was based on Java source code. Hence, the degree and complication of having a mixture of different languages in a similar application was not tested, and it does not attempt to examine the associations of the different patterns, and the purpose of having the pattern, like our proposed work.

Searching for instances of a pattern in a Web page when comprehending the source code can also be extracted from the DOM tree, HTML tree and JavaScript Abstract Syntax Tree. A study by Bellucci et al. [2012] demonstrated the applicability of using these techniques to identify the User Interface (UI) components on a Web page. However, when using these techniques, only components at an instance in time can be discovered. Furthermore, to detect widgets, other code comprehension techniques are still required on top of these techniques, to relate the different components before an inference to a widget can be determined.

Annotation & Transcoding

Developers can annotate the application either directly in the source code, or separately as documentation. Rajlich [2000] demonstrated the usefulness of having a separate incremental documentation for a piece of software using the Web. Software was developed as a hypertext notebook medium for programmers to record the understanding and observations about the software. Each component is annotated in different partitions of the documentation, starting from the root file. This form of annotation proves useful for the developers of the application. However, these documentations are not available for Websites, and documentation in the source code of the Web document

would be useful.

Annotation can also be used as a technique to insert additional information, such as semantics and notes, into the Web page's source code. Asakawa and Takagi [2000] developed an Accessibility Transcoding System, so that completed inaccessible Web pages can be converted into accessible Web pages. Two components are used for the annotations; one for structural annotations, and the other for commentary annotations. The structural annotations are used to reorder the visually fragmented groupings according to their importance, while the commentary annotations provide useful descriptions of the contents.

To overcome the fatigue of site-wide annotation authoring using the technique described by Asakawa and Takagi [2000], a new algorithm, "Dynamic Annotation Matching" was developed by Takagi et al. [2002], to automatically determine the appropriate annotations, and an authoring tool, "Site Pattern Analyzer", was also developed to support the annotator, visualising the annotation matching status. An experiment was set up to test for the effectiveness of our algorithm and tools. USA Today's Website was chosen for the experiment. It was reported that, in total, 245 annotation files were created in 30 hours and 20 minutes, thus they succeeded in creating annotation files for every target page without skipping any pages. These results demonstrated that annotation-based transcoding could improve Web Accessibility for visually impaired users; however, the time taken for annotation authoring is a bottleneck for this technique.

The approach used by Takagi et al. [2002] demonstrated the power of annotation, but it has to be used with great care to tackle the issues with this technique. This study was conducted in 2002, and during that time most Web pages still used the 'Page Model' concept described by [Maurer, 2006]. Hence, it does not include Web pages with Web widgets, and dynamic micro content.

More recently, Harper et al. [2006] suggested that creating ontologies of Cascading Style-Sheets (CSS) so that Web pages can be transformed to improve the inaccessible Web. The paper led to the introduction of the Structual-Semantics for Accessibility and Device Independence (SADIe) project, which uses the semantic annotations of a Website's CSS to transcode into a Web page format suited for screen readers that output audio sequentially. However, the SADIe concept could be applied to other applications, such as to transcode CSS annotations to generate AxsJAX framework code [Lunn et al., 2009a], and it can also be coupled with a framework for identifying strategies that visually impaired users employed to overcome with the difficulties when

interacting with Web content [Lunn, 2009].

So far, only studies applying transcoding and annotation to improve Web accessibility directly have been covered. However, Chen and Shen [2006] demonstrated that transcoding could also be applied to transform Web pages from poor compliance to become standard-compliant. This study only covers static content without addressing the accessibility of the conceptual designs of the widgets. If users are unable to grasp the operational concepts of the widget, they will not be able to operate the widget to accomplish their task, or extract the relevant content. Nonetheless, Chen and Shen [2006] study supplemented the idea that the WIMWAT project can also assist developmental tools that are intended to inject WAI-ARIA syntax into Web pages with widgets to improve the accessibility of their widgets.

2.2.4 Attempts to Identify Graphical Objects and Web Widgets

The idea to identify objects on a Web page has been attempted a few times, but due to the reverse engineering barriers raised in §2.2 these projects were often placed on the back burner. In this section, related studies will be visited.

Generally, there are two common approaches to identify an object from a Web page. One way is to sift through the source code for clues or instances of the object (textual based), or the presented content can be treated as an image (pixel based) such that image processing techniques can be used to identify instances of the object.

Text Based

There is a lot of lexical information extractable from the text in a Web page. However, the Web delivers content through the textual and multimedia medium. Karanam et al. [2011] investigated extensively the roles of text and graphics information when attempting to locate Web widgets in a Web page. This study highlights the importance of including graphical information in the analysis, both during the development stage and the reverse engineering process. It did not investigate how often graphical information is found alongside textual information. Thus, although this concept proves ideal, in practice this content does not always exist. Neither did Karanam et al. [2011] formally define the listed widgets. Objects like logo and content are classified as widgets, however, in our proposed research, these objects are considered merely as part of the layout of the page. Furthermore, the technique proposed by Karanam et al. [2011] was only evaluated over 8 Websites chosen based on a vague selection criteria. The

evaluation methodology for the study is not thorough enough to suggest any concrete conclusion, and the selection process can be manipulated to favour the experiment.

Little et al. [2007] identify the Web user interface to simplify user activities, so that end users can record their interactions with all forms filled, buttons clicked and menus selection when performing a business process. This information can be documented along with the recorded interactions and shared on a wiki. Through this method of sharing, others can automatically execute the process themselves, while Bolin et al. [2005] describe a new programming system to provide a platform for automating and customising Web applications, so that the user should never have to mess with the HTML source of the Web page.

Both studies, Little et al. [2007] and Bolin et al. [2005], are limited to identifying form fields and buttons. They do not go beyond the identification process of graphical user interface objects to understand the conceptual designs behind the processes, so that users can be informed of the purpose of conducting the task. Furthermore, their concepts are evaluated over a very small sample of six or less Websites. The issues exposed are not sufficient to conclude the approaches for the general Web.

Pixel Based

Reverse engineering the interface structure of an application can be done through examining the pixels in the graphical user interface. Since the concepts of presenting Web pages in the Web application architecture are similar to analysing the software interface structure concepts [Myers et al., 2000], we will visit some of these studies.

Dixon and Fogarty [2010]; Dixon et al. [2011] suggested the pixel-based reverse engineering technique to identify objects in the user interface structure for generic applications. This technique requires the user to specify the area to analyse with a pointing device, and the system will attempt to identify the objects. Although it can be applied across a range of applications (both desktop and Web based), it targets users with the physical capability to control a pointing device. However, it cannot be generalised for visually disabled or motor impaired users. Furthermore, relying on the visual objects to identify the tasks and processes can mislead the users, and may not provide sufficient information for the user to continue their intended task. The users must also have an idea where and how much to highlight before the system can be useful, thus the usefulness of the project is doubtful.

2.3 Using Ontology to Classify Web Widgets

Making use of available information in the content layer and DOM as a metaphor of components and objects in the page can help to identify the different user interface components, or make semantic connections of certain content in the page. This appealing approach applied with the domain's ontology is used widely for modelling different conceptual objects, especially to assist Web searching [Fazzinga and Lukasiewicz, 2010; d'Aquin and Motta, 2011] and bioinformatics analysis [Chen et al., 2009; Kim et al., 2005].

In the course of automatically identifying or predicting widgets from a Web page, a knowledge based on the different types of widget with semantic definitions will be required for the system to make these forms of deduction. Often ontologies are used to capture the concepts of the terms the vocabulary of the domain intends [Chandrasekaran et al., 1999]. Ideas of using the Semantic Web to include ready-to-use Web widgets to create an application [Mäkelä et al., 2007] have surfaced briefly with limited discussion about how widgets can be classified or modelled. In order to model the concepts of different types of widgets, a widget classification approach using the concepts of Semantic Web should be explored to aid the different types of widget definition. Presutti and Gangemi [2008] describes how to use ontology to describe the concepts instead of the logics for the design patterns. However, a simple example is presented, and it does not go deeper to explain how this concept can be applied to more complex scenarios where the widgets are often found. These studies raised interesting ideas, but they did not explore their ideas further. In WPF, these concepts are applied to a real world scenario by incorporating them to assist our widget prediction approach.

From these studies, the depth, degree and difficulty of applying these concepts as part of the classification system in WPF can be seen. Often these forms of technique require their users and developers to be experts in the area. To provide a rounded solution so that users of the Web, with a different range of technical abilities, can also understand the classification system, the technique must be flexible enough to be approached from different perspectives.

Martín et al. [2010] suggested that an abstract interface model made up of ontology widgets could be used to design and develop Accessible Web applications. However, only one type of widget was covered in the study, with only one Web page examined. Furthermore, the ontology used in the study only includes the HTML elements of the document. The extent of these studies is focused on a very small pool of Websites. They only cover a few ideal cases and they are unable to capture the vast variations of

how even a single type of widget can deviate. Unlike these studies, the proposal of our ontology covers all forms of textual content in the source code to derive the process. Combining the textual information from the content, styling and behaviour layers of the Web page and the DOM, both textual and conceptual information are included in our ontology to model a type of widget. Furthermore, a wider range of widgets is included in our prediction approach, and the location of each widget found in the Web page is returned. Again, the approach Martín et al. [2010] suggested requires users to be experts in the area. For the framework to assist broad range users of the Web, it has to be able to be simple enough for most Web users to interpret the concepts of the widget, while covering the depth so that our system will be able to predict the type of widget.

There is a range of semantic Web technologies available for use in modelling and querying the ontology. The W3C has published a number of standards, such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL), which is more expressive, to facilitate the semantically rich information exchanges [Hitzler et al., 2011]. OWL is designed to represent information about categories of objects and how objects are interrelated, and information on the objects themselves [Horrocks et al., 2003]. It is designed to fit into the rest of the Semantic Web languages, including XML and RDF, and maintain compatibility with existing languages such as SHOE [Heflin et al., 1999], OIL [Fensel et al., 2001] and DAML+OIL [Connolly et al., 2001]. These characteristics make OWL a good fit as a tool to model widgets in our framework.

More recently, in 2008, the SPARQL Protocol and RDF Query Language were introduced by W3C to provide a querying facility for RDF. Advances, such as SPARQL, are examples of the interfacing technologies available. However, as a first step to demonstrating the feasibility of our approach, and minimising the complexity of surrounding technologies, we have hardcoded the widget definitions after modelling them in WIO. Tools like Protégé[8] will provide a stable platform to work with OWL and RDF to model the widgets.

[8]Protégé – `http://protege.stanford.edu`. Last accessed 18th September 2012

2.4 Summary

In this chapter we presented existing studies and projects to approach the challenges in Web accessibility, and the issues surrounding software/Web development and maintenance. Beginning with the different Web standards and recommendations, issues faced by Web developers and Assistive Technologies developers to cope with the pace of evolution, the extent of the recommendations covered, along with the heterogeneous nature of the Web, all contribute to the complexity this thesis investigates. Scouting the approaches applied in other domains like software engineering, text mining and semantic Web for alternatives, this chapter critically reviewed some of the related approaches from the perspective of our proposed research.

These reviews expose the strengths and weaknesses of the approaches covered, and provide insights into the techniques for the type of combinations applicable to recover the concepts of widget design. A discussion was presented to highlight the difference between closely related studies and our approach. It illustrates the degree and novelty of the WPF. After reviewing the existing studies, we acknowledge that there will be flaws to every approach or set of approaches selected for our investigation. However, we believe that the WPF approach has its cause to improve the current accessibility challenges faced by dynamic content, and provide a solution by predicting widgets to indirectly improve the experience of the Web page.

Chapter 3

Classifying Web Widgets

In this chapter, we present our approach to assisting Web widget detection and to meeting the challenges faced when communicating widget design pattern concepts. Our approach addresses the ambiguity issues between different definitions of widget design patterns provided by widget design pattern libraries, as well as providing a paradigm enabling developers and end-users to communicate with a common understanding. To demonstrate our approach, the popularity of different types of widget are analysed and presented. The purpose of this analysis is to select the types of widget to include in our widget classification approach, and the WPF investigations. In §2.1.1 and §2.1.2, we discuss the importance of making Web pages accessible along with the pace of development cycles of Web pages/applications, the issues faced when communicating ideas about widgets, and reusing design patterns applied on desktop applications to Web widgets.

Due to development cost, time and functionality of task, often more than one Web widget is presented to users at the same time on a page. This way of delivering content poses additional barriers to less physically able users using Assistive technologies. This is because when more than one widget attempts to update the content in the page concurrently, the user will not be able to distinguish which update is for which widget [Brown and Jay, 2008a]. Commonly, the Assistive Technologies are also unable to pick up this relationship.

Often, the user is required to be able to work out the behaviour of the components, and relate the components to distinguish one widget from another to address the issue. However, this will require the users to have a visual cognition of the entire Web page. To improve the situation, we investigated current widgets issues, focusing especially on identifying and classifying them. From the results transcript recorded by Jay et al.

[2010], blind and visually impaired users found it beneficial to be informed about the dynamic content. Thus, by investigating the methods model widget, we have proposed an approach to formally model different types of widget. This approach will provide documentation for the definitions of different types of widget, as well as provide cognitive organisation of conceptual designs for each widget when developing the Widget Prediction System (WPS).

3.1 Developers & Users Perception

Web developers, Web page designers, and users of the Web often refer to objects in a Web page differently, and these terms are often homographic in nature. This issue commonly becomes an obstacle due to the ambiguous definitions, and it causes confusion when communicating because the users of the Web pages and the developers perceive the objects differently.

Often users perceive widgets from the interaction, the experiences they had with the widget, and the way the content was delivered, while developers view it from the interaction, processes and overall task of the widget. An example of this type of object is the Ticker widget. Users may refer to the News Ticker and the Stocks Ticker widget in a Web page as different widgets. This is because they classify these widgets based on the type of information they present, the interaction components, or how the widgets were applied. In this situation, the user assumes a Ticker widget that conveys news is a News Ticker widget, while a Ticker widget that conveys stocks results is a Stocks Ticker widget. However, a developer will approach both of these widgets from their underlying processes. Thus, both of these widgets are the same from the developer's perspective, and they may refer to these widgets with a more generic name, such as a Ticker widget. This will cause a definition mismatch between the Web developer and the user.

Although the mismatch of widget definition may seem trivial initially, in some cases it can impose false expectations on the user that will result in a poor user experience. A common understanding of the definition is even more crucial for visually impaired users, because false expectations can result in the content not being accessible.

It is suggested that a taxonomy be constructed to bridge the differences in concepts and definitions of different types of widget. Only when these differences are ironed out

can the developers of Web pages and assistive technology, as well as Web users, communicate effectively. This is a vital communication paradigm for people working with or using the Web, especially Web page and Assistive technologies developers. Using this taxonomy, a standardised definition and terms can be used to assist communication between developers and users. Similarly, assistive technologies can also use the paradigm concepts to interpret the widgets on the page, while users can be informed about the features of the widgets. Using this information, the features of the widget can be understood by the users to retrieve the content delivered by the widget.

3.2 Widget's Taxonomy

To close the gap between the definition for a type of widget from a Web user perspective and the Web developers' perspective, and to reduce the ambiguity between the definitions, a taxonomy divided into four layers is introduced to formally define widgets. To begin, let's focus on a case study between the Carousel widget (figure 3.1) and the Slide Show widget (figure 3.2) to see how these widgets are similar and often misinterpreted. Then, using the four taxonomies, we will demonstrate how these taxonomies can help to narrow the gap between the definitions of widgets and improve communications between developers and users.

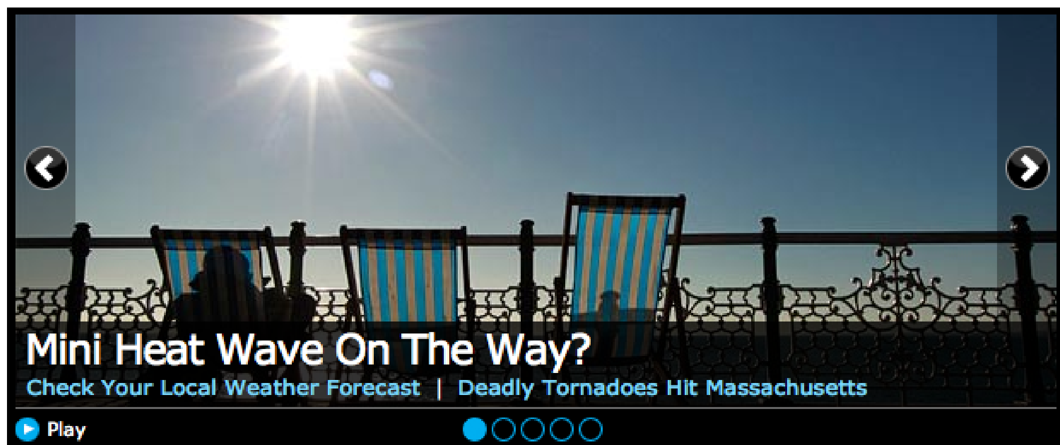


Figure 3.1: Carousel widget from Sky.com

Close examination of both figures would highlight very similar features provided by the two types of widget – for example, the definition between the 'Next' and 'Previous' buttons is ambiguous. Due to this some may classify figure 3.1 as a Slideshow

Figure 3.2: Slide Show widget from Yahoo.com

widget since it has a 'Play' feature at the bottom left hand corner. A disagreement between the definition of a Carousel and a Slide Show widget can be seen from this discussion, because the definition of a widget can vary from one perspective to another. This case study suggests a niche for a widget's taxonomy platform so that widgets can have a formal definition, making it easy to understand the conceptual designs and reducing misinterpretation.

The widget's taxonomy platform proposed suggests that the objects that form the widget can be divided into four groups – Widgets, Components, Tell-Signs and Code Constructs – to allow expansion and maintenance in the model. Each of these groups relates with the others via the relationship between the objects/classes within the group. As seen in figure 3.3, each group is represented as a layer in the diagram, and the layers have the precedence as follows: At the top, it is the Widgets group that contain the most abstract concepts of a widget, which is defined by the unique combinations of classes in groups beneath it. Each type of widget will have a number of components objects/classes in the Components group that interoperate with other classes in this group. Every component will have one or more traits that form the Tell-sign classes in the Tell-Sign group that make one component different from another. Finally, every tell-sign class/object will have one or more code construct objects in the Code Constructs group.

Using the hierarchical arrangement in the paradigm, objects/classes within each layer have to be unique. This association enables them to be modified, removed and included without affecting the rest of the group. The unique property of the classes enables them to be independent from each other, so that changes to a class will not affect other classes in the same layer. Between the layers, modification or removal of a class from a high layer does not affect the lower layers, but it is not the same for classes in the upper layers as they are composed of classes from the lower layer(s).

Now, the four layers can be approached in two ways. The first way is for users

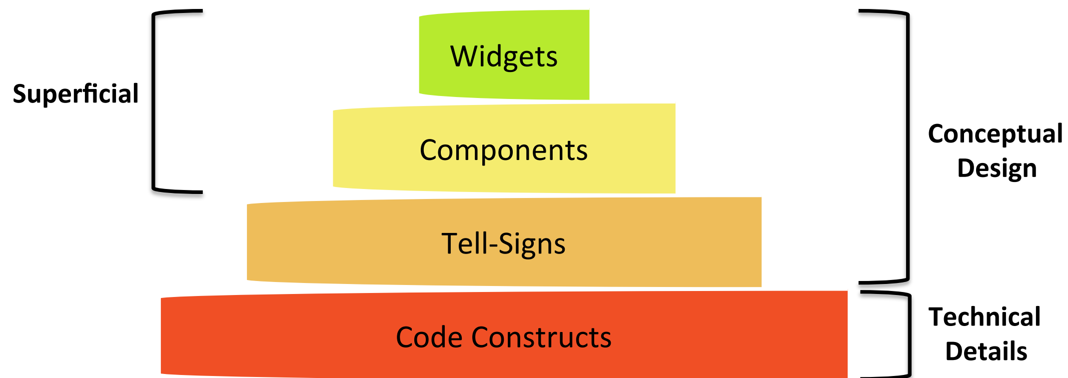


Figure 3.3: Widget Identification Ontology (WIO) paradigm: A diagram to depict how the objects in the different groupings/layers related with one another can be used to define a widget.

of the Web; these users can approach the model from the top-down approach to whatever technical abilities. Classes in the top two layers are intended to be interpretable for most laypersons. We believe this is possible because these layers only cover the overview concepts of the widget in the top layer, and the superficial concepts of the widget's components in the second layer. The second way is for the more technically inclined people. They can approach the model either from a top-down or a bottom-up approach during development. Using the bottom-up model, understanding a type of widget for the two types of actors will meet in between, and it will bridge the communication gap between the broader range of users and the developers, so that they can effectively utilise the widgets.

In our previous study Chen and Harper [2009], some objects of a widget were introduced when examining the Auto Suggest List and Carousel widget. These objects were also formerly known as tell-signs in Chen and Harper [2009] and include the Display Window and list of content to be displayed, are not to be confused with the tell-signs described in this thesis. These objects are actually components of the widget and they are classified as Components classes here. When examining the Carousel widget, a Display Pointer was also used to store the current location of the list that the widget is displaying. This additional component is also added to our Carousel widget definition.

3.2.1 Widgets Layer

Each layer in the widget classification hierarchy consists of a set of taxonomies, so that the relationships between the classes can be established to define a widget. The Widgets layer is the highest and the most abstract layer in our classification system. Classes in this layer provide the abstract concepts of a widget.

Scouting through knowledge of widget design patterns from libraries such as Yahoo! User Interface Design Patterns Library[1] and Welie.com[2], and JavaScript libraries like jQuery[3] and Dojo[4], a consensus for the different types of widget definitions is presented. Approaching the definitions from the conceptual perspective, it can be seen that nine atomistic types of widget mainly surfaced in our investigation [Chen et al., 2013]. Although many more different types of widgets are listed in these libraries, the other types of widget are mere combinations of the nine types of widget applied differently.

As seen in figure 3.4, nine types of widget are listed in this layer as an abstract concept. Two subsets of widgets are included under the Ticker widget type to demonstrate the capability of this approach to model similar concepts of widgets and the expandability of the approach.

Based on the definition of the type of widget, every type of widget shown in figure 3.4 can be thought to have unique traits that differentiate one from another. Using this taxonomy, both the News Ticker widget and the Stocks Ticker widget can be seen to be sub classes of a Ticker widget as discussed in §3.1. With this capability, widgets with similar characteristics can be grouped together under a generic categorisation of that type of widget.

3.2.2 Components Layer

The Components Layer is a layer below the Widgets layer that contains objects that are the possible components that make up a widget in the abstract Widget layer. It is divided into two groups to differentiate the types of components as seen in figure 3.5. The 'VisibleComponents' classes are concepts related to the physical User Interface

[1] Yahoo! User Interface Library - `http://developer.yahoo.com/yui/2/`. Last accessed on 18th August 2012

[2] Welie.com Patterns in Interaction Design - `http://www.welie.com/patterns/`. Last accessed on 18th August 2012

[3] jQuery JavaScript Library - `http://docs.jquery.com/Main_Page`. Last accessed on 10th November 2011

[4] The Dojo Toolkit - `http://dojotoolkit.org/api/`. Last accessed on 12th February 2009

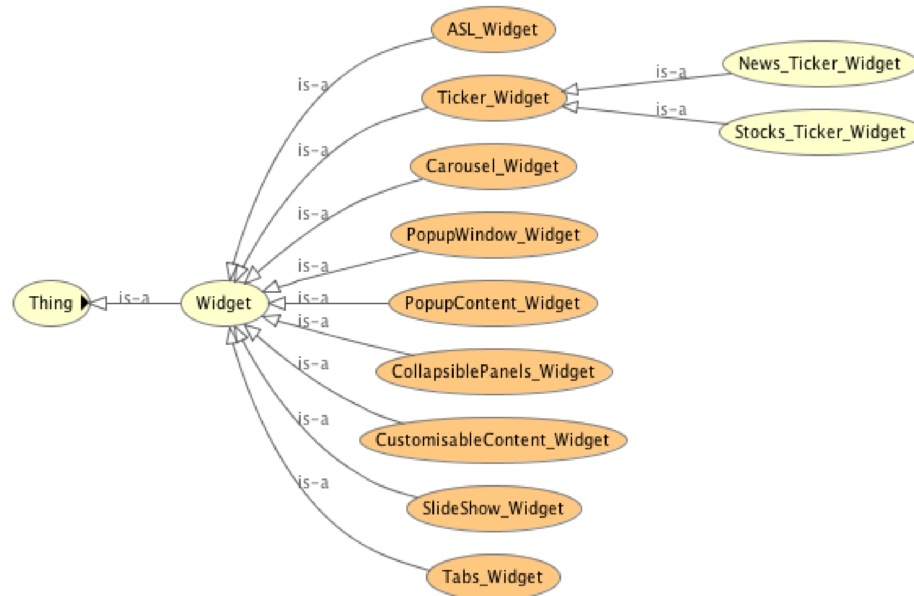


Figure 3.4: Taxonomy of the Widgets layer.

(UI) components of the widget (buttons, text fields and key strokes entered), while the 'CodeComponents' classes are conceptual processes in the code that are the behavioural aspects of the widget (an incremental process, a pointer variable and a time triggering event).

The layers in the hierarchical structure, presented in figure 3.3 demonstrates that every widget class defined in the Widgets layer must consist of a unique combinations of the Components class in the Components layer. In order for this concept to be possible, the Components classes in the Components layer must be unique. This condition is to ensure that when the Component classes are employed to define a Widget class, no ambiguity between the Widget classes definition will arise. However, the Components classes also have to be generalisable such that they can be interoperated with different combinations of Component classes. These characteristics of the classes are important because the unique combinations of widget characteristics differentiate one type of widget from another in the Widgets layer.

There are a number of different types of components used in figure 3.5. Most of these classes are formed by a combination of classes in the lower layers of the hierarchy in the WIO's paradigm. Some axiomatic definitions were assumed when creating the terms for the taxonomy. The following is a list of the axiomatic definitions of different Component classes assumed in this layer:

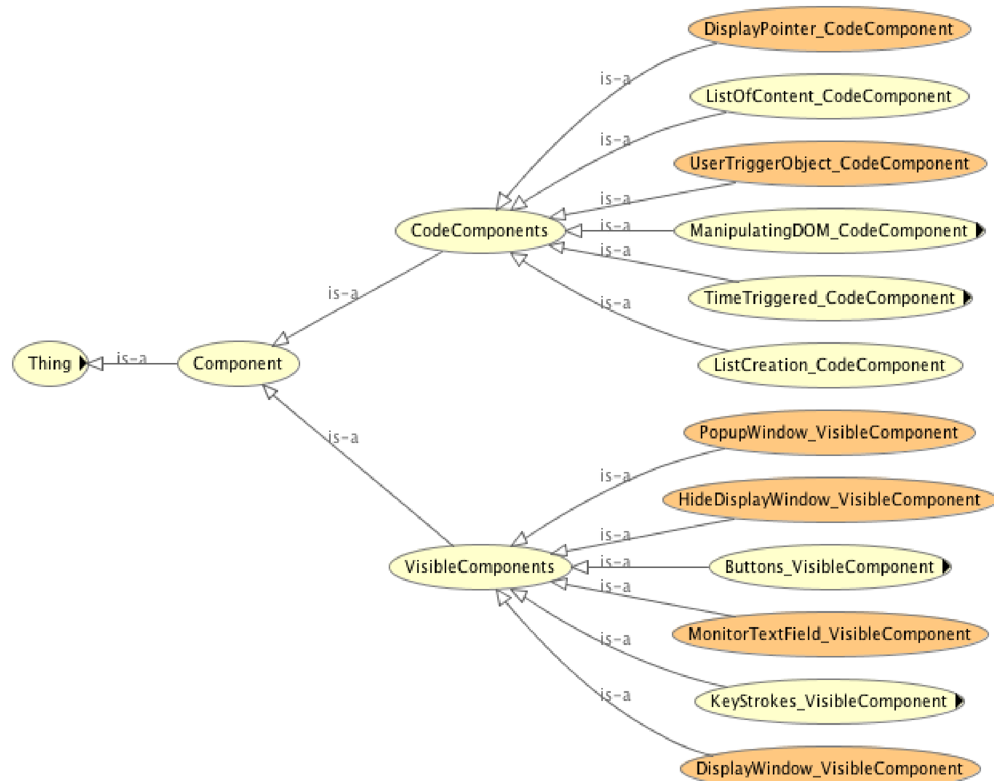


Figure 3.5: Taxonomy of the Components layer.

Button This can be a form button or a link in the form of text/image that will be triggered by a user event to execute the handler.

Display Pointer A variable that is used to store the address of the current displaying item from the list.

Display Window An area in the Web page where the content is manipulated by the widget. Often this will be an element within the DOM.

List of Content A temporary container/memory/cache to store a stack of content to be delivered to the user.

Text Field An object in the Web page that the user interacts with, such that the user can enter text for the widget to process.

Time Triggered A trigger that triggers after a delay at fixed intervals.

The full extent of the Components layer taxonomy cannot be captured in a single diagram due to the size of the diagram. However, classes/objects are divided into

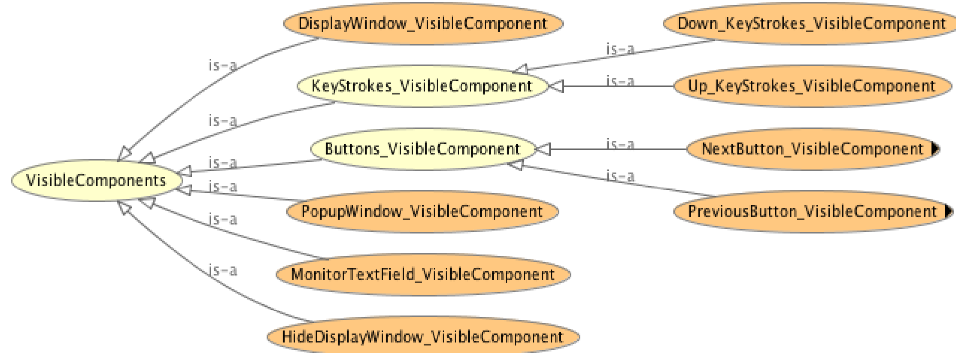


Figure 3.6: Taxonomy of Visible components objects in the Components layer.

two sub classes: one for visible components and the other for conceptual components in the code, as seen in figure 3.5. Some of the classes/objects in these classes are further grouped into smaller classes to provide more semantics in their relationships. Focusing on the Visible components classes/objects in the Components layer as seen in figure 3.6, this group consists of conceptual processes that are involved with interaction objects in the graphical user interface of the Web page. These classes/objects include users keystrokes monitoring, Display Window styling manipulation, launching popup windows and types of conceptual button components.

Zooming in on the visible buttons component classes illustrated in figure 3.7, the two types of buttons used to define the widgets covered are the 'Next' and the 'Previous' button. The generic concepts for both of these buttons will propel the presented content either forward or backward when the user clicks on the 'Next' or 'Previous' button respectively. However, this type of concept can normally be applied in two ways for each type of button. The first method will loop back to the start of the list of content when it reaches the end of the list, or vice versa, while the second method remains at the end of the list of content when it arrives at the end, or vice versa.

Due to the possible types of processes, both of these buttons are further divided into two subsets of buttons. For the 'Next' button classes, the `Loop_NextButton_VisibleComponent` class will redirect the presented content back to the start of the list when it arrives at the end of the list, while the `Finite_NextButton_VisibleComponent` class will remain presenting the content at the end of the list when it reaches the end of the list. On the contrary, for the 'Next' button classes, the `Loop_PreviousButton_VisibleComponent` class will loop the presented content to the end of the list when it arrives at the start of the list, and the `Finite_PreviousButton_VisibleComponent` class will

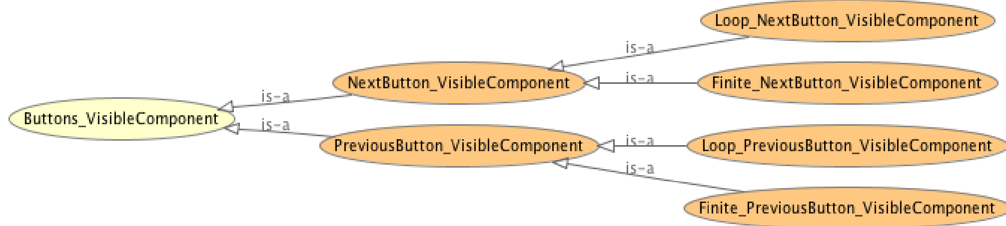


Figure 3.7: Taxonomy of Visible Buttons components objects in the Components layer.

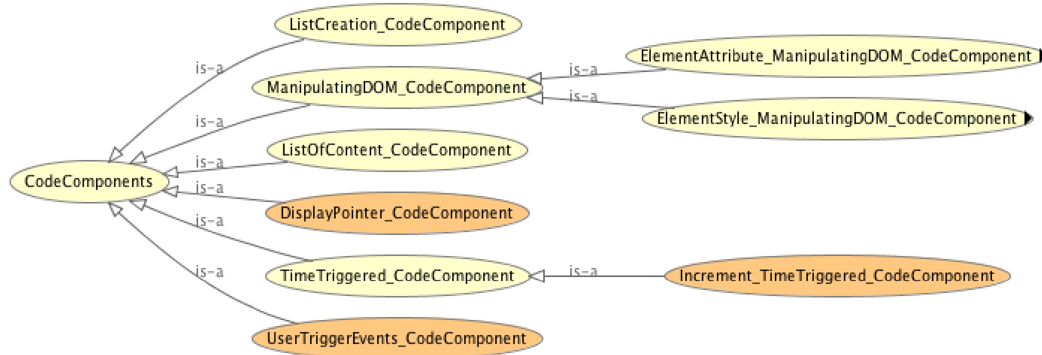


Figure 3.8: Taxonomy of Code components objects in the Components layer.

remain presenting the content at the start of the list when it reaches the start of the list.

Unlike the Visible components subset, the Code components subset in the Components layer are conceptual processes in the source code that are required by the developers to complete the processes the widget is programmed to conduct. As shown in figure 3.8, these classes/objects are conceptual processes that are required to complete the processes that are part of the developer's design. This subset includes objects/-classes such as Display Pointer, user triggered events, List Creation, List of Content, Time triggered events and manipulation of the DOM elements.

The Code components subset is further subdivided for `ManipulatingDOM_CodeComponents` and `TimeTriggered_CodeComponent` classes. `Increment_TimeTriggered_CodeComponent` class conduct an incremental process each time the time-triggered events are activated. This conceptual process will assist designs that propel processes forward automatically using the system's time or conduct computations based on the system's time. Notice that the `manipulatingDOM_CodeComponents` class is further divided into two classes. This is because, besides changing the content in the element – which is a generic concept – the element can be also manipulated, either by its attribute

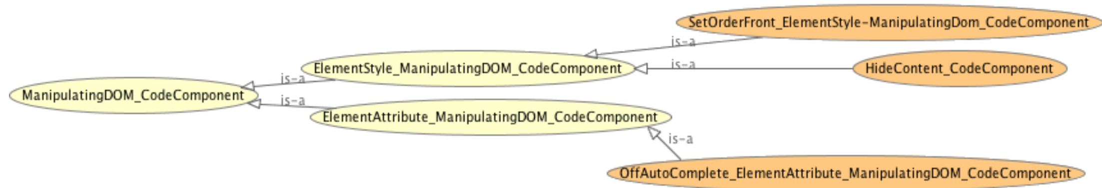


Figure 3.9: The subset taxonomy of manipulated DOM objects under the Code components objects in the Components layer.

or styling properties. We will take a closer look at these two types of manipulation in figure 3.9.

Manipulating the styling of an element to make it appear and disappear in the Web page is a method developers employ to present the targeted content, and hide the content that is not desired to be shown. From figure 3.9, two methods are covered. The first method sets the order of the element to the highest value (`SetOrderFront_ElementStyle-ManipulationDom_CodeComponent`), while the second method, `HideContent_CodeComponent` class, changes the `display` or `visibility` property to hide the element. Some widgets may alter specific element attributes that require to be checked. The `OffAutoComplete_ElementAttribute_ManipulatingDOM_CodeComponent` class ensures that the `autocomplete` attribute of the element (usually a text field) is set to off. Often this is a technique used to inform the Web browsers not to suggest content to the user when they are filling up a text field.

3.2.3 Tell-Signs Layer

Every Components class in the Components layer will have at least one trait; commonly, instances of these traits exist within the code to provide clues that a component is present in the page. Normally, one or more traits is/are required to be identified before the existence of a tell-sign can be conclusive. The tell-signs for a Components class are determined by selecting the common traits of a component that are provided by the different user interaction design patterns and JavaScript libraries.

Tell-sign classes are used to define a Component class in the Components layer as shown in figure 3.10. For this concept to be implemented, every tell-sign class must be unique, so that a unique combination of tell-signs classes defines a Components class in the layer above. Similar to the Components classes, tell-sign classes have to be unique to ensure the correctness of the Components classes' definitions and minimise

ambiguity among these classes.

Components are objects that have properties to carry out a task for the widget, thus they are vital to realise the concepts that the developer designed the widget for. To identify widgets, the concepts of the components must be identified first. However, deriving concepts from code is difficult Gamma et al. [1995]. Often, not all the required instances of these concepts can be found within the code. Instances of these traits provide clues that a tell-sign used to define a component exists, with a unique combination of tell-signs; a component can be discovered from the source code of the page. We believe that tell-signs are like the objects in the above layers – Widgets and Components. They are reusable by different components to define their properties when defined properly.

Every component will require one or more tell-signs to define them, although the tell-sign classes are reusable to define other types of components, the tell-sign classes that are related are grouped under a generic class in order to impose a semantic relationship between them. Every class in the Tell-Signs layer is unique so that a unique combination of tell-sign classes can be formed to define a component class in the Components layer. Due to the uniqueness of some components, some tell-signs are specific characteristics to define a component. Thus, in this thesis tell-signs that are used by different components to define them are known as common tell-signs, while tell-signs that are used by only one type of component will be referred to as private or specific tell-signs.

Similarly to the Components layer, most of these classes in figure 3.10 have semantic relationships with the lower layer in WIO paradigm. However, some assumptions were made when creating the terms for the taxonomy. The definitions of the terms used in this layer are listed as follows:

Auto Complete This is an attribute provided by the form's text field element. When set to off the browser will not provide the user with a list of previous history of what was entered in this field.

DOM Element Order High To change a DOM element to have higher order when displayed to the user. This means that it is less likely to be blocked by other DOM elements stacked within the same display area.

Event Handler This refers to a set of code to be executed when an event is triggered.

Event Listener This is a monitoring process to check if the specified event has occurred to a DOM element.

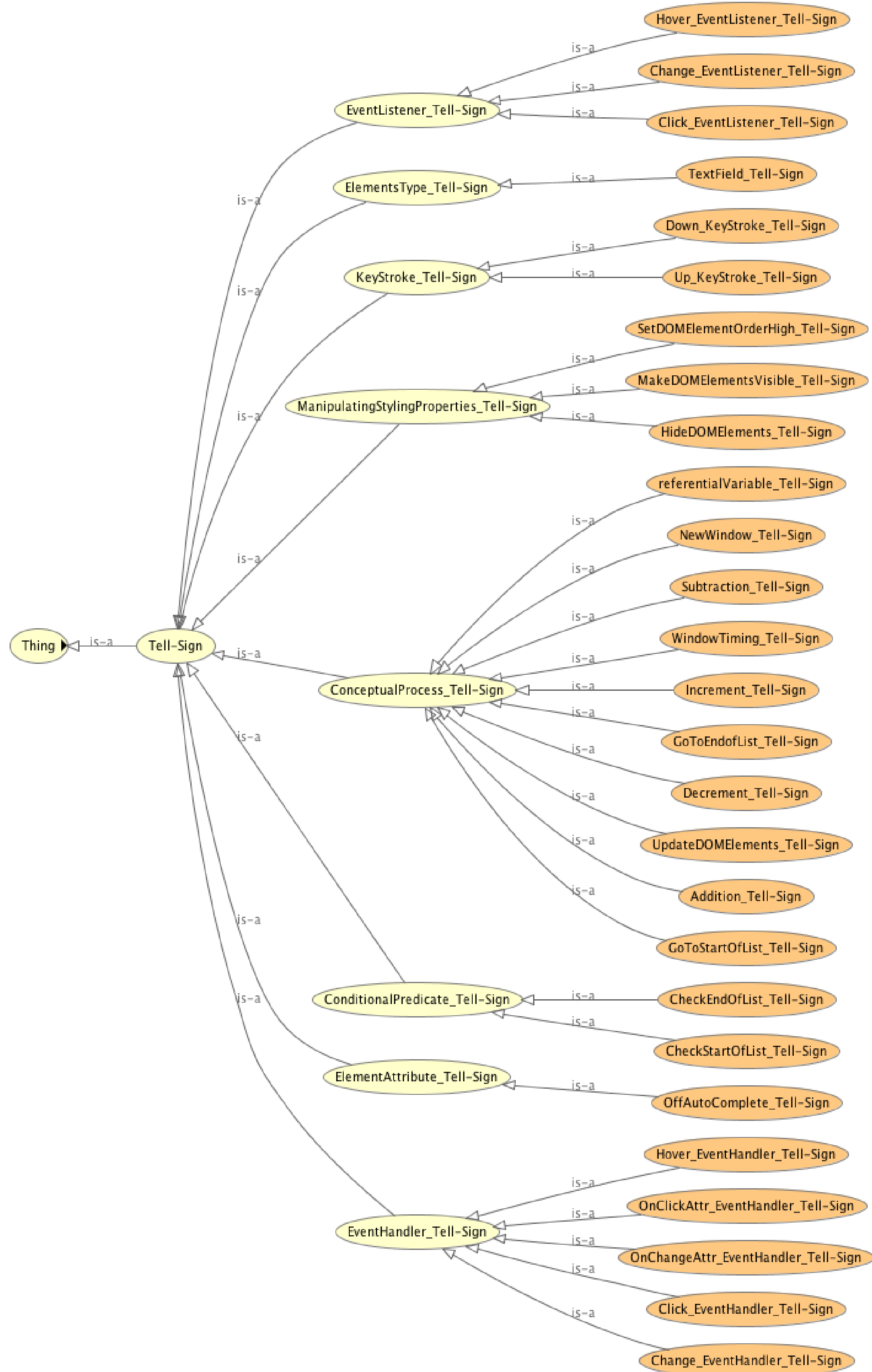


Figure 3.10: Taxonomy of the Tell-Signs layer.

Key Stroke This is a monitoring process to poll if a particular key is pressed by the user.

New Window A separate window that launches after an event is triggered.

Referential Variable A variable used to store the address location in an Array/list of content. Developers use this variable to refer to a particular location in the Array or item in the list of contents.

Window Timing A background timer provided JavaScript that triggers an event after a fixed delay.

3.2.4 Code Constructs Layer

Identifying widgets from the Web page's source code requires the system to analyse the code and the DOM, so that clues that resemble a tell-sign can be discovered. The Code Constructs layer contains classes that refer to a set of patterns in the code that a developer may use when developing the widget. Combining the pattern of code constructs found with other patterns of code constructs discovered, this combination of classes can be used to infer a tell-sign class in the Tell-Signs layer.

Figure 3.11 shows the taxonomy for the Code Constructs layer and how the different code construct classes are divided into subsets. All classes in this taxonomy are unique to ensure a unique combination of classes will correctly infer a unique tell-sign class in the Tell-Signs layer.

Commonly, when programming a conceptual process for a task, a simple concept can be approached and applied in a number of ways. Thus, the code constructs listed here are anecdotal methods to develop widgets suggested by design pattern libraries and JavaScript libraries. Since the Code Constructs layer is the lowest layer in WIO paradigm, each class defined in this layer is assumed to be the axiom patterns of an instance of physical code construct. Like the other layers, the Code Construct layer can be evolved, and additional methods can be included or modified when required. A list can be found in Appendix A that illustrates the definition of each class in the code construct form.

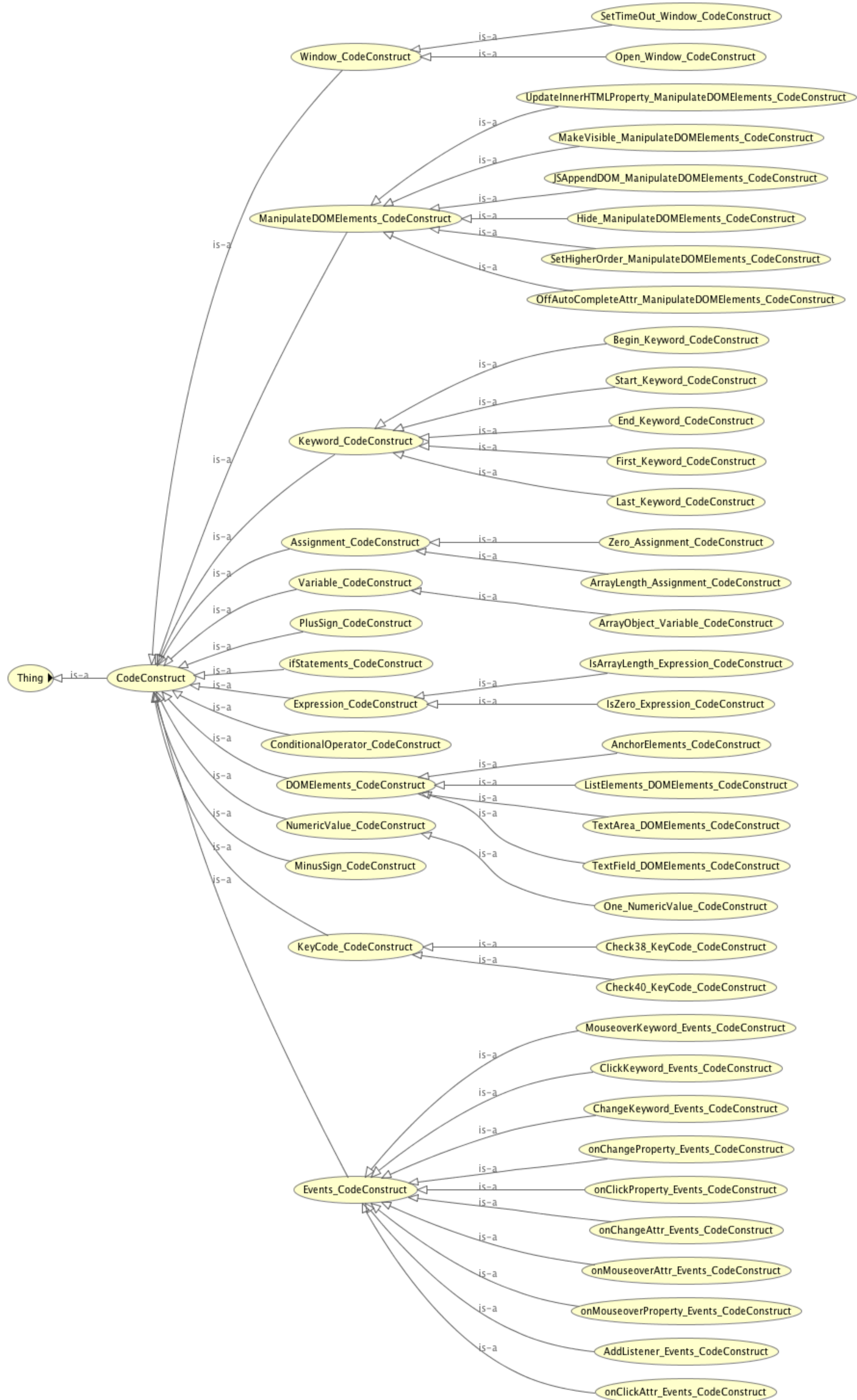


Figure 3.11: Taxonomy of the Code Constructs layer.

The definition of classes in the Code Constructs layer will affect definitions of the classes associated with it in the Tell-Sign layer. When modifying the definition of a class in the Code Construct layer, care has to be taken, as it will also affect the definition of classes in the above layers that are associated with the class. A description of the ontology for each widget will be discussed in the following sections, where a demonstration of different classes from the different layers will be presented.

3.3 Widget Identification Ontology (WIO)

Depending on individual's experiences, knowledge of the matter, application of the widget, and perception, a widget can be defined differently; sometimes this may even be due to historical reasons. Commonly, the issues faced with the naming and definition of different types of widget can cause communications to break down. This becomes a barrier that hinders development progress, communications between users and developers, and means users cannot benefit from what the widget was intended to do. An example of this would be the Ticker widget. To a Web page user, a News Ticker widget and a Stocks Ticker widget may be different types of widgets. However, from a developer's point-of-view, both of these widgets are developed in the same manner, thus they are the same, and the developer will classify this as a generic type of widget such as the Ticker widget. This example demonstrates how the different point-of-view affects the definition and the naming of the different types of widget.

Categorising the type of widget should be consistent. This is because, when communicating with these concepts, ambiguity can cause false expectations and it can affect the user's experience of the Web page. Classifying and identifying the widgets based on their User Interface (UI) components and conceptual designs will provide users with functional information and utilisation expectations; thus, it will improve the user's experience.

3.3.1 Widget's Granularity

The granularity of the reverse engineering work conducted often plays an important role in a successful recovery of development concepts and the identification of widgets. Using the taxonomy discussed, widgets can be distinguished and related to one another, thus reducing the ambiguity between the different types of widgets defined.

Choosing the correct amount of granularity can ensure the right balance between extracting sufficient information for the identification task, and not requiring too much computation power to compute the identification process.

The definition describing the Carousel and Slide Show widget in the ontology will be presented and examined to illustrate how this framework will assist widget classification and reduce ambiguity among the widgets' definition. Using the Manchester Syntax, a Carousel widget is described as presented in listing 3.4 and a Slide Show widget is described as presented in listing 3.17. It can be seen that many classes in the Components layer between these two widgets are similar. The main two differences are between how the widget deals with the request from the 'Next' and 'Previous' buttons when the widget is at either ends of the list; one widget allows the processes triggered by the buttons to loop around the list of contents, while the other prevents the content from propelling forward/backward when it comes to either ends of the list. Using these distinct identities, it will be able to identify the Carousel and Slide Show widgets without ambiguity.

Using this concept, the granularity issues between different types of widgets can be observed when attempting to define them. Well-defined axiomatic objects can be deployed and incorporated as new objects to cater for new and evolved widget designs. These new objects can be included without affecting the rest of the widgets and objects that are already well established. A relationship between these widgets can be observed from either the user's or the developer's perspectives, and by the purpose of the application of the widget.

3.3.2 Reusable Widget's Objects

Using our concepts, the examples of widgets presented exhibits the reuse of some of the objects that exist in other widgets, thus suggesting that there are relationships between some of widgets. In some cases, a type of widget can be a component of another widget. This also suggests that, solely by comprising of different types of widget, a widget can be formed.

Commonly, similar usage of widgets can be named or understood differently depending on their usage, the point of view of the user/developer, and historical reasons as discussed in §3.1 and §3.3.1. With this platform, using our widget definition, related widgets can be associated based on their objects. For example, to associate widgets with the List of Content object/class, then the Ticker, Auto Suggest List, Carousel, Slide Show, and Tabs widgets all exhibit this object/class.

The Widgets layer in the paradigm shown in figure 3.3 is the most abstract layer in the Widget Identification Ontology (WIO) paradigm – normally the classes in this layer will be the name/identifier for the type of a widget. Classes in the Components layer are the classes that represent the components of the type of widget. Every class in the Component layer will consist of at least one tell-sign class in the Tell-Signs layer that forms the individual component's class. Finally, the Code Constructs layer contains classes that refer to the instances of the patterns in the actual code, which in return form part of the metaphor of a tell-sign class in the layer above.

The widget ontology is designed to allow reusability of classes, expandability of widgets' concepts and has the capability to cater for the evolution of widget design. In order to incorporate this flexibility in the ontology design, each class within the ontology is independent of the others. Classes/objects that are used to define more than one class in the layer above are known as common classes/objects, and classes/objects that are specific for only one class in the above layer are known as private classes/objects. As see in figure 3.3, the classes in each layer are uniquely defined in the layer. In every layer, classes can be modified, added or removed without affecting the other classes in that layer or the layers below it. Thus, the model can be visualised as four separate layers in the ontology. Due to the conceptual relationships between classes in different layers that define a widget, removing or modifying classes in the lower layer will affect those in the upper layers. Modifications done to classes in the lower layers have to consider this aspect.

3.3.3 Auto Suggest List Widget

Commonly Auto Suggest List (ASL) widgets are employed to assist users when entering a query into a text field. Figure 1.3 shows an ASL widget used by `Google.com` to assist users when entering their search queries. From the suggested list of selection, the user can either select one of the options to complete the query, or they can continue to type the remainder of the search query if none of the options match their intended search query.

The ASL widget consists of five components in our widget ontology, and based on our ASL widget definition all five components must be present before the widget can be assumed to exist. Listing 3.1 described the different components that define the conceptual definition of the ASL widget ontology.

Listing 3.1: ASL widget definition

```
1  ASL ≡ hasComponents some DisplayWindow_Component
2          and hasComponents some Down_KeyStrokes_Component
3          and hasComponents some MonitorTextField_Component
4          and hasComponents some
5                  OffAutoComplete_ElementAttribute_ManipulatingDOM_Component
6          and hasComponents some Up_KeyStrokes_Component
```

Components like *DisplayWindow* have been described in §3.2.2, while components such as `Down_KeyStrokes_Component`, `MonitorTextField_Component`, `OffAutoComplete_ElementAttribute_ManipulatingDOM_Component` and `Up_KeyStrokes_Component` are defined in listing 3.2. Using only the UI components to define a widget is a loose method of defining a widget, and it can cause ambiguity in the definition between widgets. Thus, each component is further defined using the tell-sign classes. To search for these components in the Web page source code, traits of tell-signs can relate the tell-sign's classes in the Tell-Signs layer to a component class in the Component layer. Listing 3.2 illustrates the definition of the ASL widget's components using the tell-sign classes to define the components.

Listing 3.2: ASL Widget's components classes definition

```
1  DisplayWindow_Component ≡ (hasTellSign some MakeDOMElementsVisible_Tell-Sign
2                             or hasTellSign some UpdateDOMElements_Tell-Sign)
3
4  Down_KeyStrokes_Component ≡ hasTellSign some Down_KeyStroke_Tell-Sign
5
6  MonitorTextField_Component ≡ (hasTellSign some Change_EventHandler_Tell-Sign
7                                 or hasTellSign some Change_EventListener_Tell-Sign
8                                 or hasTellSign some OnChangeAttr_EventHandler_Tell-Sign)
9                               and hasTellSign some TextField_Tell-Sign
10
11 OffAutoComplete_ElementAttribute_ManipulatingDOM_Component ≡
12                                         hasTellSign some OffAutoComplete_Tell-Sign
13
14 Up_KeyStrokes_Component ≡ hasTellSign some Up_KeyStroke_Tell-Sign
```

It can be seen from listing 3.2 that often one or more Tell-sign classes are required to define a component in the widget. However, Tell-sign classes are formed by instances of code constructs that resemble a component in the widget. Listing 3.3 relates the Tell-sign classes to the patterns of code constructs found in the Web page source code. The process of defining the Tell-sign classes process is found in the fourth or bottom layer in the WIO paradigm as seen in figure 3.3.

Listing 3.3: ASL Widget's tell-sign classes definition

```
1  MakeDOMElementsVisible_Tell-Sign ≡
2                            hasCodeConstruct some MakeVisible_ManipulateDOMElements
3
4  UpdateDOMElements_Tell-Sign ≡ (hasCodeConstruct some JSAppendDOM_ManipulateDOMElements
5                 or hasCodeConstruct some UpdateInnerHTMLProperty_ManipulateDOMElements)
6
7  Down_KeyStroke_Tell-Sign ≡ hasCodeConstruct some Check40_KeyCode
8
9  Change_EventHandler_Tell-Sign ≡
10                (hasCodeConstruct some ChangeKeyword_Events_CodeConstruct
11                    or hasCodeConstruct some onChangeProperty_Events_CodeConstruct)
12
13 Change_EventListener_Tell-Sign ≡
14                    hasCodeConstruct some AddListener_Events_CodeConstruct
15                        and hasCodeConstruct some ChangeKeyword_Events_CodeConstruct
16
17 OnChangeAttr_EventHandler_Tell-Sign ≡
18                        hasCodeConstruct some onChangeAttr_Events_CodeConstruct
19
20 TextField_Tell-Sign ≡ (hasCodeConstruct some TextArea_DOMElements_CodeConstruct
21                        or hasCodeConstruct some TextField_DOMElements_CodeConstruct)
22
23 OffAutoComplete_Tell-Sign ≡
24                hasCodeConstruct some OffAutoCompleteAttr_ManipulateDOMElements
25
26 Up_KeyStroke_Tell-Sign ≡ hasCodeConstruct some Check38_KeyCode
```

3.3.4 Carousel Widget

A popular approach to organising multiple sets of content – so that users can interact with each set of content without overloading them with information that is irrelevant – is the Carousel widget. As seen in figure 3.1, this widget allows the user to interact with a list of content at their own pace, presenting one set at a time. The name of this widget came from the concept of a carousel where the user can browse through the list of content one at a time and it will loop back to the beginning when the end of the list has been reached, or looped to the end of the list when the beginning of the list has arrived. A visual perception of how a Carousel widget operates can be seen in figure 3.12; it should be noticed that the user can browse the widget in an endless loop as well as in both directions.

Besides the looping feature, figure 3.12 also shows that the widget uses a component called the Display Window to deliver the intended content to the user (The definition of a Display Window can be found in §3.2.2) and hide the irrelevant content. When

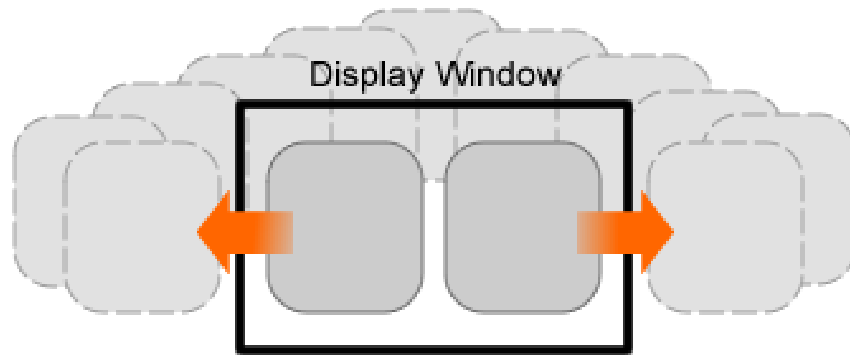


Figure 3.12: Visualisation of a Carousel widget process.

defining the Carousel widget ontology in listing 3.4 four components are used: a looping *Next* button (Loop_NextButton), a looping *Previous* button (Loop_PreviousButton), a *Display Window*, and a *Display Pointer* (The definition of a Display Pointer can be found in §3.2.2).

Listing 3.4: Carousel widget definition

```
1 Carousel ≡ hasComponents some Loop_NextButton_Component
2             and hasComponents some Loop_PreviousButton_Component
3             and hasComponents some DisplayWindow_Component
4             and hasComponents some DisplayPointer_Component
```

All four components must exist before the widget can be assumed to be a Carousel. To search for the tell-signs of these components, these components are defined by relating them to a number of tell-sign classes in the Tell-Signs layer. As discussed in §3.3.2, the taxonomy allows common objects/classes to be reused, for example the *Display Window* component. Since tell-signs of the *Display Window* component have been defined in listing 3.2, and the code constructs of these tell-signs has been defined in listing 3.3, listing 3.5 illustrates the definitions of the remaining tell-signs for the other components.

Listing 3.5: Carousel Widget's component classes definition

```
1 Loop_NextButton_Component ≡ hasTellSign some CheckEndOfList_Tell-Sign
2                                and hasTellSign some GoToStartOfList_Tell-Sign
3
4 Loop_PreviousButton_Component ≡ hasTellSign some CheckStartOfList_Tell-Sign
5                                    and hasTellSign some GoToEndofList_Tell-Sign
6
7 DisplayPointer_Component ≡ hasTellSign some referentialVariable_Tell-Sign
```

To bridge the definition of the Tell-sign classes with the code constructs, listing 3.6 defines the Tell-sign classes for the Carousel widget using the Code Constructs classes. Although the term *referentialVariable* was discussed in §3.2.3, listing 3.6 formally defines the code constructs of `referentialVariable_Tell-Sign` in listing 3.5.

Listing 3.6: Carousel Widget's tell-sign classes definition

```
1  CheckEndOfList_Tell-Sign ≡
2                          (hasCodeConstruct some IsArrayLength_Expression_CodeConstruct
3                              or assigned some End_Keyword_CodeConstruct
4                              or assigned some Last_Keyword_CodeConstruct)
5
6  GoToStartOfList_Tell-Sign ≡ (hasCodeConstruct some Zero_Assignment_CodeConstruct
7                                  or assigned some Begin_Keyword_CodeConstruct
8                                  or assigned some First_Keyword_CodeConstruct
9                                  or assigned some Start_Keyword_CodeConstruct)
10
11 CheckStartOfList_Tell-Sign ≡ (hasCodeConstruct some IsZero_Expression_CodeConstruct
12                                 or assigned some Begin_Keyword_CodeConstruct
13                                 or assigned some First_Keyword_CodeConstruct
14                                 or assigned some Start_Keyword_CodeConstruct)
15
16 GoToEndofList_Tell-Sign ≡ (hasCodeConstruct some ArrayLength_Assignment_CodeConstruct
17                                 or assigned some End_Keyword_CodeConstruct
18                                 or assigned some Last_Keyword_CodeConstruct)
19
20 referentialVariable_Tell-Sign ≡ hasCodeConstruct some Variable_CodeConstruct
21                                 and pointsTo some ArrayObject_Variable_CodeConstruct
```

Notice that the relational properties `assigned` and `pointsTo` are used to associate how different Tell-sign classes are related to different Code Construct classes. The `assigned` relationship between two classes defines how the class to the left will be accredited with the properties of the class on the right. An example would be the `CheckEndOfList_Tell-Sign` class: the conditional predicate will either search for patterns with the equality (==) or identity (===) operator followed by the length of an array, or the equality or identity operator followed by the 'End' or 'Last' keywords.

To illustrate the `pointsTo` relational property, we will examine the `referentialVariable_Tell-Sign` class. This relational property relates a variable that stores the value of a location in a `ArrayObject_Variable_CodeConstruct`, and the variable is used to refer to the location in the `ArrayObject_Variable_CodeConstruct` to load the value stored in that location.

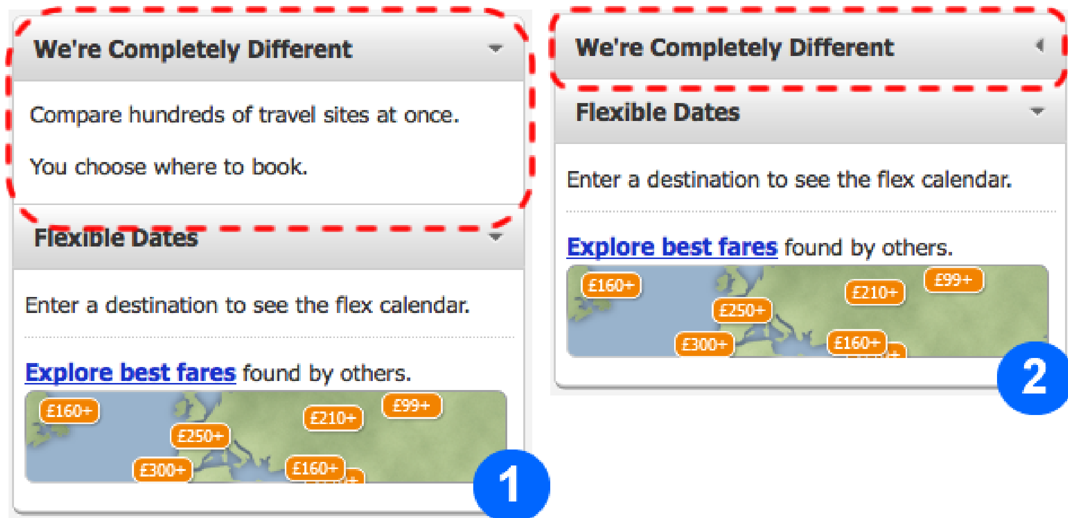


Figure 3.13: The use of Collapsible Panel widget in `kayak.co.uk`.

3.3.5 Collapsible Panel Widget

The Collapsible Panel widget is commonly used by Web developers to provide the facilities for the user to manage content heavy Web pages, so that the user can customise the content they are interested in viewing. Using this widget, users can unclutter the page, making it easier for them to read or focus on an area in the page by hiding the irrelevant content. Figure 3.13 shows how the Collapsible Panel widget is used by `kayak.co.uk` to allow its user to customise the content they are viewing. In ①, you can see the top panel (circled in red dotted lines) is expanded, and in ② this panel is collapsed, thus hiding the content from the user. The user can collapse or show the content by clicking on the arrow on the right side of the header.

Next, we will look at how the Collapsible Panel widget is defined. Three components are used to define a Collapsible Panel widget as shown in listing 3.7. Using the ontology, all three components have to be present before it can be assumed that the Collapsible Panel widget exists in the Web page.

Listing 3.7: Collapsible Panel widget definition

```
1 CollapsiblePanel ≡ hasComponents some DisplayWindow_Component
2                     and hasComponents some HideDisplayWindow_Component
3                     and hasComponents some UserTriggerObject_Component
```

The definition of some component classes is defined previously and reused to define the Collapsible Panel widget. Since `DisplayWindow_Component` has been defined already, listing 3.8 defines only the components that are newly introduced using the Tell-sign classes.

Listing 3.8: Collapsible Panel Widget's components classes definition

```
1  HideDisplayWindow_Component ≡ hasTellSign some HideDOMElements_Tell-Sign
2
3  UserTriggerObject_Component ≡ (hasTellSign some Click_EventHandler_Tell-Sign
4                                  or hasTellSign some Click_EventListener_Tell-Sign
5                                  or hasTellSign some Hover_EventHandler_Tell-Sign
6                                  or hasTellSign some Hover_EventListener_Tell-Sign
7                                  or hasTellSign some OnClickAttr_EventHandler_Tell-Sign)
```

As seen in listing 3.8, the `UserTriggerObject_Component` can be assumed to exist if either of the five user event related tell-signs is found. Current only *Click* and *Hover* events are covered. However, this can be expanded whenever required. These tell-signs are further defined with the actual code constructs presented in listing 3.9.

Listing 3.9: Collapsible Panel Widget's tell-sign classes definition

```
1  HideDOMElements_Tell-Sign ≡
2                        hasCodeConstruct some Hide_ManipulateDOMElements_CodeConstruct
3
4  Click_EventHandler_Tell-Sign ≡
5                        (hasCodeConstruct some ClickKeyword_Events_CodeConstruct
6                          or hasCodeConstruct some onClickProperty_Events_CodeConstruct)
7
8  Click_EventListener_Tell-Sign ≡ hasCodeConstruct some AddListener_Events_CodeConstruct
9                          and hasCodeConstruct some ClickKeyword_Events_CodeConstruct
10
11 Hover_EventHandler_Tell-Sign ≡
12                   (hasCodeConstruct some MouseoverKeyword_Events_CodeConstruct
13                     or hasCodeConstruct some onMouseoverProperty_Events_CodeConstruct)
14
15 Hover_EventListener_Tell-Sign ≡
16                     hasCodeConstruct some AddListener_Events_CodeConstruct
17                     and hasCodeConstruct some MouseoverKeyword_Events_CodeConstruct
18
19 OnClickAttr_EventHandler_Tell-Sign ≡
20                           hasCodeConstruct some onClickAttr_Events_CodeConstruct
```

3.3.6 Customisable Content Widget

Similar to the Collapsible Panel Widget (see §3.3.5), the Customisable Content widget extends its features not only to hide or show the user's desired information, but also

allowing them to remove content from the page, thus making room for the adding of other content.

Listing 3.10 shows the components that are used to define the Customisable Content Widget. Note that both components are common classes and discussed previously. This example demonstrates the reusability of classes in our ontology. Since this widget lacks the *DisplayWindow* component, it is only capable of removing content from the page. This example demonstrates that, when identifying a widget, not only do we use existing components to identify a widget, but we also analyse the findings of missing components to determine the type of widget.

Listing 3.10: Customisable Content widget definition

```
1 CustomisableContent ≡ hasComponents some HideDisplayWindow_Component
2                        and hasComponents some UserTriggerObject_Component
```

3.3.7 Popup Content Widget

The Popup content widget is often used to draw the user's attention to a set of content, or is used to provide further illustrations of certain parts of the Web page. As seen in figure 3.14, `AOL.co.uk` uses this widget to deliver the video and related videos based on the option the user selects. Notice that the widget arranges the order of the content to be in front of the rest of the content to attract the attention of the user. In this case, besides arranging the targeted content to be in front, the remainder of the page is masked with a translucent shade of grey to draw the visual attention of the user to the content.

Three components are used to define the Popup Content widget – `DisplayWindow_Component`, `SetOrderFront_ElementStyle-ManipulatingDom_Component` and `UserTriggerObject_Component` – as shown in listing 3.11. Notice that two of the three component classes are reused, only the `SetOrderFront_ElementStyle-ManipulatingDom` component is unique to this widget (Private class).

Listing 3.11: Popup Content widget definition

```
1 Popup Content ≡ hasComponents some DisplayWindow_Component
2     and hasComponents some SetOrderFront_ElementStyle-ManipulatingDom_Component
3     and hasComponents some UserTriggerObject_Component
```

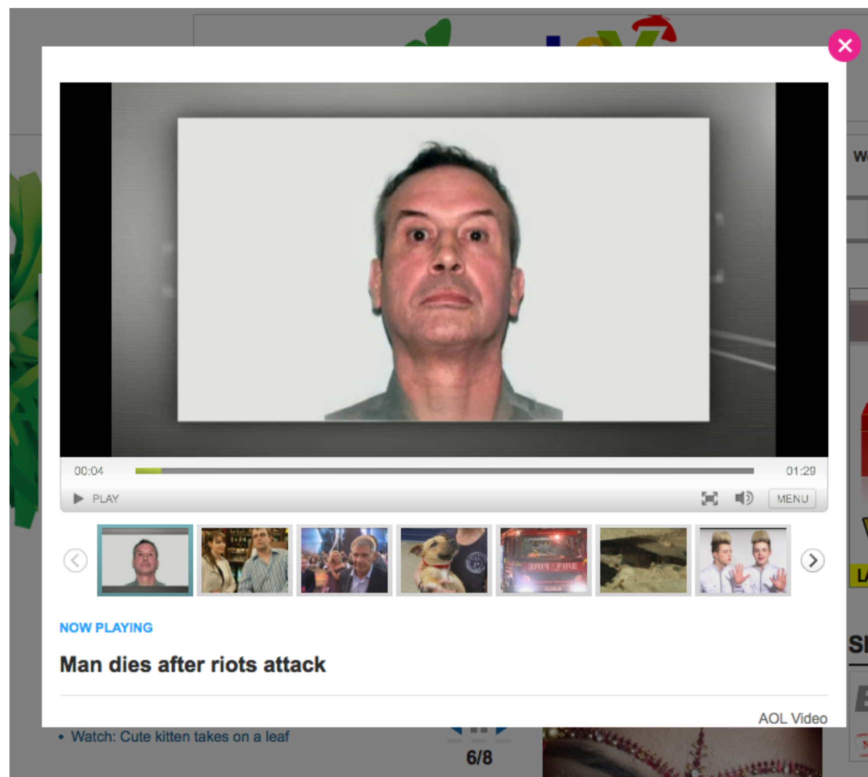


Figure 3.14: The use of Popup Content widget in AOL.co.uk website.

Listing 3.12: Popup Content Widget's components classes definition

```
1 SetOrderFront_ElementStyle-ManipulatingDom_Component ≡
2                           hasTellSign some SetDOMElementOrderHigh_Tell-Sign
```

To arrange the content to be in front of the other content, developers often change the styling property `z-index` of the element in the DOM. Thus, in listing 3.12, tell-sign *SetDOMElementOrderHigh_Tell-Sign* is used to define the components, and this tell-sign is defined as the code construct instance presented in listing 3.13. The Code Construct class `SetDOMElementOrderHigh_CodeConstruct` searches for `.style.zIndex = {>9}` pattern; where `zIndex` must be assigned to a value greater than 9.

Listing 3.13: Popup Content's Widget's tell-sign classes definition

```
1 SetDOMElementOrderHigh_Tell-Sign ≡
2          hasCodeConstruct some SetHigherOrder_ManipulateDOMElements_CodeConstruct
```

3.3.8 Popup Window Widget

Popup Window widgets are commonly used to launch content in a separate window. Normally the purpose of employing this widget is to allow the user to retrieve another Web page without leaving the existing page, and in some cases the Popup Window widget provides the user with the flexibility to switch between the popup window and the parent window. The definition of the Popup Window is presented in listing 3.14, where two components are used to define the Popup Window widget.

Listing 3.14: Popup Window widget definition

```
1 PopupWindow ≡ hasComponents some PopupWindow_Component
2               and hasComponents some UserTriggerObject_Component
```

One out of the two components discussed is a newly defined class. As seen in listing 3.15, it describes the relationship between the Components and Tell-signs classes for the `PopupWindow_Component` class.

Listing 3.15: Popup Window Widget's components classes definition

```
1 PopupWindow_Component ≡ hasTellSign some NewWindow_Tell−Sign
```

Finally, in listing 3.16, it relates the tell-sign instance to the actual code construct in the Web page source code. The possible code constructs to load a new window will look like `window.open(...)` in JavaScript, and a new window can also be launched from a link by predefining the link element with the attribute `target="_blank"`.

Listing 3.16: Popup Window Widget's tell-sign classes definition

```
1 NewWindow_Tell−Sign ≡ hasCodeConstruct some OpenNewWindow_CodeConstruct
```

3.3.9 Slide Show Widget

Similar to the Carousel widget, the Slide Show widget organises information such that it delivers multiple content in a list one set at a time. Except this time, as seen in figure 3.15, the user will not be able to loop around the list of content. Once he/she reaches either end, the user can only return to the previous content in the reverse order.

The definition of the Slide Show widget is presented listing 3.17, where it can be seen that the Slide Show widget has four components, very similar to the Carousel

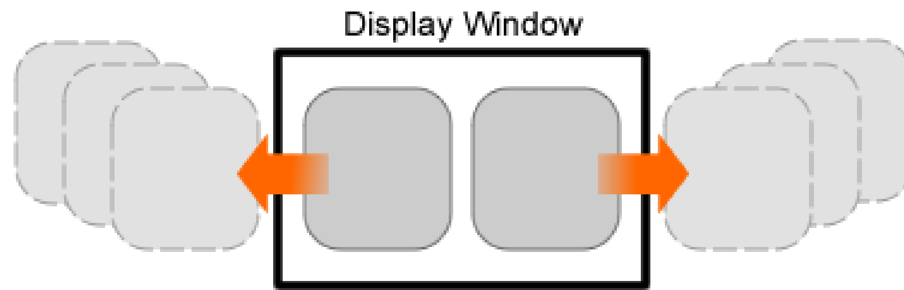


Figure 3.15: Visualisation of a Slide Show widget process.

widget. Instead of *Loop_NextButton* and *Loop_PreviousButton*, these components are replaced with *Finite_NextButton* and *Finite_PreviousButton* respectively.

Listing 3.17: Slide Show Widget Definition

```
1 SlideShow ≡ hasComponents some Finite_NextButton_Component
2             and hasComponents some Finite_PreviousButton_Component
3             and hasComponents some DisplayWindow_Component
4             and hasComponents some DisplayPointer_Component
```

The two new Components classes are defined in listing 3.18. Notice that the difference between the definition of `Loop_NextButton_Component` and `Finite_NextButton_Component`, and the definition of `Loop_PreviousButton_Component` and `Finite_PreviousButton_Component`. Comparing listing 3.18 and 3.5, the two classes defined in listing 3.18 use the same tell-signs, except they are defined with a different combination.

Listing 3.18: Slide Show Widget's components classes definition

```
1 Finite_NextButton_Component ≡ hasTellSign some CheckEndOfList_Tell-Sign
2                               and hasTellSign some GoToEndofList_Tell-Sign
3
4 Finite_PreviousButton_Component ≡ hasTellSign some CheckStartOfList_Tell-Sign
5                                   and hasTellSign some GoToStartOfList_Tell-Sign
```

3.3.10 Tabs Widget

The Tabs widget is another method of organising a stack of content so that it can be displayed to the user within a constrained area in the Web page. As seen in figure 3.16, content is stacked such that only the content at the top of the stack is presented to the user, and the user can choose which content he/she is interested in viewing. Unlike the

Carousel and Slide Show widgets, this method allows the user to retrieve the content of interest without browsing through the stack of content. At any point in time the user can swap to and fro for the desired content, as seen in ① and ② in figure 3.16.

Figure 3.16: The use of Tabs widget in `NYTimes.com`.

Listing 3.19: Tabs widget definition

```
1 Tabs ≡ hasComponents some DisplayWindow_Component
2         and hasComponents some HideContent_Component
3         and hasComponents some UserTriggerObject_Component
```

All three components used to define the Tabs widget, as shown in listing 3.19, must be present before this widget can be assumed to exist. Since on the *HideContent* component is a newly defined component, listing 3.20 describes the tell-sign for this component.

Listing 3.20: Tabs Widget's components classes definition

```
1 HideContent_Component ≡ hasTellSign some HideDOMElements_Tell−Sign
```

Instances of code constructs are searched so that the tell-signs that describe the components listed in listing 3.19 can be determined. Listing 3.21 describes the code constructs that define the tell-sign *HideDOMElements_Tell-Sign*. The code construct instances include manipulating the DOM elements so that they will not be visible to their user. The code constructs refer to patterns like `element.style.display = "none"` to manipulate the styling property of the element.

Listing 3.21: Tabs Widget's tell-sign classes definition

```
1 HideDOMElements_Tell-Sign ≡
2                         hasCodeConstruct some Hide_ManipulateDOMElements_CodeConstruct
```

3.3.11 Ticker Widget

The Ticker widget is very similar to the Carousel widget, except this time the widget automatically displays sections of the list of content to the user at fixed intervals. Another distinct visible characteristic of the Ticker widget is that it normally does not have controls for the user to manually browse through the list of content.

Three components are used to define the Ticker widget as discussed in listing 3.22. Since this widget is similar to the Carousel widget, two of the three components defined (`DisplayPointer_Component` and `DisplayWindow_Component`) are already discussed.

Listing 3.22: Ticker widget definition

```
1 Ticker ≡ hasComponents some DisplayPointer_Component
2          and hasComponents some DisplayWindow_Component
3          and hasComponents some Increment_TimeTriggered_Component
```

The Ticker widget reuses two common component classes previously defined, and introduces a third class. Listing 3.23 relates the `Increment_TimeTriggered_Component` class in the Components layer with the tell-signs classes in the Tell-sign layer to define the component in the Ticker widget.

Listing 3.23: Ticker Widget's components classes definition

```
1 Increment_TimeTriggered_Component ≡ (hasTellSign some Addition_Tell-Sign
2                                       or hasTellSign some Increment_Tell-Sign)
3                                    and hasTellSign some CheckEndOfList_Tell-Sign
4                                    and hasTellSign some GoToStartOfList_Tell-Sign
5                                    and hasTellSign some WindowTiming_Tell-Sign
```

Like the Components layer, some of the tell-signs classes are reused from the previously defined widgets. Listing 3.24 discusses the newly introduced tell-signs and relates them to the actual code construct instances in the Code Construct layer.

Listing 3.24: Ticker Widget's tell-sign definition

```
1 Addition_Tell-Sign ≡ hasCodeConstruct some PlusSign_CodeConstruct
2
3 Increment_Tell-Sign ≡ hasDoubleCodeConstruct some PlusSign_CodeConstruct
```

```
4                           or (hasCodeConstruct some PlusSign_CodeConstruct
5                               and hasCodeConstruct some One_NumericValue_CodeConstruct)
6
7   WindowTiming_Tell−Sign ≡ hasCodeConstruct some SetTimeOut_Window_CodeConstruct
```

3.4 Popularity of Widgets

Over the past nine sections (§3.3.3–§3.3.11), the nine simplest form of widget have been discussed. These widgets are in the axiom concepts of widgets derived from the Widget design pattern and JavaScript libraries examined. Using the nine categorisations of widgets as a basis to determine the types of widgets to be included in our evaluation, they will provide insights into our approach, and demonstrate the modelling and development of the widgets prediction framework.

To understand how our widget classification will affect the popular Websites, the first fifty Websites listed in Alexa's top 500 sites on the web list were taken as samples to examine the population of the nine types of widget. Using Quota Sampling, fifty Websites were selected from the highest ranked website to the least in the list based on the following prerequisites:

- Only the main domain of the Website was selected, i.e., `google.com` and `google.co.uk`, only `google.com` will be selected.
- Websites are selected from the top of the list until fifty Websites have been selected. Websites that do not satisfy the prerequisite will be omitted.
- No adult type of Websites will be included due to ethical reasons, and Websites that are unreachable from their default domain name will be excluded.

When examining each site for the type of widget that existed in its top-level index page, widgets must meet two conditions to be considered. The first condition is the widget has to meet the definition of one of the nine types of widget defined, and the second condition is that it must be developed using JavaScript. Other methods of delivering the widget, such as Flash and VBScripts, have not been included in this test.

Table 3.1 lists the results gathered for the popularity of different types of Widget investigation done on the top fifty websites. All types of widgets selected for the test had appearance on five or more Websites, with the most popular widget appearing on twenty-six Websites.

#	Websites	Auto Suggest List (ASL)	Carousel	SlideShow	Tabs	Collapsible Panels	Ticker	Popup Window	Customisable Content	Popup Content
1	google.com	1	0	0	0	0	0	0	0	0
2	facebook.com	0	0	0	0	0	0	0	0	1
3	youtube.com	1	0	0	0	1	0	0	1	1
4	yahoo.com	1	1	0	1	1	0	0	1	1
5	live.com	0	0	0	0	0	0	1	0	1
6	baidu.com	1	0	0	0	0	0	0	0	1
7	wikipedia.org	0	0	0	0	0	0	0	0	0
8	blogger.com	0	1	0	0	0	0	0	0	0
9	msn.com	1	0	1	1	0	0	0	0	0
10	qq.com	1	0	0	1	1	0	0	0	0
11	twitter.com	0	0	1	0	0	0	0	0	1
12	taobao.com	1	1	1	1	0	0	0	0	1
13	sina.com.cn	1	0	0	1	0	0	0	0	1
14	amazon.com	1	1	0	0	0	0	0	0	0
15	wordpress.com	0	1	0	0	0	0	0	0	0
16	bing.com	1	1	0	0	0	0	0	0	1
17	microsoft.com	0	0	1	1	0	0	0	0	0
18	ebay.com	1	0	0	0	0	0	0	0	0
19	yandex.ru	1	0	0	0	0	0	0	0	0
20	myspace.com	0	0	0	1	0	0	0	0	0
21	163.com	1	0	0	1	0	1	0	0	1
22	linkedin.com	0	0	0	0	0	0	0	0	0
23	conduit.com	0	0	1	1	1	0	0	0	1
24	craigslist.org	0	0	0	0	0	0	0	0	0
25	mail.ru	1	1	0	1	0	0	0	0	0
26	flickr.com	0	0	0	0	0	0	0	0	0
27	fc2.com	0	0	0	1	0	0	0	0	0
28	go.com	0	0	0	0	0	0	0	0	0

29	vkontakte.ru	0	0	0	0	0	1	0	0	1
30	rapidshare.com	0	0	0	0	0	0	0	1	0
31	bbc.co.uk	1	1	0	1	1	1	0	1	1
32	imdb.com	0	1	0	1	0	0	0	0	0
33	apple.com	1	0	0	0	0	1	0	0	0
34	adobe.com	1	0	0	0	1	0	0	0	1
35	doubleclick.com	0	0	0	0	0	0	0	0	0
36	soso.com	1	0	0	0	0	0	0	0	1
37	sohu.com	1	0	0	1	0	0	0	0	1
38	mozilla.com	0	0	0	0	0	0	0	0	0
39	fifa.com	0	1	1	1	1	1	0	0	1
40	aol.com	1	1	1	1	1	0	1	1	1
41	espn.go.com	1	1	0	1	0	0	0	0	1
42	youku.com	1	1	0	1	0	0	0	0	1
43	ask.com	1	1	0	1	1	0	0	0	1
44	cnn.com	0	1	0	1	1	0	1	0	1
45	paypal.com	0	0	0	0	0	0	1	0	1
46	photobucket.com	1	0	0	1	0	0	1	0	1
47	mediafire.com	0	0	0	1	0	0	1	0	1
48	tudou.com	1	0	0	1	0	1	0	0	0
49	orkut.com	0	0	0	0	0	0	0	0	0
50	sogou.com	1	0	0	0	0	0	0	0	0
	Total	26	15	7	23	10	6	6	5	25

Table 3.1: Popularity of widgets for the top 50 websites; where 1 means one or more of that type of widget exist on the Website and 0 is none of that type of widget exist.

Although the top fifty popular Websites are selected for this investigation, a good mix of Websites can be seen from the results. It include sites such as `Wikipedia.org`, `Flickr.com`, `Doubleclick.com`, `Mozilla.com` and `Orkut.com` that do not have any widget written in JavaScript, and sites like `BBC.co.uk`, `AOL.com` and `Yahoo.com` that incorporate a lot of different types of widget in their Website. While only 2 adult Websites were excluded, which will impact $< 5\%$ of our evaluation results.

Analysing the results in table 3.1, the different types of widgets are ranked based on their probability of occurrence in the top fifty Websites in table 3.2. The probability

of occurrence of each type of widget is computed by dividing the number of websites that the type of widget appears in by fifty (the top fifty Websites). We will use the Auto Suggest List (ASL) widget as an example; its probability of occurrence will be $26/50 = 0.52$.

#	Websites	# of Appearance in Top 50 Websites	Probability of Occurrence
1	Auto Suggest List (ASL)	26	0.52
2	Popup Content	25	0.50
3	Tabs	23	0.46
4	Carousel	15	0.30
5	Collapsible Panels	10	0.20
6	Slide Show	7	0.14
7	Ticker	6	0.12
8	Popup Window	6	0.12
9	Customisable Content	5	0.10

Table 3.2: Ranking of the types of widget based on their probability of occurrence in the top 50 Websites.

Further analysing the results in table 3.2, the probability of the number of types of widget that will appear in the top fifty Websites is 2.46. This result shows that on average more than one type of widget will be utilised by the top fifty popular Websites, and it demonstrates the extent and feasibility of using this sampling method for our widget identification evaluation in the future.

Out of the nine widgets presented here, the top 5 most popular widgets will be selected from table 3.2. However, closer examination of the top 5 widgets' definitions suggests that the Slide Show and Ticker types of widgets are similar to the Carousel widget. Therefore, these two types of widget will be evaluated along with the top 5 widgets to test for how the prediction algorithm will perform on generalised cases. Based on the probability of occurrence values, the last 4 types of widgets in table 3.2 do not appear frequently on popular Websites. Thus, the results from the last 2 types of widgets would only have minor impact on the evaluation results.

3.5 Summary

The taxonomy proposed for this framework can be expanded and applied to other types of widget. As the ontology of the different types of widget expands, the capability and accuracy of the framework will improve concurrently. This study covers the essential elements that distinguish the different widgets so that a taxonomy could be developed. It demonstrated how the concepts of the ontology can be modelled to identify and relate the different types of widgets. Nine types of widget were used as examples to illustrate how the taxonomy can be implemented to model different types of widget. The effectiveness and efficiency of the framework relies on the reuse of common classes in the taxonomy (§3.3.2) to model widgets. When defining introducing new classes to the taxonomy, the constraints of defining a class and the repercussions for the decisions to determine the classes are discussed in §3.3.1.

When the taxonomy is used effectively, it will assist developers and users to communicate by bridging the gap between their understandings of the subject matter, and in return facilitate a structure for developing widgets that can be used by more people. The use of WIO provides a platform to identify uniquely the different types of widgets and easily correlate them. Further evaluation to test the accuracy and robustness of the concepts is suggested; this should include a system to automate the detection process using the concepts defined in the ontology.

The feasibility evaluation will provide insights into the approach of applying the widgets' definition concepts modelled in WIO to the Widget Prediction System (WPS), and expose the weaknesses of this approach. This type of study should initially be done on a small scale as presented in the next chapter Feasibility Investigation for Using Tell-Signs before expanding the evaluation to a larger scale as discussed in chapter 5 - Predicting Widgets On A Web Page. By having a two-stage evaluation, it will give us an initial idea of the feasibility of the approach at an early stage, before going deeper to expose the issues in depth, and examine the spectrum of coverage of the approach.

Chapter 4

Feasibility Investigation for Using Tell-Signs

Insights into the approach, the implementation of the concepts for the widgets described in the ontology, and the techniques applied are required to be examined before expanding them across all the selected types of widgets to be evaluated. In this chapter, a feasibility investigation was conducted to examine whether the conceptual models for the two types of widgets can be found in the source code, and whether these definitions are sufficient to distinguish them. We want to understand the strengths and weaknesses of the approach from this investigation, so that it can be improved. The results from the investigation will be presented and compared with a manual analysis of the source code: to provide insights into the methods of identifying the tell-signs from the source code. A discussion of the results with the suggested improvements is also presented.

4.1 Experiment Setup

Two popular types of widget were evaluated and their results analysed. The selection of the types of widget are based on its complexity to detect the kind of widget, from the pool of the most popular widgets listed in §3.4. The most popular type of widget along with a complex type of widget in the top 5 most popular widgets are chosen to investigate how generalisable our approach will be. The two experiments were setup to evaluate the two types of widget over the same set of data to test our approach. This section explains the essential components, methodologies, and assumptions made when preparing the experiments.

4.1.1 Scripting Language

As a first step to prove that instances of tell-signs can be found in the source code of a Web page, the investigation is developed in PHP scripting language because of the availability of its Application Program Interfaces (APIs), types of regular expression syntax, and because it is freely available. Regular expressions play a crucial role when searching for clues within the source code of a Web page, so that tell-signs can be discovered. Due to the strength of the tools available to match complex patterns within a string, Perl syntax was chosen for scripting the regular expressions.

4.1.2 Experimental Data Set

The data set used to evaluate our concepts consists of twenty Websites selected from Alexa's global top five hundred sites on the Web list[1]. When choosing our set of data, a few considerations were made. One of these was to use a range of popular Websites so that it would give a good representation of the broader Web in the wild. As discovered in our earlier study on Web Evolution, the top twenty popular Websites give a good representation of the broader Web [Chen and Harper, 2008]. Hence, twenty Websites were chosen from the list of Alexa's global top five hundred Websites using the following rules.

1. No repetition of a Website's domain is allowed. For instance, google.com and google.com.br, because google.com.br is a sub-domain of google.com, thus this will be considered as a repetition.

2. If a repetition exists, it will be ignored and the next domain down the list will be examined from the first point of this list again.

3. All Websites chosen must be accessible, and broken links will be ignored.

The above requisites will be repeated for every Website, starting from the top, in the list of Alexa's global top five hundred Websites until twenty Websites are found for the experimental data set. To ensure repeatability, the set of data collected on 19 February 2009 was used when conducting both experiments. Using this method would enable us to freeze any changes to the Websites during the course of the experiments. Websites can tailor their design specifically to a user-agent. Thus, Mozilla's Firefox[2]

[1] http://www.alexa.com/topsites/global — The Alexa's global top five hundred sites on the Web list is compiled from the lists of Websites ordered by Alexa's Traffic Rank.

[2] http://www.mozilla.com/firefox/

was chosen as the user-agent when capturing this set of data. Choosing Firefox will enable us to capture Websites with Web pages that comply with one of the industry's leading user-agents.

4.2 Auto Suggest List (ASL) Widget

The ASL widget is a component to convey related suggestions to the user when he/she is completing a text field in a Web page. Most popular Web browsers provide an auto complete feature by default, similar to the ASL widget that provides suggestions to the user when completing a text field. However, the auto complete feature suggests only related entries that are previously submitted on that computer. Unlike the auto complete feature, the ASL widget is not only a replacement for the auto complete feature, but it also extends its capability by providing suggested texts computed by the server, e.g. the names of airports, and the names of train stations. This widget provides instant help and clues for users when filling in text field information. By providing such a feature to the user when filling in a text field, it will not only give assistance to users when completing the Web form, but it will also ensure that the text field gets filled in more accurately.

An example of the ASL widget in action is shown in the non-shaded region in figure 4.1. This widget will update its suggested items as the user enters or removes a character from the text field. The user can navigate through the list of suggested items either by using a mouse or by using the up and down keys on the keyboard. While the user enters the content into the text field, the widget interacts with the server to extract the relevant suggested items. This process is repeated every time the user changes the content in the text field.

Content accessibility is particularly important for this widget. This is because a constant stream of content changes when the user interacts with the widget. The content is fed back at real-time, and it is critical for the user to be aware of these changes to utilise the widget effectively. These changes may even affect the accuracy of the form completed.

4.2.1 The Tell-Signs

Five tell-signs are introduced as part of our detection methods to identify the ASL widget from the source code. Investigations were conducted on a set of training Web

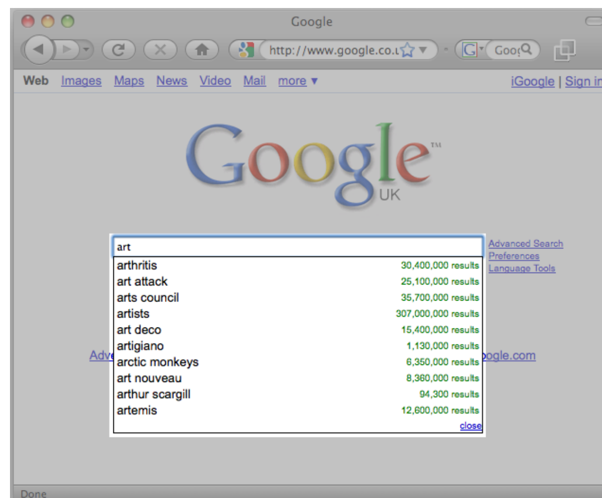


Figure 4.1: An example of ASL widget (non-shade region) in action on Google.co.uk

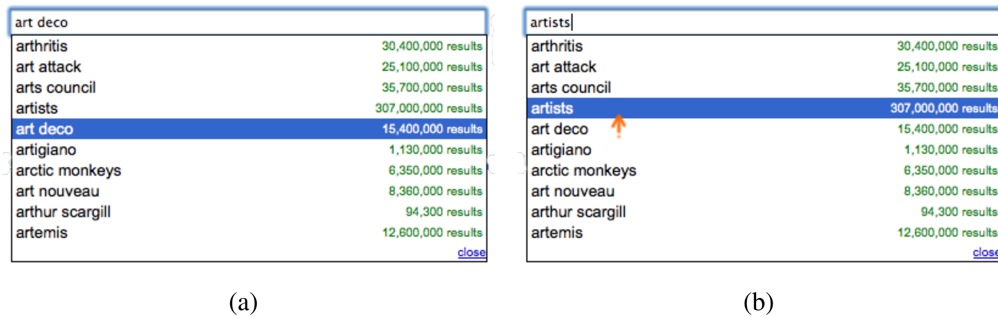
pages with ASL widgets, Yahoo! Developer Network's design pattern library[3], and Welie's pattern library[4] to identify the tell-signs for ASL widget to be included. Unlike desktop applications, some user interface objects may come in different forms on the Web. An example of this is the form button object. It can be in the form of an image, a string of texts, or a button. Thus, when searching for UI objects, different variants of it should be considered.

To code the concepts of the five tell-signs manually, each of these tell-signs follows a sequence of conditional clues when comprehending the Web page's source code. The algorithm uses the concept of a decision tree technique to arrive at a conclusion whether the ASL widget exists. The details of the five tell-signs will be covered individually in the next few sections.

'Auto Complete Off' Tell-Sign (ts1)

The Auto Complete Off tell-sign examines the Web page for clues that the Web browser's Auto Complete feature is disabled. Developers can apply this method by either setting the element's `autocomplete` attribute to 'off' in the HTML code, or manipulating the styling property of the element in the Behaviour Layer. To search within the JavaScript code for patterns that resemble the manipulation of the styling property metaphor,

[3] http://developer.yahoo.com/ypatterns/
[4] http://www.welie.com/patterns/



(a) (b)

Figure 4.2: To move the selection up as seen from subfigures (a) to (b), press the up arrow key (key code 38)

the following regular expression is used. It searches for clues that the code was programmed to set the Auto Complete feature's attribute to off.

```
/[-_\.a-z0-9]+\.setAttribute(\s)*\((\s)*(\"|\')autocomplete(\"|\'),
    (\s)*(\"|\')off(\"|\')(\s)*\)(;)?/i
```

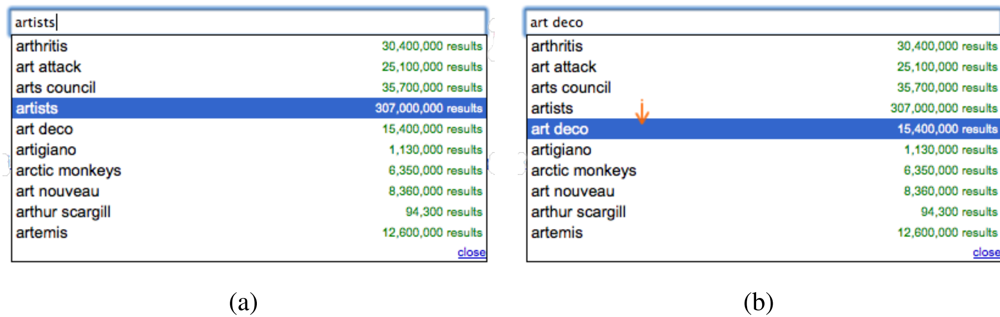
If the above regular expression could not find any instance in the JavaScript code for manipulation of the styling property, then an attempt will be made to search the HTML code. The following regular expression will be used to search for the setting of the Auto Complete feature within the HTML code. In this case, it looks for patterns that the Auto Complete feature is assigned to `off`.

```
/autocomplete(\s)*=(\s)*(\"|\')?off(\"|\')?/i
```

'Up Key' and 'Down Key' Tell-Sign (ts2)

The Up and Down Keys tell-sign checks if the code polls for either the up or down key is pressed. Commonly the users of the ASL widget can traverse through the list of suggested items using the up (key code 38) and down (key code 40) keys on the keyboard (see figure 4.2 and figure 4.3 for illustration respectively).

From our investigations into the design patterns of ASL widgets from different widget design pattern libraries, two methods are commonly found when developing the ASL widget. These methods include using existing APIs from JavaScript libraries or they are custom-made ASL widgets. Hence, a few methods can be used to poll for whether the up or down key is pressed. In the event that polling for both key codes cannot be found in the source code, the system will search for the "keyup" and "keydown" keywords within the JavaScript code. This is because some APIs may



(a) (b)

Figure 4.3: To move the selection down as seen from subfigures (a) to (b), press the down arrow key (key code 40)

provide this feature as a replacement for the key code polling. To ensure that the polling for the up and down keys is intended for the ASL widget, the event that triggers this function must be examined to ascertain the relationship between the hypertext elements, and the handler in the JavaScript code.

The following two regular expressions are applied to detect if the code polls for certain keys. The first regular expression searches whether the ‘keyCode’ function is used in the code to identify the type of key the user has pressed. If that regular expression is true, the second regular expression will be followed to check whether key codes 38 or 40 are monitored.

```
/* To check if keyCode function was called */
/[-_\.a-z0-9]+\.keyCode/i

/* To check for the correct key codes were polled */
/(38|40)/i
```

However, if Yahoo API[5] is used, the following regular expression will be used to detect if the up and down keys are polled instead of searching for key codes 38 or 40. This is because Yahoo API uses the Lazy Load method to avoid overloading the browser when initially loading the Web page. To extract the code that uses the Lazy Load method to inject code/content into the Web page, it will only be possible to retrieve the injected code when they are rendered due to the JIT techniques JavaScript engines apply. Thus, this set of code will not be available when our identification system analyses the code. Instead, the following regular expression is employed to overcome this issue by focusing on the clues in the code. This is done by searching for listeners that monitor the keyboard events.

```
/YAHOO.util.Event.addListener\([-_\.a-z0-9\,]+,(\s)*\"
      (keyup|keydown)\"/i
```

Next, the events that triggered the function or API must be detected so that we can relate the portions of JavaScript code that form the ASL widget with the elements in the Web page. To identify the methods that capture these events, either of the following regular expressions can be used.

```
/\.focus\(\)/i
```

```
/\.event/i
```

In the event of all the above checks failing, the second attempt searches for keywords such as ‘keyup’ and ‘keydown’ within the JavaScript code. Using the following regular expression, it allows the pattern to match strings (case insensitive) that are concatenated with the ‘keyup’ and ‘keydown’ keywords.

```
/\"(keyup|keydown)\"/i
```

This attempt is made because tailored APIs may provide an inbuilt key code detection feature, and keywords may be used as identifiers to refer to or trigger these features. However, in cases where APIs use the Lazy Load method, they may not be available during the widget prediction process.

‘Updates’ Tell-Sign (ts3)

The ‘Updates’ tell-sign detects if the code attempts to append a table, list or anchor element. When the list of suggested items is updated with the latest set of suggested results, the DOM will be updated too. For some pages, this may be formatted using anchor tags for each item in the list of suggested items. To alter the format of an existing Hypertext document, this is done by using the JavaScript’s `appendChild` function. Due to the available options for listing items, this tell-sign is split into two parts. The first part uses regular expressions to search for the three options available – table, list or anchor elements.

```
/(\"|\')table(\"|\')/i      /* table */
```

```
/(\"|\')td(\"|\')/i         /* table */
```

```
/(\"|\')li(\"|\')/i          /* list */

/(\"|\')a(\"|\')/i           /* anchor */
```

The second part ensures that the `appendChild` function in JavaScript is used with either of the detected elements found in part one. This part uses the following regular expression to search for the clues that the `appendChild` function is used to append the above elements.

```
/([-_\.a-z0-9])*\.appendChild(\s)*\(/i
```

'Clear List' Tell-Sign (ts4)

The 'Clear List' tell-sign searches for clues within the JavaScript code that will clear the previous list of suggested items. Commonly, a developer will clear previous results from the suggested list before new results are updated to the list.

Developers can either clear the content, or remove the nodes within the list. One can use an empty string to clear the list, but this method is 'crude' and does not tell us much about the intent of the developers. So this approach will not be searched. Instead, the following regular expression will be used to search if the node(s) of the previous list of suggestions is removed.

```
/[-_\.a-z0-9]+\.removeChild(\s)*\(/i
```

'Remote Scripting' Tell-Sign (ts5)

The 'Remote Scripting' tell-sign searches for clues that remote scripting is used by the widget to complete its task. Part of the ASL widget's service process will request the server for a list of suggestions related to the content entered in the text field. Often remote scripting methodologies are employed to extract the list of suggestions. However, remote scripting can be requested using a few methods, such as AJAX or iFrames.

Beginning with iFrames, two clues must be present before an assumption that an iFrame exists. 1) The iFrame tag must be used. This can be done either by hard coding or generated dynamically by scripting languages. Hence, a generic regular expression is used to search for these clues.

```
/iframe/i
```

2) CSS code must be used if iFrames were employed. Only by this means, when the iFrame method is employed, will the list of suggestions be able to float over the existing content. To do this in CSS, the `position` styling property must be set to 'absolute'. This is searched using either of the following regular expressions.

```
/position(\s)*=(\s)*(\"|\')absolute(\"|\')/i
```

```
/position(\s)*:(\s)*absolute/i
```

If the iFrame tag is not found, then the `HttpRequest` method is searched instead. This approach is normally employed by the remote scripting technique to communicate with the server. To determine whether the widget uses the HttpRequest method, either of the following regular expressions must be true.

```
/\.XMLHTTP/i
```

```
/XMLHttpRequest\(\)/i
```

```
/HttpRequest/i
```

4.2.2 Assumptions and Rules

The definition described for the ASL widget in this chapter may differ from the definition discussed in WIO. This is because the definitions described here are the initial definitions of the widget, while the definitions in the ontology are the refined version of the definitions described here. As a first attempt to detect the ASL widget, two assumptions were made when designing the tell-signs. In the first assumption, not all the tell-signs have to be satisfied before an ASL widget can be assumed to exist in a Web page. Two rules were created to govern the conditions when determining if an ASL widget existed.

Rule 1: The first three tell-signs ($ts1$, $ts2$, $ts3$) must be true.

Rule 2: Either tell-sign 4 or 5 (or both) must be true.

Described in equation 4.1, both of the above rules must be satisfied to assume an ASL widget exists in the Web page. The second rule is a conditional rule. This rule

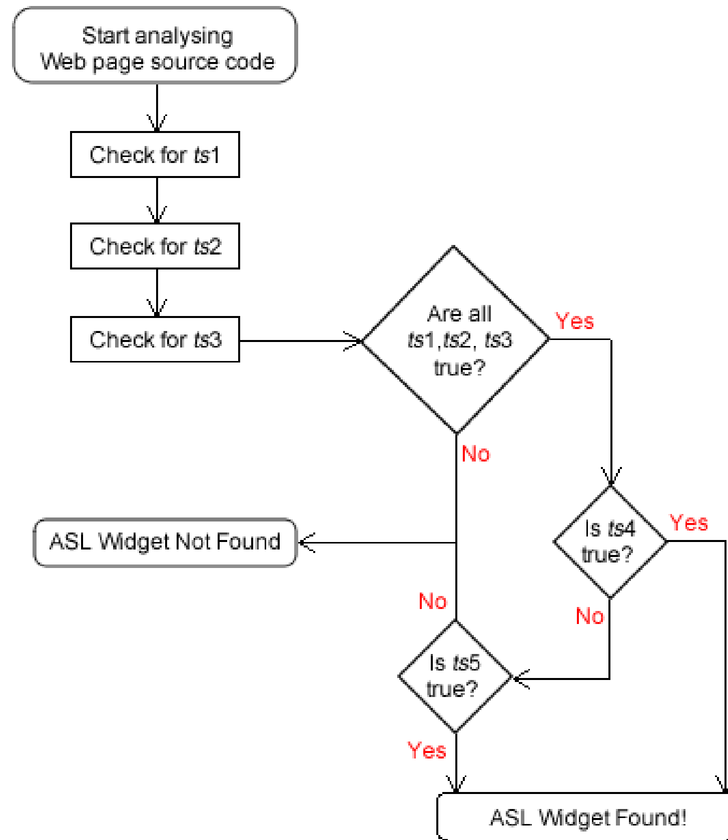


Figure 4.4: ASL widget detection's tell-sign process flow chart

is designed to cater for the variations of programming styles and practices from one Website to another.

$$ts1 \wedge ts2 \wedge ts3 \wedge \left(ts4 \vee ts5\right), \tag{4.1}$$

where $ts1$ is the 'Auto Complete Off' tell-sign, $ts2$ is the 'Up and Down Keys' tell-sign, $ts3$ is the 'Updates' tell-sign, $ts4$ is the 'Clear List' tell-sign, and $ts5$ is the 'Remote Scripting' tell-sign. Using this set of rules, the Web page's source code will be comprehended according to the flow chart illustrated in figure 4.4. The code comprehension process is structured to examine $ts1$ to $ts3$ first, before it will decide whether to proceed to $ts4$ then $ts5$. This method of coding is used to make the code leaner, and more memory efficient.

The second assumption is that we will only investigate ASL widgets written in JavaScript. This assumption is made based on a recent study conducted by Chen and Harper [2008]. It showed that VBScript (a competing client-side scripting language

with JavaScript) is poorly adopted over the last 10 years, and JavaScript was the client-side scripting language preferred by most Websites.

4.2.3 Limitations

This feasibility investigation is designed to analyse documents formatted in Hypertext, CSS, and JavaScript only. ASL widgets that are written in Flash, Shockwave, VBScripts, or any other formats other than JavaScript will not be covered.

A widget can be coded in a variety of styles and practices. Our methodologies cover a range of styles and their variants. We detect for instances of a pattern rather than the pattern itself. This is to allow a broader range of coding styles to be incorporated in our detection methods. However, it does not cover all variations. For example, if the set of code that is used to control the behaviour of an ASL widget is generated "on-the-fly", then the newly injected code will slip through our detection methods.

4.2.4 Results and Discussions

The results of the experiment for detecting the ASL widgets are tabulated in table 4.1. The results for each Website are reported in a Boolean format where'1' can be interpreted as the detection is true, '0' as the detection could not be found, and 'X' symbolises that the detection result was ignored for some reason during the investigation. In the second column, 'Manual detection', the checking of the tell-signs is done manually by a human being. If at least one ASL widget is present in the Website, it will be given a '1' in this column, and a '0' if no ASL is found. The third column, 'Auto detection' employs the rules discussed in §4.2.2 to determine if an ASL widget exists on the Web page from the tell-sign results. Following that, the results for the individual tell-signs are presented in the next few columns.

Some technical issues were noticed during the evaluation process. One of the Websites, `qq.com`, uses a licensed third party widget which made it impossible to retrieve the code. As a result, we have decided to exclude this Website from our analysis. ASL widgets were incorrectly detected on three out of nineteen Websites. Two of these Websites (`myspace.com` and `facebook.com`) reuse the same external JavaScript files on multiple Web pages in their Websites. These external JavaScript files have many functions/classes that are included even if they were not used by the page, which causes false detection to occur. After conducting a thorough investigation on both Web pages, it was found that the features of the ASL widget do exist within the external files, but

Websites \ Checks	Manual detection	Auto detection	Auto Complete Off Tell-Sign	Up and Down Keys Tell-Sign	Updates Tell-Sign	Clear List Tell-Sign	Remote Scripting Tell-Sign
yahoo.com	1	1	1	1	1	1	0
google.com	1	1	1	1	1	1	1
youtube.com	1	1	1	1	1	0	1
live.com	1	1	1	1	1	0	1
msn.com	0	0	0	0	0	0	0
myspace.com	0	1	1	1	1	1	1
wikipedia.org	0	0	0	0	0	0	0
facebook.com	0	1	1	1	1	1	1
blogger.com	0	0	1	0	1	1	1
orkut.com	0	0	0	0	0	0	0
rapidshare.com	0	0	0	0	0	0	0
baidu.com	1	1	1	1	1	1	1
microsoft.com	0	0	0	1	1	1	1
qq.com	1	X	1	0	1	1	1
ebay.com	1	1	1	1	1	1	1
hi5.com	0	0	0	1	1	1	1
aol.com	0	1	1	1	1	1	1
mail.ru	1	1	1	1	1	0	1
sina.com.cn	1	1	1	1	1	1	1
fc2.com	0	0	0	0	1	1	1
Total	**9**	**11**	**13**	**13**	**16**	**13**	**15**

Table 4.1: Feasibility investigation results for ASL widget detection. Where '1' is found, '0' is not found, and 'X' is ignored.

they were never called or used by the pages. This is an issue concerning the style of organisation on that Website that may vary from one Website to another. Another reason for this issue is the weakly typed client-side scripting languages found in Web pages. The third Website, `aol.com` had a lot of Web widgets implemented in it. Thus small bits of code from the different widgets can confuse our approach, leading to false detection that contributes to the false positive results for this Website.

The results presented in table 4.1 are promising and gave a positive outlook that our approach is feasible. Our method correctly detected all ASL widgets that are present, but it also highlights further 16% of false positive detections that suggest room for fine-tuning our methods. We are convinced that this set of results has demonstrated that our approach is feasible to work with as a platform, even though further investigations are required to refine our methodologies.

4.3 Carousel Widget

The Carousel widget is a presentation model to display one or a few items in a set of content to the user at a time. This widget provides users with the control to browse through the list of content, focusing on the information they want to concentrate on.

Three basic features are provided on the Carousel widget. Items in the list of content can be rotated round in both directions by controlling a set of controls on the Carousel widget. This is done by clicking on either the 'next' or 'previous' buttons, on some occasions the 'up' or 'down' buttons. Both sets of controls provide the user with the power to control the direction the content should rotate, depending on how the widget is developed. Thirdly, whenever the user gives no directional instruction, the Carousel widget will remain at the last requested item. In this thesis, the term 'Next' button will be used to refer to the 'next' or 'down' button, and the term 'Previous' button will refer to the 'previous' or 'up' button. An example of a Carousel widget is presented in the non-shaded region in figure 4.5. This example shows a small version of the Carousel widget with the 'next' and 'previous' buttons to the left of the widget's Display Window.

Figure 3.12 gives a visual illustration of the process and parts of a Carousel widget. The area where content is presented by the widget will be described as the Display Window. The content in the Display Window will be updated whenever the user requests a new set of item(s) within the list of contents. This is the area on the Web page where the Carousel widget will cause micro-content to update dynamically. Notice that content in the list is presented in a loop if the user keeps requesting for content in the same direction, like a Carousel at a funfair, except the user has the control over which direction the Carousel is rotating and the pace it is moving at.

The Carousel widget is chosen for the evaluation due to the nature of its behaviour: the content in the Display Window will update dynamically whenever the 'next' or 'previous' button is clicked by the user. The features of the Carousel widget are kept

Figure 4.5: An example of the Carousel widget (non-shaded region) in action on blogger.com. The arrows are used to move to the previous or next item.

basic in our definition, so that it can accommodate the different variants developers may design. Our current definition of the Carousel widget shares a number of common tell-signs with other Web widgets such as the Slide Show and Tabs. If not dealt with carefully during the development of this widget detection, this can cause ambiguity in our detection methodology, and result in a high false positive detection rate. Taking the Slide Show and Tabs widgets as an example, these widgets have very similar, sometimes identical, characteristics and features. Therefore, detection of the properties of the Carousel widgets can be misleading and it causes false positive detection. In the next two sections, we will examine these widgets in more depth.

4.3.1 Slide Show Widget

The Slide Show widget is another model for presenting a list to users. This widget can be visualised as a superset of the Carousel widget. Unlike the Carousel widget, it does not allow the users to loop around the list in either direction, and often it provides the user with additional automated features. A Slide Show widget is shown in the non-shaded region in figure 4.6. In this example, the controls are found at the bottom right hand corner of the widget's user interface.

Commonly, after loading the Web page, the Slide Show widget will play the content in the list automatically. The user can pause or resume playing it at any point. Some

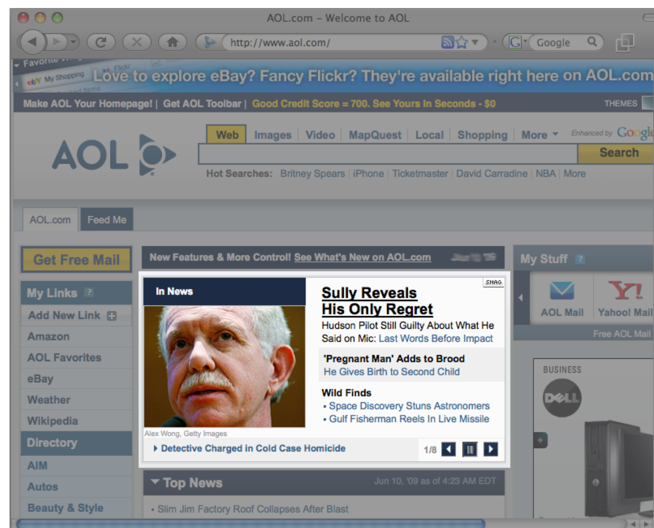


Figure 4.6: An example of the Slide Show widget (non-shaded region) in action on aol.com

Slide Show widgets even have a fast forward/skip button to allow the user to progress through the list at a faster pace. Similarly to the Carousel widget, the user can return to the previous content or progress to the next content in the list whenever he/she wants to. Besides the controls for controlling the display of content, the infrastructure of the Carousel widget is similar to the Slide Show widget as discussed in this example. These commonalities can lead to false detection for both of these widgets.

In order avoid confusion the Carousel widget is defined to have 'next' and 'previous' buttons with looping features, while the Slide Show widget has finite 'next' and 'previous' buttons. The remainder of the components are left as common objects between the widgets.

4.3.2 Tabs Widget

The Tabs widget is another widget that has similar properties to the Carousel widget. Unlike the Carousel widget, which allows its users to use the 'Next' or 'Previous' buttons to navigate through the list of content, the Tabs widget allows users to select directly the information they want to read. This means that users can skip the irrelevant content and navigate directly to the information they wish to read. Normally this widget is used to convey much more information than the Carousel widget. An example of a Tabs widget is shown in the non-shaded region of figure 4.7. In the given example, the users can change the displayed content by clicking on the relevant tabs at the top

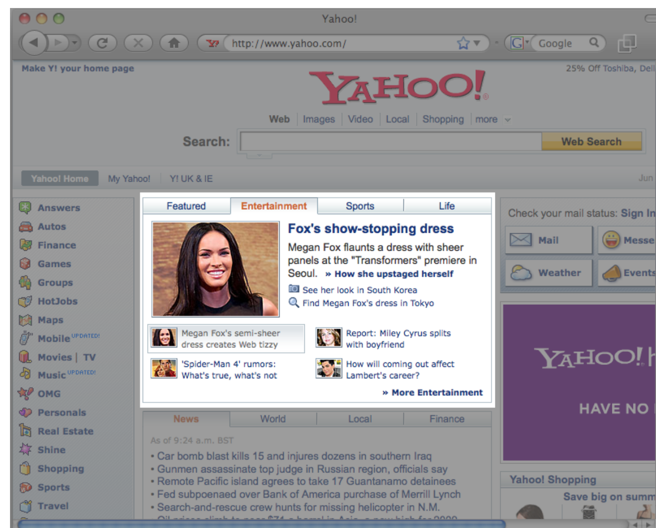


Figure 4.7: An example of the Tabs widget (non-shaded region) in action on yahoo.com

of the widget.

The main difference between the Carousel and the Tabs widget is the way it allows the user to navigate through the content. However, both widgets use a similar method to present the information to the user. This is done through the means of changing the content in the widget's Display Window.

4.3.3 The Tell-Signs

Tell-signs are used to identify clues of the Carousel widget's components in a Web page. They were defined through an investigation on Web pages containing Carousel widgets, Yahoo! Developer Network's design pattern library[5], and Welie's pattern library[6]. Additional refinements were also introduced to the tell-signs, to provide a better distinction for the Carousel widget.

Four tell-signs are used to harvest the results for this experiment. A description of the purpose, methods, and the flow of processes for each tell-sign is presented.

[5] http://developer.yahoo.com/ypatterns/
[6] http://www.welie.com/patterns/

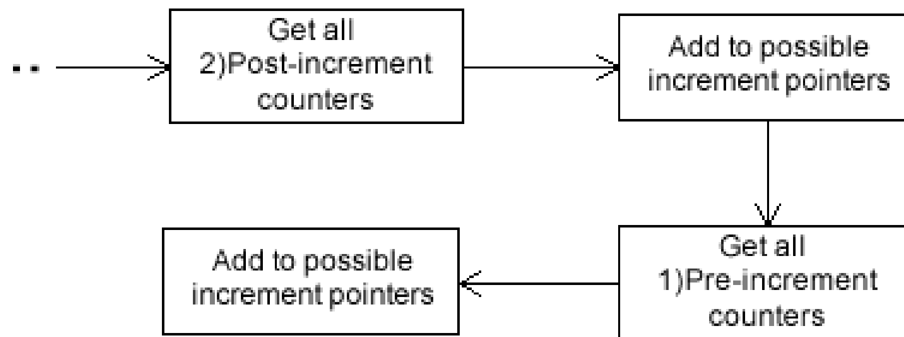


Figure 4.8: Carousel widget increment tell-sign (ts1) detection flow

'Increment' and 'Decrement' Tell-Signs (ts1 and ts2)

The Increment and Decrement tell-signs search for clues that the 'Next' and 'Previous' button functions exist within the JavaScript code. The 'Next' button can be thought of as a function that increments the value of a pointer that is pointing to a location within an array where the list of contents is stored. To find this tell-sign, an attempt is made to detect a pattern where a variable is incremented in the JavaScript code. Two methods can be employed by the developers to code an incremental pointer: 1) pre-increment and 2) post-increment. To search for these clues, the following two regular expressions were used respectively.

```
/(\s|;)\+\+([-_0-9a-z]+)/i          /* pre-increment */

/(\s|;)([-_0-9a-z]+)\+\+/i          /* post-increment */
```

The process used to determine whether variable values are incremented is shown in figure 4.8. The 'Get all 1) Pre-increment counters' block and 'Get all 2) Post-increment counters' block in this figure are where the pre-increment and post-increment regular expressions were applied respectively. Each occurrence of the identified post-increment counter and pre-increment counter variables is concatenated with the overall list of possible increment variables array.

The same process is used to search for a decrement pointer within the Web page's source code. However, this time the following two regular expressions are used instead to search for the pre-decrement and post-decrement pointers respectively. Similarly, each occurrence of the identified post-decrement and pre-decrement counter variables will be concatenated with the overall list of possible decrement variables array.

```
/(\s|;)(\-\-)([-_0-9a-z]+);/i        /* pre-decrement */

/(\s|;)([-_0-9a-z]+)\-\-/i           /* post-decrement */
```

As an initial investigation, the looping feature of the 'Next' and 'Previous' buttons will not have been covered. However, this detection has been planned to be rolled out in the second phase, after having a better understanding of the feasibility of the 'Increment' and 'Decrement' Tell-Signs. Furthermore, at this stage only the Carousel and ASL widgets are investigated, thus it will not affect the feasibility detection rates of these widgets.

'True Pointers' Tell-Sign (ts3)

To ensure that the variables identified in *ts*1 and *ts*2 are true variable pointers that point to a location within a list/array, the True Pointers tell-sign is used. This tell-sign ensures that the suspected variable refers to a location within the list/array in the code.

Two common applications employed by developers that use variables to refer to a location in a list/array are used to search for clues that the candidate variables are real. The first method searches for clues that conditional expressions were enforced on the suspected variables to restrict their value. Commonly, developers use conditional predicates to ensure that the pointer's value is restrained to the locations within the list/array. Thus, the following regular expression is employed to search for these clues. In this regular expression, the concatenated array `$possible[$i]` stores the identifiers of the candidate variables that were selected by the first two tell-signs. Using this regular expression, we can differentiate the variables that have conditional predicates enforced on them in their process.

```
/if\([\w\W]*\s*(==|<=|>=|>|<)?".$possible[$i].
      "\s*(==|<=|>=|>|<)?[\w\W]*\)/i
```

The second method checks if the suspected variable pointers are used by the developer to refer to an address location within the list/array. This is done using the following regular expression. The variable `$ptr[$i]` contains the filtered list of suspected variables that pass the first part of this tell-sign. This pattern searches if the variable identifiers from the list of suspected variables have been used to refer to a location in an array.

```
/([-_0-9a-z]+)\[".$ptr[$i]."\]/i
```

'Show and Hide' Tell-Signs (ts4)

The Show and Hide tell-sign searches within the Web page's source code for styling properties manipulation that will affect the display of the content in the widget's Display Window. This can be done using a few methods to achieve the desired content style. One of the commonly employed methods to control the styling of content displayed is to manipulate the styling properties for a particular element from the Behaviour Layer.

```
/".$fdlist[$ltc]."\[".$fptr[$ptc]."\]\.style\.display\s*
      =\s*(\'|\")block(\'|\")/i
```

```
/".$fdlist[$ltc]."\[".$fptr[$ptc]."\](\s)*(\))?\.style\.
      display\s*=\s*(\'|\")none(\'|\")/i
```

The first regular expression above searches for signs that the styling property of an element `display` is set to 'block'. The styling property `display` in CSS specifies how the appearance of the DOM element should be generated if at all. Anecdotal evidence suggests that Carousel widgets commonly employ this technique to reveal the elements content by manipulating this property of the element style. The second above regular expression searches for clues that the developer is attempting to hide the specified element. Either of these regular expressions must be satisfied for this tell-sign to be true.

4.3.4 Assumptions and Rules

A number of assumptions were made when designing the tell-signs for detecting the Carousel widget in this experiment. One of the main issues is determining the granularity of the design patterns. Due to the heterogeneous nature of the Web, defining the granularity of the patterns must be fine enough so that it will be flexible and reusable. Therefore, the agility of our approach depends a lot on our defined axiom definition for the different objects. Consider, for example, the button object. Unlike conventional programming, over the Web this can be constructed as a form button, a hyperlink with plain text, or a hyperlink with images. Furthermore, a hyperlink with images can be a link to another Web document, or a link that interacts with the client-side scripts, or a link that redirects the user to another part of the same Web document. This adds complexity to the concepts and relationships between the objects. In this case, it will

be assumed that the button object is an element within a Web page that calls a portion of the JavaScript code when activated by a user event.

Similarly to the ASL widget, the Carousel widget definitions described in this chapter may differ from the definitions described in the WIO. This is because the definitions described in this chapter are the initial definitions of the Carousel widget, while the definitions in WIO are the refined version of the definitions described here. Our method covers only Carousel widgets developed in JavaScript. This can be expanded, if required, for other scripting languages using the same concepts. Equation 4.2 introduces a rule to govern our approach for detecting the Carousel widget.

$$ts1 \wedge ts2 \wedge ts3 \wedge ts4, \tag{4.2}$$

where $ts1$ is the Increment tell-sign, $ts2$ is the Decrement tell-sign, $ts3$ is the True Pointer tell-sign, and $ts4$ is the Show and Hide tell-sign. The rule specifies that all the four tell-signs must be true before a Carousel widget can be assumed to exist in the Web page.

The identification process for the Carousel widget begins with searching for both $ts1$ and $ts2$ as illustrated in figure 4.9. If at least one of the detected variables in $ts1$ and $ts2$ matches, it will proceed to $ts3$ for separating the true variable pointers from the rest. If none of the detected variables in $ts1$ and $ts2$ match, then it will be assumed that there is no Carousel widget in the Web page. Next, if at least one true pointer is found in $ts3$, then $ts4$, otherwise no Carousel widget will be assumed to be in the Web page. Finally, if the Web page passes all of the first three tell-signs, then $ts4$ will ensure that the Carousel widgets will have characteristics that manipulate some styling features on the Web page before it is assumed that the Carousel widget existed. It is worth noting that the processing sequence for $ts3$ and $ts4$ will be repeated for each candidate variable filtered by $ts1$ and $ts2$.

4.3.5 Limitations

This experiment has been scoped to analyse code formatted in Hypertext, CSS, and JavaScript only. Thus Carousel widgets that are developed in Flash, Shockwave, VB-Scripts, or in any other format are not covered. However, if required the concepts can be applied for these coding format too.

The method employed to detect the Carousel widget is partly derived from extensive studies on the commonalities between the widget design pattern libraries and

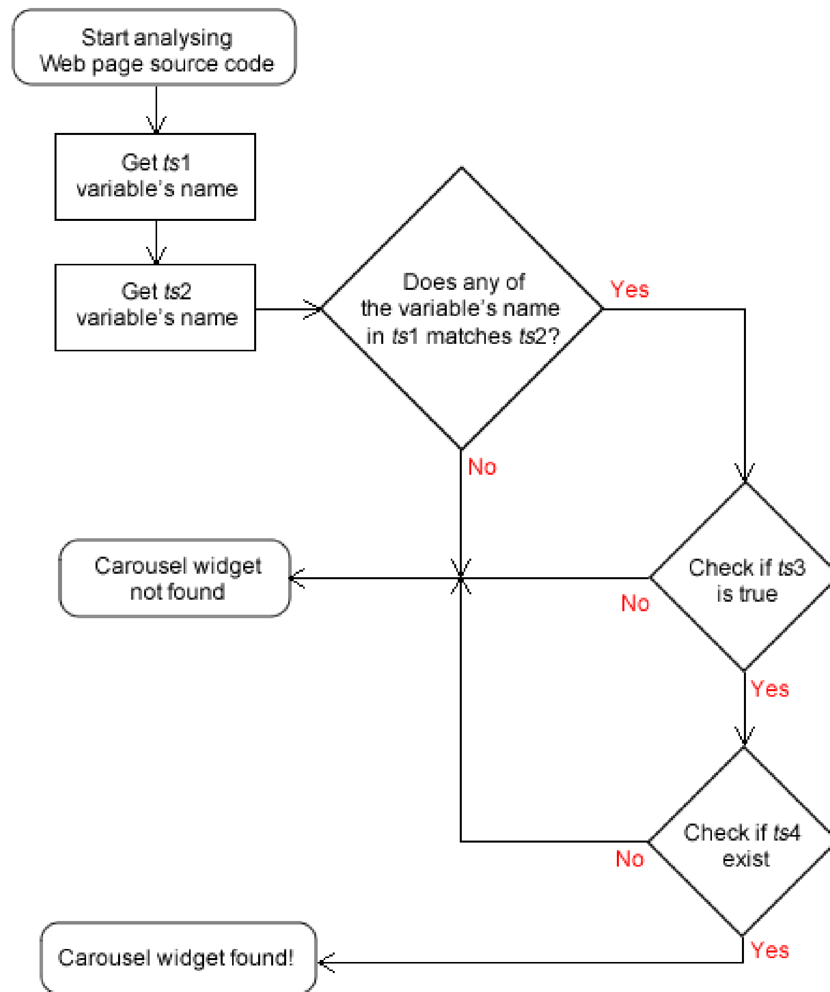


Figure 4.9: Carousel widget detection's tell-sign process flow chart

JavaScript libraries. Therefore, our approach is limited to the methods discussed by these libraries, but they can be expanded and modified to incorporate more variations.

4.3.6 Results and Discussions

The results presented in table 4.2 detail both the automated and the manual detection results for comparison, along with the results for the individual tell-signs.

The first column of table 4.2 lists, in order, the top twenty Websites selected for our experimental data set. The second column presents the results from our manual detection for the Carousel widget. The results for this detection are analysed manually by a human, going through the individual Websites to check if the tell-signs for the Carousel widget existed. The 'Auto detection' column lists the final conclusion drawn

Websites \ Checks	Manual detection	Auto detection	Increment Tell-Sign	Decrement Tell-Sign	True pointers Tell-Sign	Show and Hide Tell-Sign
yahoo.com	0	0	1	1	0	0
google.com	0	0	1	0	0	0
youtube.com	0	1	1	1	1	1
live.com	0	0	1	0	0	0
msn.com	0	X	1	1	X	X
myspace.com	0	0	1	1	0	0
wikipedia.org	0	0	0	0	0	0
facebook.com	0	X	1	1	X	X
blogger.com	1	1	1	1	1	1
orkut.com	0	0	0	0	0	0
rapidshare.com	0	0	1	0	0	0
baidu.com	0	0	1	0	0	0
microsoft.com	0	1	1	1	1	1
qq.com	0	0	1	1	1	0
ebay.com	0	0	1	1	1	0
hi5.com	0	X	1	1	X	X
aol.com	0	X	1	1	X	X
mail.ru	1	1	1	1	1	1
sina.com.cn	0	0	1	1	0	0
fc2.com	0	X	1	1	X	X
Total	**2**	**4**	**18**	**14**	**6**	**4**

Table 4.2: Feasibility investigation results for the Carousel widget detection. Where '1' is found, '0' is not found, and 'X' is ignored.

using our automated methodology as to whether the Carousel widget exists in a Web page. The next few columns show the results for the four tell-signs of the Carousel widget for each Website.

To interpret the results in table 4.2, a '1' symbolised a true, a '0' symbolised a false, and a 'X' symbolised that the investigation results were ignored due to runtime memory issues. For example, let's look at `blogger.com`. Since all the columns for this Website are '1', Manual detection and all the four tell-signs are true too, so auto

detection is true as well.

The Carousel widget is a piece of code that uses a combination of different languages to work. Hence, developers can employ a vast range of styles and practices to shape this widget. This makes it difficult to detect for all the different styles and practices for the Carousel widget. A close examination of table 4.2 reveals that the results for the Auto and Manual Detection do not tally, but a 100% detection rate is achieved when the Carousel widget is present. The investigations on Websites such as `youtube.com` and `microsoft.com` were falsely detected. Further investigating the types of Web widgets used by both Websites, it was noticed that both Websites contain the Tabs widget. As explained in §4.3.2, some parts of the Tabs widget are similar to the Carousel widget concepts, thus this issue has contributed partly to the false positive detection. On `youtube.com`, the combination of clues from both the ASL and the Tabs widget led to the false detection of the Carousel widget in this Website.

It is worth noting that the Carousel widget was also used on `ebay.com`. However, the developers of `ebay.com` chose to build them with Flash which is not included in this study. Issues with the limited memory allocation for each Web page by the PHP engine are also highlighted. This meant five Websites with a very large amount of code that use a lot of variables could not be evaluated. This is because of the amount of memory allocated by the PHP engine when interpreting a Web page. Thus for these Web pages, only the first two tell-signs were evaluated, and then the system froze when it ran out of memory. Since only the first two tell-signs were investigated, the results for these five Websites will not be taken into consideration in our evaluation.

The five Websites that were ignored are Websites with at least one column in table 4.2 that is marked with ‘X’. These are `msn.com`, `facebook.com`, `hi5.com`, `aol.com`, and `fc2.com`. Due to the runtime memory issues faced, no further work was followed up for these Websites since the majority of the Websites are fine to evaluate this investigation.

Since only 25% of the Websites from our experimental data set are affected by the memory issue, our conclusion was made with the remaining 75%. The results displayed in table 4.2 gave a promising sign for our detection method. We managed to successfully detect all the Carousel widgets that were present in the remaining 15 Websites, but two false positives were detected. Comparing these results with the ASL widget in §4.2.4, the Carousel widget gave more positive results.

Issues with PHP Memory Limit

The memory allocated to each Web page by the PHP engine imposed a limitation on our methodology. The PHP engine allocates each Web page a maximum of 8MB of memory when interpreting them. This becomes an obstacle when the system tries to comprehend large amounts of code that uses a lot of variables to complete its task.

The memory issue occurred because our Web mining methods use arrays to store the candidate variable pointers and variables identifiers that are suspected to be the list of content. The list of content on its own is an array already. Therefore, multi-dimensional arrays will be employed for our methodology. Memory management is another issue with PHP, because the interpreter manages this. An attempt was made to force the PHP memory limit to the maximum, but this did not improve the performance much, as it soon ran out of memory again. As a feasibility investigation, the memory issues faced with PHP were not pursued because it only affects a quarter of our experimental data set. Furthermore, the results are sufficient at this stage to prove the feasibility of our concepts, and our suggested refinements will also change the way the techniques will be applied in our approach.

4.4 Suggested Refinements

The results presented in tables 4.1 and 4.2 give a promising sign that our approach is feasible, but further refinements are required at this stage to fine tune and solidify our concepts before expanding the WPF. Thus, some of the methods and assumptions presented in this chapter may differ from the ontology described in chapter 3. This is because the methods and assumptions presented in this chapter are our first attempt to detect widgets, while the definitions in chapter 3 are refined approaches learned from this feasibility study.

A common problem is spotted with both sets of experiments: little bits of codes or clues from different areas within the code can be picked up as our tell-signs. This suggests an avenue to explore methods to link the clues in the source code to the User Interface objects on the widget.

4.4.1 Linking Tell-Signs User Interface Objects

Linking the user interface objects to the tell signs found will help the system to correlate the graphical user interface (GUI) objects of the Web widget with the related

source code, thus removing the non-related part of the code identified and improving the detection rate.

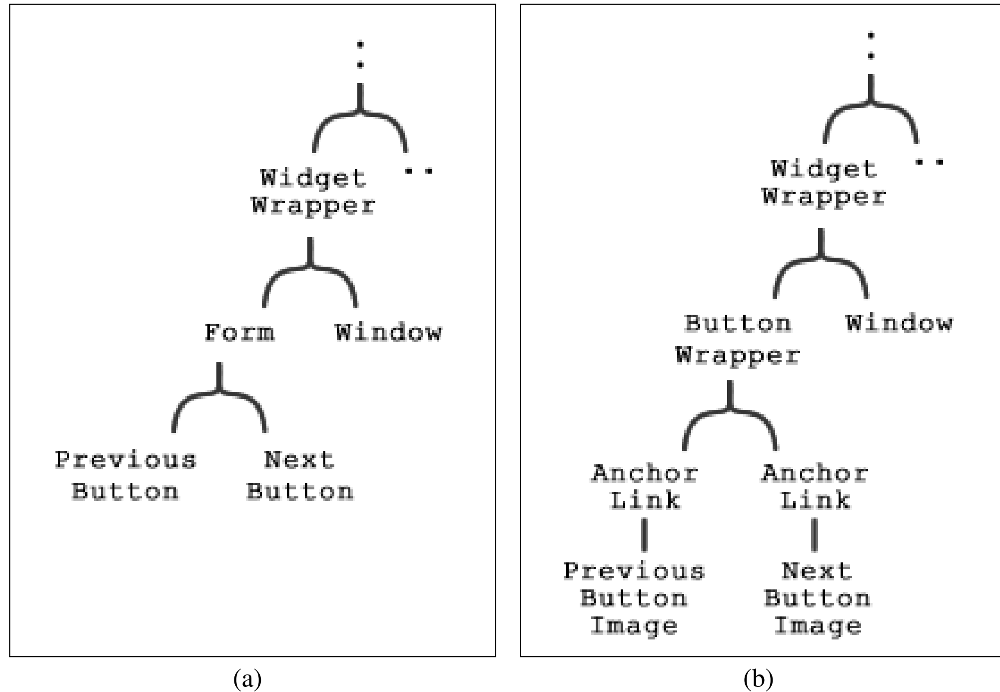


Figure 4.10: Static graph examples for Carousel widget?s UI components. Subfigure (a) shows an example of how the static graph will look like when form buttons were used for the 'Next' and 'Previous' buttons. Subfigure (b) shows an example of how the static graph will look like when images were used for the 'Next' and 'Previous' buttons.

For example, different forms of a button object (form buttons, and hyperlinks in the form of text or images) can be grouped together with the other Carousel widget's user interface components. By doing this, it will be possible for us to relate the button object with the section of code that forms the widget's engine, together with the element that is meant for displaying the content (Display Window) of the widget. The examples presented in figure 4.10 demonstrate that, by plotting the static graph of the first delivered Web page, the widget's interface elements can easily be grouped together by means of a hierarchical structure.

In some cases, the 'Window' node (Display Window) can be replaced with the actual list of contents, where each item in the list will be a node in the graph under the Widget Wrapper's node. Then, using CSS, the developer can control which node should be presented or hidden from the users. Using the hierarchical structure of the nodes, there is an assumption that a 'wrapper' will be used to interconnect the controls

of the widget with the Display Window of the Carousel widget. Hence, by identifying the wrapper, we can assume that the elements in/under the wrapper element have an association with it.

This work can be further divided into two parts so that we can comprehend the declarative code structure (hypertext) together with the procedure code structure (we refer to JavaScript for this study). By using this method, it will improve the understanding of the process and elements coherence.

4.4.2 Annotating the Source Code

Annotating the source code adds derived documentation from chunks of code before our method comprehends it. These concepts are borrowed from Takagi et al. [2002], where they demonstrated the power of annotation to develop an accessibility transcoding system. The process of annotating the source code is suggested as a pre-processing stage, to arrange and add semantic information derived from small portions of the code, so that it will provide useful machine interpretable documentation for our method to improve code comprehension. Other studies discussed in §2.2.3, under Annotation & Transcoding heading, also provide ideas to further develop the concepts. Applying these methods will assist the code comprehension processes by comprehending the code in great depth once, and not every time we need to infer something.

On the contrary, this method can be complex and it will consume a lot of additional computational power. Furthermore, the degree of automating the widget identification analysis will require consolidating the annotation to derive from the concepts of the process. This analysis phase may require another round of complex computations for each process. Thus, for the purpose of this study, this method may not be the most favourable.

4.5 Summary

In the tell-signs feasibility investigation, we demonstrated the feasibility of detecting areas where Web widgets are found in the Web page. Nevertheless, when designing these experiments, issues with defining the granularity of detecting the design pattern, and the traceability of the design patterns from the Web page's source code, were highlighted.

From the first experiment that detects the usage of the ASL widget, the feasibility

investigation results showed promising signs that our approach was applicable. The results showed that the detection for the ASL widget has successfully detected all ASL widgets present. However, a few false positive detections were also noticed. Further refinements are required to relate the suspected sections of source code with the widget's user interface components, as this may provide the solution to improve the 16% false positive detection.

The second experiment that attempts to detect the Carousel widget has brought up some critical issues with the scripting language during the feasibility investigation. It highlighted that PHP allocates only 8MB to each Web page during run time. This issue limited our evaluation capabilities, and 25% of the Websites from our experimental data set were unable to complete the investigation. Thus, this experiment was based on the remaining 75% of the Websites selected. In this experiment, all the Carousel widgets present were detected. However, two false positives were also present. These results shone a positive light on the method we employed, and when compared with the ASL widget detection methods, a lower false positive detection was noticed.

No false negative (Type II errors) detections were noticed in our methods, but to deal with false positive (Type I errors) detections, refinements should be followed up. These refinements will reduce the ambiguity in our definition of a widget, e.g. the Carousel widget, and provide a more robust methodology. A well-defined definition of the widget is an essential element to ensure a high detection rate. Both Web widget experiments successfully detected all the ASL and Carousel widgets, and through these evaluations issues were highlighted and the suggested improvements for better detections of the two widgets are presented. The suggested refinements will verify the detected widgets' presence, and help locate the parent element of the widget in the page. In the next chapter - Predicting Widgets On A Web Page, we discuss the extended work done to incorporate more types of widget and improve the reliability of our prediction method.

Chapter 5

Predicting Widgets On A Web Page

The feasibility investigation provided insights as to the effectiveness of the tell-signs in predicting widgets. Valuable suggestions arising from it were identified for refinement while expanding our approach. During the course of refining and expanding our approach, issues with false positive detection, conceptual and elements relationships, and identifying the tell-signs of the different components that form the widget surfaced and were presented in this chapter. These issues meant that some methods had to be modified to adapt them before they could be applied. The new and modified techniques are presented in this chapter, along with the tell-signs for all seven widgets.

From the feasibility investigation, it was learned that using physical coding methods to identify a widget's components as tell-signs is not the best approach. This is because often a concept can be achieved using a variety of methods depending on the developer's experience and perspective. Rather, anecdotal evidence suggests that capturing the conceptual clues of a component, from the source code to assist tell-sign identification, will make the detection process simpler and more effective in some cases. We will discuss the details of these tell-signs when presenting the methods used to capture them. Then, using the captured and derived information, the deduction of the different types of widget will be presented.

5.1 Issues With Analysing the Web Page Source Code

The heterogeneous nature of the Web and the weak-typed languages, used across various components in the page, allow these components to be loosely developed, and coupled together to make the page/application work. This architecture allows Web services to be reused, as well as complex code structures to form a page that seems

visually simple. In this section, we present the issues faced when implementing and expanding our approach, which mainly focused on the client-side methods due to the nature of the research.

5.1.1 Multiple External Resources

The beauty of the Web is to be able to interconnect multiple documents and reuse content from multiple sources in a page/application. Like desktop applications, Web pages/applications can also extract or reuse content/code from multiple documents to make them work in a page. Unlike desktop applications, Web pages/applications code/content can be hosted at multiple locations and combined later. The combination process of code/content can be nested as an iterative process that happens throughout the page's lifespan. Thus, before analysing the source code, code synthesis is required to gather the separately hosted code together.

5.1.2 Missing Code

To complicate matters, despite the complex structure discussed in §5.1.1, the process of combining code can happen as the Web page/application is interpreted by the user-agent. This means that developers can use the Lazy Load technique to request external code as a process in the page/application when it is required. Thus, the link to the code can be hidden within the code, and it is difficult to determine where in the code the Lazy Load technique is applied.

The Lazy Load technique is often employed to load a huge amount of code after the page is loaded to reduce the initial network overheads that the client has to bear [Zakas, 2010]. Although this method allows for faster page download times, while giving the developer the freedom to develop more complex pages/applications, it complicates the reverse engineering process as a whole, unless the reverse engineering process and proper documentation is available specifically for the Website or page.

5.1.3 Web Browsers Monitoring Management

JavaScript offers developers the features to monitor events occurring to a Web page/application, but often these features are either not well managed or not released by the Web browsers due to various reasons. In this section, we will look at the issues faced when attempting to recover the instances of Listeners, Handlers and Timers.

Listeners and Handlers

Listeners are often employed to monitor an element/node in the DOM for a particular type of event occurrence. When triggered, normally the browser's processes will be interrupted and the handler of the listener will execute a piece of code. These features allow developers to code reactive processes as Handlers when an event occurs. Often, a Listener is assigned to monitor an event that triggers the Handler when the event takes place on an element in the page.

However, the creations of Listeners are often not well documented by the Web browser when rendering the page. Furthermore, at times - due to the Just-In-Time (JIT) approach that Web browsers employed to render the code - some information may not be available. Thus, the approaches taken by the user-agents to improve rendering times make it difficult to detect the full extent of Listeners and Handlers initialised by the page. This issue suggests the need to provide a customised source code profiler, to keep track of Listeners and Handlers initialised. These records will aid the widget prediction process to determine whether an identified widget will be used by the Web page's user interface.

Window Timers

Timers provide developers with the feature to delay or execute certain processes at a fixed interval. Again, this feature is not well documented by the Web browser, or else, due to the JIT structure employed to interpret the code, the information is not available. For widgets that use this feature, extracting instances of this feature from the source code is difficult, and often requires different approaches to detect them. This is another scenario that suggests the source code profiler will aid by keeping track of initialised timers by the code.

5.1.4 Weakly-Typed Scripting Languages

JavaScript is widely used in Websites [Harper and Chen, 2012] to provide interactive features to the user and allow flexibility to developers to orchestrate the content in the page. However, JavaScript is a Weakly-Typed language, hence an even wider variety of coding styles and approaches can be taken by developers to deliver their ideas or concepts compared to Strongly-Typed languages. The flexible nature of the code makes it difficult to trace certain design concepts and techniques applied during development. Often, traces of concepts get lost in the code while attempting to derive them.

Using the declaration of Objects and Methods as examples, these declarations can be done using two approaches for Objects and four approaches for Methods. We will discuss this issue in more depth later in §5.4.2 when we look at how we can search for Methods and Objects from the code.

The Weakly Typed traits can also be found when declaring variables. This can be done in three ways too (`a=1;` `a:1;` and passed as arguments in a function), and normally the variable's data type is determined and managed by the interpreter. Due to these issues, determining the variable type is fussy and misleading at different stages of the code.

5.1.5 Non-Standardisation Interpretation

JavaScript literals supported by Web browsers are implemented in many variations from the standard specifications by ECMA-International [2009]. The different dialects of JavaScript vary between browsers, and often Web pages need to customise their code and content specifically for a particular browser. However, this does not stop at JavaScript; even the different versions of Markup Languages and style sheet languages have variants from one Web browser to another.

Websites commonly include a few variants of the same set of code for different Web browsers in a Web page. This technique is used to ensure their pages can accommodate a wider range of audiences. For our investigation, the Mozilla Firefox dialect was chosen due to its popularity[1] during the time this research was conducted. Furthermore, it provides a benchmark to base our proposed research on a leading industrial Web browser.

5.1.6 Just-In-Time (JIT) Interpreters

To improve code interpreting and loading speeds, and to accommodate the desire of Web developers and designers to include a large set of code in their Web page – so that more interactivity, usability and various application specific purpose can be incorporated – many Web browsers resort to the Just-In-Time (JIT) approach used to interpret Web documents. Although this technique has its perks, it also makes it difficult to reverse engineer a Web page unless it is done specifically for a particular Website and the documentation of the designs for the Website is available.

[1] W3Schools, Browser Statistic. Accessed on 24 Feb 2011, http://www.w3schools.com/browsers/browsers_stats.asp

5.2 Profiling The Web Page Source Code

Applications working in the Web domain sometimes require some tools to assist them to complete their task when faced with the variety of technological issues. Many Rich Internet Applications (RIA) use Web 2.0 technologies to enable them to communicate and monitor different processes and events from different conceptual layers within the page – such as the Presentation, Behaviour and Content layers as seen in figure 5.1. The content within the different layers can change dynamically throughout the timespan when the page is viewed. Mainly, the content is manipulated by the Behaviour layer, where it can be updated with new content when it is designed to do so. All these interactions enable the Web page to be dynamic, hence providing better interactivity with its user, and reducing the number of page reloads, thus improving the network communication efficiency.

To accommodate the advances, Web developers often use tools that incorporate special techniques to assist and speed up their development process. These techniques use sophisticated concepts to include the desired technology or concepts in Web pages, so that they are generic for most cases. Comprehending pages that use such tools often requires in-depth analysis of the code to make inferences that these techniques are used, so that the technologies and concepts used can be inferred. Using the technologies and concepts derived from these techniques, together with other concepts derived from the code, the type of widgets used on the page can be inferred.

Due to the wide choice of technologies available, most available tools only tackle the generic issues. These tools are targeted to provide assistance for Web development, or they only solve part of the problem. The process required for our investigation involves comprehending the code and reverse engineering the Web page, so that widgets can be logically inferred and, when possible, the location of the widgets in the Web page interface can be identified. Thus, a specially tailored code profiler is introduced here to assist our widget identification investigation. This profiler will only record the necessary information required by our deduction process, providing crucial and detailed records about the code.

The Web 2.0 concepts indirectly allowed Web pages to be more complex with technologies and recommendations that are less structured, more flexible and lightweight. Such a nature was chosen for the Web to make it more compelling to others to use it [Millard and Ross, 2006]. Furthermore, the advances in these technologies and recommendations do not always get adopted [Chen and Harper, 2008] or used effectively. Due to these reasons, a wide combination of methods is commonly employed for a

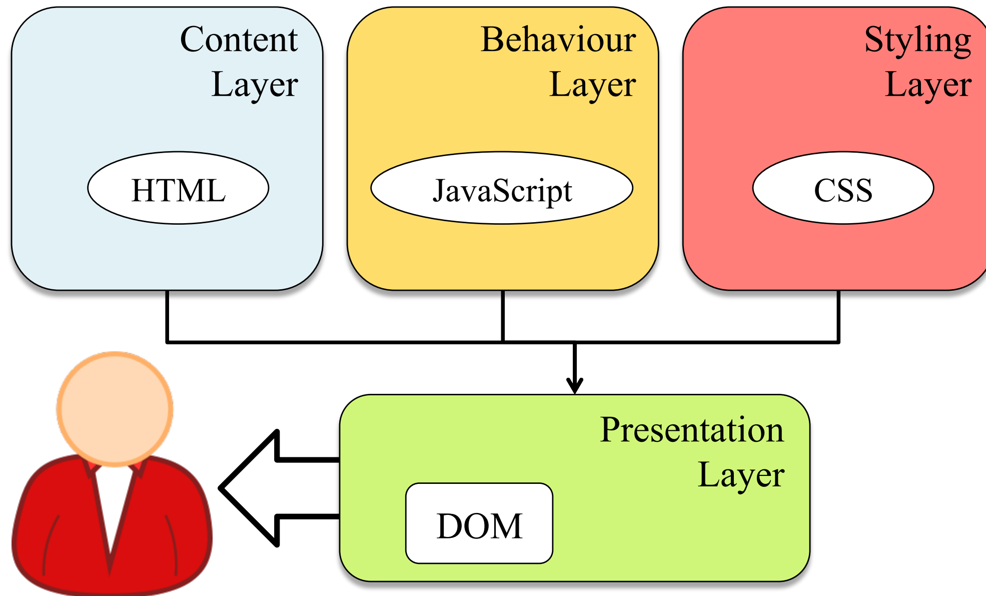


Figure 5.1: Overview of the browser interpretation process.

task. This makes it difficult to comprehend the code to derive from the methods taken by developers and the task it was designed to do when conducting reverse engineering.

In order to understand how to mine useful information that correlates with what is presented by the Web browser at an instant in time, a general concept of how the browser renders the source code is necessary. Every time a Web page is loaded, the Web browser renders it with a similar interpretation structure shown in figure 5.1. When the page is first loaded, the browser compiles the Content layer and uses a Scripting engine to interpret the scripting code in the Behaviour layer. Then, the Document Object Model (DOM) is rendered using the information provided by the Content, Behaviour and Styling layer in the Presentation layer. Using the DOM, the browser renders the Web page's interface and presents it to the user in the Presentation layer.

Within the load lifespan of a page, each time an event (user or machinery) is triggered, the DOM will be reinterpreted, changing the page's interface where required. This process repeats itself until the page is unloaded. Most major browsers use the generated DOM as a form of communication to relate the current interpreted interface of the Web page with other technologies. Due to the dynamic nature of generating presented content and the source code, both the source code of the page and the DOM have to be analysed alongside each other.

5.2.1 Handling Events

Many Rich Internet Applications (RIA) consist of three conceptual layers – such as the presentation, behaviour and content layers – when interpreted by the Web browsers, as seen in figure 5.1. The content within the different layers can change dynamically throughout the lifespan when the page is viewed. This is mainly manipulated by the behaviour layer to mutate the DOM in the Content Layer.

Commonly, event listeners are employed to monitor these events. Upon triggering the event, the behaviour layer executes the event handling set of code to execute the task the developer designed it to do. Each event will require a separate listener to monitor it. Therefore, a few listeners are often required by a dynamic Web page. Due to this need, developers use a Factory method technique to semi-autonomously generate the event listeners. This technique sometimes makes it difficult to trace if an instance of an event listener was created.

5.2.2 Dynamic Code Generation

Using Remote Scripting technologies allows developers to insert content into the Web page without reloading it. Through this means, code in the Behaviour layer can also be dynamically inserted for various purposes. One of the uses of this method is to improve the initial loading speed. This technique sends the necessary content to the client-side first, so that the structure and the important content the author intended for the user can be loaded. After loading the page, the rest of the code – usually the "behind the scenes" code that reacts to the events – will be transmitted. Using this technique, developers aim to give their users a misperception that the page is loaded quickly or in an acceptable time.

Since both content and code can be generated dynamically, some RIAs use this feature to enhance their application interactivity. In the course of doing this, event listeners may also need to be generated semi-autonomously employing the technique discussed in §5.2.1.

5.2.3 Redundant Code

Most developers often built a number of widgets for the Website, and these are commonly stored in a library that is attached along with most pages in the Website (see §5.1.1). In some cases, these libraries contain all the most commonly used widgets. In

most cases, this is done by specifying the `src` attribute in the `<script>` tag with the location of the library to attach it to the Web page.

This technique is encouraged because the reuse of methods as objects is deemed to be good programming practice, and it reduces development time. However, unlike traditional programming, over the Web these libraries can be either located on the server-side of the Website, or located on a separate server so that it will be a centralised location for a few Websites. These practices may be more efficient and cost effective during development, but the cost is being transferred to the client. Now, end-users have more to download that includes redundant code that is never going to be used by the Web page.

The Web constantly evolves due to its popularity, competition between Websites, and the developers' desire to meet users' expectations. In such an environment, having an efficient, iterative and incremental development cycle is important to promote continuous improvements to the code and design [Maurer and Martel, 2002]. Thus, developers often resort to attaching external libraries of widgets they have created onto Web pages, so that the library acts as a single point of access to distribute their improved code.

Using this technique often results in false positive detection when trying to detect widgets [Chen and Harper, 2009]. Hence, additional inferences have to be made before a widget can be affirmed to exist. The additional inferences process searches for the elements within the DOM that trigger the suspected set of code. If such a link can be bridged, then the identified widget can be assumed to exist. On the contrary, if a link between the set of code and at least one element within the DOM cannot be formed, then the widget will be assumed not to exist. In the next few sections, we will discuss the approaches explored and the final approach taken on-board in our system.

5.3 Approaches to Identify Web Widgets

Finding the tell-tale signs of Web widgets in a Web page can be done using the tell-sign approach, as proven in Chen and Harper [2009]. Due to the advances and nature of the Web, issues relating to the inclusion of Web 2.0 technologies can be improved by informing users about the widgets, as noticed from Brown [2012] evaluation transcripts. One of the more critical issues was to deal with false positive widget detection. As discussed in §5.2.3, developers often consolidate there code into libraries and embed them in the pages they have developed. Using this development technique, developers

are able to deploy agile development life cycles for their widgets, while maintaining the different versions of the code.

Currently, widgets are designed from concepts described by design patterns, but these concepts are often lost within the code because programming/scripting languages do not support them [Bosch, 1998]. The fusion of different technologies that makes the Web work will require different types of techniques to deal with the heterogeneous nature of the Web. The widget ontology was built to uniquely categorise the types of widget and provide shared documentation of the widgets' definition in the automated identification process. Different approaches to finding the location of widgets within a Web page were experimented with before arriving at the current approach of using a tailored design "Profiler".

5.3.1 Bottom-up Approach

From the beginning, a bottom-up approach was experimented with in order to reverse engineer the Web pages. This method analyses the entire set of code and makes its way to the abstract concepts derived from the code. With these concepts, inferences to identify the type of widgets in the page can be made from the source code. As seen in Chen and Harper [2009], it is feasible to use this approach to assist widget identification. However, further research and tweaks are required to improve its accuracy.

One of the major hindrances of this approach is false positive detection due to issues such as those described in §5.2.3. Additional steps and the fusion of techniques are required to minimise false positive detection. Records from the analysed code must be cached to assist later inference to determine whether a widget detected is truly used by the page. Depending on the size of the page, this approach may require a huge overhead to compute its deductions. Hence, additional components will also be required to manage the overheads.

5.3.2 Top-down Approach

A top-down approach was investigated in the attempt to overcome the issues highlighted by our bottom-up approach experiments. This method examines the Web page's DOM, then it tries to make its way down to the JavaScript code in the Behaviour Layer. Only JavaScript code that is related to the DOM element(s) is examined, and hence lesser overhead is required to compute and store the deduction.

Since the introduction of Remote Scripting, developers commonly employ listeners

to monitor other technologies and the targeted elements within the DOM. These evolutionary concepts become an issue when the system attempts to bridge the connection between the DOM elements and the JavaScript code. Often this link is lost when Remote Scripting methods are employed. Thus, techniques such as a fusion of both top-down and bottom approach are suggested, or implementing a tracer component so that the system can follow the clues that lead to the JavaScript code can be employed to tackle these issues. The fusion of the Top-down and Bottom-up approaches was investigated first because it was the easiest choice to adapt our script, and it requires the least amount of time.

5.3.3 Fusion of Top-down and Bottom-up Approach

To reduce overheads and assist the system to determine the location in the Web page where a widget is used, an experiment to combine the top-down and bottom-up approaches is conducted. To apply the concept, a reference point is required to determine the starting point of the approach. This point must be selected carefully so that the advantage of both approaches can be harnessed.

Selecting the Reference Point

As seen from the top-down approach, the link between the widget's DOM elements and the JavaScript code is often lost. This is due to the different approaches developers take to trigger a piece of JavaScript code. Some of the more popular methods include implementing listeners and handlers either from JavaScript frameworks or manually coding them, or in some cases, inline JavaScript is used to trigger a set of code. On some occasions, even inline JavaScript is used to implement the concepts directly. From these examples, the magnitude of diversity to link the DOM elements to a piece of JavaScript code can be seen.

The reference points chosen are where the missing connections are suspected, to bridge the widget's DOM elements together with the JavaScript code that manipulates them. Although it is difficult to include all possible approaches in our system, often the reference point is related to either event handlers or listeners of the DOM element(s). Hence, our system uses the event handlers or listeners as the reference points. Therefore, every Web page will have a different amount of reference points, and the code comprehension process is repeated for every reference point detected.

Mechanics of the Approaches

Once the reference points in the Web page are selected, the top-down approach is used to trace the JavaScript code to the suspected set of code for analysis. Meanwhile, the bottom-up approach will determine the DOM elements that are monitored by the event handler/listener. After a widget is identified, inferences can be made from the relationships and groupings of the DOM elements, together with other DOM elements manipulated by the suspected set of code, to determine whether the widget exists in the page.

Issues with the Fusion Approach

Using the fusion of approaches will give the best of both worlds, and reduces both overheads and false positive widget detection, but it also introduces new problems. The accuracy of determining the reference points is crucial. If the reference points cannot be identified, then the widgets will never be discovered. Another issue with this approach is that it does not always resolve the bridging issues faced by the solely top-down approach. Thus, this suggests implementing a tracer component to assist with the searching of the connection between the event handlers/listeners and the set of code in the Behaviour layer.

5.3.4 Implementing a Tracer

Following the clues in the code to link up the event handlers/listeners to the set of code that manipulates the DOM was suggested. A tracer was believed to be able to produce the trace and provide the connection for searching the widget's location.

Existing JavaScript tracers were commonly found within debuggers - such as Hopkins [2011]; Firebug [2011] - to assist developers when debugging, as well as Just-In-Time (JIT) compilers to assist the code interpretation process [Gal et al., 2006]. Tracers used by JIT compilers only interpret JavaScript code that is related to the process during that instance in time. To identify widgets, the entire set of JavaScript code used by the Web page is required to make inference of a widget. Available tracers such as jsTracer [Hopkins, 2011] only provide a platform for developers to insert messages and the type of error detected into either a log file or a viewer to assist developers with their development. These types of tracer are useful for development and dealing with specific Web pages, but not useful when attempting to reverse engineer generic Web pages. Debuggers such as Firebug [2011] use the browser's JavaScript engine to trace

the JavaScript code. This type of tracer only provides another layer on top of the JIT interpretation concept to assist developers with their development. In order to use it, additional components are required to guide the tracer, so that the entire JavaScript code can be traced.

5.3.5 Implementing a Profiler

Documenting relevant information about the JavaScript code is another technique to assist code comprehension. Often compilers and interpreters, such as Rhino [2011] and V8 [2011], profile code to benchmark their performance. This form of profiling the code structure includes the time taken by the compilers/interpreter to execute them. Debuggers such as Firebug [2011] use profilers to assist in tracing the code as well as to evaluate the performance of the code.

After investigating the usage and implementation of profilers, it can be concluded that, depending on the objectives, profilers can be created to document information about the code that is useful for the task. The profiler used by Firebug is useful for our work, but it relies on the DOM generated by the browser, thus JIT concepts are used and they do not profile the entire set of code. In order to use Firebug, additional implementation is required to direct the profiler, so that it will document the entire set of code. This implementation will be difficult as every page is different, and additional overheads will be incurred.

5.3.6 Tailor Designed Profiler

From our investigation of different approaches and the available tools, a tailor-designed profiler is suggested to assist bridging the DOM elements and the inferred set of JavaScript code. The tailor-designed profiler will document the structure of the code as well as the locations of the structure and other essential details. Basic inferences will also be made so that tracing and more in-depth analysis of the code can be conducted.

5.4 Analysing The Code

Collecting information about the code and recording it for further code comprehension is a technique used to reverse engineer programs. Using the records profiled by the code profiler, different types of extensive analysis can be conducted for a variety of

purposes. Depending on the purpose of profiling, different parameters within the profiled records of the code are analysed. The structure of the code, the listeners within the code, and the location of different methods and objects are of interest. This is because, with this information, inferences of different suspected code for our widget prediction process can be made.

The profiler plays an important role in the widget's detection investigation, but a few additional blocks are also required to prepare the code for profiling. Figure 5.2 depicts the overall architecture of the Widget Prediction System (WPS) and the flow of the system. In the beginning of the page analysis cycle, the page is interpreted and the DOM is parsed ①. Using the DOM structure, the paths of external coding files embedded in the page are extracted for downloading. Then, the downloaded sets of code are synthesised with the original page in the 'Code Synthesiser' block ②. The profiler resides in the 'Code Profiler' block ③, which taps into the results produced by the 'Code Synthesiser' block. Finally, widgets are inferred in the 'Widget Inferences' block ④ to predict the types of widget in the page. In the 'Widget Inferences' block, the concepts modelled in WIO are hard coded to make the deduction of the types of widget.

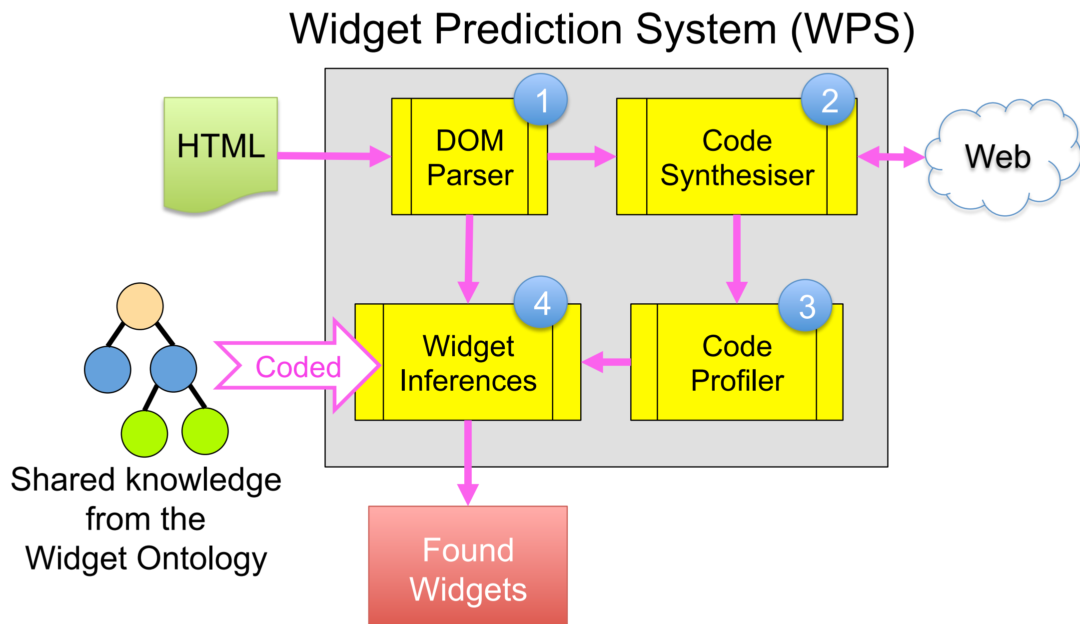


Figure 5.2: The Widget Prediction System (WPS) overall architecture depicting where the shared knowledge from the Widget Ontology is hard coded.

The 'Widget Inferences' block is extensible, thus tell-signs can be included or excluded, and different types of widget inferences can be added or removed when required. The requirements and steps for the expansion process will be covered in §5.4.4, and the next few sections will describe the different components and processes within the 'Widget Inferences' block.

The entire WPS architecture is extensible as well. Additional blocks can be included into the architecture if required. Thus, the data structures used to transmit information between the blocks will also be discussed.

5.4.1 Synthesising the Code

Since HTML can be parsed and analysed, only the client-side scripting will need to be comprehended. As previously discussed in Chen and Harper [2009], the investigation will only cover widgets that are developed using JavaScript. JavaScript code can be included within HTML or embedded from an external source, therefore the "Code Synthesiser" block ② will compile the code from different places into a string for further analysis.

5.4.2 Inside the Profiler

Methods, objects, event listeners and instances of tell-signs are searched and recorded in the "Code Profiler" block ③ in figure 5.2. The "Code Profiler" block captures and records instances of the code specifically for our investigation. This part of the system follows the process flow shown in figure 5.3 to mine and make important inference of the results gathered. As most JavaScript interpreters use the JIT technique, the code's syntax tree will be compiled for an instant in time. Commonly, this instance will be the first instance when the page is loaded. Instead, regular expressions are employed to sniff through the code for the code construct patterns.

Searching for Methods

There are four ways to declare a method or function that is covered by WPS. Developers can deploy these approaches anywhere throughout the code. Thus, keeping track of where methods are declared, what are included in each method, and how each method communicates with one and the other has to be managed properly. It is assumed that

Figure 5.3: Widget Prediction System (WPS) Code Profiler block's internal process flow.

tell-signs found within the same method are related due to their parentheses association. Using this concept, recording the method's identifier, input arguments, its location in the code, and where it begins and ends will be useful information for inferring the tell-signs relationship.

The profiler covers four common types of function declaration. The first way (M1) is as follows,

```
/*M1*/  function identifier (arguments) { function body }
```

In order to detect this form of declaration, two phases of the detection process are employed. The following regular expression is used to search for instances of M1 code construct patterns in the first phase.

```
function\s+[_a-z0-9\$]+\([\w_\s,]*\)
```

Once all the instances of this form of function declaration are found, the following regular expression is used in the second phase. This phase extracts the properties of the method, such as the method's identifier and the input arguments for each record found by the first phase.

```
\s([_a-z0-9\$]+)\s*\(([\w_\s,]*)\)
```

The second way (M2) is often used to declare methods within an object. Instances with this form of method declaration are mined at this stage rather than when mining for objects. With this approach, the profiler can provide a quick and thorough search for all instances of methods, leaving the semantic relationship of an object's method to be derived later.

Functions can also be assigned to a variable so that the variable can act as an identifier for the function (M3). This form of declaration is an assignation approach similar to M2 when declaring a method within an object.

```
/*M2*/  identifier : function (arguments) { function body };
/*M3*/  var identifier = function (arguments) { function body };
```

The approach for M2 and M3 will be mined in three stages. First, instances of both of these approaches can be searched using the following regular expression:

```
[_a-z0-9\.\[\]\$\)\(\'\"]+\s*[=\:]\s*function\s*\([\w_\s,]*\)
```

After instances of M2 and M3 are mined, the following regular expression is used to segregate the identifier from the function declaration syntax for each record in the second stage.

```
\s*[=\:]\s*
```

By segregating the identifier from the function declaration syntax, the input arguments of the function can be extracted using the following regular expression for each record in the third stage.

```
function\s*\(([\w_\s,]*)\)
```

Functions can also be declared without an identifier (M4) in JavaScript, such as `function () { function body }`. To identify instances of this type of declaration, the following regular expression is employed.

```
[\;\(\),\{\}\|\s]function\s*\([\w_\s,]*\)\s*\{
```

Functions declaration M2 and M3 can easily be mixed up by the system to be M4. Thus, notice in this regular expression that additional conditions are included to reduce the false positive detection for this type of declaration. Table 5.1 illustrates the different coding patterns that will and will not be picked up by the regular expression designed for M4.

Additional filters are included to exclude the false positive records picked up by M4's regular expression. These filters check the starting location of the current record and compare it with previous records. If a match is found, the current record will be discarded, leaving only the true positive records remaining.

Next the input arguments of the remaining records identified by M4's regular expression use the following regular expression to detect it.

Will Not Be Picked Up	`var a=function() { function body }`
	`var a =function() { function body }`
	`var a:function() { function body }`
	`var a :function() { function body }`

Will Be Picked Up	`var a = function() { function body }`
	`var a= function() { function body }`
	`var a :  function() { function body }`
	`var a:  function() { function body }`

Table 5.1: Code patterns for functions that will and will not be identified by M4's regular expression

```
function\s*\(([\w_\s,]*)\)\s*\{
```

Searching for the starting and ending locations for the function can be tricky and challenging, since within the input arguments of the function and the body of the function, almost any type of character can be included. Thus, the following code in listing 5.1 is used to check character by character for its type and the inset level of the parentheses. Variables `startChar` and `endChar` will store the character location of both the starting and ending location of the function body respectively.

```
1  // Declare necessary variables
2  var inset = 0;  // depth of the code
3  var startChar = false; // method starting character location
4  var endChar = 0; // method ending character location
5
6  // Search method ending
7  for(var fmedi = mStart; fmedi < doc.length; fmedi++)
8  {
9      // locate code depth
10     if(doc[fmedi] == '{')
11     {
12         startInMethod = true; //flag that method has started
13         startChar = fmedi; //function starting location
14         inset++;  //increment inset counter
15     }  // if
16     else if(doc[fmedi] == '}')
17         inset--;  //decrement inset counter
18
```

```
19        // If true end is found
20        if(startInMethod && inset == 0)
21        {
22            endChar = fmedi;   //function ending location
23            break;
24        }  // if
25    }     // for
```

Listing 5.1: Searching for the encapsulated body of code within curly brackets ({...}).

Finally, the data structure object as seen in figure 5.4 is used to store the discovered functions. In the data structure object, every record of a function is divided into five arrays of methods so that the features of the function can be stored. To access a record, the location for every method in data structure object will be the same. For an example, if a function has an identifier called 'top' in record number 1 (`method.identifier[1]`), then to access the ending character location of its body it will be `method.end[1]`.

identifier	location	start location	end location	input arguments
[0]	[0]	[0]	[0]	[0]
[1]	[1]	[1]	[1]	[1]
:	:	:	:	:

Figure 5.4: Data structure to store the functions found in the JavaScript code of a Web page.

The 'identifier' method in the data structure object stores the identifier for the function found. The 'location' and 'input arguments' variables store the location in the code where the function was found, and the identifiers of the input arguments from the respective functions. Then, the 'start location' and 'end location' variables store the starting and ending character locations of the function's body respectively.

For example, using the first instance of a function found in line 1 in listing 5.3, this record will be stored as `identifier = setListen, location = 0, start location = 31, end location = 130, input arguments = 'a, b, c'`. Storing this type of information will assist the later code comprehension process, and assist the semantic relationship inferences between different tell-signs and sets of code.

During development, often developers use comments to describe a piece of code or

temporarily exclude a set of code. The system does not deal with instances of patterns that are commented upon; hence, these instances may be profiled as well.

Searching for Objects

Objects are not commonly associated with JavaScript because it is known as weakly typed and has first-class functions. However, the Objects concept can be incorporated and they can be declared in various ways. Two approaches to declare Objects are covered by the profiler. The first approach uses the literal notation as follows,

```
var object_identifier = { ...;
                          method_identifier : ...;  }
```

To identify the patterns of this type of Object declaration, the following regular expression is employed to search for the declaration and starting location of the object.

```
([_a-z0-9\[\]]+)\s*=\s*\{
```

After locating the Object, patterns to search for identifiers assigned with content encapsulated within a pair of curly brackets will be searched. The second approach to declare an Object is by using `new Object()`. The following regular expression is used to detect patterns of this form of declaration applied to an instance.

```
([_a-z0-9]+)\s*=\s*new\sObject
```

Once an object is identified, the algorithm in listing 5.1 is used again to search for the starting and ending location of its body. For example, in listing 5.3, the first instance of an object on line 11 exhibits a similar pattern to the literal notation approach to declare the object. In this case, `z =` { will be picked up by the first approach's regular expression. Then the starting and ending location of the object's body will be searched using the algorithm in listing 5.1.

Comments left by developers may exhibit similar patterns to those approaches to declare an Object. These instances within the comments will be picked up and recorded by the profiler as well.

When an object is found, the data structure used to store these records is similar to figure 5.4, except this time only four parameters are stored instead of five. As seen in figure 5.5, these are the identifier of the object, the starting and the ending location of the object. The data structure object for the found objects can be accessed in the same

way as the function's data structure object. Referring to listing 5.3 as an example, the first instance of an object on line 11 will be stored as `identifier = z, location = 327, start location = 331, end location = 332` in the data structure object.

identifier	location	start location	end location
[0]	[0]	[0]	[0]
[1]	[1]	[1]	[1]
:	:	:	:

Figure 5.5: Data structure to store the objects found in the JavaScript code.

After the methods/functions and objects in the code are found, methods within an object can be inferred from these two sets of extractions. Methods that are found within an object can be rightly assumed to be the methods of that object.

Searching for Event Listeners

Event listeners can provide the widget detection process with the link between the DOM elements that interact with the user, and the set of JavaScript code that handles the request from the triggering event. With this knowledge, it can be affirmed whether a widget detected is used in the Web page, thus removing false positive widget detection.

The method to include an event listener is not part of the standard ECMA-262 specification [ECMA-International, 2009]. Commonly, these methods are part of a JavaScript dialect specific to a Web browser. The differences between the JavaScript dialects are divided into two main categories: Internet Explorer (IE) and non-IE. Since most popular websites provide scripts for both of them, only the non-IE scripting such as the following will be covered.

```
element.addEventListener(EventType, Handler, useCapture);
```

Instances of this pattern can be searched within the code using a two-part process. The first part uses a regular expression such as the following to capture the first instance of the pattern within the code.

```
[\(\)\{\}\;\:\s\?\,]([_a-z0-9\.]*)addEventListener\(\s*([a-z0-9'"]+)
       \s*,\s*([\w\W]+)
```

Since the handler can be either an identifier for a function or an unnamed function (inline function declaration), it will be difficult to use only regular expression to capture the handler. The above regular expression will return three sets of data, with the last one returning the remainder of the code after the argument that specifies the type of event that triggers the listener. Hence, a second part of the process is required to analyse the remaining code for the remaining features of the event listener's detection. This process uses the following JavaScript code in listing 5.2 to scan through each character in the string ('doc' variable) for the start and the end of the function if an unnamed function approach is employed. In the case where the function identifier is used, the search in the remaining code will be aborted and the 'found' variable will be assigned with a boolean value `false`.

```
1    // flag to check if the start of the method is found
2    var methodStart = false;
3
4    // Search method ending
5    for(var fmedi = 0; fmedi < doc.length; fmedi++)
6    {
7        // locate code depth
8        if(doc[fmedi] == '{')  // if { is found
9        {
10           // flag that the start of the method is found
11           methodStart = (!methodStart)?true:methodStart;
12           inset++;    // increment inset count
13       } // if
14       else if(doc[fmedi] == '}')     // if } is found
15           inset--;    // decrement inset count
16
17       // if a comma is found before method
18       if(doc[fmedi] == ',' &&  inset == 0 && !methodStart)
19       {
20           found = false;    // No method found.
21           /* Indicate endChar so that method identifier can
22           be located by the callee */
23           endChar = fmedi;
24           fmedi = doc.length;    // End loop
25       }  // if
26       else
```

```
27                  found = found + doc[fmedi];   // Cache data
28
29              // if the method is found and the ending is detected
30              if(methodStart && inset == 0)
31              {
32                  /* Store current location as ending character
33                  location */
34                  endChar = fmedi;
35                  fmedi = doc.length;    // End loop
36              }  // if
37          }     // for
```

Listing 5.2: Algorithm to deal with complex cases within a function's input argument.

Notice the variables 'inset' and 'methodStart' are to help the search process to determine the depth of the code, and determine if the start of the methods is found respectively. These variables are important because if 'methodStart' is boolean `true`, then an unnamed function is found, otherwise a function's identifier will be assumed. When an unnamed function is found, the 'inset' variable will behave like an indicator to locate the end of the function. Next, this two-part process is repeated to search for more instances of the pattern to add event listeners in the remaining code. This is an iterative process until the string 'doc' that contains the code is exhausted. Similar to the previous detections, patterns within developer's comments that are similar to those discussed for the event listeners detection will be picked up by the profiler too.

Finally, the details of the found event listeners are recorded in a data structure object containing four parameters. As seen in figure 5.6, these are the location of the event listener, the trigger event, the element to be monitored, and the handler of the triggered event on the element monitored.

Location	Triggering Event	Monitoring Element	Handler
[0]	[0]	[0]	[0]
[1]	[1]	[1]	[1]
:	:	:	:

Figure 5.6: Data structure to store the event listeners found in the JavaScript code.

Using the example in listing 5.3, an event listener will be picked up on line 2.

In the data structure object that stores the profile of the event listener, it will look something like `location = 60, triggering event = b, monitoring element = a, handler = c`. Notice that in this case, the triggering event, the monitoring element and the handler can be variables. Unless it is a Factory Method design pattern concept, the details of these variables are not crucial for our deduction.

Event Listener's Factory Method

Quite often event listeners are included in the Web page by calling a function that instantiates the event listeners when required; similar to the concepts of a factory method design pattern [Gamma et al., 1995]. This technique is commonly used for a variety of reasons and recording the inferences of this technique will be useful when trying to infer a widget.

Since the event listener's factory methods have a transitive relationship between event listener that is encapsulated within the function, and the function's callees, determining these instances can only be inferred from our profiled records. Profiles of functions and event listeners will be analysed to find the qualified candidates.

Checking within the function for event listener declaration can help to decide whether the event listeners factory method technique is applied. This is done by searching within the profiled functions for a function with a starting location smaller than the event listener's location, and ending location larger than the event listener. The segregated function's input arguments will be compared with the event listener's triggering event, element and handler. If at least one of the three parameters matches then a factory method to declare event listeners is discovered.

After the factory method of the event listeners is found, these details are recorded in a data structure object containing three parameters as seen in figure 5.7. These parameters are the identifier of the method in which the factory pattern is housed, the starting and ending locations of the method.

Referring to listing 5.3 as an example, an inference of an event listener's factory method will be made on line 1. This is because the `monitoring element`, `triggering event` and `handler` of the event listener are the same as the input arguments of function `setListen`. In this case, the event listener factory method will record this as follows `method's identifier = setListen, starting location = 31, ending location = 130`.

Callees of the event listener's factory method are also searched and recorded by the profiler. This process searches within the JavaScript code for patterns similar to the

Method's Identifier	Starting Location	Ending Location
[0]	[0]	[0]
[1]	[1]	[1]
:	:	:

Figure 5.7: Data structure to store the event listeners factory method found in the JavaScript code.

factory function's identifier. However, this process will pick up the function declaration and callees locations. Thus, filters are used to exclude instances with locations similar to any profiled function's location.

5.4.3 Widget's Inference

The "Widget Inferences" block ④ in WPS (see figure 5.2) makes the decision whether or not a widget is detected at a location, or whether there are sufficient clues that a widget will exist. To do this the "Widget Inferences" block examines the DOM, along with the synthesised JavaScript in ② and the "Code Profiler" block ③ results to make such a deduction. The deduction for inferring a widget is based on the tell-signs for each type of widget described in §5.6.1 to §5.6.6. The "Widget Inferences" block ④ was developed based on the concepts in WIO for each type of widget. A confidence percentage is awarded for each candidate suspected to aid the prediction if a widget exists at a location. Therefore, it is not required for all essential tell-signs to be present to infer a widget. Keywords and content structure can also be used to assist the prediction process.

When the deductions of the page are done, the "Widget Inferences" block results will be returned as WPS final decision of the types of widgets and their details (see figure 5.2). Each results return will consist of the type of widget WPS has deducted, the confidence percentage of the detection/prediction, and the XPath of where the widget is located.

5.4.4 Expanding the System

The system's architecture is designed to be extensible for adding or removing tell-sign objects and widget's inferences, and modifying the current processes to detect

methods, objects and event listeners in the code. Only the inclusion or removal of tell-sign concepts will be discussed here because modifying existing approaches can be done by extending the process flow in their respective blocks in figure 5.3. All mining processes for tell-sign's objects are executed in the "Tell-sign's Objects" block as seen in figure 5.2.

The widget detection process is split into two parts: the first part is in the "Code Profiler" block and the other part in the "Widget's Inference" block. Using this method, all code mining and useful inferences are conducted in the "Code Profiler" block, followed by the widget's derivation in the "Widget's Inference" block. The reason for splitting the widget detection process is that, when the system is deducing the type of widgets and linking them to the DOM element(s) that the user interacts with, there is no need to return to mine the code whenever inferences are required. Another reason for splitting the widget detection process is to allow additional tell-sign objects to be included or excluded whenever required.

Packaging the Tell-signs

Once the JavaScript code for detecting and making inferences that a tell-sign exist is developed, each tell-sign object must be packaged as a method/function. The packaged function can be either on the same file as the Tell-sign's Objects block or residing on a separate file. In the Tell-sign's Objects block each tell-sign function will be interconnected, as illustrated in figure 5.8. Thus, a set of input arguments, and an output data structure should be included.

Integrating the Tell-signs

Integrating the packaged tell-sign's function with the existing system is done by inserting the function's callee in the "Tell-sign's Objects" block. As seen in figure 5.8, each tell-sign will be mined sequentially with some standard input and outputs. Similarly, to remove a tell-sign from the system, the callee of the function must be from the Tell-sign's Objects block.

Some necessary arguments will be supplied to every tell-sign object, and some standard parts/methods within the returned data structure are required. Since every tell-sign object miner will require the entire set of JavaScript code, this input argument should be supplied to the tell-sign's object. If the mining process requires additional information this can also be requested optionally. The returned output of each tell-sign object will be compiled and returned as a single data structure. Therefore, the methods

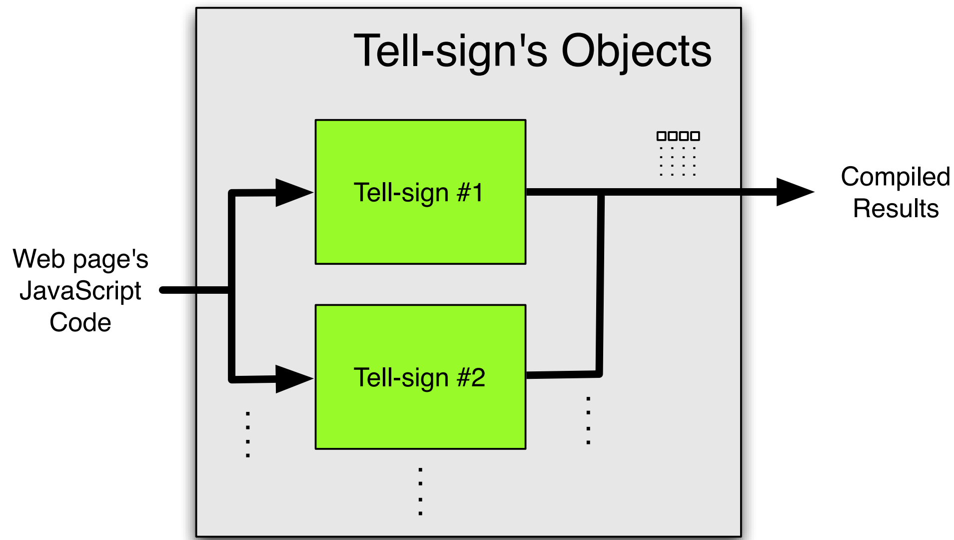


Figure 5.8: The architecture and connections of the Tell-sign's Object Block that is found in the overall WPS architecture as seen in figure 5.2.

Tell-sign	Location	Optional Outputs
[0]	[0]	[0]
[1]	[1]	[1]
:	:	:

Figure 5.9: Data structure to store the tell-signs found in the JavaScript code.

in the data structure object of the compiled results will have a similar structure as figure 5.9.

The compiled results returned from each tell-sign search should include the type of tell-sign with which the search was conducted, where in the JavaScript code the tell-sign exists, and if there are additional parameters that need to be returned with the results, these should be included as semi-structured data under optional outputs. In some cases, if the tell-sign search is dependent on the results of another tell-sign, then tell-signs can be interconnected. In these cases, the sequence of executing the tell-sign's objects must be dealt with carefully; the first to execute having the highest precedence. A skeleton of how the tell-sign detection method should be composed when scripting in JavaScript is illustrated as follows.

```
function findTellSignObject (JavaScriptCode)
{
      // Search for tell-sign method body

      // Return data structure
      return {type:Tell-sign,
              location:InstanceOfTellSignLocation,
              optional:AnyOptionalOutputs};
}
```

5.5 Evaluating the Profiler

The code profiler is an intermediate stage within WPS. However, it is a crucial phase to determine the success of the WIMWAT project as we have discussed. A set of tests designed to examine the accuracy of the profiler was conducted. This evaluation examines the default page of the top ten Websites selected from Alexa Top 500 Global Sites[2] on 28 February 2011. The selection process for the top ten websites selected uses a quota sampling method and it is governed by the three rules.

Rule 1: Exclude all sub-domains. For example, `google.com`, `google.co.jp`, and `google.de`, only `google.com` will be selected.

Rule 2: Exclude all adult related Websites due to ethical reasons.

Rule 3: Allow only functional Websites to be included in the list.

To determine the performance of the profiler, the results will be compared against the manual detection results using the same requisite, as well as a human with no prior experience with the profiler. Since techniques discussed in §5.2.3 can be employed to include external JavaScript code, Web pages may contain huge sets of JavaScript code when the entire set of code is downloaded; sometimes this may consist of more than ten thousand lines of code. In order to cope with the massive size of code during the manual detection, the concept of evaluating over a subset of code from the top ten Websites selected is suggested. This step was taken to reduce human fatigue and ensure the quality of the manual detection. To determine an appropriate size for the subset, the character sizes of the top thirty most popular Websites' default pages were

[2] `http://www.alexa.com/topsites`

	Websites	**Character Size**
1	google.com	206,505
2	facebook.com	43,320
3	youtube.com	75,712
4	yahoo.com	448,042
5	live.com	39,952
6	blogger.com	30,713
7	baidu.com	12,536
8	wikipedia.org	1,732
9	twitter.com	213,012
10	qq.com	186,074
11	msn.com	97,857
12	sina.com.cn	350,285
13	taobao.com	84,438
14	amazon.com	172,438
15	linkedin.com	276,885
16	bing.com	10,870
17	wordpress.com	151,622
18	yandex.ru	49,298
19	microsoft.com	182,068
20	ebay.com	251,274
21	163.com	115,011
22	mail.ru	240,966
23	paypal.com	224,502
24	fc2.com	257,115
25	flickr.com	1,882
26	apple.com	379,702
27	craigslist.org	101,572
28	imdb.com	156,433
29	sohu.com	680,537
30	bbc.co.uk	19,983
	Total	5,062,336.00
	Mean	**168,744.53**
	Variance	22,579,260,836.32
	Standard Deviation	**152,832.77**
	High	**321,577.30**
	Low	**15,911.76**

Table 5.2: Character sizes of the default page for the top 30 most popular Websites ranked by Alexa Top 500 Global Sites selected on 09 March 2011.

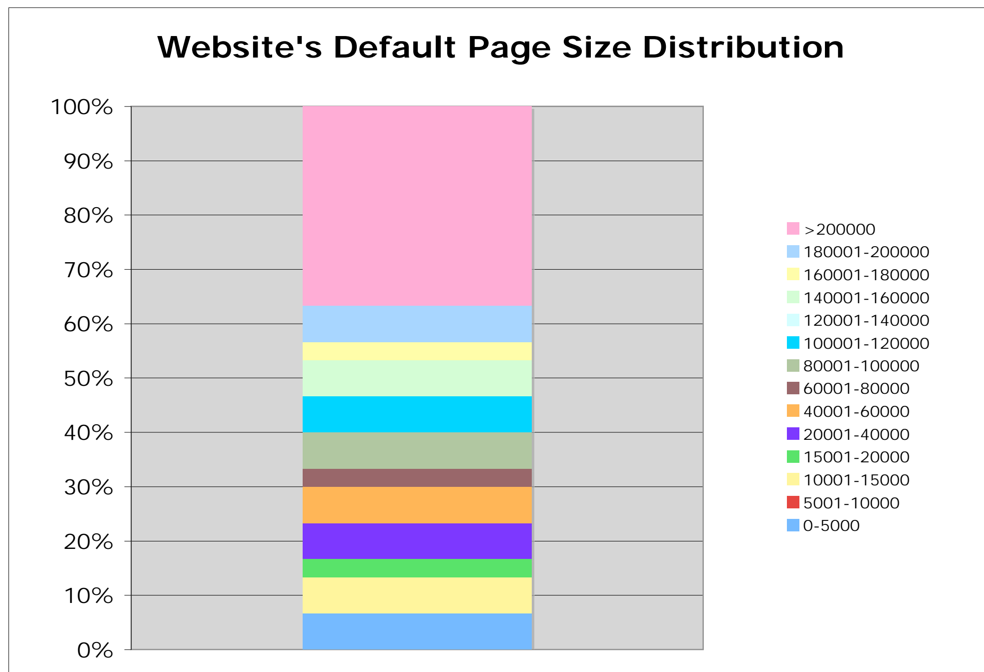


Figure 5.10: Website's default page character size distribution for the top 30 most popular Websites ranked by Alexa Top 500 Global Sites selected on 09 March 2011.

examined; see table 5.2. These Websites were selected from Alexa Top 500 Global Sites using the same three rules to govern the selection process.

Examining table 5.2, the character size of a Website from our samples ranged between 680,537 characters to 1,732 characters. It will be noticed from the compiled character size of the default pages of the top thirty Websites in figure 5.10 shows that more than 60% of Websites have default pages with character size larger than a hundred thousand characters. Eleven Websites, which constitute 36.67% of the top thirty Websites, have default pages containing more than two hundred thousand characters. Only about 20% of Websites have default pages with twenty thousand characters or less.

This investigation highlights the magnitude of the size of the JavaScript code that has to be examined manually. Thus, a manageable subset of JavaScript code for each Website default page is required. When selecting the size of the subset for the manual detection, it should be large enough to thoroughly evaluate the profiler, and get a

flavour of the coding style for each Website. Firstly, the computed mean of the samples is 168,744.53, and then to arrive at the lower limit of character size, the sample standard deviation of 152,832.77 was also computed. Using these values, a high of 321,577.30 and a low of 15,911.76 character size can be computed.

The subset character size to evaluate the profiler was chosen to be 20,000 characters– a round figure above the lower limit of 15,911.76 characters computed. Then, to give a flavour of how the different parts of Websites' code will affect the evaluation, a breakdown of the results for the subset into 10,000, 15,000 and 20,000 character sizes was done. This will illustrate the accuracy of the profiler at different parts of the code.

5.5.1 Profiler's Evaluation Setup

The interim evaluation setup enables the profiler to analyse the subset of the Web page's JavaScript code for instances that match the patterns described in §5.4.2. The findings are recorded and necessary inferences made through records found earlier in the profiling process. This cycle repeats itself as depicted in figure 5.3 until the whole process is completed.

In order to understand how the different sizes of characters will affect the code profiler performance, three sets of data were investigated. The first set evaluates the profiler with up to 15,000 characters; a figure very close to the lower limit computed. This investigation will allow us to have an insight into the performance of the profiler when the character size is close to the suggested lower limit. The second set does it over 10,000 characters and the third set over 20,000 characters. Both of these tests will investigate the effects of the character size when it is under or over the suggested amount.

Using these three sets of test results, we can examine if the character size has any effect on the profiler's performance. This evaluation will help to segregate the investigation process of WPS into phases, such that the earlier phases up to the profiler phase will be thoroughly tested. Then issues with future phases can be isolated and addressed without having to deal with the issues faced during this phase.

5.5.2 Profiler Evaluation's Results and Discussions

Results obtained by the profiler and manual detection are compiled and tabulated in table 5.3. This table lists the breakdown of items detected from the three tests obtained by the profiler and manual detection for comparison. A good result was achieved as

the manual and profiler detection results never differ.

The evaluation uses all the patterns detected and inferences discussed in §5.4.2 and the results obtained are grouped into five general groupings according to the item detection tested.

Methods This grouping consolidates the different patterns covered to declare a method-/function.

Objects This grouping consolidates the different patterns covered to declare an object.

Event Listeners Details of all instances of event listener declaration covered are placed together under this grouping.

Factory Event Listeners Using the profiled results from the event listeners detection and the function detection, the inference of a factory method is stored under this grouping.

Factory Callee From the identifier of the inferred factory event listeners detection, the callees of these functions detected are consolidated under this grouping.

In practice, using the set of code in listing 5.3 as an example, four methods will be identified on lines 1, 6, 12 and 25, one object on line 11 and an event listener on line 2. An instance of an event listener's factory method will be inferred on line 1 with the identifier called "setListen", and no instances of the event listener factory method's callee will be identified.

```
1  setListen = function (a, b, c) {
2      a.addEventListener ? a.addEventListener(b, c, j) :
3                      a.attachEvent(‘‘on” + b, c)
4  }; // function setListen ()
5
6  setUnlisten = function (a, b, c) {
7      a.removeEventListener ? a.removeEventListener(b, c, j) :
8                      a.detachEvent(‘‘on’’ + b, c)
9  }; // function setUnlisten()
10
11 var z = {},
12     A = function (a, b, c, d) {
13         var e = c === undefined ? h : c,
14             f = c === j,
```

```
15            g = b && b[0] === c;
16
17        if (a in z)
18        {
19            if (d === undefined)
20                d = j;
21
22            var l;
23
24            l = typeof d ==``function'' ? d :
25                          function (y) { return y === d };
26
27            for (var x = 0, n; n = z[a][x++];)
28            {
29                n = n.apply(i, b || []);
30
31                if (f)
32                        e = e || n;
33                else
34                {
35                    if (g)
36                        b[0] = n;
37
38                    e = n;
39
40                    if (l(e))
41                        return e
42                } // else
43            } // for
44        } // if
45        if (typeof d == ``function'')
46           return c;
47
48        return e
49    };  // function A()
```

Listing 5.3: Example JavaScript code

		Methods						Objects						Event Listeners						Factory Event Listeners						Factory Callee					
	Websites	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K
1	google.com	60	60	108	108	141	141	19	19	25	25	29	29	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	2	2
2	facebook.com	34	34	56	56	88	88	12	12	15	15	17	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	youtube.com	18	18	26	26	43	43	0	0	3	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	yahoo.com	38	38	71	71	93	93	12	12	19	19	21	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	live.com	0	0	0	0	0	0	1	1	2	2	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	blogger.com	2	2	4	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	baidu.com	69	69	86	86	86	86	6	6	6	6	6	6	3	3	3	3	3	3	2	2	2	2	2	2	8	8	13	13	13	13
8	wikipedia.org	8	8	8	8	8	8	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3
9	twitter.com	34	34	56	56	78	78	3	3	6	6	9	9	0	0	1	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1
10	qq.com	32	32	48	48	68	68	8	8	8	8	8	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.3: Compiled evaluation results breakdown for 10, 15 and 20 thousand characters comparison between the Code Profiler results (Auto) and manual detection results (Manual). The full extent of the results can be found in Appendix B, C, D.

In table 5.3, under each item detection's evaluation column, six sub-columns are aligned such that the profiler results (Auto) and the manual detection results (Manual) for the different character sizes can be compared. Using `google.com` as an example, the number of Methods/Functions detected when evaluating over 20 thousand (20K) characters for the profiler and manual detection, both results yielded 141 instances. Moreover, the profiler identified more instances of Methods/Functions in the 20 thousand (Auto 20K) characters test than the 15 thousand (Auto 15K) characters test. The full extent of the profiler's evaluation results can be found in Appendix B, C, D for tests conducted with 10 thousand, 15 thousand and 20 thousand character size respectively.

Examining the Methods column in table 5.3, it can be seen that the profiler has managed to identify as many items as the manual detection. This trend continues for all components evaluated except the Event Listeners test. As the character size increases, more items are also identified and no signs of error detection were noticed for the Methods, Objects and Factory Callee tests.

Since only part of the Website's JavaScript source code is evaluated for most of these sites, these results do not reflect the true results of the individual sites. The purpose of the evaluation is to check the accuracy and integrity of the profiler, so that problems with the other blocks of WPS can be isolated. However, due to the relatively small size of `wikipedia.org` JavaScript code–1,732 characters as shown in table 5.2, the entire Website default page's JavaScript source code is evaluated. This is because even for the 10 thousand (10K) character size evaluation, this is larger than the entire `wikipedia.org` default page's JavaScript character size. Thus, the results remain the same throughout all the five items detection tests.

The results from the profiler and the manual detection returned very similar values as seen in table 5.3, and the overall computed values for "Total items mined" column in table 5.4. These figures provide a good coarse indication to highlight issues pertaining to the profiler's detection method. However, they are not fine-grain enough to pick up the specific problems.

Another parameter is used to check the integrity of the results between the profiler and manual detection results. This test is known as the similarity score where the percentage difference between the correct profiler results and the actual results is identified by manual detection. The computational process for the similarity score illustrated in equation 5.1 first sums both the correct items detected by the profiler and actual items identified by the manual detection individually, before computing the percentage difference. Since every result's item will have its distinct features ($ItemsFeatures$) that

differ them from the previous, the summation processes of the correct and actual results follow the same steps to compute the individual features for their respective results. Using the test, we will be able to capture the depth of the profiler's accuracy by examining the detailed features of each result.

$$Similarity = \frac{\sum_n (CorrectItems[n] + \sum_i CorrectItemsFeatures[n][i])}{\sum_n (ActualItems[n] + \sum_i ActualItemsFeatures[n][i])} \tag{5.1}$$

The similarity test is a value between zero and one that checks for the soundness of every item picked up by the profiler. A point is given for every item found and for every correct feature belonging to that item. Using this scoring system, comparing the summed score achieved by each item will give the accuracy of the detection. For example, when verifying the Methods detection results, every item identified by the profiler will consist of five features: the Method's identifier, the Method's location, the Method body's starting location, the Method body's ending location, and the input arguments to the Method. Thus, if a Method is detected correctly, it will score 6 points. Every Object detected will consist of four features: the Object's identifier, the location where it is spotted, the beginning and ending location of the Object's body. A total of 5 points will be awarded if the Object is detected correctly.

The Factory Event Listeners detection has a transitive relationship to the latter Factory Callee detection, and the former Event Listeners detection. Thus, the Event Listeners detection test will include four features: its location, the element's name that the listener is monitoring, the handling function, and the triggering event. The Factory Event Listeners detection test consists of two features: the factory method's identifier and the factory method's location, and the Factory Callee detection test consists only of the locations of where the factory's callee(s) is/are found.

Using listing 5.3 to illustrate the similarity scoring computation for Methods evaluation, the first method item recorded on line 1 will be inspected. The features of the items will be checked. It will score a point for identifying it, another point for the correct location found, 2 points for correct start and end locations respectively, and 3 points for correctly identifying the three input arguments. A total of 7 points will be awarded to this record if all features are detected correctly. This is an iterative process that repeats itself until all the items detected by the profiler are computed.

Concurrently, another actual score similar to the correct score is also computed. The actual score sums all the items and their features, whether they are correct or not,

then as seen in equation 5.1, it divides the correct score by the actual score to compute the similarity score for each Website for each set of evaluations as seen in table 5.4.

		Total items mined						**Similarity**		
	Websites	Auto 10K	Manual 10K	Auto 15K	Manual 15K	Auto 20K	Manual 20K	10K	15K	20K
1	google.com	81	81	136	136	174	174	1.0	1.0	1.0
2	facebook.com	46	46	71	71	105	105	1.0	1.0	1.0
3	youtube.com	18	18	29	29	46	46	1.0	1.0	1.0
4	yahoo.com	50	50	90	90	114	114	1.0	1.0	1.0
5	live.com	1	1	2	2	3	3	1.0	1.0	1.0
6	blogger.com	2	2	4	4	4	4	1.0	1.0	1.0
7	baidu.com	88	88	110	110	110	110	1.0	1.0	1.0
8	wikipedia.org	13	13	13	13	13	13	1.0	1.0	1.0
9	twitter.com	37	37	65	65	90	90	1.0	1.0	1.0
10	qq.com	40	40	56	56	76	76	1.0	1.0	1.0

Table 5.4: Overall evaluation results for 10, 15 and 20 thousand characters comparison between the Code Profiler results (Auto) and manual detection results (Manual), as well as the similarity scoring between both sets of results.

Table 5.4 list the computed Similarity score for the different sets of character size evaluated for the correct results between the profiler and manual detection, and the actual results collected by the manual detection for each Website. For this evaluation the similarity score for all tests scored 1.0. These results demonstrated that the profiler achieved a 100% detection accuracy of what it was set up to do.

The total items of mined results listed in table 5.4 demonstrated that more items were discovered when the sample character size increased. This result is expected for Websites that has JavaScript code with character size larger than the evaluated sample character size. In the case of `wikipedia.org`, because even the smallest evaluated character size is larger than the Website's JavaScript code character size, it returned the same results constantly. These results prove that the profiler is a reliable platform for conducting the next investigation phase of our widget identification research, inferring the tell-signs from the profiler's results.

The optimal character size sufficient for evaluating Web pages depends on the type of the Websites chosen for the evaluation. In our case, although the suggested size is

16 thousand characters, table 5.4 results demonstrated that the variance of 30% has no impact on our evaluation results. The results demonstrated that using the standard deviation method to arrive at the lowest amount of characters for evaluation is adequate for this type of investigation.

5.6 Techniques to Predict Widgets

In this section, the techniques applied to predict all seven widgets in the 'Widget Inferences' block ④ figure 5.2 are presented. A confidence percentage is awarded for each widget inferred in the page. Using this value, inferred widgets that do not have all the components available can also be included in the inference process. Then, the widget inference process will use the confidence percentage to predict widgets that are partially available in the source code. The rationale behind the confidence percentages is to provide a weighting to measure how confident the system is when predicting a candidate for a type of widget. This parameter not only allows fine-tuning to the approach, but also lays the foundation for incorporating machine learning algorithms into our approach in the future.

Every component and tell-sign is awarded a confidence percentage when it is inferred or found. This percentage will be computed as part of the confidence percentage for the widget. To demonstrate how these concepts can be applied, a simple approach to distribute the percentages evenly was used. However, often the tell-sign may require more than one clue to determine its existence. In such a situation, the percentage awarded to the tell-sign is further evenly distributed among the key clues that determine the tell-signs. Since developers can apply numerous coding styles to deliver a concept, the similar process of evenly distributing the percentage is repeated for sub-processes concepts. In cases where no clues for a tell-sign are found, keywords are used to provide clues. Since keywords are language dependent and they are not a concrete method to assume that the concepts of a clue exist, a small amount of percentage is awarded – starting with 10%. Depending on the concepts of the clue the keyword determines, the percentages are evenly distributed and rounded down, as the concept is sub-divided. This approach of distributing the percentages may not be the best approach, but since this is only a demonstration of our framework concepts, this basic approach is used. Future work could focus on the possibility of implementing a feedback loop, as well as Machine Learning techniques, to improve the accuracy when using the WPF approach by modulating the Confidence Percentages.

5.6.1 Ticker Widget Tell-Signs

The components and processes that dominate the classification of a Ticker widget are the timer and the ability to automatically display the next set of content after each interval, as described in §3.3.11. Due to the issues relating to Window Timer (see §5.1.3), techniques to trace for Window Timers will be required before a Ticker widget can be determined, or other techniques will be required to spot the areas in the Web page where they change constantly.

Ideally, as described in listing 3.23, the main component and processes to determine a Ticker widget require tell-signs such as `WindowTiming`, `GoToStartOfList`, `CheckEndOfList` and `Increment` to exist. Conversely, tracing for Window Timers in the source code can be difficult, consume a lot of computing power, can take a very long time to sniff through the code and issues such as those raised in §5.1 and §5.2, can all contribute to the failure of detecting this component. Therefore, we approach this issue by determining the areas in the Web page where it changes constantly, and examine these areas for clues that exhibit signs of a Ticker widget.

After WPS is activated and the page is loaded, a Listener is injected to monitor the change in DOM tree modification. At this stage two scenarios can happen. The first scenario is when the DOM tree mutates; this event will trigger the Ticker widget detection process to begin. The second scenario is if nothing happens. In order to free WPS from endlessly monitoring the DOM for a change, a 15 seconds Window Timer is also injected to clear the event monitoring process if nothing happens after 15 seconds, and make a decision whether any Ticker widget exists in the page.

Conceptually, the Ticker widget is composed of 3 components: display window, triggering mechanism and content manipulation mechanism. Thus, if each of these is found, 30% confidence will be awarded while leaving 10% confidence for keywords detected. When the Ticker widget detection process is triggered, the content in elements that have changed will be considered as the candidate elements, and a 40% confidence will be awarded to them. Checks for the type of content within the candidate element's opening and closing tags will be conducted on every suspected candidate. The 40% awarded is composed of the detection of the Display window (30%) and the detection of linking the Triggering mechanism to the Display window (10%). The Trigger mechanism is usually a Window timer with a handler that points to a set of code. However, in this case only a linkage to a Window timer can be identified. The actual instance of the Window timer (10%) and triggering factor (10%) has not been determined. Since a Ticker widget's content is commonly text based, we check

the content of the element for multimedia objects. Tag names such as `img`, `applet`, `embed`, `object`, `param` are considered as multimedia objects. If none of these tags are found 30% confidence is awarded to the candidate element.

If no multimedia objects are found within the element, keywords such as 'ticker', 'marquee', 'scroller' or 'flasher' will also be searched to improve the detection confidence. When any of these words are found, 5% confidence will be awarded. Since multimedia objects are partial concepts of an increment tell-sign, the existence of these keywords is only awarded half of the full 10% awarded to keywords detected.

In the case where the `<marquee>` element is found within the page, this element will be given the highest detection confidence (100%) since this type of element was originally created for scrolling content across the page. This detection process is included to cover Web pages that are scripted using popular, but non-standard HTML elements.

5.6.2 Popup Content Widget Tell-Signs

Visually, the Popup Content and Collapsible Panel widget are different widgets, as seen in figures 3.14 and 3.13 respectively. However, developing these types of widget is very similar in concepts, and our detection methods for them are very similar as well.

In this section the processes used to detect the Popup Content widget are presented. Figure 5.11 illustrates the process flow and decisions taken to detect the Popup Content widget. Note that the process flow presented is conducted to all candidate elements identified during the detection process.

Conceptually, the Popup Content widget consists of 4 main components with Confidence Percentages distributed evenly. These components include a Display Window (25% confidence), a button (25% confidence), a triggering mechanism (25% confidence) and the visual effect to make it float (25% confidence). However, if either of the UI components: Display Window and button, are not found, 10% confidence is awarded for keywords that gave clues of a button, and 10% confidence is awarded for keywords that gave clues of a floating Display window.

When an event listener is created, most major Web browsers often record the creation of the listener's instance along with its related details. We can search in the interfaces component management to examine all the recorded event listeners created. In Firefox, this service is provided by including the following,

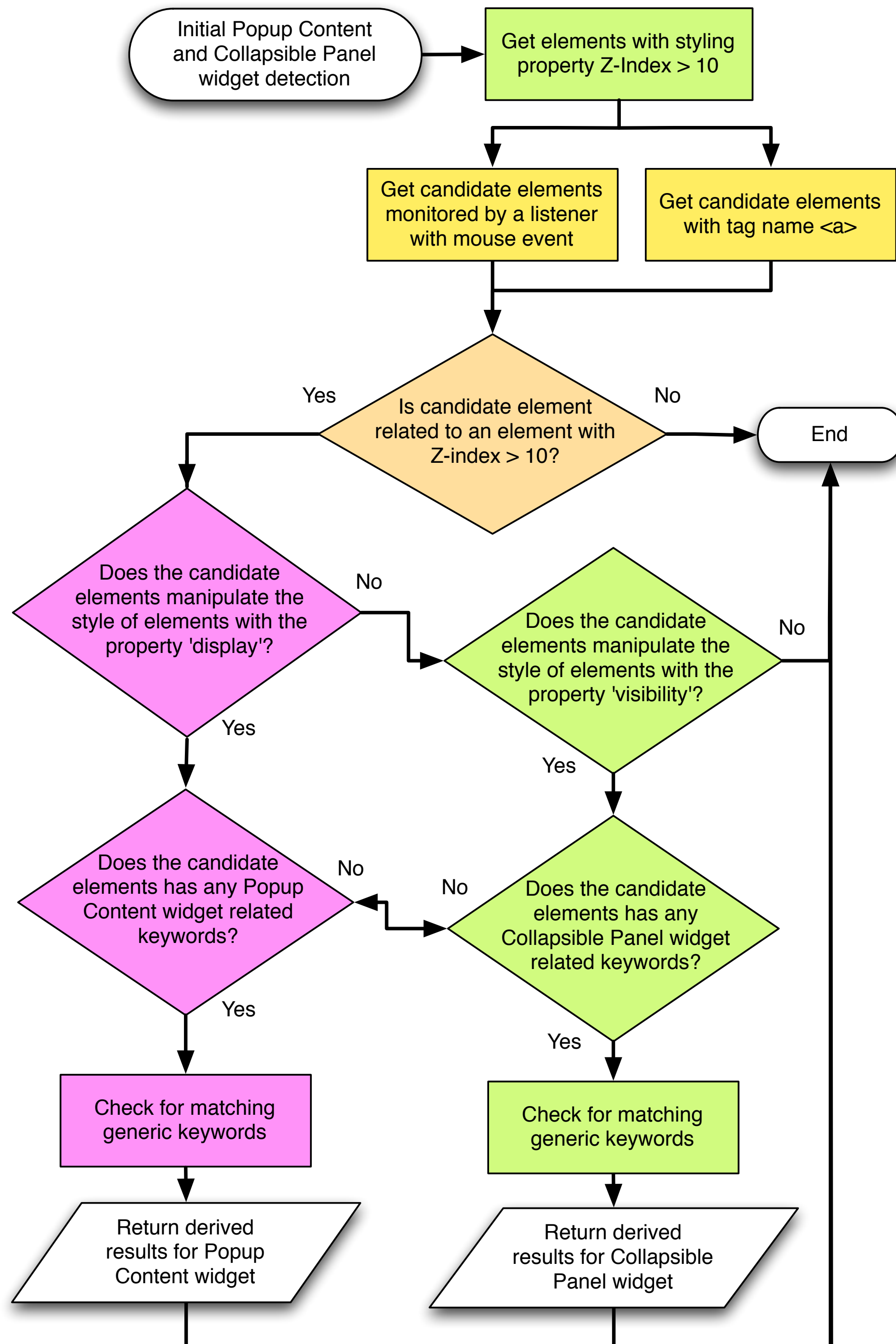


Figure 5.11: The final WPS Popup Content and Collapsible Panel widget prediction process flow for each candidate.

```
Components.classes["@mozilla.org/eventlistenerservice;1"].
    getService(Components.interfaces.nsIEventListenerService)
```

This service records the details about the event listeners and scripts relating to the element. However, not all details are always recorded well. Elements with tag name Anchor `<a>` or an element that is monitored by a listener for mouse events such as 'onclick', 'onmouseover' and 'onmouseout' are considered as candidate elements at the start of the process. These elements are then analysed if any neighbouring elements have styling property `z-index` set to more than 10. The confidence percentages awarded for these clues found are: 10% confidence for qualifying to be a candidate element (`UserTriggerObject` component), and 20% confidence for styling property `z-index` is more than 10, and will meet the `SetDOMElementOrderHigh` tell-sign requirements.

The styling property `z-index` allows the developers to work with different visual layers overlaid on top of each other - where the larger the number the closer the lay is to the user, while the smaller the number the lower the layer is. To extract the `z-index` values of the element, because this value is not always available immediately after the page is loaded, the following regular expression is employed to search for patterns that the `z-index` property is assigned and extract its value.

```
z\-index\s*:\s*[1-9][0-9]+[;\}\s]{1}
```

To avoid the complication when detecting whether developers employ remote scripting techniques to deliver the content, our technique instead analyses the envelope elements for clues. This is done through analysing the candidate elements for scripting code that manipulate the `visibility` styling property value to either `none`, `block` or `inline-block`, so that the element will be visible to the user or not. When found, this evidence will be awarded with 20% confidence.

Since WAI-ARIA 1.0 became a candidate recommendation in January 2011 [Craig et al., 2009], on rare occasions, anecdotal evidence highlights WAI-ARIA attributes are applied to Websites. For developers to be able to apply these techniques, the developer's intention to develop a piece of code is presented in the code. Thus, for Web pages that use WAI-ARIA, the candidate elements with WAI-ARIA attribute `aria-haspopup` are given an additional 30% confidence. This is because it shows that the developer intends this set of code to popup some content.

The remaining confidence percentage is added up via keywords for the different components, generic keywords for Popup Content and Collapsible Panel widgets, and

more specifically for a Popup Content widget. Because this evidence is language dependent and does not necessarily translate to factual evidence of a tell-sign, the percentages for each of these instances are much lower.

5.6.3 Collapsible Panel Widget Tell-Signs

Using the definition of WIO, listing 3.8 shows the components that form the Collapsible Panel widget. Similar to the Popup Content widget, elements Anchor `<a>` tag name or an element that is monitored for mouse events such as ‘onclick’, ‘onmouseover’ and ‘onmouseout’ are included in the list of candidate elements for further examination.

As seen in figure 5.11, most of the processes are similar to the Popup Content widget, except this time candidate elements that have processes in the code that manipulate the `display` styling property to affect the changes to the appearance of content are searched. This is because the `display` styling property, unlike the `visibility` styling property, will remove the space within the page instead of only hiding the content in the element.

The confidence percentages for each of these pieces of evidence are awarded on the same scale with the Popup Content widget since the detection processes are quite similar. Thus, similar to the Popup Content widget, the remaining confidence percentage is added up via keywords for the different components, generic keywords for Popup Content and Collapsible Panel widgets, and more specifically for Collapsible Panel widget. Since this evidence is language dependent and does not necessarily translate to factual evidence of a tell-sign, the percentages for each of these instances are much lower.

The keywords used to enhance the Display Window detection are mainly divided into two types. The first type is generic to both the Popup Content and Collapsible Panel widget. These keywords include ‘panel’ or ‘content’, and when found 2% confidence is awarded for either of these words found. This is because these keywords only partially provide clues that the Display Window for the Collapsible Panel widget exists. The second type is specific keywords to Collapsible Panel widget. These words, which include ‘offsetHeight’ or ‘offsetWidth’, are awarded 5% confidence, and if words like ‘expand’ or ‘shrink’ or ‘contract’ or ‘enlarge’ or ‘collapsible’ or ‘drawer’ are also found, they will be awarded 10%.

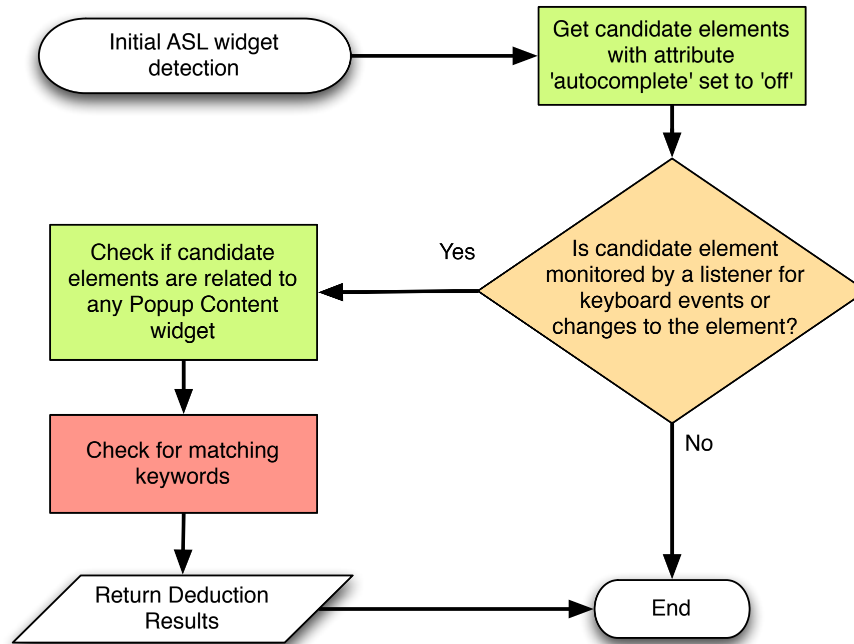


Figure 5.12: The final WPS Auto Suggest List (ASL) widget prediction process flow for each candidate.

5.6.4 Auto Suggest List (ASL) Widget Tell-Signs

The Auto Suggest List (ASL) widget is considered to be a simpler type of widget which was used in our feasibility study in chapter 4. However, challengers with false positive detection were highlighted in the study. In this section, the techniques applied to our detection methods are presented, and refinements to the previous techniques used in our feasibility study are discussed.

Figure 5.12 provides a visual illustration for the process flow of the detection methods employed in detection of the ASL widget. The ASL widget consists of 2 main components with equal weighting: the triggering mechanism (50%) and the Display window (50%). The triggering mechanism is composed of 2 parts: the triggering element and a visual mechanism to make the Display Window float. To recommend bespoke suggestions from the content entered by the user, a common practice is to prepare the text field by turning off the `autocomplete` attribute. Thus, elements with attribute `autocomplete` set to 'off' are considered as candidate elements for further analysis. If instances of the pattern `autocomplete` set to 'off' are found, 25% confidence is awarded to the candidate element.

Then the candidate elements are checked if they are monitored for changes or keyboard events such as `keypress`, `keyup`, `keydown`. Candidate elements with this event monitored will be awarded 30% confidence towards their overall confidence percentage.

Checking for a list that will contain the suggested items was found to be more deceptive than originally thought, since WPS will be activated immediately after the page is visually loaded, and before the user will interact with the page. Thus, the list of suggested items will be either empty or the list elements will not be available at that time. This challenged us to seek for other alternatives such as looking for the Popup Content widget identified in the neighbouring elements and keywords that provide clues that such an instance may exist. Indeed, searching for instances of manipulating the table/list is a key process in ASL widgets, but the purpose of this task normally updates the suggested content presented in a Popup Content widget or a Collapsible Panel widget. Using the concepts of detecting Popup Content and Collapsible Panel that has resulted from a change event from a text field, can also achieve the same objectives. In this case, detecting the conceptual processes rather than the physical code processes is employed to refine our detection.

If a candidate element is related to the records identified by the Popup Content and Collapsible Panel widget detection, then a further 25% confidence is awarded. The records (candidate elements) from the Popup Content and Collapsible Panel widget detection, used for the comparison, do not need to meet the threshold confidence percentage assigned for these widgets. This is because the suggested list presented in the ASL widget is often wrapped within a Popup Content and Collapsible Panel widget. However, it will not fulfil all the requirements of either a Popup Content or Collapsible Panel widget.

5.6.5 Tabs Widget Tell-Signs

Commonly, the technical concepts of the Tabs widgets are formed by two list sets. The first list displays the available tabs, while the other list contains the content of each tab. As described in listing 3.19, the concepts of three main components are used to define the Tabs widget: Display Window (normally this is the list that contains the content of each tab), Hide Content process, and User Trigger Object (list containing the available tabs). These tabs are awarded 30% confidence each. When manually coding these concepts, additional steps are required to materialise them. Figure 5.13 illustrates the process flow and decisions to detect the Tabs widget.

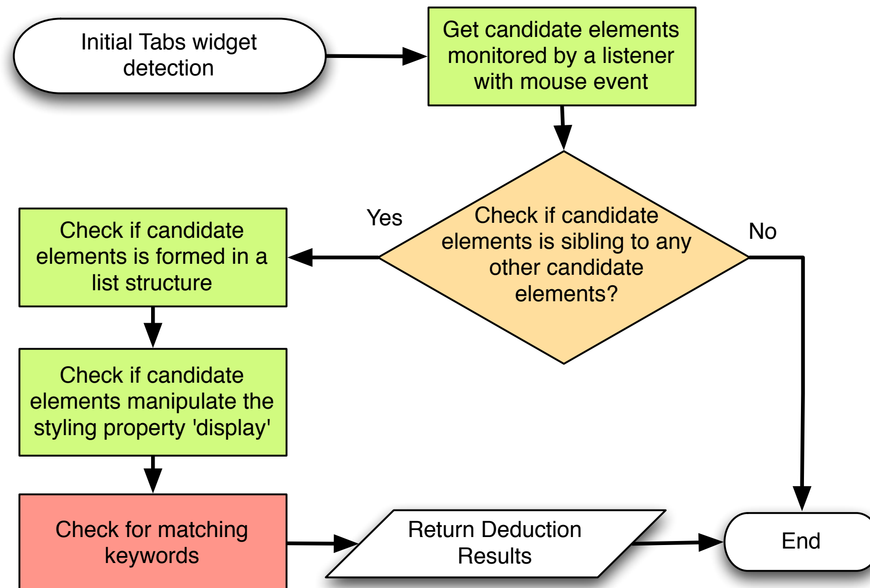


Figure 5.13: The final WPS Tabs widget prediction process flow for each candidate.

The process begins with identifying the elements with either Anchor `<a>` tag name, or elements that are monitored by listeners for mouse events such as `onclick` and `onmouseover`. These are traits that provide clues that the list displaying the available tabs may be present. All candidate elements identified as the `UserTriggerObject` component are initially awarded 10% confidence for associating themselves with traits that may react to certain processes when mouse events are applied to them. However, to determine it is the list that displays the available tabs, a list structure (10% confidence), and a linkage to another list (10% confidence) is required.

Usually a Tab widget will consist of more than one navigation link to different sets of content. Therefore, a check is put in place to examine other candidate elements if they are related to the existing candidate element via a list structure, or if they are in the neighbourhood up to 3 degrees separation. For either of the instances found, a 10% confidence will be awarded to the candidate elements for each relative found, and up to a maximum of 2 relatives will be awarded.

Since the Tabs widget often reuses the same space within the page to present the content of a different tab, investigation to search for clues of these candidate elements uses scripting code to manipulate the `display` styling property. This detection technique was employed because it does not matter whether the developer preloads the content or loads the content on the fly via remote scripting methods. In this case we

are examining the container of the tab content instead. If these types of instances are found, 10% confidence will be awarded and the HideDOMElements tell-sign.

For developers who comply to WAI-ARIA guidelines and provide the tabindex attribute, 20% confidence will be awarded to the candidate elements. A further 30% confidence will be awarded if the elements are arranged in a list structure. The following regular expression is employed to detect for patterns that exhibit this form of structure.

```
\/(ul|ol|dl)(\[.*?\])?\/(li|dt|dd)(\[.*?\])?(\/a)?(\[.*?\])?$
```

This regular expression includes a number of list formations such as <ul>, <ol> and dl, where the Anchor <a> tag can be present or absent within the list during the detection.

Once again, keywords are employed in the Tabs widget detection to enhance the detection process. As we mentioned before, because keywords are language dependent and do not necessarily reflect the true concept of the developers, a lower percentage is given for each test that is found. Here words such as 'tab', 'nav', 'folder', 'panel', 'section', 'selected' will be awarded with 10% confidence when this test is true.

5.6.6 Carousel and Slide Show Widgets Tell-Signs

Both the Carousel and Slide Show widgets are conceptually and visually similar, and they are often confused with each other. Thus, we will discuss the detection for these two widgets together. In this section, we present the processes and decisions to detect the two types of widget. Figure 5.14, illustrate the overall process flow and decision to detect and distinguish between the two types of widget.

The Carousel and Slide Show widgets are very similar widgets as defined in §3.3.4 and §3.3.9. Thus, we will discuss both of these widgets together, and point out their differences in this section. Conceptually, these widgets are composed of 4 main physical components: Display window (20% confidence), Next button (20% confidence), Back button (20% confidence) and Display Pointer (20% confidence). The process of looping or terminating the list of content is awarded 20%, because it is a main process component to distinguish between the Slide Show and Carousel widgets. Notice that the weighting of the Confidence percentage is distributed evenly among the 5 main components again.

At the start of the detection process, elements with Anchor <a> tag names and elements that are monitored for mouse events such as 'onclick' and 'onmouseover', will

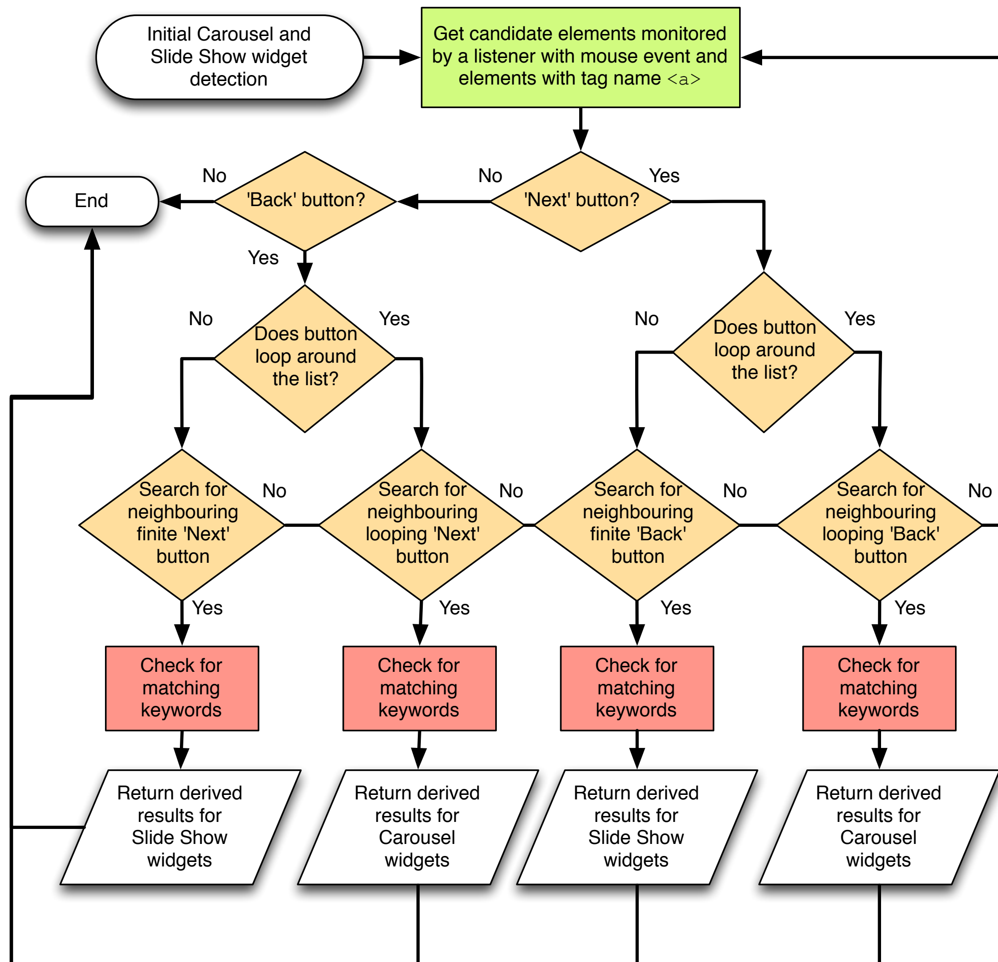


Figure 5.14: The final WPS Carousel and Slide Show widget overall prediction process flow for each candidate.

be selected as candidate elements for closer examination. Then, two tests to identify which candidate elements are ‘Next’ or ‘Back’ (‘Previous’ button) buttons. Combining candidate elements that are related, and one detected as the ‘Next’ button, and the other as the ‘Back’ button, we can begin filtering the buttons that are finite or looping together. Through this evidence, `Loop_NextButton`, `Loop_PreviousButton`, `Finite_NextButton` and `Finite_PreviousButton` components can be discovered from the source code.

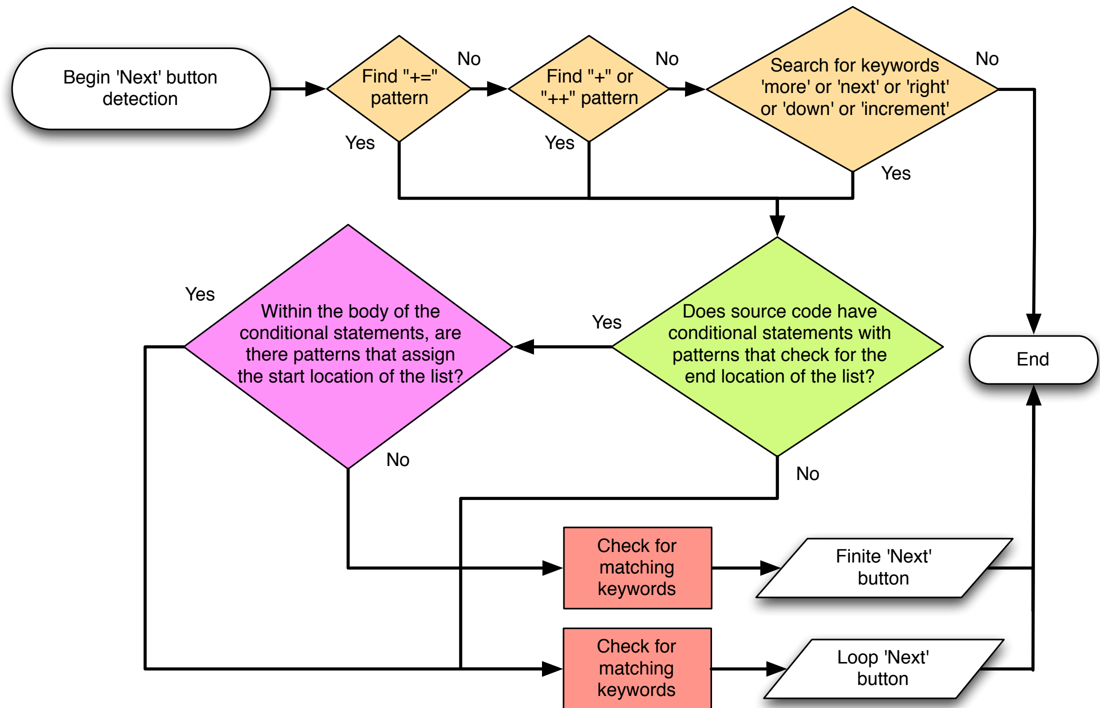


Figure 5.15: The final WPS Carousel and SlideShow widget 'Next' button prediction process flow for each candidate.

Next Buttons

Described in listing 3.4 and 3.17, both types of widget have controls that consist of very similar features. However, in this section, the 'Next' button feature will be discussed in more depth using figure 5.15 to illustrate the process flow and decisions for determining whether the element is a 'Next' button.

The related source code of the candidate elements are examined to determine if += or + or ++ patterns exist at the beginning of this leg of the analysis. When either of these patterns is found, a 20% confidence is added. However, if none of the patterns are found, keywords 'more', 'next', 'right', 'down' and 'Increment' are searched. If either of the previous checks is true, then the element is considered as a candidate element.

Next, the related source code of the candidate elements is examined for patterns that conditional statements are included by developers to check if the widget has reached the end of the list. Since the 'Next' button feature allows the user to transverse forward in the list, the system searches for clues that the Display Pointer is redirected to the start of the list of content. The following regular expression is used to search for

a conditional statements pattern that checks if the widget has arrived at the end of the list. This regular expression includes conditional operations ==, ===, >, >=, < and <=.

```
\([^\{;]+?(((\={2,3}|>|>\=)\s*\.*?(length|end|last|finish|max))|
      ((length|end|last|finish|max)\s*(\={2,3}|<|<\=)))
```

If the conditional statement pattern is found, 10% confidence is added and the `CheckEndOfList` tell-sign is determined. Since these clues are only evidence of half of the looping process, the encapsulated code within the body of the conditional statements is examined for patterns that the developer has programmed the assignment to the start of the list. Commonly this pattern will be found for looping buttons. For this examination, it will consist of two tests, the first consisting of the following regular expression.

```
\=.*?(start|begin|0|1)[\s\n;\)\]]
```

When found, another 10% confidence will be awarded. Otherwise, if the second test for the `loop` keyword is found, it will be awarded 10% confidence and the `GoToStartOfList` tell-sign is derived. If none of these tests pass, the button will be assumed to be finite and a 5% confidence will be added, and the `GoToEndofList` tell-sign will be assumed to be found.

Whether either of the checks is found or not, a check to match keywords that may resemble a finite or looping ‘Next’ button is conducted to improve the detection confidence percentage. The searching for keywords also helps candidates that do not have sufficient evidence to derive from any of the tell-signs. Through these little clues, available from the source code, when possible assumptions can be made.

Back/Previous Buttons

The ‘Back’ or ‘Previous’ button is similar to the ‘Next’ button except it is conceptually opposite. In this section, the ‘Back’ button feature will be discussed in more depth using figure 5.16 to illustrate the process flow and decisions for determining whether the element is a ‘Back’ button.

To begin, this time the following patterns `-=` and `-` and `--` are used for the initial pattern searches instead. Then, the keywords search to determine the ‘Back’ or ‘Previous’ buttons employs words like ’less’, ’prev’, ’left’, ’back’, ’up’ and ’decrement’.

Conditional statements to check if the widget has reached the start of the list are searched for this time. This search is done using the following regular expression that includes conditional operations such as ==, ===, >, >=, < and <=.

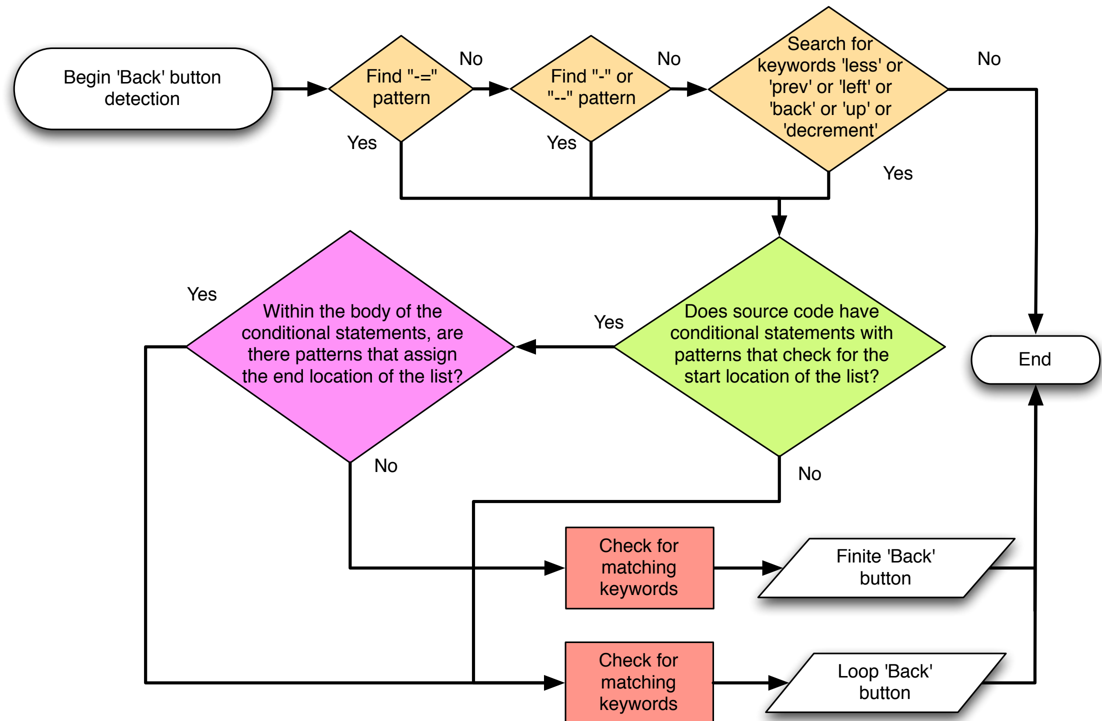


Figure 5.16: The final WPS Carousel and SlideShow widget 'Back' or 'Previous' button prediction process flow for each candidate.

```
\([^\{;]+?(((\={2,3}|<|<\=)\s*\.*?(start|begin|first|min|0|1))|
    ((start|begin|first|min|0|1)\s*(\={2,3}|>|>\=)))
```

When found, 10% confidence will be awarded. Since these clues are only evidence of half of the looping process, the system searches for clues that the Display Pointer is redirected to the start of the list of content. Next, the body of the conditional statements is examined to find clues that assignation to the end of the list for a loop button is found. Two tests are used for analysing this task, with the following regular expression employed for the first test.

```
\=.*?(end|last|length|max)[\s\n;\)\]]
```

When the pattern is found, 20% confidence will be awarded, while – if the second test for the `loop` keyword is true – it will be awarded 10% confidence, and the `GoToEndOfList` tell-sign is derived. If none of these tests pass, the button will be assumed to be finite and a 5% confidence will be added, and the `GoToStartofList` tell-sign will be assumed to be found.

Finally, all candidate elements will be checked for keywords that may resemble a finite or looping ‘Back’ or ‘Previous’ button. The searching for keywords also helps candidates that do not have sufficient evidence to derive from any of the tell-signs. Through these little clues available from the source code, when possible, assumptions can be made.

5.6.7 Limitations

With every approach there are strengths and weaknesses. In this section, the limitations of the prediction methods and approaches chosen are discussed. Often keywords are applied to improve the different widget prediction approaches. Using keywords is not the core method in an approach; they are included to improve prediction rates in cases where instances of tell-signs are not available. We acknowledge that this method is language dependent, but the terms used are generic to the widget concepts, and anecdotal evidence suggests that these terms are used widely across most languages. Further research can be done to analyse the impact of this method on the prediction approach.

5.7 Summary

Expanding our approach is not direct due to the additional types of widget and the diverse nature of them. In this chapter, we have covered the techniques and refinements applied to solidify the detection concepts of the different types of widget. Challenges with analysing the Web page source code, and the techniques introduced to overcome these issues, as well as the issues raised in chapter 4 – Feasibility Investigation for Using Tell-Signs are discussed. The final set of tell-signs private to all seven types of widget, and common tell-signs shared between the different types of widget, can be found in Appendix E. Furthermore, a confidence percentage for each widget was introduced to aid the prediction. The percentage will indicate how confident WPS is when predicting widgets.

Future research could explore other techniques to distribute the confidence percentages. An avenue could investigate how these values will affect the detection rates. Finally, an evaluation of our approach is presented in the next chapter to test and expose the spectrum of the prediction coverage and its weaknesses.

Chapter 6

Widget Prediction Evaluation

In this chapter, the approaches applied in WPS are evaluated in the broader Web, where Websites selected from Alexa's Global Top 500 sites on the Web[1] list were used in our evaluation. The selection methodology for our evaluating data set and the evaluation methodology are also presented.

The approaches presented in chapter 3 provide a mode to model and classify different types of widgets. These concepts are then applied to our widget prediction approach by identifying the instances or code constructs of tell-signs from the Web page source code. Using a combination of tell-signs, the components for a type of widget can be derived from this evidence, which will be conceptually deducible in the "Widget Inferences" block (see figure 5.2) to predict the widget. In chapter 5, the technical details of our approach and the techniques employed were discussed, while fine tuning of the methods that were highlighted from the issues raised in chapter 4 were also covered.

Due to the vast variation of possible routes a developer can take to develop a widget, we will not be able to cover every style and method. Only the methods used for developing the different types of widget discussed in chapters 3 and 5 will be examined in our evaluation. This is because the purpose of this evaluation is to demonstrate the proof-of-concepts of our approach.

The aim of the evaluation is to examine the possibility of predicting widgets from the Web page source code using the approaches proposed (see chapters 3 and 5). More

[1] Alexa's Global top 500 sites on the web - `http://www.alexa.com/topsites`. Last accessed 21 September 2012.

analysis of the results was also conducted and presented to expose the qualitative aspects of our approach. These investigations aim to answer the research questions proposed in §1.3, as well as to contribute an extensive understanding of the subject matter for future research.

6.1 Evaluation Setup

Setting up the evaluation through to the evaluation phase consists of a few stages. The different phases of preparing the evaluation are discussed in this section. The evaluation preparation consists of three parts: the data set used to evaluate our approach, the setup of our automated prediction system – WPS – and the manual analysis setup to provide a benchmark for our evaluation.

6.1.1 Data Collection

Selecting the data corpus is one of the most important elements to ensure the repeatability of the experiment and the thoroughness of the evaluation. We use the list provided by Alexa's Global Top 500 sites on the Web to select the Websites for our data corpus. This list was chosen because, from previous studies conducted, we found that the top Websites listed provide us with a good reflection of the broader Web [Harper and Chen, 2012]. Fifty Websites were selected from the list of five hundred Websites. The first Website in the list is chosen, then the tenth, and then a Website at every interval of ten selected repeatedly until fifty Websites were chosen.

Using this method, a broad spectrum of Websites at different levels of popularity can be included in our study. Websites like `Google` that have many regional sites to customise their user experience for the country. Due to their popularity across the world, even the regional Websites often make it into the top 500 sites globally and some of these sites may be included in our data corpus due to our selection methodology.

Fortunately, only regional Websites from `Google` were selected using our methodology on Friday, 13 July 2012. Although similar in nature, Web pages from these Websites do vary depending on the culture and language of the region. Occasionally, additional features are provided by these Websites to improve the usability of the pages for the region because of the language and culture. Thus, examining these Websites will provide a different degree of thoroughness in our evaluation, by evaluating the generalisability of our approach across different languages.

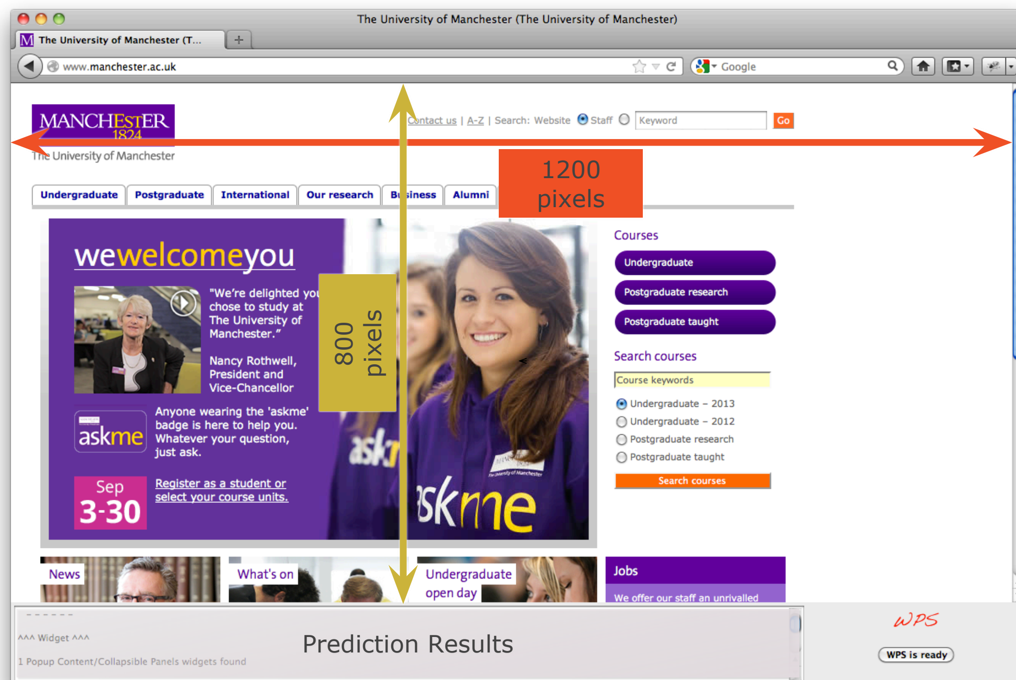


Figure 6.1: The setup of WPS as an Add-On in Mozilla's FireFox Web browser.

Once the Websites are chosen, during the evaluation only the default page for each Website is evaluated, and the content of the page is collected for future analysis. Since the Web is ever evolving, the default pages for each Website are stored to freeze the evaluation data corpus to the collection date, and provide the means to repeat the experiments when required.

6.1.2 WPS Setup

WPS can be set up as an Add-On in a Web browser or deployed as a Restful service depending on the application. For the evaluation, WPS is set up as an Add-On in Mozilla's Firefox version 14.0.1 as seen in figure 6.1. Using this medium, the presentation layout of Web pages and features conforming to one of the industry's leading Web browsers is used to evaluate our approach.

Some Websites are developed to scale their pages, or add and remove features to the page depending on the user's Web browser screen size. To maintain the repeatability of the experiment, the Web browser screen's size is preset to 1200 x 800 pixels for

evaluating all Websites.

Remote scripting allows developers to actively load scripts, code and content after the Web browser loads the page. This feature allows the developer to improve initial download speed by employing Lazy Load techniques, and provide more interactivity on the page, as well as real-time responses. When Lazy Load methods are employed, developers can use this method to further load less important content after the browser loads the page. In such scenarios, the page loads in two stages. The first stage loads the Web page, where the process is monitored by the browser, and upon successful loads, handler `onLoad` and the handler of event `DOMContentLoaded` will be triggered. The second stage is tailored to the page, where it loads the content using remote scripting or Lazy Load techniques to the relevant sections in the page. This customised process is specific to the Web page and it is difficult to monitor the entire process. To overcome this issue, a button is included to the Add-On interface as seen in figure 6.1, so that WPS can be fired after the first instance of the page is visually loaded. Finally, the results of the prediction are printed in the area on the left of the button.

6.1.3 Manual Analysis

In order to provide verification of the results predicted by WPS, a manual analysis is applied. The manual analysis provides a fine granularity of analysis over the results provided by WPS. This analysis consists of a person going through the same Web pages that WPS has analysed. Using the same setup and widgets definition, except this time a human searches the Web page source code for these tell-signs and compares it with the prediction results returned by WPS. We recognised that humans are not always consistent when it comes to repeated tasks, and they are often error prone. Thus, the analysis was done in stages to reduce fatigue. To reduce the deviation of concepts between the manual analysis and WPS, the person conducting the analysis will use the definition provided by WIO.

The results from the manual analysis are tabulated and can be found in `http://wel-data.cs.manchester.ac.uk/data_files/8`. There are four columns in the results presented. The first column details the type of widget WPS has predicted or the manual analysis has derived. The second column gives the Confidence percentage returned by WPS. In the case where the manual analysis does not tally with the WPS prediction, a zero will be awarded in this column. The third column indicates whether the manual analysis matches the results provided by WPS. This is symbolised by having $\checkmark$ as conform, $\times$ as disagree, $\checkmark*$ as conform but it is a repeat, and $\times*$ as

disagree and it is a repeat. Finally, in the last column the report generated for each widget detected is presented, along with the remarks from the manual analysis being recorded.

6.2 WPS Evaluation Results

A discussion of the results collected by WPS and the manual analysis are presented in this section. These results are proof-of-concept that WPF is a feasible approach for predicting widgets. The raw population of widgets predicted (y-axis) are tabulated at an interval of 5% confidence (x-axis) in the graphs. A point in the graphs for each type of widget will be selected as a cut-off point, to decide whether a candidate widget detected exists or not. This point is called the threshold confidence percentage. Using this spectrum of granularity will allow us to spot the threshold confidence percentage for the different types of widget at an accuracy of 5%.

The threshold confidence percentage is the point in the graph with the smallest confidence percentage, where the True Positives have the largest population, and the smallest population of False Negatives, and the smallest population of False Positives. In the evaluation results presented, the threshold confidence percentages derived from our results are merely a demonstration of how to use the WPF to predict widgets. This should not lead to the impression that we are proving WPS as a more accurate widget prediction approach/system.

It is worth noting that the Website `uimserv.net` could not be accessed during the evaluation. Thus, the result from this site is not considered in our evaluation and only forty-nine Websites are examined in our evaluation process.

6.2.1 Ticker Widget Evaluation

The Ticker widget, although the least popular among the seven types of widget chosen for this investigation, has the best prediction rate, as seen in figure 6.2. Despite the indirect approach used to overcome the issues faced when identifying the `WindowTiming` tell-sign (see §5.6.1), this technique demonstrated its reliability during our evaluation.

After scrutinising the predicted results from WPS, even though only five occurrences of this widget are predicted, our manual analysis confirms that all predictions are accurate. Figure 6.2 illustrates the True Positives, False Negatives, and the False Positives results from our evaluation for this widget, where the Actual line is the sum

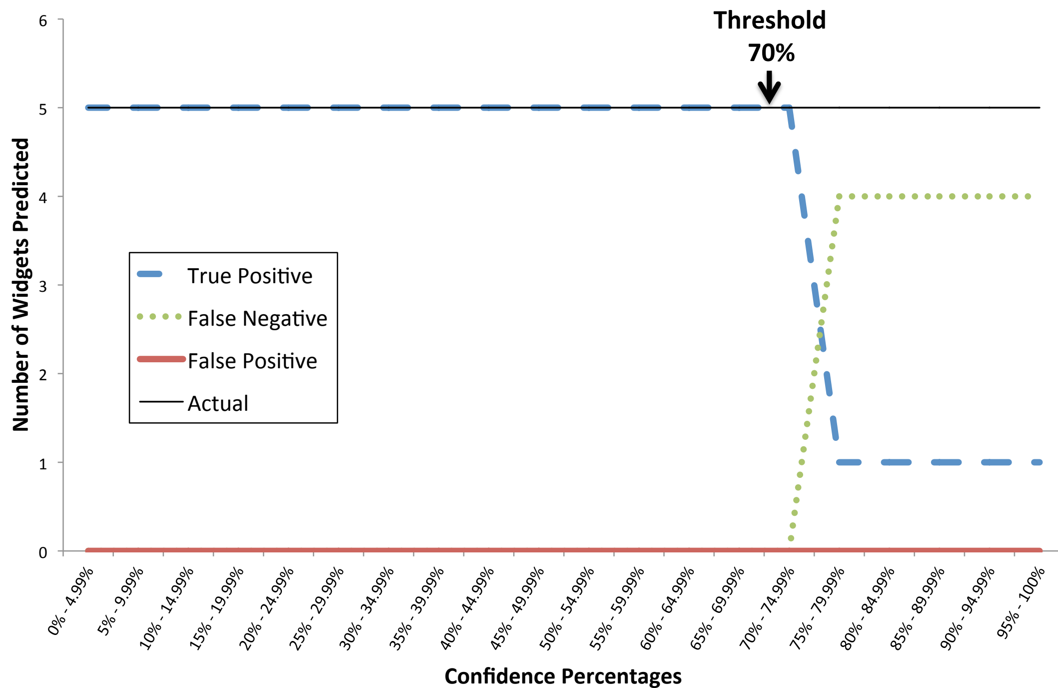


Figure 6.2: Prediction results for Ticker widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

of True Positives and False Negatives. These results are plotted at a confidence percentage interval of 5%, against the number of widgets predicted for each line. The threshold confidence percentage for the Ticker widget is chosen to be 70% using the selection process described. This is because that is the point just before the True Positives begin to fall and False Negatives begin to rise.

Using this set of results with the chosen threshold confidence percentage, an excellent prediction rate of 100% can be achieved. From this demonstration, we have covered how to choose the threshold confidence percentage, and in this ideal case 100% accuracy was achieved. Although the prediction rate for this widget is 100% accurate for the evaluation, however, this may be due to the low popularity of this widget. From the manual analysis, we noticed that only one Ticker widget was found on each Website. Comparing this result with our previous popularity study presented in §3.4, both results are close, and this demonstrates the accuracy of our prediction techniques employed for the Ticker widget.

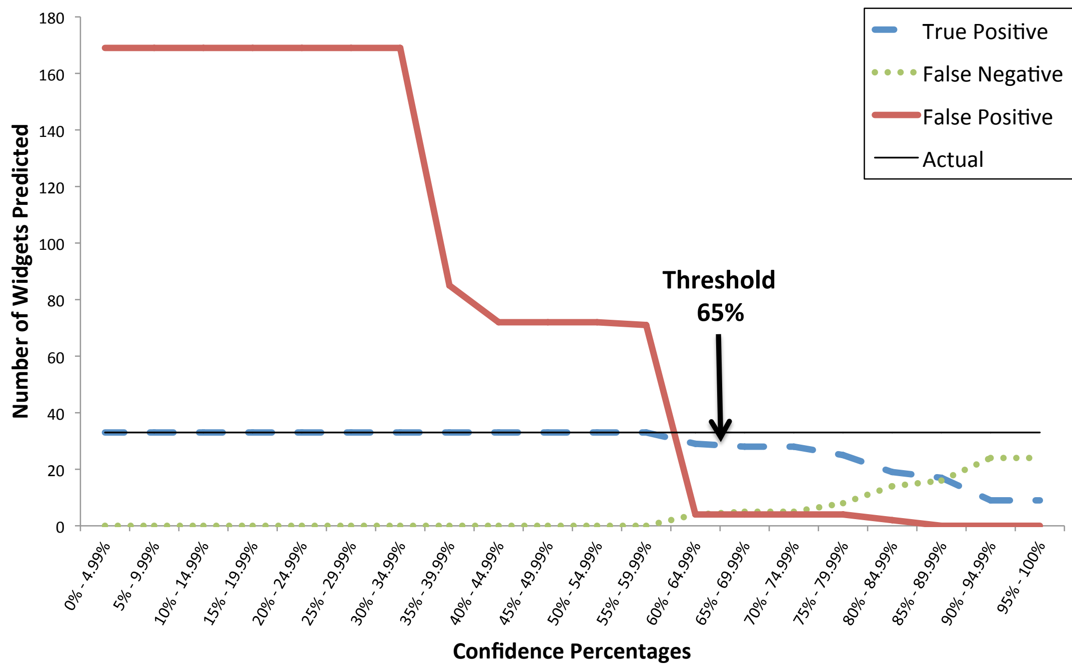


Figure 6.3: Prediction results for Auto Suggest List (ASL) widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

6.2.2 Auto Suggest List (ASL) Widget Evaluation

Identified as one of the simpler types of widget to detect in chapter 4 - Feasibility Investigation for Using Tell-Signs, the ASL widget prediction performed fairly accurately as presented in figure 6.3. The prediction results exhibit trends that are expected, such as a high False Positive at the lower end of the confidence percentage scale, and then rolled off to almost zero at the higher end. These results demonstrated that some of the tell-signs' instances are generic – a consequence of the initial high False Positives. However, the steep decline of False Positives after 35% and 60% suggest that these generic tell-signs are vital components of the overall tell-sign combination to assist the ASL widget predictions.

Using the same selection process for the ASL widget threshold confidence percentage, it is selected as 65%. This is because 65% is the optimum value where the False Positive values or Type I errors are low and the True Positive detection remains at an accurate level. Indeed, the choice of the threshold confidence percentage misses some

True Positives, but it is the best compromise for prediction accuracy. This will be a typical scenario for predicting widgets using WPF. Developers applying WPF would want to model their widget prediction to achieve results like this, if not better. Using the selected threshold confidence percentage, 28 out of the 33 ASLs are detected, thus achieving an accuracy of 84.85%. From the evaluation results, the techniques and the selected tell-signs employed to detect the ASL widget have demonstrated that they are appropriated for the task.

6.2.3 Popup Content Widget Evaluation

Described in §3.3.7, the tell-signs used to classify and identify the Popup Content widgets can be easily deceived as a straightforward detection process. The result of this widget presented in figure 6.4 shows a presence of a high number of False Negative or Type II errors, and the correct prediction and False Positives or Type I errors trending in a similar fashion. The results for the Popup Content widget suggest that the threshold confidence percentage should be 50%. Using this value, 26 out of 165 Popup Content widgets are detected, achieving an accuracy of only 15.76%.

Some instances of Popup Content in Websites such as `56.com`, `google.com.tw`, `digg.com`, `letitbit.net`, `irctc.co.in` were not detected due the relationship issues between the suspected element that has the `z-index` styling property value greater than 9, and the element that is monitored for mouse events or has a tag name `<a>` to trigger the process. The issue arises because of the way WPS was designed. Nodes that are within the proximity of ± 3 nodes apart will be considered to be related. However, none of these pairs of elements fulfil this specification.

More recently after our technological freeze §1.2 dateline, anecdotal evidence suggests that more Websites introduce the concept of float panels to improve usability in their pages. However, this method causes confusion to our detection method as noticed in `digg.com`. Since this method is an evolutionary issue, this issue will not be pursued in this evaluation. Nevertheless, future work should examine what is the best approach to develop tell-sign objects, so that they can model the `z-index` styling property. This may mean breaking it down into finer grain. Therefore, results from widgets that use the `SetDOMElementOrderHigh_Tell-Sign` class under the `ManipulatingStylingProperties_Tell-Sign` class will be affected. Hence, in this thesis only Popup Content and Collapsible Panel widgets will be affected.

Due to the technical issues faced when evaluating `digg.com`, it contributed to a massive 77 records of False Negative or Type II errors to the prediction results. Thus,

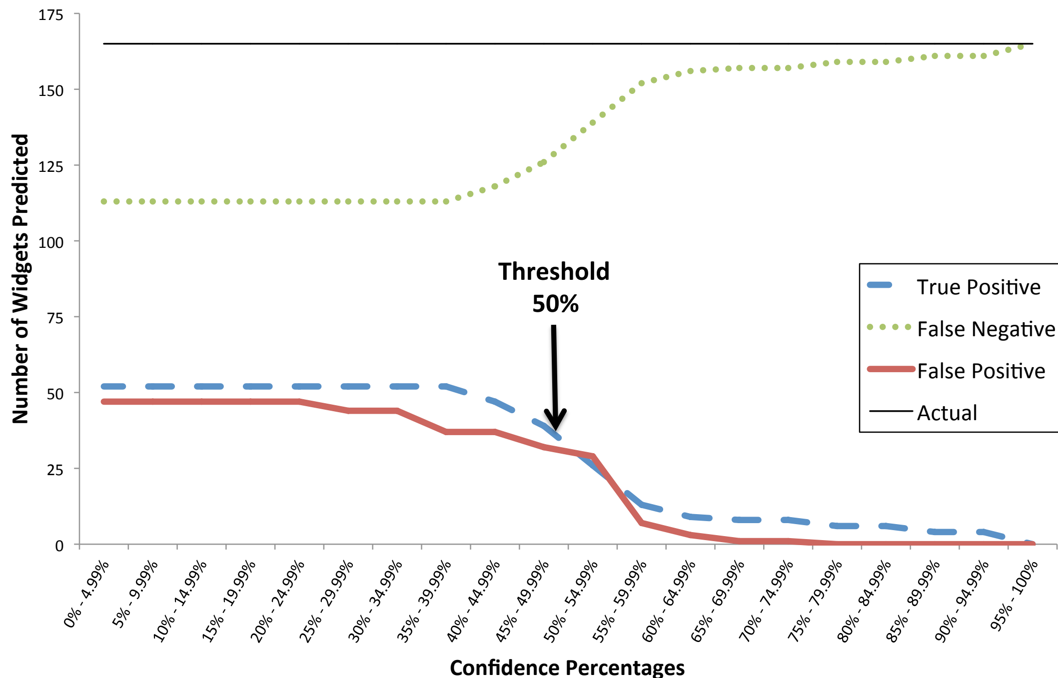


Figure 6.4: Prediction results for Popup Content widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

it is recommended that the analysis for this type of widget should be done again without including `digg.com` prediction results.

Issues faced when tracing for the related section of the code limit the prediction process. Sometimes following the trails of clues can be confusing due to the Weakly Typed scripting language JavaScript involves. This suggests that further research to follow Weakly Typed language clues and semantics will allow for better prediction rates.

6.2.4 Collapsible Panel Widget Evaluation

The tell-signs of the Collapsible Panel widget are at times difficult to identify due to its generic characteristics. Hence, often the Collapsible Panel prediction is easily confused with other widgets such as the Popup Content widget if some of the tell-signs are not present. As illustrated in figure 6.5, a high presence of False Negatives suggests that the prediction mechanism lacks evidence during its analysis to deduce

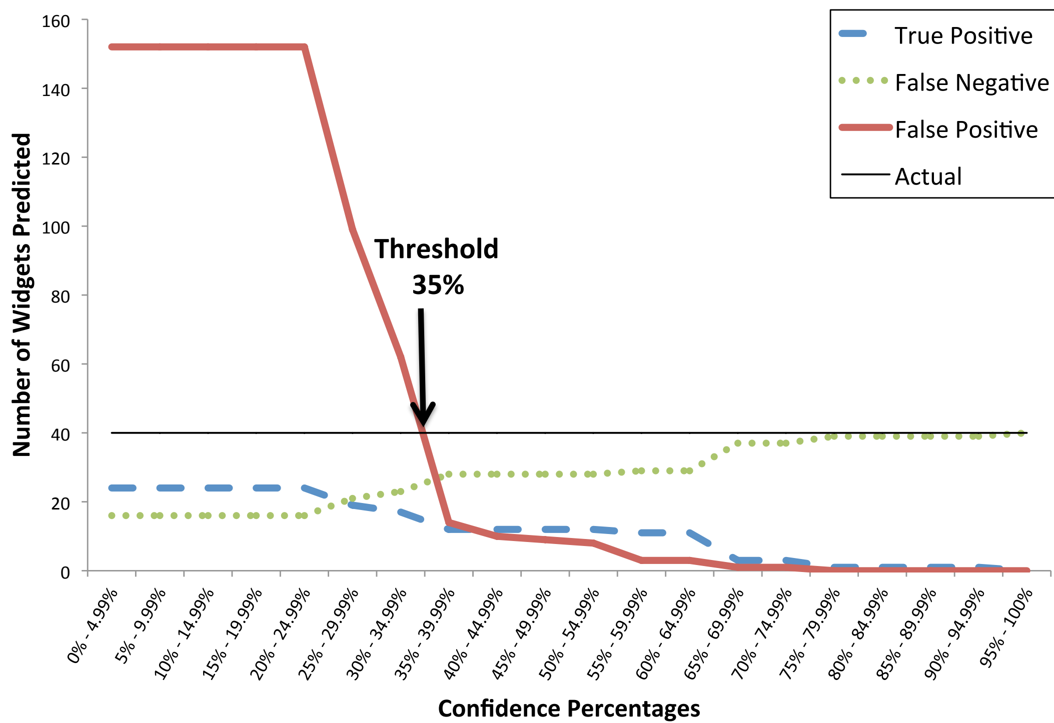


Figure 6.5: Prediction results for Collapsible Panel widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

the identity of the Collapsible Panel widget.

A close examination of the results together with the manual analysis results suggests that often the system confuses the Collapsible Panel widget prediction with the Popup Content widget. The behaviour that causes this type of confusion often exists when some instances of a tell-sign are not present, thus resulting in the tell-sign being inconclusive.

The threshold confidence percentage for the Collapsible Panel widget was chosen to be 35%. Although from the prediction results for this widget, to determine the threshold confidence percentage is difficult, 35% gives a fair compromise between the True Positive, False Negative and False Positive values. With the selected threshold confidence percentage, 12 out of 40 Collapsible Panels widgets are predicted, achieving a 30% accuracy. A considerable amount of False Positives is noticed even at low confidence percentages. This suggest that the methods applied do not stretch wide

enough to detect the different variations of the missed widgets. However, we are only demonstrating how WPF can be applied to predict the Collapsible Panel widgets. Thus, these results do demonstrate broadly that Collapsible Panel widgets can be detected, and the methods applied are feasible. On the other hand, they also suggest that some of the techniques applied may need to be revisited. They may be too generic, or a wider spectrum of methods could be used to develop a tell-sign.

Further investigations are required to identify the issues to improve the accuracy in the prediction for the collapsible Panel widget, with particular emphasis on displaying the Display Window. Instead of just searching for clues to hide and display the Display Window, anecdotal evidence suggests that the height and width of the Display Window can also be manipulated to give the visual effect that the Display Window expands and collapses. Another suggestion for future investigations on the Collapsible Panel widget prediction would be revisiting the design patterns library, to check if there have been any updates to this widget since the detection techniques were developed. Since the introduction of HTML 5.0 standard, new techniques may have emerged.

6.2.5 Tabs Widget Evaluation

The metaphor of a Tab widget usually consists of two lists – one for navigating through the different tabs in the Widget, and the other list to store the content of the tabs. Although the definition of the Tab widget can be well designed, however, in practice when translating the concepts into code, structurally it is generic and this form of coding structure is widely used to develop Web pages and widgets. This makes it difficult to detect. Figure 6.6 demonstrated the complexity of determining this kind of widget between the region of 50% and 75% confidence.

Choosing the threshold confidence percentage for this widget is tricky based on the evaluation results. However, we advocate 65% as the best compromise between the True Positive, False Negative and False Positive predicted values. Using 65% as the threshold confidence percentage, 29 out of 62 Tabs widgets are predicted, resulting in 46.77% accuracy.

There are many ways this widget can deliver the content to its user. Based on the anecdotal evidence gathered during the manual analysis, two methods were widely used. The first method either uses remote scripting to load the respective tab content or preload the content to the respective tab. Then the Behaviour layer is used to manipulate the styling properties of the element to make the selected content visible to the user. The second method uses similar methods to load the tabs content. However,

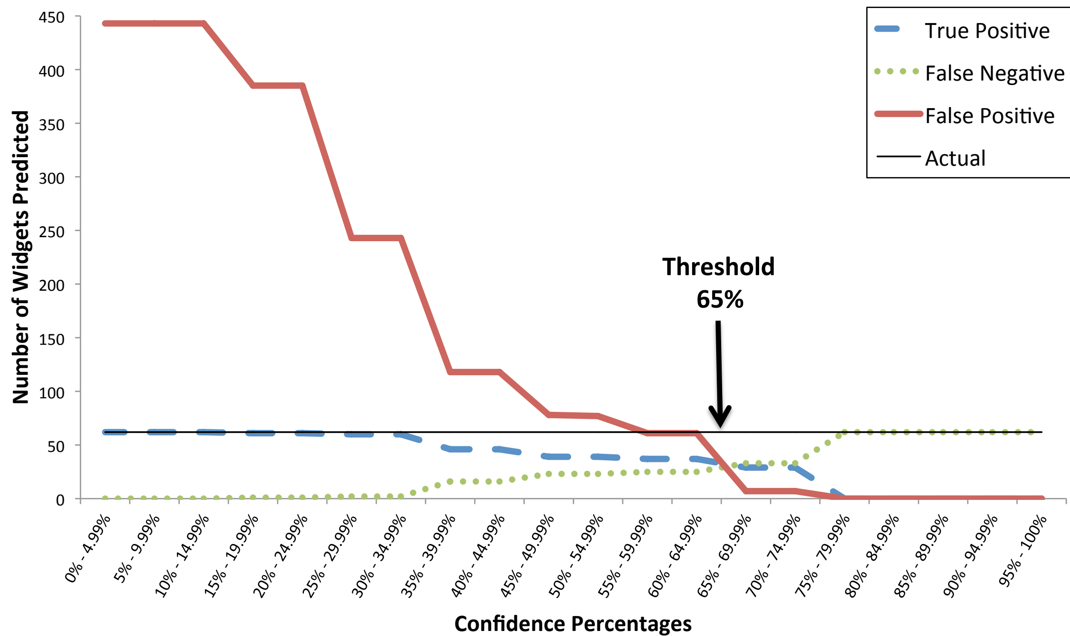


Figure 6.6: Prediction results for Tabs widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

this time the Behaviour layer is used to manipulate the styling properties `z-index` of the elements such that all tabs are stacked on top of one another, with the selected tab having the highest layer order. If the second method is employed, our prediction methods will confuse the Tabs widget for a Popup Content widget, which has occurred on a few occasions.

6.2.6 Carousel Widget Evaluation

During the feasibility study for using the tell-signs defined as a technique in our detection approach (chapter 4), Carousel and Slide Show widgets were considered to be the widgets that were more complex to identify due to the processes these widgets possess. The results presented in figure 6.7 confirm this judgement as clues of the Carousel widget prove difficult to find within the code.

Depending on the approach the developers decide to take on, sometimes even some essential components of the widget are not present until the developer requires the user to interact with the widget. Scenarios like this make it difficult to predict if the widget

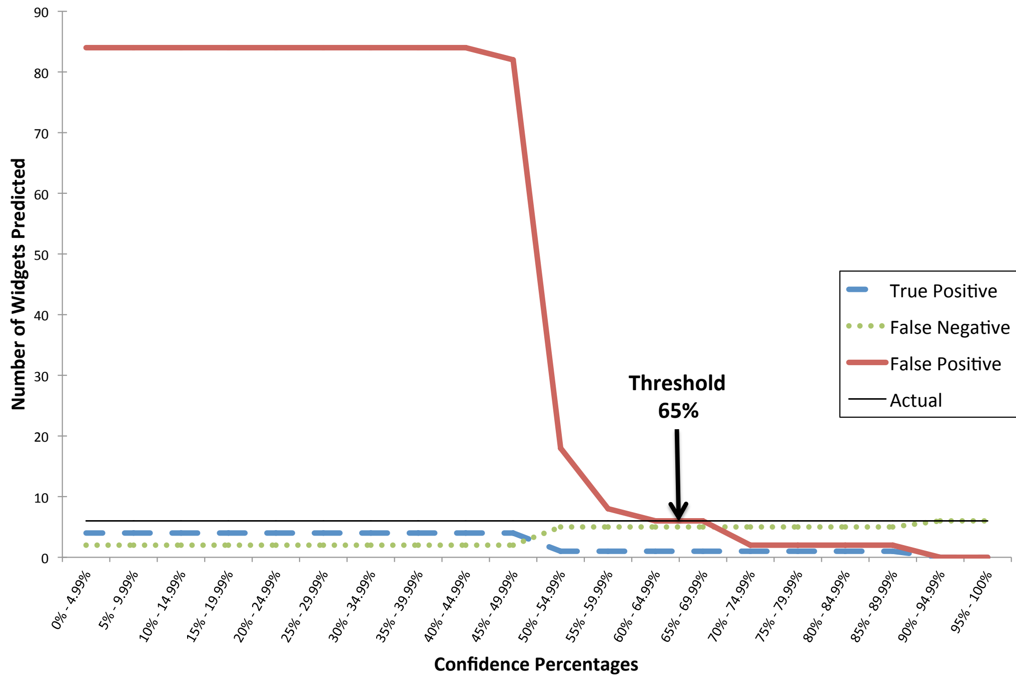


Figure 6.7: Prediction results for Carousel widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

exists. Even for humans, a trial-and-error approach is required to determine widgets that operate in this way. Thus, more loosely defined detection has to be incorporated to accommodate the vast variety of methods, and a high number of False Positives prediction or Type I errors is expected as reflected in figure 6.7.

Searching for the best compromise for the threshold confidence percentage is not easy and 65% is believed to fit the best point. Using the selected threshold confidence percentage, 1 out of 6 Carousel widgets are predicted (accuracy of 16.67%). The early dip in True Positive values, contrasting with the rise in False Negative values, suggests that either the prediction methods are too rigid, or the range of variations in which the Carousel widget can be developed is broader than previously thought.

6.2.7 Slide Show Widget Evaluation

Defined in §3.3.9, the difference between a Slide Show and Carousel widget is the way these widgets allow their user to browse through the list of contents. With such close

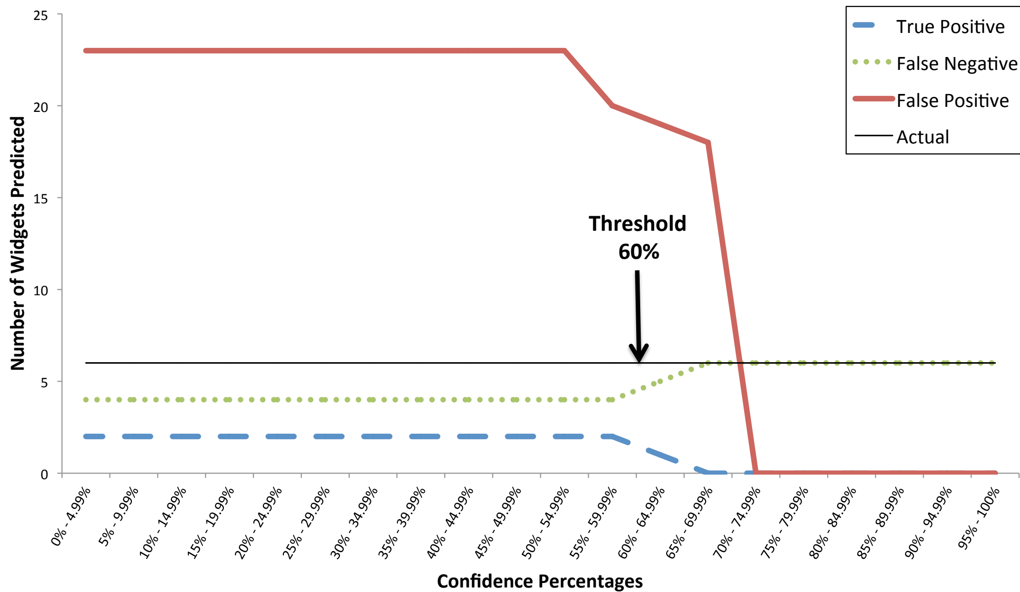


Figure 6.8: Prediction results for Slide Show widget. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

definition proximity, the results presented figure 6.8 are not surprising, especially when not all the clues of a tell-sign are readily available from the code.

Selecting the confidence threshold percentage for the results collected for this widget is straightforward as there are not many choices. The threshold is chosen at 60% for this widget, as it is the only point with the least False Positives and highest True Positives. The selection of the threshold confidence percentage resulted in only 1 out of 6 Slide Show widgets being detected; an accuracy of 16.67%.

With similar trends for the Carousel widget, the Slide Show widget also exhibits high numbers of False Positives due to the vast variations of styles and methods with which this widget can be developed. It can also be seen that the number of False Negatives is double that of the True Positives. This suggests that either the prediction methods are too rigid, or the range of variations with which the Slide Show widget can be developed is broader than previously thought. This also suggests that the detection methods need redressing and generalising to capture the wide range of techniques applied by developers.

6.3 Analysis and Discussion

The results from all seven widgets have been presented and compared with our manual analysis results. In this section, we analysed the prediction results and exposed the insights of the results. The results demonstrated that the robustness of our approach varies between different types of widget. In one aspect, this issue could be narrowed down to the selection of tell-signs, and the approaches employed to detect them. On the other hand, the heterogeneous nature of the Web, and the approaches developers take to loading information/code on the fly, are not to be overlooked.

Good results were achieved in the Ticker and ASL widgets prediction as demonstrated in §6.2.1 and §6.2.2 respectively. Nevertheless, widgets like Popup Content, Collapsible Panel, Carousel and Slide Show will require supplementary techniques to assist them with the prediction process.

Poor prediction results do not mean the technique is weak. The results from the manual analysis suggest that sometimes insufficient evidence is present in the code to detect the tell-signs, or when tracing identifiers in the code the clues can be misleading or not derivable. Similar issues were reported by Dong et al. [2008b]; Gamma et al. [1995], when attempting to reverse engineer design patterns from the source code for desktop applications. Further research is suggested to investigate the possibilities of meeting this challenge.

From the empirical observations gathered during the manual analysis, it is suggested that the major challenge when identifying tell-signs is tracing the clues in the source code and the methods developers employed to create the widgets. Depending on the approach the developers decide to take on, sometimes even the components of the widget are not present. This means that prediction has to be done at a level where not all the components of the widget will be present.

Recreating the relationships of different nodes from the DOM without documentation is another area worth further investigation. Although annotating the code could be a possibility, this means higher computational overheads and poorer response time.

The similarity between the definitions of the Carousel and Slide Show widgets as seen in §3.3.4 and §3.3.9 respectively suggest that the prediction methods applied in WPS can be combined for further analysis. Referring to figure 3.7 and figure 6.9 to illustrate the alteration of the Carousel and Slide Show widgets' definition, the 'Next' and 'Previous' buttons are now simplified by ignoring whether the buttons are finite or looping.

After updating the code in WPS to match the simplified 'Next' and 'Previous'

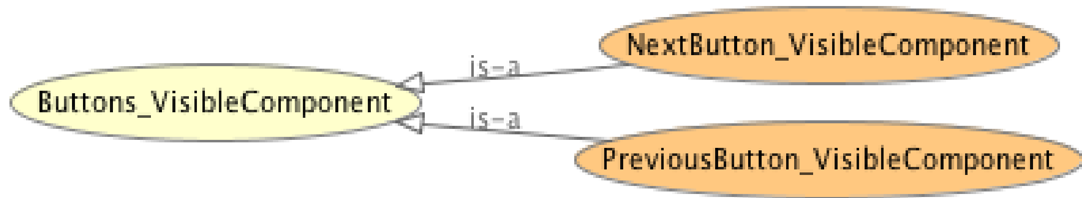


Figure 6.9: Simplified 'Next' and 'Previous' buttons concepts.

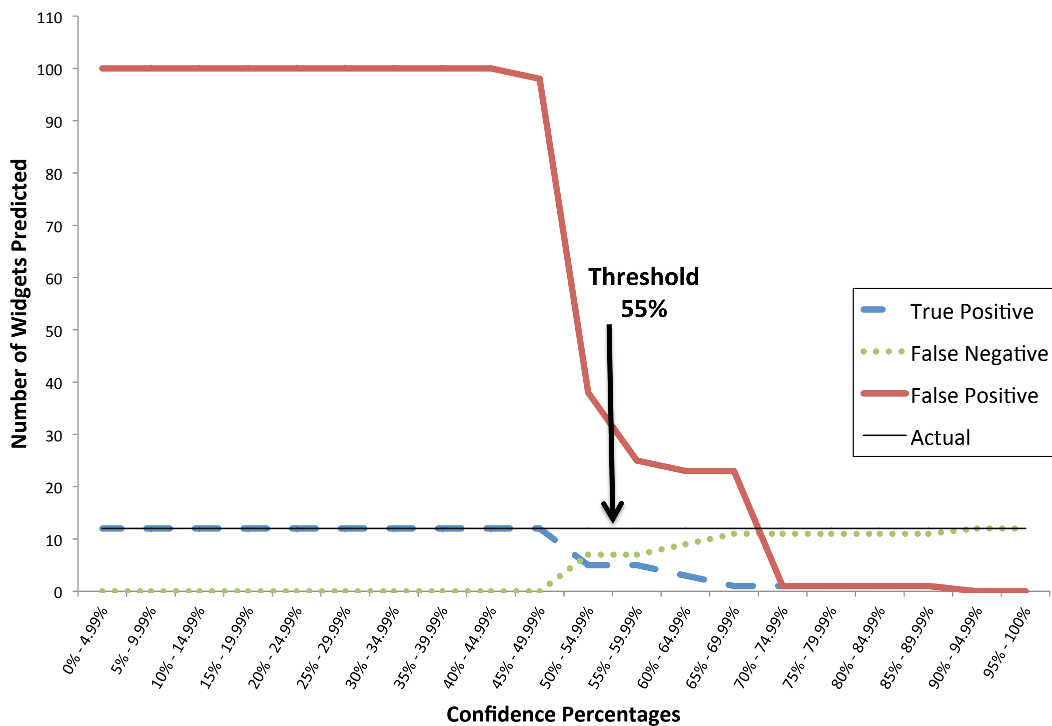


Figure 6.10: Prediction results when Carousel widget and Slide Show widget are combined. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

buttons concepts, to combine the Carousel and Slide Show widgets, the new set of prediction results can be seen in figure 6.10. Now, the selected confidence percentage threshold is 55%, which is the best compromise between the True Positive, False Negative and False Positive predicted values. Using the selected threshold confidence percentage, 71 out of 194 widgets are predicted. There is an improvement to the accuracy for the Popup Content widget and Collapsible Panel widget which have risen from 15.76% and 30% respectively to 36.67%. The improvement suggests further research

should be conducted to understand more about the characteristics of the Carousel and Slide Show widgets. User studies are also suggested to understand the effects on user experience when both of these widgets are misinterpreted.

As discussed in §6.2.3 and §6.2.4, both the Popup Content and Collapsible Panel widgets are often found to be confused by WPS during the prediction process. The main challenge lies in there being too many ways these types of widget can be developed. A few contributing factors include the heterogeneous nature of the Web, the Weakly-Typed languages involved, and the fact that Websites are often developed by practitioners, graphic designers and people with little knowledge of developing Web pages.

The main difference between the Popup Content and Collapsible Panel widget is the order of the widget wrapper elements `z-index`. Without this distinction, both widgets are conceptually the same. Figure 6.11 shows the compiled results for both of these widgets together when the `SetOrderFront_ElementStyle-ManipulatingDom` component in the Popup Content widget is ignored. The order of elements in the z-axis of the user interface is merely a visual effect and may have little importance for Assistive technologies users. This suggests venues for user experience studies in the future, to investigate the degree of usage, to understand the extend of the differences of these widgets have on the user, as well as analysing the impact systems such as WPS have on the user.

By combining the detection methods for these two widgets, the extent to which these widgets get misinterpreted by using our definitions is demonstrated. An improvement to the prediction results is observed in figure 6.11 when compared with the individual prediction results of both widgets (see figure 6.4 for the Popup Content widget's prediction results and figure 6.5 for the Collapsible Panel widget's prediction results).

The new confidence threshold percentage chosen is 40% as it is the best compromise between the True Positive, False Negative and False Positive predicted values. However, in this case the confidence threshold percentage can be variable by $\pm 5\%$, whoever provides the optimal result. Using 40% as the threshold confidence percentage, 5 out of 12 Popup Content and Collapsible Panel widgets are detected, an accuracy of 41.67%. A significant improvement to the prediction accuracy from 16.67% when detecting the widgets individually, to 41.67% when combined, was achieved. These results suggest that future user studies should examine the effects of these types of widget when they are misinterpreted.

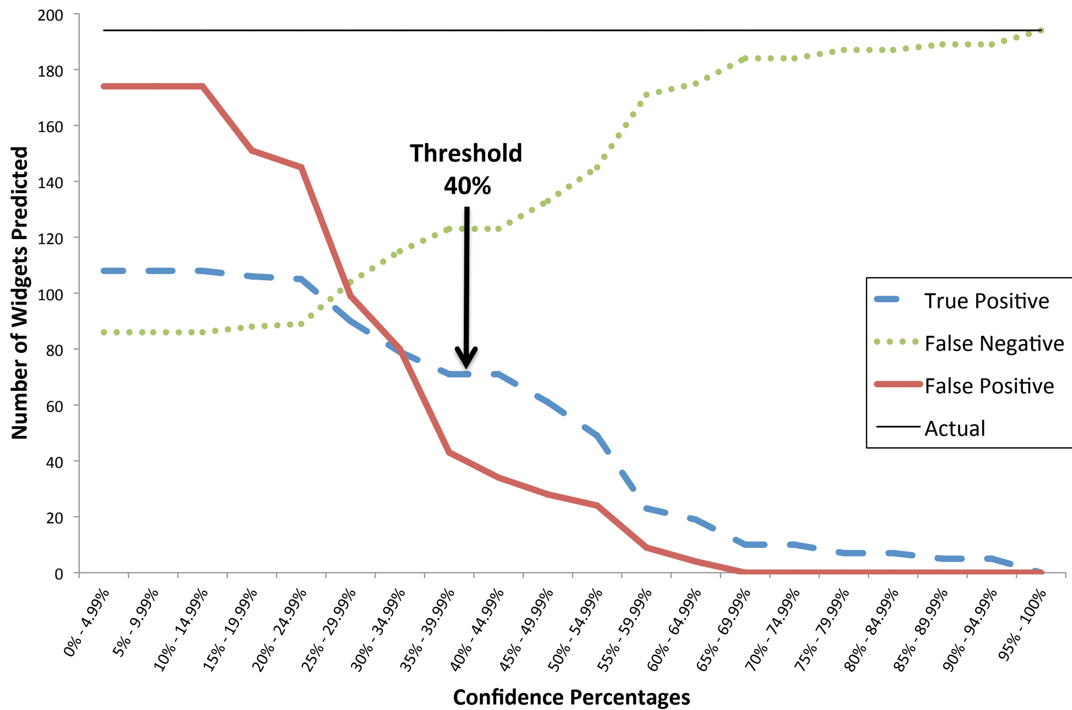
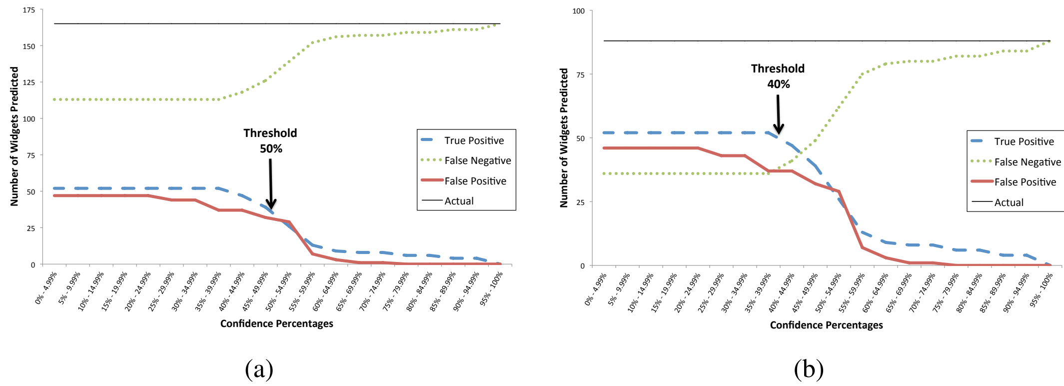


Figure 6.11: Prediction results when Popup Content widget and Collapsible Panel widget are combined. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

Focusing on `digg.com` in particular, this Website used widgets that were more popular after our technical freeze dateline. Thus, when designing the classification of the widgets in this thesis, conflicting methods of detecting tell-signs for some widgets were noticed. Therefore, this Website was taken out of our evaluation as an outlier Website, and the affected types of widget prediction were re-analysed.

Only the Popup Content widget is affected by our widget's definition after isolating the outlier Website. This is because only widgets that use the `z-index` styling property in their definition are affected. Figure 6.12 contrasted the results of Popup Content widget prediction. It showed how casting `digg.com` as an outlier affected the results of the prediction methods chosen for the Popup Content widget.

An improvement to the prediction results can be noticed in figure 6.12(b) when compared to figure 6.12(a) after removing the outlier Website. The new confidence threshold percentage is now selected as 40% – a point in the graph that has the biggest number of correctly predicted widgets, and the lowest number of False Positive and



(a) (b)

Figure 6.12: Comparison of Popup Content widget results with (a) and without (b) `digg.com`. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positives.

	Threshold Confidence %	Accuracy	True Positives	Manual Analysis
With `digg.com`	50.00%	15.76%	26	165
Without `digg.com`	40.00%	53.41%	47	88

Table 6.1: Comparison of results with and without `digg.com` for Popup Content widget.

False Negative results. Table 6.1 lists the results of the Popup Content widget in detail. By removing the outlier Website `digg.com`, a three fold improvement is observed. This is because it was noticed that `digg.com` uses a new type of widget called "Floating Panel" that was introduced after our technical freeze dateline. A key characteristic of the Float Panel widget is the `z-index` styling property. This characteristic confuses our system with the `z-index` tell-sign. As a result, a low confidence percentage is awarded due to the insufficient evidence that can be concluded for the Popup Content widget. These results exhibit the impact on WPF, which can be affected due to the evolution of widget designs.

Using the same concept of removing the outlier Website results from the Popup Content widget evaluation, this method is also done for the Collapsible Panel widget results. However, from the manual analysis, the conflict between the methods only affects widgets with tell-signs that have high `z-index` styling property. Thus, only Tabs and Popup Content widgets use this component as part of their detection methods. Nonetheless, the detection methods employed by the Tabs widget only use this

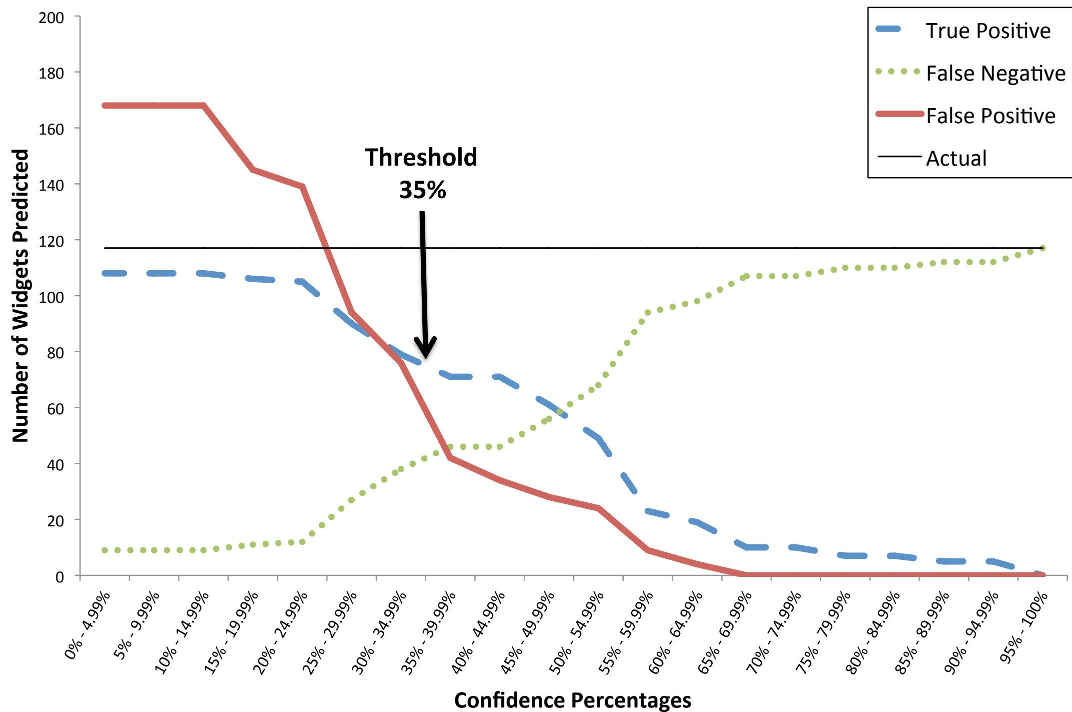


Figure 6.13: Prediction results when Popup Content widget and Collapsible Panel widget are combined without `digg.com`. Where Actual is the sum of True Positives and False Negatives, and the Threshold value is the anecdotal confidence percentage, in the graph, that will give the highest ratio of True Positives to the sum of False Negatives and False Positive.

component as an optional approach, but only Popup Content widget results were affected.

As discussed earlier in figure 6.11, an improvement to the prediction is noticed when combining the Popup Content widget results with the Collapsible Panel widget results. A similar analysis is done this time by removing the outlier Website, as shown in figure 6.13. Notably, an improvement to the prediction results is demonstrated. The confidence threshold percentage is chosen at 35%, where it is the spot with the most True Positive or correct prediction, and the lowest False Positives and False Negatives.

Further improvement can be observed in table 6.2 by removing the outlier Website `digg.com`. A jump in accuracy strongly suggests that further investigations should examine how to model the `z-index` order, and redress the current method. The Floating Panel widget, found in `digg.com`, provides a clue that the granularity of `z-index` order should be more fine grained.

The high number of False Negatives or Type II errors suggests improvements to

	Threshold Confidence %	Accuracy	True Positives	Manual Analysis
With digg.com	55.00%	36.60%	71	194
Without digg.com	35.00%	60.68%	71	117

Table 6.2: Comparison of results with and without digg.com for the combination of Popup Content and Collapsible widget.

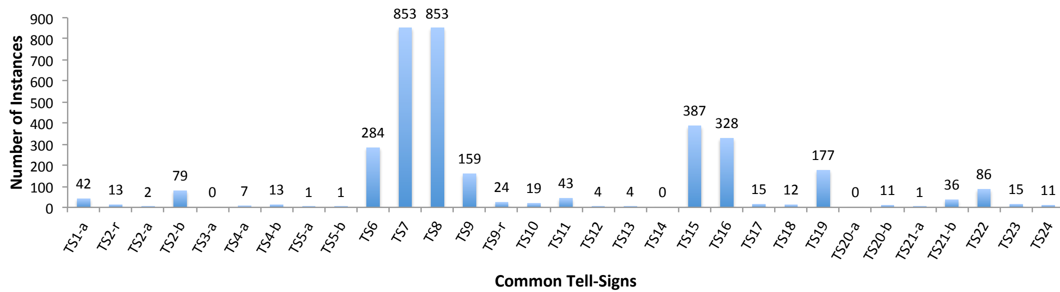


Figure 6.14: The usage of common tell-signs analysed from our evaluation results. The description of the identification reference for each tell-sign can be found in Appendix E.

some of the tell-signs could be made. To understand the problems pertaining to the tell-signs, further analysis was conducted to find out how they performed during the evaluation.

Generically, the components of a widget are formed by two kinds of tell-signs defined in our ontology. The first type is private or specific to the type of widget, while the second type is commonly used by all types of widget because it refers to generic methods or objects in the code. Therefore, when investigating the performance of the tell-signs, they will be discussed separately; each set of the private tell-signs is analysed separately according to the type of widget, while the common tell-signs are done as a group.

Figure 6.14 presents the usage of the common tell-signs that can be reused as functional objects to define different types of widget. Some of these tell-signs are significantly more popular than others. The more popular tell-signs include TS7, TS8, TS15 and TS16; on the other hand, tell-signs such as TS3-a, TS14, TS20-a were never used during the evaluation. See chapter 3 for the details of the tell-sign identifiers listed.

Common tell-signs that are less often used do not mean that they are not useful indicators of a type of widget. It was the intention to pack more variations of tell-signs

to cover a broader spectrum of the development approach of the type of widget. In this case, these common tell-signs are not as popular among the Websites selected for our data corpus. However, there is no denying the fact that some of these common tell-signs may not be the best method to develop or identify the respective type of widget.

The analysis of the private tell-signs for a type of widget is presented in figure 6.15. The performance of the private tell-signs employed by the Auto Suggest List (ASL) widget is shown in figure 6.15(a), and the Ticker widget's private tell-signs performance is shown in figure 6.15(b). The combination of Popup Content widget and Collapsible Panel widget private tell-signs performance is shown in figure 6.15(c), while the Tabs widget's private tell-sign performance is shown in figure 6.15(d). Finally, the combination of Carousel widget and Slide Show widget private tell-sign performance is presented in figure 6.15(e).

Evidence of tell-signs not used during the evaluation can be found in the ASL widget – A-TS1, Ticker widget – TK-TS1, TK-TS2 and TK-TS3, the combination of Popup Content and Collapsible Panel widgets – PC-TS2, PC-TS3, PC-TS5 and PC-TS6, and the combination of Carousel and Slide Show widget – CS-TS5-a, CS-TS5-b, CS-TS5-c, CS-TS6-b, CS-TS9-a, CS-TS9-b, CS-TS9-c and CS-TS10-b. These results suggest that the techniques modelled by these tell-signs may not be popular among the Websites chosen for our evaluation.

Drawing attention to the Ticker widget's private tell-signs: interestingly, we have witnessed that the prediction methods for this type of widget produced the best results. However, figure 6.15(b) demonstrated that only one of its private tell-signs (TK-TS4) is found throughout the evaluation. These findings show that developers prefer to develop the Ticker widget using a combination of more generalisable methods, rather than methods that are specific for the type of widget. Another reason for this phenomenon to occur may be due to the developers' choice of approach to develop these Websites. Often, developers prefer to use readily available industrial libraries to code their Websites.

Unlike the Ticker widget, figure 6.15(a) illustrates that ASL widgets are often developed using more specific methods for the widget. Analysing the results discussed about the ASL and Ticker widgets with the rest of the other types of widget, it can be noticed that widgets with more tightly formed methods perform better in our prediction approach, compared to those that have more loosely formed methods or that have a wider variation of development methodologies. These issues can be narrowed down

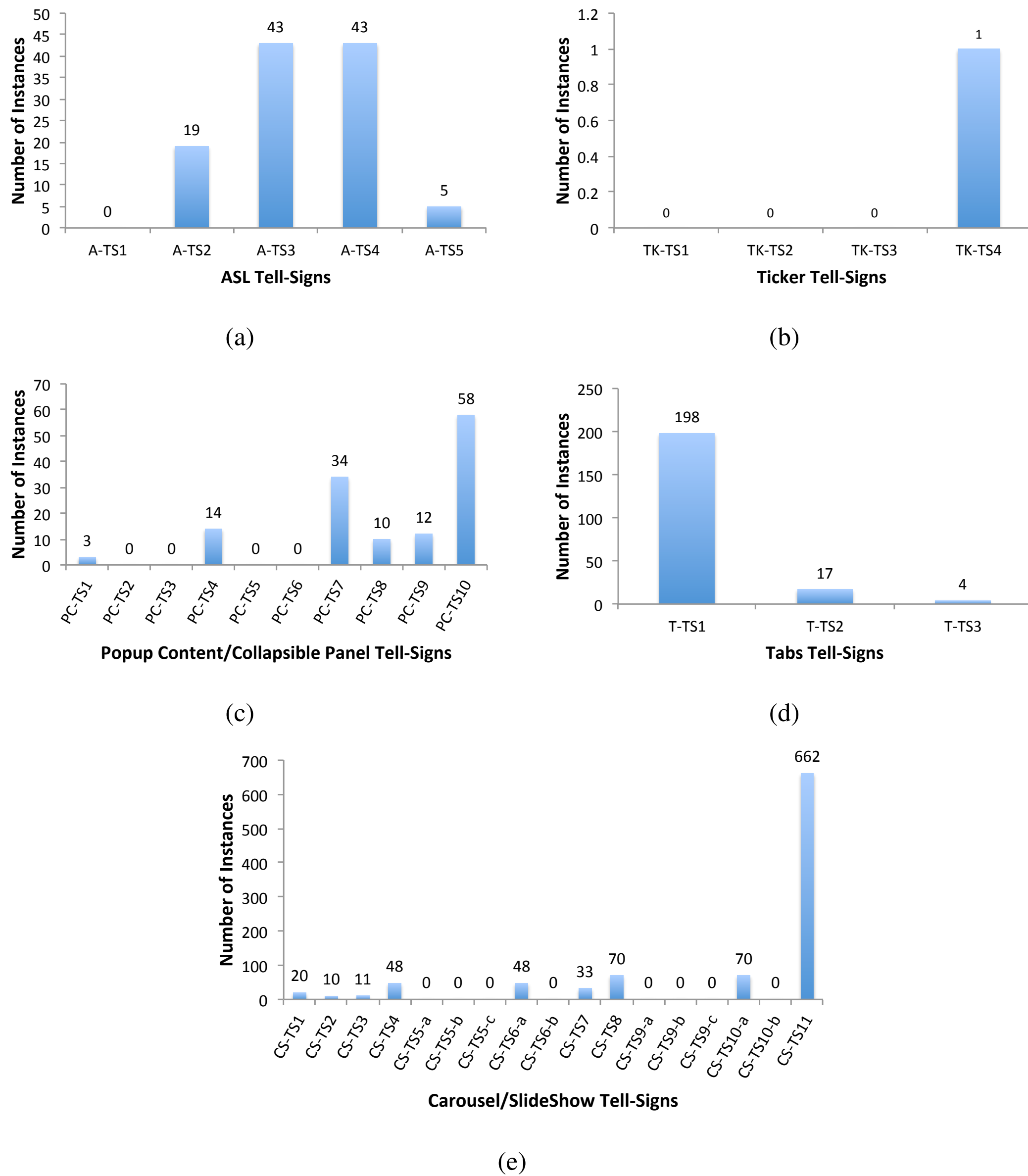


Figure 6.15: The usage of specific types of widget tell-signs analysed from our evaluation results. The description of the identification reference for each tell-sign can be found in Appendix E.

to choosing the right balance of detection granularity.

Similar usage trends are exhibited among the usage of common and widget-specific private tell-signs. Again, it can be noticed that more than necessary tell-signs are included in the detection to allow a broader range of developing styles to be incorporated in our prediction approach. These results provide a good mix of well-designed tell-signs that are popular and those which are less often used, along with tell-signs

that not as well designed. It provides the proof-of-concepts that tell-signs can be automatically detected from the source code, and when different types of tell-signs are combined to form specific sets of combinations, widgets can be predicted from the source code.

6.4 Summary

In this chapter, we have successfully demonstrated how WPF can be applied using WPS, to predict the widgets in a Web page. The evaluation over fifty popular websites obtained an average accuracy of 61.98%, with two widgets achieving $> 84\%$ accuracy. These results are proofs-of-concepts that WPF is feasible. A demonstration of how the widget ontology concepts can be altered using the Slide Show and Carousel widgets is presented. These changes impacted the accuracy of these widgets, and an improvement was observed. This demonstration has shown the importance of selecting the correct balance of detection granularity. Issues raised in the Popup Content and Collapsible Panel widgets prediction techniques suggest an avenue for future research. On the other hand, the weaknesses of the framework are also revealed. The case study of `digg.com` exposes the fact that the techniques used to model the order (`z-index`) of an element are insufficient and require redressing. Furthermore, the study stretches the boundaries of WPF, to reveal how the framework will react to evolving widget designs. The results analysed for the other widgets also provided interesting highlights of how their prediction methods are employed.

Analysis on the different kinds of tell-sign performance were presented and discussed. The trends in the techniques and approaches developers use to develop widgets were exposed. These results contribute insights for research intending to understand widget design patterns, for reverse engineering and widget design evolution purposes. Furthermore, they reveal the popularity of the seven types of widget employed by the fifty popular Websites. These results will provide insights for Web designers/developers with regards to widget designs and the tools intended to understand and detect widgets, to aid aged and visually impaired users.

Chapter 7

Conclusions and Future Work

At the beginning, we discussed the myth that providing accessible content means delivering content that can be interpreted by most people. This concept, as explored, is more difficult to achieve with dynamic content and the ever evolving Web. More recently, the model of delivering content over the World Wide Web (Web) has incorporated richer interactivity into Web pages, thus requiring users to interact even more intensively with the page. The investigation of the current state of the Web is discussed in our literature review from the perspective of Web Accessibility, along with related techniques that can help improve Web widget Accessibility. Our literature review unveiled the lack of coverage of developers' attempts to make the Web accessible due to a range of reasons. Lack of knowledge and compliancy that developers overlooked when designing and developing Websites, as well as the lack of understanding of how the extra interactivity affects the users, are some of the reasons reported. Ongoing studies conducted by Brown [2012] and Lunn et al. [2009b] provided insights into the issues and studies surrounding this method of content delivery. Using this method of delivery requires users to be more engaging, thus expecting their users to acquire more knowledge of using the Web, to cope with it. Often elderly and disabled users will be affected most by these changes. They either have to relearn the Web pages, or have the capability of figuring out a way of achieving their intended task.

This thesis highlights the need to assist humans with all forms of abilities to operate Web widgets, so that the experience of the Web can be improved. It successfully presented research that demonstrated methods to classify Web widgets, and described how to apply these concepts to predict widgets from the Web page's source code. Research questions set out in §1.3 investigate the applicability of formally classifying Web widgets, and the feasibility of predicting widgets from the Web page's source

code using the Widget Prediction Framework (WPF) suggested. The novelty of this approach defines widgets based on visual components combined with the widget's design concepts. A widget classification paradigm is proposed to model the widgets from a high-level perspective (see figure 3.3), as well as sufficiently fine grained to detect instances of the widget from the source code. We believe this framework addresses the issues laid out by this thesis from both the developer's and end-user's perspectives, providing a high-level approach to detect widgets across multiple computer languages, and that it contributes a rounded solution for the proposed research.

In chapter 5, a demonstration of how WPF can be applied to predict widgets, together with its techniques, is presented. The evaluation of the Widget Prediction System (WPS) exposes many interesting insights into WPF and the methods taken to predict the widgets. Using fifty popular Websites to evaluate WPS (chapter 6), we have shown the proof-of-concepts for the WPF. Later, analysis of the results revealed insights into the issues identified during the evaluation and the trends of widget development. The effects resulting from the selected granularity of the prediction methods are shown, highlighting the importance of balancing the right amount of detection granularity. This is demonstrated by altering the concepts of the 'Next' and 'Previous' buttons from the Carousel and Slide Show widgets. A closer examination of the Popup Content widget results uncovered the evolutionary transformations of widget designs.

7.1 Contribution of the Thesis

This thesis reports the studies conducted to address the research questions set out in §1.3. A novel high-level approach to code comprehension over multiple computer languages to reverse engineer widgets is demonstrated. The approach analyses the Web page's source code scripted in JavaScript, HTML and CSS, to deduce the conceptual designs for a type of widget using the WPF. Since the source code of the Web page can be sourced and remixed from multiple locations, conjointly, scripted commonly with multiple computer languages in a Web page, the WPF uses a high-level approach to capture the abstract concepts to determine the widgets. We have successfully shown how to apply the framework by using the definition of different types of widget from the widget ontology and hardcoding them in WPS to predict widgets. Through this demonstration, design trends of widgets were exposed, along with the variants of styles developers take to develop the seven types of widgets. Consequently, this work contributes to the fields of Web Accessibility and Web engineering – affecting end-users of

all levels of technical knowledge, Web designers, authors and developers with different technical expertise. In the next three sections, a discussion of the research questions set out in §1.3 is presented from the perspectives of the contributions from the research.

7.1.1 Widgets Can Be Formally Modelled and Classified

In chapter 3 it was shown that using the framework suggested, Web widgets can be modelled by creating a widget ontology for the type of widget using the proposed taxonomy. Designing different types of widget ontologies was demonstrated using seven types of widget. In the demonstrations, it was shown how a widget can use components and tell-signs[1] to form a new widget ontology and, when required, new common or private components and tell-signs can be designed for the type of widget.

The proposed framework provides a stable framework for developers to model and communicate their design concepts among developers and end-users. Unlike design patterns, the framework in figure 3.3 provides the building blocks from the widget's superficial level (Widget layer), to the conceptual designs of its components (Components layer and Tell-Signs layer), and progresses as deep as the technical detail aspects (Code Constructs layer). Thus, designers, authors and developers can be involved with development of the widget at different stages of the project. Using the WIO paradigm, the documentation at this level will provide a uniform understanding of the design. Furthermore, this form of documentation will allow people with different technical ability to take on widget development, as well as being sufficiently abstract for end-users to have knowledge of the different kinds of widget.

Compliance is the cornerstone of the framework. From the proof-of-concepts that widgets can be formally modelled to assist development of our framework, a proposal to standardise the definition of different types of widget is suggested. When the consensuses between the different definitions of widgets are formed, the framework can provide both developers and end-users with a common platform to understand the definition of the different types of widget.

[1]Tell-signs are traces and clues of a component, often used to model components in the Components layer of our widget ontology paradigm.

7.1.2 Tell-Signs Can Be Discovered From the Web Page Source Code

The key to the prediction methodology is the tell-signs employed to detect the components of the widgets. Using the components discovered, design concepts of the widget can be derived. Indeed, there are numerous approaches to construct the design concepts in the source code. However, tell-signs allow these concepts to be listed at a finer granularity in our ontology. By identifying the instances of the tell-signs, deriving the widget's components from the tell-signs, this enables WPS to reconstruct the design concepts.

Chapter 4 has proven that instances of tell-signs can be identified from the Web page source code. To provide a sound approach to model widgets, tell-signs are grouped into two main types. The first type of tell-signs, known as common tell-signs, allows them to be reused by other components. The second type of tell-signs is specific (private tell-signs) for the pattern or concept of a component. Using the same concepts of model tell-signs, different types of widgets possess two types of component: common and private components. The method of modelling widgets allows each layer to be modelled independently (see figure 3.3), and encourages the objects in the layers to be reusable.

Analysis of the tell-signs popularity provides insights into the trends developers take when developing widgets. It demonstrates the feasibility of identifying tell-signs from the Web source code. The results from the tell-sign popularity analysis will not only facilitate future work on widget prediction, but also demonstrates that the approach of reverse engineering Web widgets can assist Website maintenance, by identifying the set of codes related to the widget for refactoring.

7.1.3 Widgets Prediction Can Be Done Automatically From the Web Page Source Code

The evaluation and analysed results for our prediction system (WPS), in chapter 6, demonstrated the feasibility of predicting widgets from the Web page source code. Predicting Web widgets not only informs the type of widgets available in the page, but also provides the location of the widgets. Reported by Brown [2012], it can be noticed from the evaluation transcripts that providing users with the knowledge of the type of widget was deemed helpful when they were interacting with the page. WPF provides a stable foundation to develop widget prediction systems as demonstrated by the seven

types of widgets covered. An average of 61.98% accuracy is achieved for WPS, while the prediction accuracy of two widgets attaining > 84%. This result demonstrates the framework feasibility as a concept. However, weaker prediction rates from Carousel and Slide Show widgets suggest an avenue for future research.

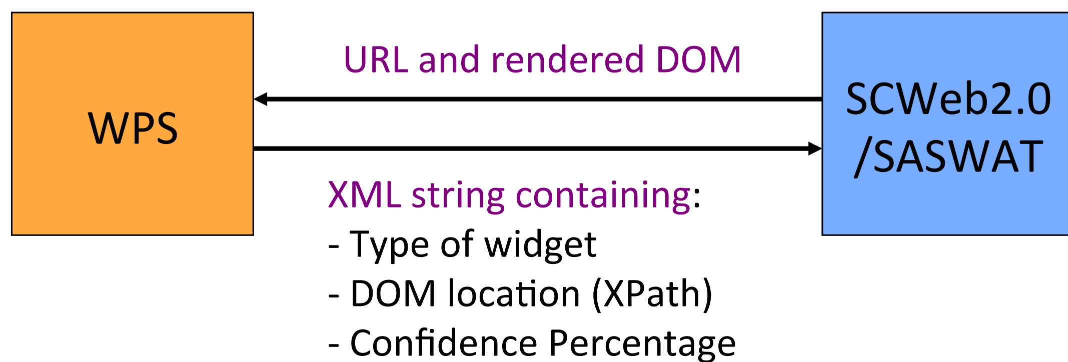


Figure 7.1: WPS interface as a service.

WPF can be applied as the backend engine for the SCWeb2.0 project to predict the area in the Web page where the widgets are found. The interfacing format between SCWeb2.0 and WPS is shown in figure 7.1. As a RESTful service, WPS can predict the location and type of the widget when provided with the Web page's Universal Resource Locator (URL) and the rendered DOM. The results will be returned in an XML string containing the XPath of the widget location, the type of widget predicted, and the confidence percentage for each prediction result. Providing the rendered DOM ensures that WPS analyses the Web page structure that is consistent with what is presented to the user.

To provide a rounded solution, the benefits of predicting widgets do not stop here. Developers can benefit from the prediction results to help identify areas to insert WAI-ARIA code, or create AxsJAX code. The approach can also be applied to provide validity checks for developers to ensure the widgets created conform to their design; an alternative solution to the Web engineering and reverse engineering issues pertaining to poor documentation.

7.2 Insights to WPS

The WPF successfully contributes approaches to predict widgets. Implementing WPF in WPS highlights that, for certain aspects in the widget prediction process, things can

be dealt with differently. This section provides insights into the problems and suggests possible approaches for future work of this nature.

Web widget design patterns are heavily dependent on the user's needs and trends. Often, when developing a widget, it is subjected to how the widget is applied, the current trends and the developer's perspective. These factors contribute mainly to the types and styles of coding the widgets that will indirectly affect the accuracy of the widget prediction. The Web is constantly evolving; this means that keeping ahead of the widget design trends will be a challenge. Commonly, projects and studies of the proposed research will always be one step behind the current trends. This means that frameworks like WPF have to be constantly revisited to align themselves.

Although a snapshot of every Web page analysed during our evaluation is captured for repeatability purposes, techniques such as Lazy Load and remote scripting prevent archiving and synthesising the entire set of code. Thus, during the "widget's inferences" phase (see figure 5.2) of the prediction process, parts of the code may not be available. Future work of this nature should research methods to identify the region in the code when these techniques are applied, so that simulation of the code can be done, to extract the set of code that is currently not available.

Unlike most computer languages, HTML allows inline coding in tags/elements for JavaScript and CSS. Although this provides easy access for developers to code their Web page/application, it is not a good programming practice that developers often use. Again, future work should consider this issue. Using JavaScript as an example, this is a sequential programming language, and inline coding will be dependent on when the code is executed in the Web page's rendered timeline. In certain cases, identifiers of variables may be assigned to different objects or values during a phase in the timeline. Thus, tracing the code for design concepts can be confusing and difficult.

We recommend that future work should not rely on the rendered DOM by the user agents. Although WPS does not rely on the DOM documentation for its prediction, it is a caution for future work, attempting to predict objects in the page, not to rely on the user agent's rendered DOM. Most popular Web browsers employ a Just-In-Time (JIT) approach to render the page. Thus, properties or attributes that are not available during the time of rendering are recorded as 'Undefined'. The term 'Undefined' by these browsers does not equate to absence of the object, but it means that the browser is not aware of it. This will be the same for elements with properties or attributes. 'Undefined' value does not equate to a value `false` or 0. Like WPS, synthesising the source code to derive the values is recommended.

7.3 Outstanding Issues

Although the work presented came up with significant insights and made contributions to the fields of Web Accessibility and Web engineering, a number of practical issues should be redressed. A list of four outstanding issues accompanied by a detailed discussion of the matter is presented.

Improving Weaker Widgets' Prediction Accuracy: The evaluation results exposed the strengths and weaknesses of our proposed approach. Weaker tell-sign detection was noticed from the evaluation results and should be redressed. It is not always the case that poor detection results for a tell-sign imply that the tell-sign was designed poorly. In many cases, the poor detection results are merely due to the trends of coding methods and choices taken due to developers' perspectives. Nevertheless, filtering through the weaker tell-signs to segregate the ones with issues is one way of enhancing the detection process. Identifying the tell-signs that need rectification for redressing the tell-signs or even removing them may improve the robustness of the prediction process. In some cases, new tell-signs will be required to fill the gap. However, more research is required to determine the robustness of the chosen tell-signs. It is not always the case that tell-signs are the only contributing factor that will affect the prediction accuracy. We cannot rule out the method of selecting the threshold confidence percentage. Changing the value for the threshold confidence percentage can affect the accuracy of the approach as well. Therefore, future investigation should explore the effects of this value.

Update Tell-Signs for HTML5 and Improved Compliancy With WAI-ARIA: A Technological Freeze is imposed on the studies covered in this thesis, so that the evolutionary process can be halted while investigations into the challenges surrounding widgets can be conducted. The Web is ever changing, and from the Technological Freeze dateline until now, it has evolved. New Web technologies such as HTML5, CSS3 and more formalised WAI-ARIA recommendations have been introduced since. Thus, the framework needs to be updated and refined to align itself with existing Web technologies.

Updating the Types of Widget Covered: Besides newer Web technologies being introduced, Web developers have also revolutionised their designs by including more complex interactivity and features to their Web pages. Some less popular widgets have evolved or disappeared, while the popular ones evolved into more

complex concepts. The technological freeze has provided a means of ceasing the constant changes to the Web for WPS evaluation. However, updating the widget ontology and definition is strongly recommended to include the current types of widget. Removal of older, less popular widgets is not recommended because some Web pages in the archives may still employ these widgets.

Including a Mapping Phase: A grey area between the Styling and Behaviour layers can affect the way widgets are designed. Indeed, normally the Behaviour layer handles events from a user or machine, but the Styling layer also provides some styling features for dynamic content. Using the Anchor tag `<a>` as an example, styles for pseudo-classes `hover`, `visited`, `active` and `link` can be manipulated in the Styling layer, without invoking the Behaviour layer. Developers can use this feature as a loophole to orchestrate the visibility of content. Anecdotal evidence from the evaluation results suggests that some developers used this technique to create the Collapsible Panel widget. This can be done by changing the element's `height` or `width` styling properties to achieve their design. To strengthen the current prediction architecture applied in WPS so that it includes detection of this form of coding, a "Mapping" phase is suggested. This phase will map the styles to each element, so that our widget prediction framework can pick up pseudo-classes in the Styling layer.

7.4 Future Research

The demonstration of predicting widgets to improve Web Accessibility, along with the modelling of different types of widget, provide a proof-of-concepts that should be used as an infrastructure for future research. Eight suggestions for future work are presented from the perspective of improving the proposed WPF approach, ideas to interface with different types of tools, and possible application venues for the approach.

Widget Prediction System (WPS) as a Service in the Cloud: Deploying WPS as a service in the Cloud will mean systems such as Assistive Technologies can easily tap in to take advantage of this service. Through this service, developers of Assistive Technologies can incorporate the concepts of WPF as part of their service and provide useful information about the widgets on the page. An example of parts of WPS services that can be hosted in the cloud include, instead of hard-coding the widget definitions, WIO can be queried directly using SPARQL. The

popularity of using the Cloud will allow WPS to be reached by more developers, and provide the capability of expanding the framework so that the robustness of the framework can be improved by covering a broader range of Websites. The agile development process will allow us to spread the test process of the framework to the users, as well as facilitating the evaluation of a wide range of Websites and issues. Through this process, generic issues can be highlighted and addressed as part of the expansion of the framework. Since Web pages are rendered depending on the type and version of the user-agent, hosting WPS as a service in the Cloud has to ensure that the analysed DOM is the same as the rendered DOM at the client-end.

Working with Assistive Tools: Currently, WPS returns the predicted results containing the type of widget, the confidence percentage and the XPath of where the widget is likely to be located. Although this result format is greatly driven by the motivation inspired by ScWeb2.0, it can be expanded for other tools and requirements. Tools that assist developers to insert AxsJAX or WAI-ARIA code may require more detailed information about the widget during their processes. However, further research should be done to improve our tell-sign detection process.

An Assistive Tool for Developers: Although the purpose of predicting a widget is presented as a means of improving the accessibility of Web content delivered by Web widgets in this thesis, developers may also find WPF useful for reverse engineering purposes for detecting areas in the code so that WAI-ARIA code can be inserted or AxsJAX code can be created. Although hosting it as a service in the Cloud is a viable source, more detailed information may be required for reverse engineering processes. Therefore, packaging it as part of the IBM Accessibility Tools Framework (ACTF) could enable the WPF to work more closely with the developers.

Better Determination of Related Elements for Conceptual Designs: A common issue raised by the evaluation results was that often the relationships between elements are difficult to determine without available documentation during the widget prediction process. Empirical observations gathered from the manual analysis suggest that the distance between the elements, or the structure of the elements, do not always provide sufficient evidence that two elements are related. Since issues pertaining to tracing code were highlighted, as well as parallel studies reported similar issues when attempting to reverse engineer design patterns

from the source code for desktop applications [Gamma et al., 1995; Dong et al., 2008b], this suggests further research to investigate other alternatives. Resolving this issue will assist the false negative (Typed II errors) and false positive (Typed I errors) predictions to be eliminated.

Identify When to Apply Prediction: Due to the popularity of the remote scripting technique, often determining the most appropriate time to execute the prediction process is ambiguous. Investigation into this type of challenge is vital to WPS's prediction process because it affects the accuracy of the approach. Anecdotal evidence from the manual analysis suggests that often researchers resort to using a timer at the start of their analysis process to overcome this problem. This is not ideal, and for Websites with a huge amount of content, this method is not appropriate.

Machine Learning Algorithms: Machine learning algorithms such as the Markov chain can be included to aid widget prediction. In future, when some key tell-signs are not available, predictions can be done to determine whether a tell-sign will exist based on the tell-signs discovered. Being able to predict tell-signs will further assist more accurate widget prediction when combined with our approach.

Assistant for Mobile Devices: As discussed in Yesilada et al. [2010], mobile users make similar errors to disabled desktop users. An interesting piece of future research could include investigations into how users of mobile devices react to tools that inform them about the different types of widgets in the Web page. These types of studies could understand the effects of how tools such as WPS can make a difference to mobile users' experience and productivity, and possibly open another usage venue.

Accessible Conceptual Designs: Inserting WAI-ARIA code attempts to improve accessibility to a widget through the widget's superficial level and the code construct aspect of the entire widget design building blocks (see figure 3.3). However, WPF addresses the entire design process including the conceptual aspects of the widget. Through our investigation, it can be noticed that little has been done to improve the accessibility of the conceptual design of a widget. Although this thought may sound difficult, in order to understand the operational purpose of a widget the user will have to know the cognition ability of the widget. This cognitive knowledge can be achievable via assisting the user to access the conceptual designs of the widget.

7.5 Summary

The work presented in this thesis has demonstrated the feasibility of WPF, and shown that it is applicable, adaptable and scalable. Our widget ontology paradigm allows developers to model their widget design concepts, and provides a platform to communicate their ideas to other developers and end-users. This paradigm can be approached from the superficial concepts or detailed code construct aspects of the widget modelled. Applying these concepts to WPS, widgets in the page can be predicted. Translating the concepts from the widget classification paradigm is not direct, and sometimes requires special techniques to overcome the accessibility barriers imposed by widgets. These techniques and methods applied were evaluated to prove the feasibility of the framework, and expose the weakness of the methods chosen. Finally, the evaluation results suggest that some of the practicalities should be redressed based on the recommendations suggested by the analysis conducted.

The contribution of the work presented has supplemented the fields of Web Accessibility and Web engineering. It demonstrated the feasibility of predicting widgets from the Web page source using our framework. The popularity of common techniques applied by developers, for different types of widgets, was presented with a discussion of how the evolution of widget designs will affect the developers and end-users. Discussions of the applicability of how this work can provide important information to Assistive tools and mobile devices were suggested to improve the accessibility of Web widgets. The beauty of the WPF allows people with different levels of technical knowledge to understand a type of widget. When applied to an end-user's tool, it can facilitate the tool with the necessary information about the widgets, thus indirectly providing a better user experience. For developers, WPS as a backend tool provides them with the knowledge of the widget in the page, allowing them to make important decisions during development or maintenance. When applied to an assistive development tool, it can help developers to identify the areas where WAI-ARIA can be inserted, or spot the parts of the widgets that require more conformance to guidelines and design concepts. The evaluation results exposed the weakness of our detection methods and suggested future work. Further analysis of tell-signs revealed the conceptual design trends, as well as coding trends taken up by developers and widget design libraries. The results reported will also provide reverse engineering processes with development trends for Website maintenance.

Bibliography

Abascal, J., Arrue, M., Fajardo, I., Garay, N., and Tomás, J. (2004). The use of guidelines to automatically verify web accessibility. *Universal Access in the Information Society*, 3(1):71–79.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Anttonen, M., Salminen, A., Mikkonen, T., and Taivalsaari, A. (2011). Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 800–807, New York, NY, USA. ACM.

Arvola, M. (2006). Interaction design patterns for computers in sociable use. *International Journal of Computer Applications in Technology*, 25(2/3):128–139.

Asakawa, C. and Takagi, H. (2000). Annotation-based transcoding for nonvisual web access. In *Assets '00: Proceedings of the fourth international ACM conference on Assistive technologies*, pages 172–179, New York, NY, USA. ACM.

Axelsson, J., Birbeck, M., Dubinko, M., Epperson, B., Ishikawa, M., McCarron, S., Navarro, A., and Pemberton, S. (2006). XHTML 2.0. `http://www.w3.org/TR/xhtml2`.

Axtell, N., Bacon, L., and Windall, G. (2007). COMPACT Web Design Approach: A Methodology and Modelling Technique for communicating the High-level Design of Struts Web Applications.

Bellucci, F., Ghiani, G., Paternò, F., and Porta, C. (2012). Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms. In *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*, IUI '12, pages 217–226, New York, NY, USA. ACM.

Benatallah, B., Dumas, M., Fauvet, M. C., Rabhi, F. A., and Sheng, Q. Z. (2002). Overview of some patterns for architecting and managing composite web services. *SIGecom Exchanges*, 3(3):9–16.

Bergman, M. K. (2001). The deep web: Surfacing hidden value. Digital Web Magazine, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.363.

Bernardi, M. L., Di Lucca, G. A., and Distante, D. (2008). Reverse engineering of Web Applications to abstract user-centered conceptual models. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 101–110.

Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R. C. (2005). Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 163–172, New York, NY, USA. ACM.

Borodin, Y., Bigham, J. P., Raman, R., and Ramakrishnan, I. V. (2008). What's new?: making web page updates accessible. In *Assets '08: Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*, pages 145–152, New York, NY, USA. ACM.

Bos, B. (2009). Cascading Style Sheets. http://www.w3.org/Style/CSS/.

Bosch, J. (1998). Design Patterns as Language Constructs. *Journal of Object-Oriented Programming*, 11(2).

Brown, A. (2012). Technical evaluation of the update classification system. WEL Technical Report, School of Computer Science, The University of Manchester. http://wel-eprints.cs.manchester.ac.uk/156/.

Brown, A. and Jay, C. (2008a). A Review of Assistive Technologies: Can users access dynamically updating information? WEL Technical Report, School of Computer Science, The University of Manchester. http://wel-eprints.cs.man.ac.uk/70/.

Brown, A. and Jay, C. (2008b). SASWAT Technical Requirements. WEL Technical Report, School of Computer Science, The University of Manchester. http://wel-eprints.cs.man.ac.uk/79/.

Brown, A., Jay, C., and Harper, S. (2009). Audio Presentation of Auto-Suggest Lists. In *Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibililty (W4A)*, W4A '09, pages 58–61, New York, NY, USA. ACM.

Brown, A., Jay, C., and Harper, S. (2010). Audio Access to Calendars. In *W4A 2010: Proceedings of the 2010 International Cross-Disciplinary Conference on Web Accessibility (W4A)*. ACM.

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

Caldwell, B., Cooper, M., Reid, L. G., Vanderheiden, G., Chisholm, W., Slatin, J., and White, J. (2008). Web Content Accessibility Guidelines (WCAG) 2.0. http://www.w3.org/TR/WCAG20.

Canfora, G., Cimitile, A., and Munro, M. (1996). An Improved Algorithm for Identifying Objects in Code. *Software: Practice and Experience*, 26(1):25–48.

Ceri, S., Fraternali, P., and Bongio, A. (2000). Web Modeling Language (WebML): a modeling language for designing web sites. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 137–157, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R. (1999). What are ontologies, and why do we need them? *Intelligent Systems and their Applications, IEEE*, 14(1):20–26.

Chen, A. Q. (2008). Web Evolution. Master's thesis, The University of Manchester, School of Computer Science, Kilburn Building, Oxford Road, Manchester, M13 9PL, UK.

Chen, A. Q. and Harper, S. (2008). Web Evolution: Method and Materials. WEL Technical Report, School of Computer Science, The University of Manchester. http://wel-eprints.cs.man.ac.uk/74/.

Chen, A. Q. and Harper, S. (2009). Identifying Web Widgets. WEL Technical Report, School of Computer Science, The University of Manchester. http://wel-eprints.cs.man.ac.uk/115/.

Chen, A. Q., Harper, S., Lunn, D., and Brown, A. (2013). Widget Identification: A High Level Approach. *World Wide Web*, 16(1):73–89.

Chen, B. and Shen, V. Y. (2006). Transforming web pages to become standard-compliant through reverse engineering. In *Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A): Building the mobile web: rediscovering accessibility?*, W4A '06, pages 14–22, New York, NY, USA. ACM.

Chen, C. L. and Raman, T. V. (2008). AxsJAX: a talking translation bot using google IM: bringing web-2.0 applications to life. In *Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A)*, W4A '08, pages 54–56, New York, NY, USA. ACM.

Chen, H., Ding, L., Wu, Z., Yu, T., Dhanapalan, L., and Chen, J. Y. (2009). Semantic web for integrated network analysis in biomedicine. *Briefings in Bioinformatics*, 10(2):177–192.

Chisholm, W., Vanderheiden, G., and Jacobs, I. (1999). Web Content Accessibility Guidelines 1.0. `http://www.w3.org/TR/WCAG10`.

Conallen, J. (2003). *Building Web Applications With UML*. Addison-Wesley, 2nd edition.

Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2001). DAML+OIL (March 2001) Reference Description. `http://www.w3.org/TR/daml+oil-reference`.

Cooper, M. (2007). Accessibility of emerging rich web technologies: web 2.0 and the semantic web. In *W4A '07: Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 93–98, New York, NY, USA. ACM.

Cooper, M. (2012). Indie UI Working Group Charter. `http://www.w3.org/2012/05/indie-ui-charter` Last accessed on 3rd September 2012.

Craig, J., Cooper, M., Pappas, L., Schwerdtfeger, R., and Seeman, L. (2009). Accesible Rich Internet Applications (WAI-ARIA) 1.0. `http://www.w3.org/TR/wai-aria`.

d'Aquin, M. and Motta, E. (2011). Watson, more than a Semantic Web search engine. *Semantic Web*, 2(1):55–63.

Deitel, P. and Deitel, H. (2007). *JAVA: How to Program*. Pearson Education, Inc., 7th edition.

Dencker, T., Fischer, C., and Röessler, A. (2008). Javascript client framework. United States Patent, `http://www.google.co.uk/patents?id=c5WpAAAAEBAJ`.

Di Lucca, G. A., Distante, D., and Bernardi, M. L. (2006). Recovering conceptual models from web applications. In *SIGDOC '06: Proceedings of the 24th annual ACM international conference on Design of communication*, pages 113–120, New York, NY, USA. ACM.

Di Lucca, G. A., Fasolino, A. R., and Tramontana, P. (2005a). Recovering Interaction Design Patterns in Web Applications. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 366–374.

Di Lucca, G. A., Fasolino, A. R., and Tramontana, P. (2005b). Web site accessibility: identifying and fixing accessibility problems in client page code. In *Web Site Evolution, 2005. (WSE 2005). Seventh IEEE International Symposium on*, pages 71–78.

Distante, D., Parveen, T., and Tilley, S. (2004). Towards a technique for reverse engineering Web transactions from a user's perspective. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 142–150.

Dixon, M. and Fogarty, J. (2010). Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 1525–1534, New York, NY, USA. ACM.

Dixon, M., Leventhal, D., and Fogarty, J. (2011). Content and hierarchy in Pixel-Based methods for Reverse-Engineering interface structure. In *CHI '11*.

Dong, J., Lad, D. S., and Zhao, Y. (2007). DP-Miner: Design Pattern Discovery Using Matrix. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 371–380.

Dong, J., Sun, Y., and Zhao, Y. (2008a). Design pattern detection by template matching. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 765–769, New York, NY, USA. ACM.

Dong, J. and Zhao, Y. (2007). Experiments on Design Pattern Discovery. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 12+, Washington, DC, USA. IEEE Computer Society.

Dong, J., Zhao, Y., and Peng, T. (2008b). A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*.

Dong, J., Zhao, Y., and Sun, Y. (2009). XSLT-based evolutions and analyses of design patterns. *Software: Practice and Experience*, 39(8):773–805.

ECMA-International (2009). Standard ECMA-262:ECMAScript language specification, 5th edition. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

Fazzinga, B. and Lukasiewicz, T. (2010). Semantic search on the Web. *Semantic Web*, 1(1):89–96.

Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D., and Patel-Schneider, P. (2001). Oil: an ontology infrastructure for the semantic web. *Intelligent Systems, IEEE*, 16(2):38–45.

Finkelstein, A., Savigni, A., Kappel, G., Retschitzegger, W., Schwinger, W., and Feichtner, C. (2002). Ubiquitous Web Application Development - A Framework for Understanding. In *Proceedings of 6th World Multiconference On Systemics, Cybernetics And Informatics*, volume 1, pages 431–438.

Firebug (2011). Firebug: Web development evolved. `http://getfirebug.com/`. Accessed on 22 March 2011.

Flanagan, D. (2002). *JavaScript: The Definitive Guide*. O'Reilly, 4th edition.

Fowler, M. (1996). *Analysis Patterns Reusable Object Models*. Addison Wesley.

Gal, A., Probst, C. W., and Franz, M. (2006). HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garzotto, F., Paolini, P., and Schwabe, D. (1993). HDM—a model-based approach to hypertext application design. *ACM Transactions on Information Systems (TOIS)*, 11(1):1–26.

Guha, A., Krishnamurthi, S., and Jim, T. (2009). Using static analysis for Ajax intrusion detection. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 561–570, New York, NY, USA. ACM.

Gwardak, L. and Påhlstorp, L. (2007). Exploring Usability Guidelines for Rich Internet Applications. Master's thesis, Department of Informatics, Lund University.

Hanson, V. L. (2009). Age and Web Access: The Next Generation. In *W4A '09: Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibility (W4A)*, pages 7–15. ACM.

Hanson, V. L. and Richards, J. T. (2005). Achieving a More Usable World Wide Web. *Behaviour & Information Technology*, 24(3):231–246.

Hardesty, J. L. (2011). Bells, whistles, and alarms: HCI lessons using AJAX for a page-turning web application. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 827–840, New York, NY, USA. ACM.

Harper, S. (2008). Web evolution and its importance for supporting research arguments in web accessibility. In De Roure, D. and Hall, W., editors, *Proceedings of the First International Workshop on Understanding Web Evolution (WebEvolve2008): A prerequisite for Web Science*, pages 51–54, Southampton, UK. Web Science Research Initiative.

Harper, S., Bechhofer, S., and Lunn, D. (2006). Taming the inaccessible web. In *Proceedings of the 24th annual ACM international conference on Design of communication*, SIGDOC '06, pages 64–69, New York, NY, USA. ACM.

Harper, S. and Chen, A. Q. (2012). Web Accessibility Guidelines: A Lesson From The Evolving Web. *World Wide Web*, 15:61–88.

Harper, S. and Yesilada, Y. (2008). Web accessibility and guidelines. In Harper, S. and Yesilada, Y., editors, *Web Accessibility: A Foundation for Research*, pages 61–78. Springer.

Hartley, A. A. (1992). Attention. In Craik, F. I. and Salthouse, T. A., editors, *The Handbook of Aging and Cognition*, chapter 1, pages 3–53. Laurence Erlbaum Associates. ISBN: 0–8058-0713-6.

Heflin, J., Hendler, J., and Luke, S. (1999). SHOE: A knowledge representation language for internet applications. Technical Report CS-TR-4078, Department of Computer Science, University of Maryland. http://hdl.handle.net/1903/1044.

Hendrix, D., Cross, J. H., and Maghsoodloo, S. (2002). The effectiveness of control structure diagrams in source code comprehension activities. *Software Engineering, IEEE Transactions on*, 28(5):463–477.

Henry, S. L. and May, M. (2008). Authoring Tool Accessibility Guidelines (ATAG) Overview. http://www.w3.org/WAI/intro/atag.php.

Henry, S. L. and May, M. (2012). User Agent Accessibility Guidelines (UAAG) Overview. http://www.w3.org/WAI/intro/uaag.php.

Hickson, I. and Hyatt, D. (2009). HTML 5. http://www.w3.org/TR/html5.

HiSoftware Inc. (2009). HiSoftware Cynthia Says. http://www.contentquality.com.

Hitzler, P., Krotzsch, M., and Rudolph, S. (2011). *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC.

Hopkins, N. (2011). jsTracer: Debugging Web 2.0. http://jstracer.sourceforge.net/. Accessed on 22 March 2011.

Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7 – 26.

IBM (2009). RAVEn On Accessibility. http://www-03.ibm.com/able/resources/raven.html.

Jacobs, I. and Brewer, J. (1999). Accessibility Features of CSS. http://www.w3.org/Style/CSS/.

Jay, C., Brown, A., and Harper, S. (2010). Internal Evaluation of the SASWAT Audio Browser. WEL Technical Report 6, School of Computer Science, The University of

Manchester. SASWAT Technical Report. http://wel-eprints.cs.manchester.ac.uk/125/.

Karanam, S., Oostendorp, H., Melguizo, M. C., and Indurkhya, B. (2011). Interaction of textual and graphical information in locating web page widgets. *Behaviour & Information Technology*, pages 1–13.

Kim, S. Y., Nam, S. W., Lee, S. H., Park, W. S., Yoo, N. J., Lee, J. Y., and Chung, Y.-J. (2005). ArrayCyGHt: a web application for analysis and visualization of array-CGH data. *Bioinformatics*, 21(10):2554–2555.

Koch, N. and Kraus, A. (2002). The Expressive Power of UML-based Web Engineering. http://dx.doi.org/http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7588.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*, page 7. Addison-Wesley, Boston, MA, 2nd edition.

Kurniawan, S., King, A., Evans, D., and Blenkhorn, P. (2006). Personalising Web Page Presentation for Older People. *Interacting with Computers*, 18(3):457–477. Human Factors in Personalised Systems and Services.

Larman, C. (1998). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR.

Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M., and Kandogan, E. (2007). Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 943–946, New York, NY, USA. ACM.

Liu, S. S. and Wilde, N. (1990). Identifying objects in a conventional procedural language: an example of data design recovery. In *Software Maintenance, 1990., Proceedings., Conference on*, pages 266–271.

Lunn, D. (2009). *Towards Behaviour-Driven Transcoding of Web Content Through an Analysis of User Coping Strategies*. PhD thesis, The University of Manchester, School of Computer Science, Oxford Road, Manchester, M13 9PL, UK.

Lunn, D. and Harper, S. (2010). Using Galvanic Skin Response Measures To Identify Areas of Frustration for Older Web 2.0 Users. In *W4A '10: Proceedings of the 2010 International Cross-Disciplinary Conference on Web Accessibility (W4A)*. ACM.

Lunn, D. and Harper, S. (2011). Providing assistance to older users of dynamic web content. *Computers in Human Behavior*, 27(6):2098–2107.

Lunn, D., Harper, S., and Bechhofer, S. (2009a). Combining SADIe and AxsJAX to improve the accessibility of web content. In *Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibililty (W4A)*, W4A '09, pages 75–78, New York, NY, USA. ACM.

Lunn, D., Yesilada, Y., and Harper, S. (2009b). Barriers Faced by Older Users On Static Web Pages: Criteria Used In The Barrier Walkthrough Method. WEL Technical Report WP1D1, School of Computer Science, The University of Manchester. http://wel-eprints.cs.manchester.ac.uk/108/ SCWeb2 Technical Report.

Mäkelä, E., Viljanen, K., Alm, O., Tuominen, J., Valkeapää, O., Kurki, J., Sinkkilä, R., Lindroos, R., Suominen, O., Ruotsalo, T., and Hyvönen, E. (2007). Enabling the Semantic Web with Ready-to-Use Web Widgets. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.3018.

Martín, A., Rossi, G., Cechich, A., and Gordillo, S. (2010). Engineering Accessible Web Applications. An Aspect-Oriented Approach. *World Wide Web*, 13:419–440.

Maurer, D. (2006). Usability for rich internet applications. Digital Web Magazine, http://www.digital-web.com/articles/usability_for_rich_internet_applications/.

Maurer, F. and Martel, S. (2002). Extreme programming. Rapid development for Web-based applications. *IEEE Internet Computing*, 6(1):86–90.

Memmert, D. (2006). The Effects of Eye Movements, Age, and Expertise on Inattentional Blindness. *Consciousness and Cognition*, 15(3):620–627.

Mikkonen, T. (1998). Formalizing Design Patterns. In *Software Engineering, International Conference on*, volume 0, Los Alamitos, CA, USA. IEEE Computer Society.

Millard, D. E. and Ross, M. (2006). Web 2.0: hypertext by any other name? In *Proceedings of the seventeenth conference on Hypertext and hypermedia*, HYPERTEXT '06, pages 27–30, New York, NY, USA. ACM.

Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28.

O'Neill, E. T., Lavoie, B. F., and Bennett, R. (2003). Trends in the evolution of the public web (1998 - 2002). *D-Lib Magazine*, 9(4).

O'reilly, T. (2007). What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Social Science Research Network Working Paper Series*.

Overmyer, S. P. (2000). What's Different about Requirements Engineering for Web Sites? *Requirements Engineering*, 5(1):62–65.

Power, C. and Petrie, H. (2007). Accessibility in non-professional web authoring tools: a missed web 2.0 opportunity? In *W4A '07: Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 116–119, New York, NY, USA. ACM.

Presutti, V. and Gangemi, A. (2008). Content Ontology Design Patterns as Practical Building Blocks for Web Ontologies. In Li, Q., Spaccapietra, S., Yu, E., and Olivé, A., editors, *Conceptual Modeling - ER 2008*, volume 5231 of *Lecture Notes in Computer Science*, chapter 11, pages 128–141. Springer Berlin / Heidelberg, Berlin, Heidelberg.

Rajlich, V. (2000). Incremental redocumentation using the Web. *Software, IEEE*, 17(5):102–106.

Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, volume 0, pages 271–278, Los Alamitos, CA, USA. IEEE Computer Society.

Rhino (2011). Rhino: Javascript for java. `http://www.mozilla.org/rhino/`. Accessed on 23 March 2011.

Richards, J. T. and Hanson, V. L. (2004). Web accessibility: a broader view. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 72–79, New York, NY, USA. ACM.

Ridpath, C. and Chisholm, W. (2000). Techniques For Accessibility Evaluation and Repair Tools. `http://www.w3.org/TR/AERT/`.

Rossi, G., Urbieta, M., Ginzburg, J., Distante, D., and Garrido, A. (2008). Refactoring to Rich Internet Applications. A Model-Driven Approach. In *International Conference on Web Engineering*, volume 0, pages 1–12, Los Alamitos, CA, USA. IEEE Computer Society.

Sàenz, M., Buracas, G. T., and Boynton, G. M. (2003). Global Feature-Based Attention for Motion and Color. *Vision Research*, 43(6):629 – 637.

Schwerdtfeger, R. (2012). HTML5 accessibility Coming soon are you ready? `http://www-03.ibm.com/able/news/html5.html` Last accessed on 3rd September 2012.

Shawn (2009). Web Accessibility Initiative. `http://www.w3.org/WAI/`.

Stencel, K. and Wegrzynowicz, P. (2008). Detection of Diverse Design Pattern Variants. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 25–32.

Takagi, H., Asakawa, C., Fukuda, K., and Maeda, J. (2002). Site-wide annotation: reconstructing existing pages to be accessible. In *Assets '02: Proceedings of the fifth international ACM conference on Assistive technologies*, pages 81–88, New York, NY, USA. ACM.

Thatcher, J., Waddell, C., Henry, S., Swierenga, S., Urban, M., Burks, M., and Bohman, P. (2003). *Constructing Accessible Web Sites*. Apress.

Thiessen, P. and Chen, C. (2007). Ajax live regions: chat as a case example. In *W4A '07: Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 7–14, New York, NY, USA. ACM.

Tilley, S. and Huang, S. (2001). Evaluating the reverse engineering capabilities of web tools for understanding site content and structure: a case study. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 514–523, Washington, DC, USA. IEEE Computer Society.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design Pattern Detection Using Similarity Scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909.

V8 (2011). V8 javascript engine. http://code.google.com/apis/v8/. Accessed on 23 March 2011.

Vlissides, J. M., Coplien, J. O., and Kerth, N. L. (1996). *Pattern Languages of Program Design 2*. Addison-Wesley Publishing Company.

Vora, P. (2009). *Web Application Design Patterns*, chapter Pattern Libraries, pages 389–403. Morgan Kaufmann Publishers.

WebAIM (2009). WAVE. http://wave.webaim.org.

xhtml.com (2008). X/HTML 5 Versus XHTML 2. http://xhtml.com/en/future/x-html-5-versus-xhtml-2/.

Yesilada, Y., Harper, S., Chen, T., and Trewin, S. (2010). Small-device users situationally impaired by input. *Computers in Human Behavior*, 26(3):427–435.

Zakas, N. C. (2010). *High Performance JavaScript*. O'Reilly.

Appendix A

Code Constructs Objects

Assignment_CodeConstruct The set of different kinds of assignment to a variable in the code.

ArrayLength_Assignment_CodeConstruct This refers to instances within the code that the length/size of the array is assigned to a variable. We search for the following pattern "`= array.length`".

Zero_Assignment_CodeConstruct This refers to instances within the code that the numeric '0' is assigned to a variable. We search for the following pattern "`= 0`".

ConditionalOperator_CodeConstruct This Code Constructs class refers to a pattern commonly used by developers to incorporate a condition when assigning the variable with a value/object/instance. An example of the pattern for the code construct is `Variable = (conditions)? ... : ...;`

DOMElements_CodeConstruct The super set that refers to the different types of elements in the DOM. This can include term given to refer to a group of different types of elements.

AnchorElements_DOMElements_CodeConstruct This is the Anchor elements in the DOM document. i.e., `<a>...</a>`.

ListElements_DOMElements_CodeConstruct This Code Construct object refers to a group of elements in the DOM that is a subset of a parent element such as Ordered(`<ol>`), Unordered(`<ul>`), and Definition(`<dl>`) List elements, nested `<div>` elements, nested Anchor elements (`<a>`), and nested `<p>` elements.

TextArea_DOMElements_CodeConstruct This DOM element is usually declared within the form element (`<form>...</form>`) that allow the user to input a lot of text when filling up the form. Normally the textarea element (`<textarea>...</textarea>`) consist of more than one row with multiple columns.

TextField_DOMElements_CodeConstruct This DOM element is usually declared within the form element (`<form>...</form>`) that allow the user to input text when filling up the form. It uses the input element with it type property set to "text" – i.e., `<input type="text">`. Normally this element allows the user to input lesser text than the textarea element, and only has one row with multiple columns.

Expression_CodeConstruct This is a set of expression to check if an object in the code meets certain criterial.

isArrayLength_Expression_CodeConstruct This a control condition to check if the value is the same as the array length.

isZero_Expression_CodeConstruct This is a control condition to check if the value is zero.

Events_CodeConstruct The is a set of code constructs that relates to the monitoring of different type of events that happens within the page session.

AddListener_Events_CodeConstruct This is an instance where event listeners are added to monitor a event on an element.
i.e., `element.addEventListener(event, handler, boolean).`

ChangeKeyword_Events_CodeConstruct This refers to the keyword 'change' to assist searching for variables assigned to monitor this event.

ClickKeyword_Events_CodeConstruct This refers to the keyword 'click' to assist searching for variables assigned to monitor this event.

MouseoverKeyword_Events_CodeConstruct This refers to the keyword 'mouseover' to assist searching for variables assigned to monitor this event.

onChangeAttr_Events_CodeConstruct This refers to the attribute 'onChange' declared in the DOM with the element to monitor the onChange event for that particular element.

onChangeProperty_Events_CodeConstruct This refers to the 'onChange' property of an element in the code; declared to monitor the changes of content in that particular element.

onClickAttr_Events_CodeConstruct This refers to the attribute 'onClick' declared in the DOM with the element to monitor the onClick event for that particular element.

onClickProperty_Events_CodeConstruct This refers to the 'onClick' property of an element in the code; declared to monitor if that particular element is clicked.

onMouseoverAttr_Events_CodeConstruct This refers to the attribute 'onMouseover' declared in the DOM with the element to monitor the onMouseover event for that particular element.

onMouseoverProperty_Events_CodeConstruct This refers to the 'onMouseover' property of an element in the code; declared to monitor mouse is hovering over the element.

ifStatements_CodeConstruct This refers to a code pattern structure where the 'if' statement is used in the code. i.e., `if(Expression)` {...}

KeyCode_CodeConstruct This is a set of conditions to check the key codes referring to a particular key on the keyboard.

Check38_KeyCode_CodeConstruct This checks if the 'Up' key is monitored in the code. In this case the key code is 38.

Check40_KeyCode_CodeConstruct This checks if the 'Down' key is monitored in the code. In this case the key code is 40.

Keyword_CodeConstruct This is a set of keywords to be search within a specified area in the code.

Begin_Keyword_CodeConstruct Within a specific area in the code look for the 'Begin' word.

End_Keyword_CodeConstruct Within a specific area in the code look for the 'End' word.

First_Keyword_CodeConstruct Within a specific area in the code look for the 'First' word.

Last_Keyword_CodeConstruct Within a specific area in the code look for the 'Last' word.

Start_Keyword_CodeConstruct Within a specific area in the code look for the 'Start' word.

ManipulateDOMElements_CodeConstruct The set of properties that are commonly used to manipulate the DOM elements.

Hide_ManipulateDOMElements_CodeConstruct This is a set of instances that changes a particular DOM element's styling property so that the element becomes not visible to the user. This concept can be done either by changing the styling visibility property to hide (`element.style.visibility = hide`) or the styling display property to none (`element.style.display = none`).

JSAppendDOM_ManipulateDOMElements_CodeConstruct This class refers to an instance in the code where it attempts to append a DOM document node. i.e., `element.appendChild(new node to be inserted).`

MakeVisible_ManipulateDOMElements_CodeConstruct This is the opposite to *Hide_ManipulateDOMElements_CodeConstruct* class where here we are checking that a particular DOM element's styling property is changed, so that the element becomes visible to the user.
i.e., `element.style.visibility = visible` and `element.style.display = block`.

OffAutoCompleteAttr_ManipulateDOMElements_CodeConstruct This is a series of pattern in the code that changes the text field or textarea element's property – autocomplete to off. By doing this it informs the browser not to suggest words that were previous entered.

SetHigherOrder_ManipulateDOMElements_CodeConstruct An instance in the code that changes the styling property of an element so that it is high visibility order. i.e., `element.style.zIndex="1";`

UpdateInnerHTMLProperty_ManipulateDOMElements_CodeConstruct This instance of code assigns the DOM element with a new set of content. i.e., `element.innerHTML = ...`

MinusSign_CodeConstruct This code construct is an instance of a '−' sign within the code.

PlusSign_CodeConstruct This code construct is an instance of a '+' sign within the code.

Variable_CodeConstruct This is an instance of a declaration or assignation of a variable in the code.

ArrayObject_Variable_CodeConstruct This refers to a set of code patterns that declares a variable as an array. This can be in the form of `variable = newArray()` or `variable[...]`.

Window_CodeConstruct Instances of code that is used to manipulate the Window object.

Open_Window_CodeConstruct This set of code launches a new browser window. i.e., `window.open(...)`.

SetTimeOut_Window_CodeConstruct This set of code sets a delay for an event to occur to the browser window. i.e., `setTimeout(...)`.

Appendix B

Results for Profiler's 10K Character Size Evaluation

#	Identifier	Location	Start	End	Input Arguments
1	_gjuc	1437	1453	1679	NULL
2	_gjp	1680	1695	1754	NULL
3	_gjp	2521	2536	2585	NULL
4	wgjp	2855	2870	3022	NULL
5	n	3737	3749	3786	NULL
6	c	4762	4774	4775	NULL
7	c	5883	5898	5944	F,G
8	ml	277	290	291	NULL
9	time	316	331	358	NULL
10	log	360	380	560	c,d,b
11	a.onabort	440	461	473	NULL
12	l	278	290	291	NULL
13	e	319	331	358	NULL
14	window.onpopstate	744	772	780	NULL
15	c[a]	896	911	936	NULL
16	window.google.startTick	1061	1098	1146	a,b
17	window.google.tick	1148	1182	1245	a,b,c
18	google.x	1769	1791	1825	e,g
19	window.rwt	1827	1863	2278	a,f,g,k,l,h,c,m
20	qs	2294	2307	2308	NULL

21	tg	2310	2324	2398	e
22	g	3496	3513	3628	b,c,a
23	google.med	3787	3809	3981	b
24	google.register	3984	4013	4121	b,c
25	google.save	4123	4148	4316	b,c
26	google.initHistory	4318	4347	4428	NULL
27	google.exportSymbol	4474	4509	4684	a,b,c
28	google.exportProperty	4686	4723	4730	a,b,c
29	google.inherits	4732	4761	4833	a,b
30	o	4930	4943	5830	a
31	p	5867	5882	6409	a,b
32	google.browser.isEngineVersion	6500	6542	6567	a
33	google.browser.isProductVersion	6569	6612	6637	a
34	u	6683	6696	6766	a
35	v	6768	6783	7027	a,b
36	w	7029	7042	7080	a
37	create	7104	7124	7225	a,b
38	get	7227	7244	7264	a,b
39	insert	7285	7307	7361	a,b,c
40	remove	7363	7381	7433	a
41	set	7435	7450	7736	a
42	google.listen	7739	7768	7837	a,b,c
43	google.unlisten	7839	7870	7945	a,b,c
44	A	7956	7975	8251	a,b,c,d
45	d	8083	8096	8109	y
46	listen	8266	8283	8384	NULL
47	unlisten	8386	8405	8549	NULL
48	D	8604	8621	8897	a,b,c
49	E	8899	8911	9182	NULL
50	H	9184	9196	9270	NULL
51	I	9272	9287	9481	a,b
52	search	9501	9521	9840	a,b
53	getQuery	9885	9904	9918	NULL
54	J	9925	9937	9990	NULL
55	NULL	717	728	942	NULL
56	NULL	883	895	937	a
57	NULL	2374	2385	2396	NULL

58	NULL	3344	3355	3461	NULL
59	NULL	3467	3478	4430	NULL
60	NULL	4436	4447	NULL	NULL

Table B.1: The Functions results extracted from the profiler for google.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	window.google	17	31	715
2	window.chrome	966	980	981
3	window.google.timers	1037	1058	1059
4	i[a]	1099	1104	1145
5	google.y	1757	1766	1767
6	window.gbar	2281	2293	2399
7	o	2329	2331	2341
8	google.j[1]	2586	2598	2820
9	e	29	31	715
10	c[d]	4677	4682	4683
11	k	4839	4841	4861
12	m	4863	4865	4897
13	google.browser.engine	5621	5643	5703
14	google.browser.product	5705	5728	5829
15	google.dom	7083	7094	7737
16	z	7951	7953	7954
17	google.msg	8254	8265	8557
18	google.nav	9484	9495	9919
19	d	9565	9567	9568

Table B.2: The Objects results extracted from the profiler for google.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory

1	a	7788	b	c	google.listen at 7739 with no callee.

Table B.3: The Listener results extracted from the profiler for google.com when evaluating over 10K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	run_if_loaded	893	920	953	a,b
2	run_with	954	978	1031	b,a,c
3	wait_for_load	1032	1061	1447	c,b,e
4	bind	1448	1466	1699	c,b
5	env_get	1731	1750	1777	a
6	hasArrayNature	1902	1928	2114	a
7	$A	2115	2129	2271	b
8	eval_global	2273	2296	2673	c
9	copy_properties	2675	2704	2888	b,c
10	add_properties	2889	2917	2970	a,b
11	is_empty	2971	2991	3120	b
12	Arbiter	3195	3213	3339	NULL
13	window.async_callback	3126	3183	3193	a,b
14	subscribe	3746	3771	4275	k,b,i
15	unsubscribe	4277	4300	4619	e
16	inform	4621	4643	5419	i,c,b
17	query	5421	5438	5563	b
18	_getInstance	5580	5604	5717	a
19	registerCallback	5719	5749	6261	b,d
20	_updateCallbacks	6263	6293	6560	d,c
21	Function.prototype.deferUntil	6565	6612	6811	a,h,b,i
22	c	6683	6695	6778	NULL
23	configurePage	6959	6984	7389	b

24	loadComponents	7391	7419	7593	d,b
25	loadResources	7595	7626	8329	h,b,g,k
26	_fetchWithIframe	8331	8359	8793	d
27	_addResourceToIframe	8795	8827	9198	e
28	requestResource	9200	9231	9888	j,g,e
29	_runCSSPolls	9926	9949	NULL	NULL
30	NULL	712	724	745	a
31	NULL	853	865	891	a
32	NULL	1280	1291	1424	NULL
33	NULL	1519	1530	1697	NULL
34	NULL	6918	6930	NULL	a

Table B.4: The Functions results extracted from the profiler for facebook.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	__rm	145	152	153
2	window[a]	2953	2963	2964
3	h	4717	4719	4720
4	i	5924	5926	5927
5	a._listen[j]	6061	6074	6075
6	a._callbacks[h]	6188	6204	6250
7	h	4717	4719	4720
8	this._cssLinkMap[f]	7295	7315	7325
9	e	7674	7676	7677
10	this._earlyResources	7845	7866	7867
11	this._cssLinkMap[e]	9677	9697	9713
12	this._cssLinkMap[e]	9677	9697	9713

Table B.5: The Objects results extracted from the profiler for facebook.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for facebook.com when evaluating over 10K character size.

#	Identifier	Location	Start	End	Input Arguments
1	checkChromePromoAlert	2763	2796	2977	NULL
2	dismissChromePromoAlert	3038	3073	3224	NULL
3	hideFbPromoAlert	5982	6010	6049	NULL
4	isRtl	7054	7071	7114	NULL
5	setRtlYva	7119	7146	9617	suffix
6	yt.timing.tick	86	129	359	label, opt_time
7	yt.timing.info	369	409	549	label, value
8	gaiaChangedCallback	3892	3934	4388	autoshare
9	fn	4230	4246	4341	NULL
10	canConnectCallback	4462	4503	4778	autoshare
11	serviceChangedCallback	4850	4895	5385	autoshare
12	window.dismissFbPromoAlert	5809	5849	5972	NULL
13	fixYvaDom	7793	7822	8142	suffix
14	richMedia.clipFlashObject	8644	8753	9096	asset, width, height, offsetTop, offsetRight, offsetBottom, offsetLeft
15	richMedia.unclipFlashObject	9171	9232	9560	asset, width, height
16	NULL	3258	3270	6061	NULL
17	NULL	3276	3288	6052	NULL
18	NULL	6483	6495	6567	NULL

Table B.6: The Functions results extracted from the profiler for youtube.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects and event listeners were found for facebook.com when evaluating over 10K character size.

#	Identifier	Location	Start	End	Input Arguments
1	c	2756	2773	2896	r,s,q
2	o	2897	2914	3032	r,s,q

3	enable	3048	3065	3108	NULL
4	disable	3110	3128	3171	NULL
5	dispatch	3173	3192	3261	NULL
6	reset	3263	3279	3285	NULL
7	a	5457	5478	6172	A,l,u,n,h
8	e.augment	6376	6405	6841	f,w,j,u,o
9	k[i]	6537	6552	6672	NULL
10	e.aggregate	6843	6872	6902	h,g,f,i
11	e.extend	6904	6930	7182	i,h,f,k
12	e.each	7184	7208	7390	i,h,j,g
13	e.clone	7392	7421	7914	j,k,n,p,i,m
14	e.bind	7916	7936	8115	g,i
15	e.rbind	8117	8138	8317	g,i
16	before	8502	8526	8636	i,k,l,m
17	after	8638	8661	8771	i,k,l,m
18	_inject	8773	8798	9035	h,j,k,m
19	k[m]	8906	8921	8961	NULL
20	detach	9037	9055	9081	h
21	_unload	9083	9104	9105	i,h
22	e.Do.Method	9108	9133	9209	h,i
23	e.Do.Method.prototype.register	9211	9257	9304	i,j,h
24	e.Do.Method.prototype._delete	9306	9347	9391	h
25	e.Do.Method.prototype.exec	9393	9430	9926	NULL
26	e.Do.AlterArgs	9928	9956	9983	i,h
27	NULL	185	198	2118	NULL
28	NULL	2158	2169	2548	NULL
29	NULL	2588	2599	3288	NULL
30	NULL	5404	5416	6204	g
31	NULL	6308	6320	8319	e
32	NULL	6522	6536	6770	r,i
33	NULL	7742	7756	7846	o,f
34	NULL	7867	7881	7894	o,f
35	NULL	7999	8010	8113	NULL
36	NULL	8201	8212	8315	NULL
37	NULL	8417	8429	NULL	e
38	NULL	8464	8475	NULL	NULL

Table B.7: The Functions results extracted from the profiler for yahoo.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	k	2747	2749	2750
2	j.OnloadCache	3033	3047	3286
3	r	3197	3199	3211
4	k	2747	2749	2750
5	g	6499	6501	6502
6	k	2747	2749	2750
7	q	6509	6511	6512
8	e.Env.evt	8430	8440	8462
9	e.Do	8488	8493	9106
10	this.objs[n]	8838	8851	8852
11	this.before	9180	9192	9193
12	this.after	9195	9206	9207

Table B.8: The Objects results extracted from the profiler for yahoo.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for yahoo.com when evaluating over 10K character size. No Functions were found for live.com when evaluating over 10K character size.

#	Identifier	Location	Start	End
1	srf_oTemplate	3886	none	none

Table B.9: The Objects results extracted from the profiler for live.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for live.com when evaluating over 10K character size.

#	Identifier	Location	Start	End	Input Arguments
1	gaia_onLoginSubmit	6764	6794	6903	NULL
2	window.onload	6373	6401	6662	NULL

Table B.10: The Functions results extracted from the profiler for blogger.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects and event listeners were found for blogger.com when evaluating over 10K character size.

#	Identifier	Location	Start	End	Input Arguments
1	addEV	644	665	767	C,B,A
2	G	768	781	808	A
3	F	1560	1572	1692	NULL
4	H	1693	1708	2431	W,S
5	U	1843	1855	2082	NULL
6	J	2432	2445	2541	S
7	C	2542	2555	2676	T
8	I	2955	2968	3002	C
9	K	3003	3016	3049	C
10	S	3050	3063	3163	C
11	U	3164	3177	3245	C
12	P	3246	3263	3374	G,X,C
13	N	3375	3388	3439	C
14	R	3440	3453	3665	X
15	H	3666	3679	3874	G
16	C	3777	3792	3873	Z,b
17	O	3875	3888	4054	Y
18	L	4055	4068	4278	G
19	C	4411	4424	4531	b
20	G	4532	4545	4639	Y
21	Z	4809	4821	4880	NULL

22	e	4881	4894	5304	n
23	k	5305	5317	5514	NULL
24	l	5515	5527	5547	NULL
25	g	5548	5560	5577	NULL
26	i	5578	5590	5667	NULL
27	b	5668	5681	5733	n
28	G	5734	5747	5793	n
29	n	6642	6654	6716	NULL
30	e	6717	6729	6907	NULL
31	i	6908	6920	6972	NULL
32	G	6973	6985	7009	NULL
33	b	7010	7023	7111	q
34	c	7112	7125	7221	q
35	f	7222	7235	7390	q
36	X	7391	7403	7627	NULL
37	Y	7628	7643	7778	r,q
38	p	7779	7794	7882	q,s
39	j	7883	7895	8347	NULL
40	Z	8348	8360	8454	NULL
41	k	8455	8467	8736	NULL
42	g	8737	8749	9007	NULL
43	G	9401	9413	9514	NULL
44	X	9531	9543	9558	NULL
45	Y	9559	9572	9603	Z
46	X	9803	9816	9919	C
47	Y	9920	9933	9979	C
48	a[i].onclick	246	269	426	NULL
49	w.onunload	2738	2759	2760	NULL
50	ini	4647	4662	4701	X
51	bdsug.sugkeywatcher.on	5843	5876	5969	NULL
52	bdsug.sugkeywatcher.off	5971	6005	6101	NULL
53	rm	6222	6236	6563	n
54	rm	9022	9036	9351	q
55	rm	9618	9632	9768	Z
56	NULL	428	439	516	NULL
57	NULL	874	885	2695	NULL
58	NULL	2002	2013	2044	NULL

59	NULL	2714	2725	2735	NULL
60	NULL	2840	2851	NULL	NULL
61	NULL	3291	3303	3332	Y
62	NULL	3310	3321	3331	NULL
63	NULL	4399	4410	4703	NULL
64	NULL	4714	4725	6566	NULL
65	NULL	5381	5392	5422	NULL
66	NULL	6577	6588	9354	NULL
67	NULL	7140	7151	7220	NULL
68	NULL	9365	9376	9771	NULL
69	NULL	9782	9793	NULL	NULL

Table B.11: The Functions results extracted from the profiler for baidu.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	bdimeHW	813	821	822
2	window.bdsug	4341	4354	4355
3	bdsug.sug	4357	4367	4368
4	bdsug.sugkeywatcher	4370	4390	4391
5	X.__MSG_QS__	4663	4676	4677
6	G	9798	9800	9801

Table B.12: The Objects results extracted from the profiler for baidu.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	C	736	B	A	addEV at 644 with callees at 2098, 2677, 2700
2	G	3344	X	C	P at 3246 with callees at 6128, 6147, 6190, 8039, 8058, 8076, 8095, 9515

3	X	5924	keydown	e	None

Table B.13: The Listener results extracted from the profiler for baidu.com when evaluating over 10K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	$	96	109	143	a
2	addLoadEvent	145	169	280	a
3	getLang	281	299	437	NULL
4	convertChinese	713	739	880	a
5	convertZhLinks	881	906	1126	NULL
6	setLang	1157	1176	1416	a
7	null	450	461	710	NULL
8	null	1429	1440	1720	NULL

Table B.14: The Functions results extracted from the profiler for wikipedia.org when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects were found for wikipedia.org when evaluating over 10K character size.

#	Element	Location	Event	Handler	Factory
1	Document	197	load	a	addLoadEvent at 145 with callees at 438, 1128, 1417

Table B.15: The Listener results extracted from the profiler for wikipedia.org when evaluating over 10K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	fn	948	961	1147	NULL
2	each	3865	3897	3999	a, fn, opt_scope
3	Animate	4515	4548	4745	el, prop, opts
4	onCondition	723	752	861	D,C,A,B
5	window.self.onload	1081	1115	1145	evt
6	Array.prototype.filter	3081	3128	3346	fn, thisObj
7	Array.prototype.indexOf	3396	3442	3593	el, start
8	Animate.canTransition	4857	4892	5054	NULL
9	Animate.prototype._setStyle	5158	5202	5460	val
10	Animate.prototype._animate	5530	5570	6019	NULL
11	Animate.prototype.start	6079	6116	6261	NULL
12	TWTR.Widget	6433	6462	6487	opts
13	matchUrlScheme	6757	6788	6872	url
14	getClassRegEx	7007	7035	7273	c
15	getByClass	7285	7328	7785	c, tag, root, apply
16	browser	7797	7818	7922	NULL
17	byId	7936	7956	8066	id
18	trim	8078	8099	8148	str
19	getViewportHeight	8160	8191	8518	NULL
20	getTarget	8530	8571	8660	e, resolveTextNode
21	resolveTextNode	8672	8703	8862	el
22	getRelatedTarget	8874	8905	9178	e
23	insertAfter	9190	9228	9302	el, reference
24	removeElement	9314	9343	9429	el
25	getFirst	9441	9465	9499	el
26	withinElement	9511	9539	9828	e
27	getStyle	9840	9862	NULL	NULL

28	NULL	30	42	168	g
29	NULL	821	832	855	NULL
30	NULL	1035	1046	1076	NULL
31	NULL	3631	3643	NULL	NULL
32	NULL	6201	6213	6251	NULL
33	NULL	6493	6505	NULL	NULL
34	NULL	9953	9977	NULL	el, property

Table B.16: The Functions results extracted from the profiler for twitter.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	3 objects found			
2	page	711	716	717
3	twttr	6544	6552	6553
4	reClassNameCache	6928	6947	6948

Table B.17: The Objects results extracted from the profiler for twitter.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for twitter.com when evaluating over 10K character size.

#	Identifier	Location	Start	End	Input Arguments
1	$	7777	7790	7851	o
2	CreatAjax	7854	7879	8092	a,b,c
3	CreatXmlRequest	8095	8122	8301	NULL
4	parseInfo	8304	8329	8375	a,b,c
5	swTabs	8380	8398	9038	e
6	swLabs	9041	9074	9665	sobject,sid,snum

7	loadNewsMap	9668	9690	9760	NULL
8	tagOver	9763	9782	NULL	a
9	QosS.isAllLoaded	372	400	609	B
10	QosS.checkLoad	611	636	1519	NULL
11	(QosS.c.onerror	1277	1303	1317	NULL
12	QosS.topSpan	1521	1547	1772	A,B
13	QosS.killTimer	1774	1799	1892	NULL
14	QosS.endCheck	1894	1918	2533	NULL
15	MiniSite.$	2925	2947	3007	s
16	load	3031	3060	3586	sUrl,fCallback
17	_script.onreadystatechange	3335	3372	3466	NULL
18	_script.onload	3506	3531	3553	NULL
19	set	3610	3654	3911	name, value, expires, path, domain
20	get	3915	3933	4075	name
21	clear	4079	4111	4278	name,path,domain
22	insertFlash	4639	4672	5449	elm,url,w,h
23	_print	5469	5494	5524	province
24	print	5528	5544	6348	NULL
25	ok	5552	5565	5596	NULL
26	window.onerror	6482	6511	6529	NULL
27	window.setTimeout	6583	6639	6897	fCallback, nDelay, oObject
28	obj.onreadystatechange	7986	8022	8091	NULL
29	NULL	5758	5769	6177	NULL
30	NULL	6774	6785	6819	NULL
31	NULL	9835	9846	9856	NULL
32	NULL	9967	9978	9993	NULL

Table B.18: The Functions results extracted from the profiler for qq.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	QosS.C	81	88	172

2	QosS.G	174	181	370
3	MiniSite.Browser	2656	2673	2922
4	MiniSite.JsLoader	3010	3028	3588
5	MiniSite.Cookie	3591	3607	4280
6	MiniSite.Home	4283	4297	6350
7	QosS	63	none	none
8	MiniSite	2633	none	none

Table B.19: The Objects results extracted from the profiler for qq.com when evaluating over 10K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for qq.com when evaluating over 10K character size.

Appendix C

Results for Profiler's 15K Character Size Evaluation

#	Identifier	Location	Start	End	Input Arguments
1	_gjuc	1437	1453	1679	NULL
2	_gjp	1680	1695	1754	NULL
3	_gjp	2521	2536	2585	NULL
4	wgjp	2855	2870	3022	NULL
5	n	3737	3749	3786	NULL
6	c	4762	4774	4775	NULL
7	c	5883	5898	5944	F,G
8	j	14491	14503	14540	NULL
9	k	14541	14556	14614	a,b
10	m	14624	14636	14669	NULL
11	ml	277	290	291	NULL
12	time	316	331	358	NULL
13	log	360	380	560	c,d,
14	b				
15	a.onabort	440	461	473	NULL
16	l	278	290	291	NULL
17	e	319	331	358	NULL
18	window.onpopstate	744	772	780	NULL
19	c[a]	896	911	936	NULL
20	window.google.startTick	1061	1098	1146	a,b

21	window.google.tick	1148	1182	1245	a,b,c
22	google.x	1769	1791	1825	e,g
23	window.rwt	1827	1863	2278	a,f,g,k,l,h,c,m
24	qs	2294	2307	2308	NULL
25	tg	2310	2324	2398	e
26	g	3496	3513	3628	b,c,a
27	google.med	3787	3809	3981	b
28	google.register	3984	4013	4121	b,c
29	google.save	4123	4148	4316	b,c
30	google.initHistory	4318	4347	4428	NULL
31	google.exportSymbol	4474	4509	4684	a,b,c
32	google.exportProperty	4686	4723	4730	a,b,c
33	google.inherits	4732	4761	4833	a,b
34	o	4930	4943	5830	a
35	p	5867	5882	6409	a,b
36	google.browser.isEngine-Version	6500	6542	6567	a
37	google.browser.isProduct-Version	6569	6612	6637	a
38	u	6683	6696	6766	a
39	v	6768	6783	7027	a,b
40	w	7029	7042	7080	a
41	create	7104	7124	7225	a,b
42	get	7227	7244	7264	a,b
43	insert	7285	7307	7361	a,b,c
44	remove	7363	7381	7433	a
45	set	7435	7450	7736	a
46	google.listen	7739	7768	7837	a,b,c
47	google.unlisten	7839	7870	7945	a,b,c
48	A	7956	7975	8251	a,b,c,d
49	d	8083	8096	8109	y
50	listen	8266	8283	8384	NULL
51	unlisten	8386	8405	8549	NULL
52	D	8604	8621	8897	a,b,c
53	E	8899	8911	9182	NULL
54	H	9184	9196	9270	NULL
55	I	9272	9287	9481	a,b

56	search	9501	9521	9840	a,b
57	getQuery	9885	9904	9918	NULL
58	J	9925	9937	9990	NULL
59	K	9992	10009	10385	a,b,c
60	L	10387	10400	10559	a
61	M	10561	10574	10763	a
62	N	10765	10778	10834	a
63	O	10836	10849	10906	a
64	P	10908	10921	10947	a
65	Q	10949	10962	10992	a
66	R	10994	11009	11083	a,b
67	addClass	11232	11254	11356	a,b
68	removeClass	11358	11383	11498	a,b
69	U	11581	11594	11638	a
70	escape	11654	11672	11746	a
71	unescape	11748	11768	11842	a
72	stopPropagation	11858	11885	11962	a
73	getSelection	11964	11987	12139	NULL
74	xjsol	12141	12158	12240	a
75	xjsl	12242	12260	12344	a,b
76	V	12351	12368	12566	a,b,c
77	isSerpLink	12581	12603	12640	a
78	isSerpForm	12642	12664	12697	a
79	updateLinksWithParam	12699	12737	13177	a,b,c,d
80	qs	13179	13193	13463	a
81	google.xhr	13466	13487	13720	NULL
82	google.getHeight	13781	13809	13821	a
83	google.getWidth	13823	13850	13862	a
84	google.getComputedStyle	13864	13903	13919	a,b,c
85	google.getPageOffsetTop	13921	13956	13968	a
86	google.getPageOffsetLeft	13970	14006	14018	a
87	google.getPageOffsetStart	14020	14057	14069	a
88	google.hasClass	14071	14100	14114	a,b
89	google.getColor	14116	14143	14155	a
90	google.append	14157	14182	14194	a
91	google.rhs	14196	14217	14218	NULL
92	google.eventTarget	14221	14251	14263	a

93	google.bind	14265	14292	14313	a,b,c
94	google.unbind	14315	14344	14367	a,b,c
95	google.History.client	14420	14453	14481	a
96	google.History.addPostInit-Callback	14670	14716	14726	a
97	google.History.save	14728	14761	14771	a,b
98	google.History.initialize	14774	14811	14936	a
99	NULL	717	728	942	NULL
100	NULL	883	895	937	a
101	NULL	2374	2385	2396	NULL
102	NULL	3344	3355	3461	NULL
103	NULL	3467	3478	4430	NULL
104	NULL	4436	4447	NULL	NULL
105	NULL	10055	10069	10092	e,f
106	NULL	12433	12447	12456	e,f
107	NULL	12520	12536	12564	e,f,g
108	NULL	14375	14386	NULL	NULL

Table C.1: The Functions results extracted from the profiler for google.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	window.google	17	31	715
2	window.chrome	966	980	981
3	window.google.timers	1037	1058	1059
4	i[a]	1099	1104	1145
5	google.y	1757	1766	1767
6	window.gbar	2281	2293	2399
7	o	2329	2331	2341
8	google.j[1]	2586	2598	2820
9	e	29	31	715
10	c[d]	4677	4682	4683
11	k	4839	4841	4861
12	m	4863	4865	4897
13	google.browser.engine	5621	5643	5703
14	google.browser.product	5705	5728	5829

15	google.dom	7083	7094	7737
16	z	7951	7953	7954
17	google.msg	8254	8265	8557
18	google.nav	9484	9495	9919
19	d	9565	9567	9568
20	google.style	11086	11099	11499
21	google.util	11641	11653	12345
22	google.srp	12569	12580	13464
23	google.History	14387	14402	14403
24	i[g]	14568	14573	14574
25	n.json	14950	14957	14958

Table C.2: The Objects results extracted from the profiler for google.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	a	7788	b	c	google.listen at 7739 with callee at 14293.

Table C.3: The Listener results extracted from the profiler for google.com when evaluating over 15K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	run_if_loaded	893	920	953	a,b
2	run_with	954	978	1031	b,a,c
3	wait_for_load	1032	1061	1447	c,b,e
4	bind	1448	1466	1699	c,b
5	env_get	1731	1750	1777	a

6	hasArrayNature	1902	1928	2114	a
7	$A	2115	2129	2271	b
8	eval_global	2273	2296	2673	c
9	copy_properties	2675	2704	2888	b,c
10	add_properties	2889	2917	2970	a,b
11	is_empty	2971	2991	3120	b
12	Arbiter	3195	3213	3339	NULL
13	set_ue_cookie	14251	14276	14405	a
14	window.async_callback	3126	3183	3193	a,b
15	subscribe	3746	3771	4275	k,b,i
16	unsubscribe	4277	4300	4619	e
17	inform	4621	4643	5419	i,c,b
18	query	5421	5438	5563	b
19	_getInstance	5580	5604	5717	a
20	registerCallback	5719	5749	6261	b,d
21	_updateCallbacks	6263	6293	6560	d,c
22	Function.prototype.deferUntil	6565	6612	6811	a,h,b,i
23	c	6683	6695	6778	NULL
24	configurePage	6959	6984	7389	b
25	loadComponents	7391	7419	7593	d,b
26	loadResources	7595	7626	8329	h,b,g,k
27	_fetchWithIframe	8331	8359	8793	d
28	_addResourceToIframe	8795	8827	9198	e
29	requestResource	9200	9231	9888	j,g,e
30	_runCSSPolls	9926	9949	10735	NULL
31	_startCSSPoll	10737	10762	11228	d
32	done	11230	11248	11524	f,c
33	requested	11526	11547	11610	c
34	enableBootload	11612	11638	11707	b
35	_unloadResource	11709	11736	12082	e
36	getHardpoint	12084	12107	12259	NULL
37	setResourceMap	12261	12287	12368	c
38	resolveResources	12370	12400	12590	e,b
39	loadEarlyResources	12592	12622	12808	d
40	b	13372	13385	13429	l
41	a	13435	13448	14047	m
42	l.onload	13827	13846	13859	NULL

43	d	13834	13846	13859	NULL
44	h	14912	14925	0	r
45	NULL	712	724	745	a
46	NULL	853	865	891	a
47	NULL	1280	1291	1424	NULL
48	NULL	1519	1530	1697	NULL
49	NULL	6918	6930	12984	a
50	NULL	10840	10851	11144	NULL
51	NULL	11082	11093	11137	NULL
52	NULL	11157	11168	11190	NULL
53	NULL	13040	13051	14245	NULL
54	NULL	14055	14071	14243	n,l,m
55	NULL	14155	14166	14240	NULL
56	NULL	14422	14433	NULL	NULL

Table C.4: The Functions results extracted from the profiler for facebook.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	__rm	145	152	153
2	window[a]	2953	2963	2964
3	h	4717	4719	4720
4	i	5924	5926	5927
5	a._listen[j]	6061	6074	6075
6	a._callbacks[h]	6188	6204	6250
7	h	4717	4719	4720
8	this._cssLinkMap[f]	7295	7315	7325
9	e	7674	7676	7677
10	this._earlyResources	7845	7866	7867
11	this._cssLinkMap[e]	9677	9697	9713
12	this._cssLinkMap[e]	9677	9697	9713
13	this._activeCSSPolls	10193	10214	10215
14	e	7674	7676	7677
15	r	14534	14536	14684

Table C.5: The Objects results extracted from the profiler for facebook.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for facebook.com when evaluating over 15K character size.

#	Identifier	Location	Start	End	Input Arguments
1	checkChromePromoAlert	2763	2796	2977	NULL
2	dismissChromePromoAlert	3038	3073	3224	NULL
3	hideFbPromoAlert	5982	6010	6049	NULL
4	isRtl	7054	7071	7114	NULL
5	setRtlYva	7119	7146	9617	suffix
6	RichMediaCore_58_12	11230	11261	13229	NULL
7	yt.timing.tick	86	129	359	label, opt_time
8	yt.timing.info	369	409	549	label, value
9	gaiaChangedCallback	3892	3934	4388	autoshare
10	fn	4230	4246	4341	NULL
11	canConnectCallback	4462	4503	4778	autoshare
12	serviceChangedCallback	4850	4895	5385	autoshare
13	window.dismissFbPromoAlert	5809	5849	5972	NULL
14	fixYvaDom	7793	7822	8142	suffix
15	richMedia.clipFlashObject	8644	8753	9096	asset, width, height, offsetTop, offsetRight, offsetBottom, offsetLeft
16	richMedia.unclipFlashObject	9171	9232	9560	asset, width, height
17	RichMediaCore_58_12.prototype.setCsiEventsRecordedDuringBreakout	13339	13425	13529	creative
18	RichMediaCore_58_12.prototype.csiHasValidStart	13532	13600	13678	creative

19	RichMediaCore_58_12.prototype.shouldReportCsi	13681	13748	13838	creative
20	RichMediaCore_58_12.prototype.shouldCsi	13841	13913	14292	asset, creativeType
21	RichMediaCore_58_12.prototype.trackCsiEvent	14295	14357	14406	event
22	RichMediaCore_58_12.prototype.getCsiServer	14409	14465	14642	NULL
23	RichMediaCore_58_12.prototype.reportCsi	14645	14706	NULL	creative
24	NULL	3258	3270	6061	NULL
25	NULL	3276	3288	6052	NULL
26	NULL	6483	6495	6567	NULL

Table C.6: The Functions results extracted from the profiler for youtube.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	self.dartLoadedGlobalTemplates_58_12	11057	11096	11097
2	dartCreativeDisplayManagers	undefined	10893	none
3	csiTimes	undefined	13314	none

Table C.7: The Objects results extracted from the profiler for youtube.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for youtube.com when evaluating over 15K character size.

#	Identifier	Location	Start	End	Input Arguments
1	c	2756	2773	2896	r,s,q
2	o	2897	2914	3032	r,s,q

3	enable	3048	3065	3108	NULL
4	disable	3110	3128	3171	NULL
5	dispatch	3173	3192	3261	NULL
6	reset	3263	3279	3285	NULL
7	a	5457	5478	6172	A,l,u,n,h
8	e.augment	6376	6405	6841	f,w,j,u,o
9	k[i]	6537	6552	6672	NULL
10	e.aggregate	6843	6872	6902	h,g,f,i
11	e.extend	6904	6930	7182	i,h,f,k
12	e.each	7184	7208	7390	i,h,j,g
13	e.clone	7392	7421	7914	j,k,n,p,i,m
14	e.bind	7916	7936	8115	g,i
15	e.rbind	8117	8138	8317	g,i
16	before	8502	8526	8636	i,k,l,m
17	after	8638	8661	8771	i,k,l,m
18	_inject	8773	8798	9035	h,j,k,m
19	k[m]	8906	8921	8961	NULL
20	detach	9037	9055	9081	h
21	_unload	9083	9104	9105	i,h
22	e.Do.Method	9108	9133	9209	h,i
23	e.Do.Method.prototype.register	9211	9257	9304	i,j,h
24	e.Do.Method.prototype._delete	9306	9347	9391	h
25	e.Do.Method.prototype.exec	9393	9430	9926	NULL
26	e.Do.AlterArgs	9928	9956	9983	i,h
27	e.Do.AlterReturn	9985	10015	10044	i,h
28	e.Do.Halt	10046	10069	10095	i,h
29	e.Do.Prevent	10097	10121	10133	h
30	e.EventHandle	10384	10411	10434	f,g
31	detach	10461	10478	10591	NULL
32	e.CustomEvent	10594	10621	10845	f,g
33	applyConfig	10872	10897	10923	g,f
34	_on	10925	10946	11264	j,h,g,f
35	subscribe	11266	11289	11376	h,g
36	on	11378	11394	11481	h,g
37	after	11483	11502	11586	h,g

38	detach	11588	11608	11772	k,h
39	unsubscribe	11774	11796	11838	NULL
40	_notify	11840	11863	12027	i,h,f
41	log	12029	12046	12065	g,f
42	fire	12067	12082	12333	NULL
43	fireSimple	12335	12357	12505	f
44	fireComplex	12507	12530	12571	f
45	_procSubs	12573	12598	12759	j,g,f
46	_broadcast	12761	12783	12949	g
47	unsubscribeAll	12951	12976	13021	NULL
48	detachAll	13023	13043	13065	NULL
49	_delete	13067	13086	13178	f
50	e.Subscriber	13181	13209	13286	h,g,f
51	_notify	13312	13335	13561	j,h,i
52	notify	13563	13583	13794	g,i
53	contains	13796	13818	13889	g,f
54	l	14261	14274	14588	m
55	on	14603	14623	NULL	r,v,p,w
56	NULL	185	198	2118	NULL
57	NULL	2158	2169	2548	NULL
58	NULL	2588	2599	3288	NULL
59	NULL	5404	5416	6204	g
60	NULL	6308	6320	8319	e
61	NULL	6522	6536	6770	r,i
62	NULL	7742	7756	7846	o,f
63	NULL	7867	7881	7894	o,f
64	NULL	7999	8010	8113	NULL
65	NULL	8201	8212	8315	NULL
66	NULL	8417	8429	NULL	e
67	NULL	8464	8475	10156	NULL
68	NULL	13892	13903	NULL	NULL
69	NULL	13951	13965	14029	m,n
70	NULL	14042	14056	14258	o,q
71	NULL	14871	14885	NULL	x,n

Table C.8: The Functions results extracted from the profiler for yahoo.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	k	2747	2749	2750
2	j.OnloadCache	3033	3047	3286
3	r	3197	3199	3211
4	k	2747	2749	2750
5	g	6499	6501	6502
6	k	2747	2749	2750
7	q	6509	6511	6512
8	e.Env.evt	8430	8440	8462
9	e.Do	8488	8493	9106
10	this.objs[n]	8838	8851	8852
11	this.before	9180	9192	9193
12	this.after	9195	9206	9207
13	e.EventHandle.prototype	10436	10460	10592
14	this.subscribers	10728	10745	10746
15	this.afters	10748	10760	10761
16	e.CustomEvent.prototype	10847	10871	13179
17	e.Subscriber.prototype	13288	13311	13890
18	l.prototype	14590	14602	NULL
19	G	14831	14833	14834

Table C.9: The Objects results extracted from the profiler for yahoo.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for yahoo.com when evaluating over 15K character size.
No Functions were found for live.com when evaluating over 15K character size.

#	Identifier	Location	Start	End

1	srf_oTemplate	3886	none	none
2	g_DO	10032	none	none

Table C.10: The Objects results extracted from the profiler for live.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for live.com when evaluating over 15K character size.

#	Identifier	Location	Start	End	Input Arguments
1	gaia_onLoginSubmit	6764	6794	6903	NULL
2	gaia_setFocus	12081	12106	12349	NULL
3	window.onload	6373	6401	6662	NULL
4	gaia_loginForm.Email.onkey-press	11993	12038	12076	NULL

Table C.11: The Functions results extracted from the profiler for blogger.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects and event listeners were found for blogger.com when evaluating over 15K character size.

#	Identifier	Location	Start	End	Input Arguments
1	addEV	644	665	767	C,B,A
2	G	768	781	808	A
3	F	1560	1572	1692	NULL
4	H	1693	1708	2431	W,S
5	U	1843	1855	2082	NULL
6	J	2432	2445	2541	S

7	C	2542	2555	2676	T
8	I	2955	2968	3002	C
9	K	3003	3016	3049	C
10	S	3050	3063	3163	C
11	U	3164	3177	3245	C
12	P	3246	3263	3374	G,X,C
13	N	3375	3388	3439	C
14	R	3440	3453	3665	X
15	H	3666	3679	3874	G
16	C	3777	3792	3873	Z,b
17	O	3875	3888	4054	Y
18	L	4055	4068	4278	G
19	C	4411	4424	4531	b
20	G	4532	4545	4639	Y
21	Z	4809	4821	4880	NULL
22	e	4881	4894	5304	n
23	k	5305	5317	5514	NULL
24	l	5515	5527	5547	NULL
25	g	5548	5560	5577	NULL
26	i	5578	5590	5667	NULL
27	b	5668	5681	5733	n
28	G	5734	5747	5793	n
29	n	6642	6654	6716	NULL
30	e	6717	6729	6907	NULL
31	i	6908	6920	6972	NULL
32	G	6973	6985	7009	NULL
33	b	7010	7023	7111	q
34	c	7112	7125	7221	q
35	f	7222	7235	7390	q
36	X	7391	7403	7627	NULL
37	Y	7628	7643	7778	r,q
38	p	7779	7794	7882	q,s
39	j	7883	7895	8347	NULL
40	Z	8348	8360	8454	NULL
41	k	8455	8467	8736	NULL
42	g	8737	8749	9007	NULL
43	G	9401	9413	9514	NULL

44	X	9531	9543	9558	NULL
45	Y	9559	9572	9603	Z
46	X	9803	9816	9919	C
47	Y	9920	9933	9979	C
48	G	10128	10141	10387	Y
49	C	10669	10681	10755	NULL
50	G	10756	10768	10879	NULL
51	a	11050	11062	11141	NULL
52	d	11142	11154	11340	NULL
53	b	11341	11354	11489	e
54	X	11490	11502	11529	NULL
55	G	11530	11542	11998	NULL
56	a[i].onclick	246	269	426	NULL
57	w.onunload	2738	2759	2760	NULL
58	ini	4647	4662	4701	X
59	bdsug.sugkeywatcher.on	5843	5876	5969	NULL
60	bdsug.sugkeywatcher.off	5971	6005	6101	NULL
61	rm	6222	6236	6563	n
62	rm	9022	9036	9351	q
63	rm	9618	9632	9768	Z
64	rm	9994	10008	10090	C
65	rm	10402	10416	10517	Y
66	bdsug.sug	10525	10546	10586	C
67	bdsug.initSug	10588	10612	10636	NULL
68	rm	10894	10908	10974	X
69	rm	12033	12047	12106	e
70	NULL	428	439	516	NULL
71	NULL	874	885	2695	NULL
72	NULL	2002	2013	2044	NULL
73	NULL	2714	2725	2735	NULL
74	NULL	2840	2851	12618	NULL
75	NULL	3291	3303	3332	Y
76	NULL	3310	3321	3331	NULL
77	NULL	4399	4410	4703	NULL
78	NULL	4714	4725	6566	NULL
79	NULL	5381	5392	5422	NULL
80	NULL	6577	6588	9354	NULL

81	NULL	7140	7151	7220	NULL
82	NULL	9365	9376	9771	NULL
83	NULL	9782	9793	10093	NULL
84	NULL	10104	10115	10520	NULL
85	NULL	10657	10668	10977	NULL
86	NULL	10988	10999	12109	NULL

Table C.12: The Functions results extracted from the profiler for baidu.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	bdimeHW	813	821	822
2	window.bdsug	4341	4354	4355
3	bdsug.sug	4357	4367	4368
4	bdsug.sugkeywatcher	4370	4390	4391
5	X.__MSG_QS__	4663	4676	4677
6	G	9798	9800	9801

Table C.13: The Objects results extracted from the profiler for baidu.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	C	736	B	A	addEV at 644 with callees at 2098, 2677, 2700.
2	G	3344	X	C	P at 3246 with callees at 6128, 6147, 6190, 8039, 8058, 8076, 8095, 9515, 11624, 11650.
3	X	5924	keydown	e	None

Table C.14: The Listener results extracted from the profiler for baidu.com when evaluating over 15K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	$	96	109	143	a
2	addLoadEvent	145	169	280	a
3	getLang	281	299	437	NULL
4	convertChinese	713	739	880	a
5	convertZhLinks	881	906	1126	NULL
6	setLang	1157	1176	1416	a
7	NULL	450	461	710	NULL
8	NULL	1429	1440	1720	NULL

Table C.15: The Functions results extracted from the profiler for wikipedia.org when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects were found for wikipedia.org when evaluating over 15K character size.

#	Element	Location	Event	Handler	Factory
1	Document	197	load	a	addLoadEvent at 145 with callees at 438, 1128, 1417

Table C.16: The Listener results extracted from the profiler for wikipedia.org when evaluating over 15K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	fn	948	961	1147	NULL
2	each	3865	3897	3999	a, fn, opt_scope
3	Animate	4515	4548	4745	el, prop, opts
4	HexToR	11725	11744	11800	h
5	HexToG	11808	11827	11883	h
6	HexToB	11891	11910	11966	h
7	getRest	13398	13417	13721	NULL
8	onCondition	723	752	861	D,C,A,B
9	window.self.onload	1081	1115	1145	evt
10	Array.prototype.filter	3081	3128	3346	fn, thisObj
11	Array.prototype.indexOf	3396	3442	3593	el, start
12	Animate.canTransition	4857	4892	5054	NULL
13	Animate.prototype._setStyle	5158	5202	5460	val
14	Animate.prototype._animate	5530	5570	6019	NULL
15	Animate.prototype.start	6079	6116	6261	NULL
16	TWTR.Widget	6433	6462	6487	opts
17	matchUrlScheme	6757	6788	6872	url
18	getClassRegEx	7007	7035	7273	c
19	getByClass	7285	7328	7785	c, tag, root, apply
20	browser	7797	7818	7922	NULL
21	byId	7936	7956	8066	id
22	trim	8078	8099	8148	str
23	getViewportHeight	8160	8191	8518	NULL
24	getTarget	8530	8571	8660	e, resolveTextNode
25	resolveTextNode	8672	8703	8862	el
26	getRelatedTarget	8874	8905	9178	e
27	insertAfter	9190	9228	9302	el, reference
28	removeElement	9314	9343	9429	el
29	getFirst	9441	9465	9499	el
30	withinElement	9511	9539	9828	e
31	getStyle	9840	9862	10515	NULL
32	has	10616	10637	10724	el, c
33	add	10734	10755	10872	el, c
34	remove	10882	10906	11062	el, c

35	add	11204	11232	11462	el, type, fn
36	remove	11471	11502	11677	el, type, fn
37	hex_rgb	11695	11716	12068	NULL
38	bool	12162	12180	12227	b
39	def	12237	12254	12306	o
40	number	12316	12336	12397	n
41	string	12406	12426	12472	s
42	fn	12482	12498	12546	f
43	array	12556	12575	12689	a
44	absoluteTime	12849	12876	13781	s
45	hour	13022	13040	13315	NULL
46	timeAgo	13973	14004	NULL	dateString
47	NULL	30	42	168	g
48	NULL	821	832	855	NULL
49	NULL	1035	1046	1076	NULL
50	NULL	3631	3643	NULL	NULL
51	NULL	6201	6213	6251	NULL
52	NULL	6493	6505	NULL	NULL
53	NULL	9953	9977	10236	el, property
54	NULL	10344	10368	10500	el, property
55	NULL	11379	11391	11442	NULL
56	NULL	11981	11996	12060	hex

Table C.17: The Functions results extracted from the profiler for twitter.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	page	711	716	717
2	twttr	6544	6552	6553
3	reClassNameCache	6928	6947	6948
4	classes	10598	10608	11068
5	events	11187	11196	11683
6	is	12149	12154	12695

Table C.18: The Objects results extracted from the profiler for twitter.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	el	11279	type	fn	add at 11204 with a callee at 11143.

Table C.19: The Listener results extracted from the profiler for twitter.com when evaluating over 15K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	$	7777	7790	7851	o
2	CreatAjax	7854	7879	8092	a,b,c
3	CreatXmlRequest	8095	8122	8301	NULL
4	parseInfo	8304	8329	8375	a,b,c
5	swTabs	8380	8398	9038	e
6	swLabs	9041	9074	9665	sobject,sid,snum
7	loadNewsMap	9668	9690	9760	NULL
8	tagOver	9763	9782	10001	a
9	tagOut	10004	10022	10055	a
10	getNames	10059	10090	10276	obj,name,tij
11	cplay	10280	10296	10522	NULL
12	formatPageZone	10526	10553	11114	NULL
13	send_request	11118	11146	11665	url
14	loadPage	11669	11690	12352	NULL
15	tail	12355	12372	12982	NULL
16	getExpires	13067	13089	13181	a

17	setdefSkin	13184	13205	13642	NULL
18	setdef	13646	13663	13999	NULL
19	QosS.isAllLoaded	372	400	609	B
20	QosS.checkLoad	611	636	1519	NULL
21	(QosS.c.onerror	1277	1303	1317	NULL
22	QosS.topSpan	1521	1547	1772	A,B
23	QosS.killTimer	1774	1799	1892	NULL
24	QosS.endCheck	1894	1918	2533	NULL
25	MiniSite.$	2925	2947	3007	s
26	load	3031	3060	3586	sUrl,fCallback
27	_script.onreadystatechange	3335	3372	3466	NULL
28	_script.onload	3506	3531	3553	NULL
29	set	3610	3654	3911	name, value, expires, path, domain
30	get	3915	3933	4075	name
31	clear	4079	4111	4278	name,path,domain
32	insertFlash	4639	4672	5449	elm,url,w,h
33	_print	5469	5494	5524	province
34	print	5528	5544	6348	NULL
35	ok	5552	5565	5596	NULL
36	window.onerror	6482	6511	6529	NULL
37	window.setTimeout	6583	6639	6897	fCallback, nDelay, oObject
38	obj.onreadystatechange	7986	8022	8091	NULL
39	f[i].onclick	10915	10938	10984	NULL
40	f[i].onmouseover	10989	11016	11050	NULL
41	f[i].onmouseout	11052	11078	11112	NULL
42	$("tailorArrow").onclick	11966	12003	12079	NULL
43	$("tailorArrow").onmouseover	12081	12122	12143	NULL
44	$("tailorArrow").onmouseout	12146	12186	12206	NULL
45	NULL	5758	5769	6177	NULL
46	NULL	6774	6785	6819	NULL
47	NULL	9835	9846	9856	NULL
48	NULL	9967	9978	9993	NULL

Table C.20: The Functions results extracted from the profiler for qq.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	QosS.C	81	88	172
2	QosS.G	174	181	370
3	MiniSite.Browser	2656	2673	2922
4	MiniSite.JsLoader	3010	3028	3588
5	MiniSite.Cookie	3591	3607	4280
6	MiniSite.Home	4283	4297	6350
7	QosS	63	none	none
8	MiniSite	2633	none	none

Table C.21: The Objects results extracted from the profiler for qq.com when evaluating over 15K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for qq.com when evaluating over 15K character size.

Appendix D

Results for Profiler's 20K Character Size Evaluation

#	Identifier	Location	Start	End	Input Arguments
1	_gjuc	1437	1453	1679	NULL
2	_gjp	1680	1695	1754	NULL
3	_gjp	2521	2536	2585	NULL
4	wgjp	2855	2870	3022	NULL
5	n	3737	3749	3786	NULL
6	c	4762	4774	4775	NULL
7	c	5883	5898	5944	F,G
8	j	14491	14503	14540	NULL
9	k	14541	14556	14614	a,b
10	m	14624	14636	14669	NULL
11	d	17842	17854	17934	NULL
12	k	18350	18363	18372	b
13	l	18374	18386	18854	NULL
14	m	18856	18875	18911	b,c,g,e
15	n	18912	18925	19013	b
16	h	19865	19880	19910	a,c
17	j	19911	19923	NULL	NULL
18	ml	277	290	291	NULL
19	time	316	331	358	NULL
20	log	360	380	560	c,d,

21	b				
22	a.onabort	440	461	473	NULL
23	l	278	290	291	NULL
24	e	319	331	358	NULL
25	window.onpopstate	744	772	780	NULL
26	c[a]	896	911	936	NULL
27	window.google.startTick	1061	1098	1146	a,b
28	window.google.tick	1148	1182	1245	a,b,c
29	google.x	1769	1791	1825	e,g
30	window.rwt	1827	1863	2278	a,f,g,k,l,h,c,m
31	qs	2294	2307	2308	NULL
32	tg	2310	2324	2398	e
33	g	3496	3513	3628	b,c,a
34	google.med	3787	3809	3981	b
35	google.register	3984	4013	4121	b,c
36	google.save	4123	4148	4316	b,c
37	google.initHistory	4318	4347	4428	NULL
38	google.exportSymbol	4474	4509	4684	a,b,c
39	google.exportProperty	4686	4723	4730	a,b,c
40	google.inherits	4732	4761	4833	a,b
41	o	4930	4943	5830	a
42	p	5867	5882	6409	a,b
43	google.browser.isEngineVersion	6500	6542	6567	a
44	google.browser.isProductVersion	6569	6612	6637	a
45	u	6683	6696	6766	a
46	v	6768	6783	7027	a,b
47	w	7029	7042	7080	a
48	create	7104	7124	7225	a,b
49	get	7227	7244	7264	a,b
50	insert	7285	7307	7361	a,b,c
51	remove	7363	7381	7433	a
52	set	7435	7450	7736	a
53	google.listen	7739	7768	7837	a,b,c
54	google.unlisten	7839	7870	7945	a,b,c
55	A	7956	7975	8251	a,b,c,d

56	d	8083	8096	8109	y
57	listen	8266	8283	8384	NULL
58	unlisten	8386	8405	8549	NULL
59	D	8604	8621	8897	a,b,c
60	E	8899	8911	9182	NULL
61	H	9184	9196	9270	NULL
62	I	9272	9287	9481	a,b
63	search	9501	9521	9840	a,b
64	getQuery	9885	9904	9918	NULL
65	J	9925	9937	9990	NULL
66	K	9992	10009	10385	a,b,c
67	L	10387	10400	10559	a
68	M	10561	10574	10763	a
69	N	10765	10778	10834	a
70	O	10836	10849	10906	a
71	P	10908	10921	10947	a
72	Q	10949	10962	10992	a
73	R	10994	11009	11083	a,b
74	addClass	11232	11254	11356	a,b
75	removeClass	11358	11383	11498	a,b
76	U	11581	11594	11638	a
77	escape	11654	11672	11746	a
78	unescape	11748	11768	11842	a
79	stopPropagation	11858	11885	11962	a
80	getSelection	11964	11987	12139	NULL
81	xjsol	12141	12158	12240	a
82	xjsl	12242	12260	12344	a,b
83	V	12351	12368	12566	a,b,c
84	isSerpLink	12581	12603	12640	a
85	isSerpForm	12642	12664	12697	a
86	updateLinksWithParam	12699	12737	13177	a,b,c,d
87	qs	13179	13193	13463	a
88	google.xhr	13466	13487	13720	NULL
89	google.getHeight	13781	13809	13821	a
90	google.getWidth	13823	13850	13862	a
91	google.getComputedStyle	13864	13903	13919	a,b,c
92	google.getPageOffsetTop	13921	13956	13968	a

93	google.getPageOffsetLeft	13970	14006	14018	a
94	google.getPageOffsetStart	14020	14057	14069	a
95	google.hasClass	14071	14100	14114	a,b
96	google.getColor	14116	14143	14155	a
97	google.append	14157	14182	14194	a
98	google.rhs	14196	14217	14218	NULL
99	google.eventTarget	14221	14251	14263	a
100	google.bind	14265	14292	14313	a,b,c
101	google.unbind	14315	14344	14367	a,b,c
102	google.History.client	14420	14453	14481	a
103	google.History.addPostInit-Callback	14670	14716	14726	a
104	google.History.save	14728	14761	14771	a,b
105	google.History.initialize	14774	14811	14936	a
106	n.typeOf	14988	15008	15592	a
107	n.isArray	15594	15615	15643	a
108	n.json.j	15645	15665	15960	a
109	n.json.parse	15963	15987	16091	a
110	n.json.unsafeParse	16093	16123	16146	a
111	n.json.a	16148	16168	16203	a
112	n.json.Serializer	16205	16233	16234	NULL
113	n.json.Serializer.prototype.a	16236	16277	16316	a
114	n.json.Serializer.prototype.b	16319	16362	16683	a,b
115	n.json.Serializer.prototype.d	16924	16967	17223	a,b
116	n.json.Serializer.prototype.g	17225	17268	17308	a,b
117	n.json.Serializer.prototype.e	17311	17354	17450	a,b
118	n.json.Serializer.prototype.h	17452	17495	17654	a,b
119	google.rhs	18002	18022	18313	NULL
120	animate	19026	19049	19383	b,c,g
121	finish	19341	19358	19381	NULL
122	easeInAndOut	19385	19409	19427	b
123	easeOut	19429	19448	19473	b
124	getFrameCount	19475	19499	19508	NULL
125	unwrap	19520	19538	19591	b
126	wrap	19522	19538	19591	b
127	dispose	19725	19743	19776	NULL
128	NULL	717	728	942	NULL

129	NULL	883	895	937	a
130	NULL	2374	2385	2396	NULL
131	NULL	3344	3355	3461	NULL
132	NULL	3467	3478	4430	NULL
133	NULL	4436	4447	NULL	NULL
134	NULL	10055	10069	10092	e,f
135	NULL	12433	12447	12456	e,f
136	NULL	12520	12536	12564	e,f,g
137	NULL	14375	14386	17816	NULL
138	NULL	17008	17020	17216	f
139	NULL	17822	17833	18315	NULL
140	NULL	18321	18332	19780	NULL
141	NULL	19786	19797	NULL	NULL

Table D.1: The Functions results extracted from the profiler for google.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	window.google	17	31	715
2	window.chrome	966	980	981
3	window.google.timers	1037	1058	1059
4	i[a]	1099	1104	1145
5	google.y	1757	1766	1767
6	window.gbar	2281	2293	2399
7	o	2329	2331	2341
8	google.j[1]	2586	2598	2820
9	e	29	31	715
10	c[d]	4677	4682	4683
11	k	4839	4841	4861
12	m	4863	4865	4897
13	google.browser.engine	5621	5643	5703
14	google.browser.product	5705	5728	5829
15	google.dom	7083	7094	7737
16	z	7951	7953	7954
17	google.msg	8254	8265	8557

18	google.nav	9484	9495	9919
19	d	9565	9567	9568
20	google.style	11086	11099	11499
21	google.util	11641	11653	12345
22	google.srp	12569	12580	13464
23	google.History	14387	14402	14403
24	i[g]	14568	14573	14574
25	n.json	14950	14957	14958
26	n.json.Serializer.c	16685	16705	16819
27	google.fx	19015	19025	19703
28	j	19262	19264	19292
29	google.event.back	19828	19846	19847

Table D.2: The Objects results extracted from the profiler for google.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	a	7788	b	c	google.listen at 7739 with callee at 14293, 17939.

Table D.3: The Listener results extracted from the profiler for google.com when evaluating over 20K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	run_if_loaded	893	920	953	a,b
2	run_with	954	978	1031	b,a,c
3	wait_for_load	1032	1061	1447	c,b,e

4	bind	1448	1466	1699	c,b
5	env_get	1731	1750	1777	a
6	hasArrayNature	1902	1928	2114	a
7	$A	2115	2129	2271	b
8	eval_global	2273	2296	2673	c
9	copy_properties	2675	2704	2888	b,c
10	add_properties	2889	2917	2970	a,b
11	is_empty	2971	2991	3120	b
12	Arbiter	3195	3213	3339	NULL
13	set_ue_cookie	14251	14276	14405	a
14	Metaprototype	18249	18273	18274	NULL
15	__metaprototype	19081	19110	19350	c,a
16	__metaprototype_construct	19351	19388	19632	a
17	__metaprototype_init	19633	19665	19997	d
18	window.async_callback	3126	3183	3193	a,b
19	subscribe	3746	3771	4275	k,b,i
20	unsubscribe	4277	4300	4619	e
21	inform	4621	4643	5419	i,c,b
22	query	5421	5438	5563	b
23	_getInstance	5580	5604	5717	a
24	registerCallback	5719	5749	6261	b,d
25	_updateCallbacks	6263	6293	6560	d,c
26	Function.prototype.deferUn-til	6565	6612	6811	a,h,b,i
27	c	6683	6695	6778	NULL
28	configurePage	6959	6984	7389	b
29	loadComponents	7391	7419	7593	d,b
30	loadResources	7595	7626	8329	h,b,g,k
31	_fetchWithIframe	8331	8359	8793	d
32	_addResourceToIframe	8795	8827	9198	e
33	requestResource	9200	9231	9888	j,g,e
34	_runCSSPolls	9926	9949	10735	NULL
35	_startCSSPoll	10737	10762	11228	d
36	done	11230	11248	11524	f,c
37	requested	11526	11547	11610	c
38	enableBootload	11612	11638	11707	b

39	_unloadResource	11709	11736	12082	e
40	getHardpoint	12084	12107	12259	NULL
41	setResourceMap	12261	12287	12368	c
42	resolveResources	12370	12400	12590	e,b
43	loadEarlyResources	12592	12622	12808	d
44	b	13372	13385	13429	l
45	a	13435	13448	14047	m
46	l.onload	13827	13846	13859	NULL
47	d	13834	13846	13859	NULL
48	h	14912	14925	15083	r
49	k	15085	15097	15500	NULL
50	a	15502	15523	15864	u,q,s,t,r
51	$	16051	16064	16120	a
52	hasClass	16140	16162	16221	b,a
53	addClass	16223	16245	16317	b,a
54	removeClass	16319	16344	16439	b,a
55	toggleClass	16441	16466	16517	b,a
56	conditionClass	16519	16549	16595	c,b,a
57	show	16597	16613	16647	a
58	hide	16649	16665	16696	a
59	conditionShow	16698	16725	16765	b,a
60	byTag	16781	16800	16866	a,b
61	byClass	16868	16889	16942	b,a
62	b.onclick	17008	17029	17778	d
63	b.onsubmit	17780	17802	18029	d
64	Function.prototype.extend	18036	18073	18247	a
65	makeFinal	18306	18327	18328	a
66	_queue	18342	18362	18551	b,c
67	_onbootload	18553	18578	18594	b,a
68	_update	18596	18614	18824	NULL
69	_apply	18826	18846	19077	a,c
70	NULL	712	724	745	a
71	NULL	853	865	891	a
72	NULL	1280	1291	1424	NULL
73	NULL	1519	1530	1697	NULL
74	NULL	6918	6930	12984	a
75	NULL	10840	10851	11144	NULL

76	NULL	11082	11093	11137	NULL
77	NULL	11157	11168	11190	NULL
78	NULL	13040	13051	14245	NULL
79	NULL	14055	14071	14243	n,l,m
80	NULL	14155	14166	14240	NULL
81	NULL	14422	14433	16042	NULL
82	NULL	15872	15892	16040	u,q,s,t,r
83	NULL	16019	16030	16035	NULL
84	NULL	17247	17258	17312	NULL
85	NULL	17385	17396	17425	NULL
86	NULL	17565	17576	17605	NULL
87	NULL	17707	17718	17740	NULL
88	NULL	17981	17992	18013	NULL

Table D.4: The Functions results extracted from the profiler for facebook.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	__rm	145	152	153
2	window[a]	2953	2963	2964
3	h	4717	4719	4720
4	i	5924	5926	5927
5	a._listen[j]	6061	6074	6075
6	a._callbacks[h]	6188	6204	6250
7	h	4717	4719	4720
8	this._cssLinkMap[f]	7295	7315	7325
9	e	7674	7676	7677
10	this._earlyResources	7845	7866	7867
11	this._cssLinkMap[e]	9677	9697	9713
12	this._cssLinkMap[e]	9677	9697	9713
13	this._activeCSSPolls	10193	10214	10215
14	e	7674	7676	7677
15	r	14534	14536	14684
16	c	15963	15965	15966
17	Parent	16773	16780	16943

Table D.5: The Objects results extracted from the profiler for facebook.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for facebook.com when evaluating over 20K character size.

#	Identifier	Location	Start	End	Input Arguments
1	checkChromePromoAlert	2763	2796	2977	NULL
2	dismissChromePromoAlert	3038	3073	3224	NULL
3	hideFbPromoAlert	5982	6010	6049	NULL
4	isRtl	7054	7071	7114	NULL
5	setRtlYva	7119	7146	9617	suffix
6	RichMediaCore_58_12	11229	11260	13228	NULL
7	yt.timing.tick	86	129	359	label, opt_time
8	yt.timing.info	369	409	549	label, value
9	gaiaChangedCallback	3892	3934	4388	autoshare
10	fn	4230	4246	4341	NULL
11	canConnectCallback	4462	4503	4778	autoshare
12	serviceChangedCallback	4850	4895	5385	autoshare
13	window.dismissFbPromoAlert	5809	5849	5972	NULL
14	fixYvaDom	7793	7822	8142	suffix
15	richMedia.clipFlashObject	8644	8753	9096	asset, width, height, offsetTop, offsetRight, offsetBottom, offsetLeft
16	richMedia.unclipFlashObject	9171	9232	9560	asset, width, height
17	RichMediaCore_58_12.prototype.setCsiEventsRecordedDuringBreakout	13338	13424	13528	creative
18	RichMediaCore_58_12.prototype.csiHasValidStart	13531	13599	13677	creative

19	RichMediaCore_58_12.prototype.shouldReportCsi	13680	13747	13837	creative
20	RichMediaCore_58_12.prototype.shouldCsi	13840	13912	14291	asset, creativeType
21	RichMediaCore_58_12.prototype.trackCsiEvent	14294	14356	14405	event
22	RichMediaCore_58_12.prototype.getCsiServer	14408	14464	14641	NULL
23	RichMediaCore_58_12.prototype.reportCsi	14644	14705	16118	creative
24	RichMediaCore_58_12.prototype._isValidStartTime	16121	16191	16234	startTime
25	RichMediaCore_58_12.prototype._convertDuration	16237	16305	16552	duration
26	RichMediaCore_58_12.prototype.convertUnit	16555	16613	16816	pos
27	RichMediaCore_58_12.prototype._isValidNumber	16819	16880	17000	num
28	RichMediaCore_58_12.prototype.writeSurveyURL	17003	17069	17244	creative
29	RichMediaCore_58_12.prototype.postPublisherData	17247	17330	17492	creative, publisherURL
30	RichMediaCore_58_12.prototype.isInterstitialCreative	17495	17569	17700	creative
31	RichMediaCore_58_12.prototype.isBrowserComplient	17703	17771	17958	plugin
32	RichMediaCore_58_12.prototype.shouldDisplayFloatingAsset	17961	18039	18143	duration
33	RichMediaCore_58_12.prototype.isWindows	18146	18199	18259	NULL
34	RichMediaCore_58_12.prototype.isFirefox	18262	18315	18829	NULL
35	RichMediaCore_58_12.prototype.isSafari	18832	18884	19209	NULL

36	RichMediaCore_58_12.prototype.isChrome	19212	19264	19499	NULL
37	RichMediaCore_58_12.prototype.isMac	19502	19551	19607	NULL
38	RichMediaCore_58_12.prototype.isInternetExplorer	19610	19672	19773	NULL
39	RichMediaCore_58_12.prototype.isIPhone	19776	19828	19900	NULL
40	RichMediaCore_58_12.prototype.isIPad	19903	19953	NULL	NULL
41	NULL	3258	3270	6061	NULL
42	NULL	3276	3288	6052	NULL
43	NULL	6483	6495	6567	NULL

Table D.6: The Functions results extracted from the profiler for youtube.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	self.dartLoadedGlobalTemplates_58_12	11056	11095	11096
2	dartCreativeDisplayManagers	10892	none	none
3	csiTimes	13313	none	none

Table D.7: The Objects results extracted from the profiler for youtube.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for youtube.com when evaluating over 20K character size.

#	Identifier	Location	Start	End	Input Arguments
1	c	2756	2773	2896	r,s,q
2	o	2897	2914	3032	r,s,q
3	enable	3048	3065	3108	NULL

4	disable	3110	3128	3171	NULL
5	dispatch	3173	3192	3261	NULL
6	reset	3263	3279	3285	NULL
7	a	5457	5478	6172	A,l,u,n,h
8	e.augment	6376	6405	6841	f,w,j,u,o
9	k[i]	6537	6552	6672	NULL
10	e.aggregate	6843	6872	6902	h,g,f,i
11	e.extend	6904	6930	7182	i,h,f,k
12	e.each	7184	7208	7390	i,h,j,g
13	e.clone	7392	7421	7914	j,k,n,p,i,m
14	e.bind	7916	7936	8115	g,i
15	e.rbind	8117	8138	8317	g,i
16	before	8502	8526	8636	i,k,l,m
17	after	8638	8661	8771	i,k,l,m
18	_inject	8773	8798	9035	h,j,k,m
19	k[m]	8906	8921	8961	NULL
20	detach	9037	9055	9081	h
21	_unload	9083	9104	9105	i,h
22	e.Do.Method	9108	9133	9209	h,i
23	e.Do.Method.prototype.register	9211	9257	9304	i,j,h
24	e.Do.Method.prototype._delete	9306	9347	9391	h
25	e.Do.Method.prototype.exec	9393	9430	9926	NULL
26	e.Do.AlterArgs	9928	9956	9983	i,h
27	e.Do.AlterReturn	9985	10015	10044	i,h
28	e.Do.Halt	10046	10069	10095	i,h
29	e.Do.Prevent	10097	10121	10133	h
30	e.EventHandle	10384	10411	10434	f,g
31	detach	10461	10478	10591	NULL
32	e.CustomEvent	10594	10621	10845	f,g
33	applyConfig	10872	10897	10923	g,f
34	_on	10925	10946	11264	j,h,g,f
35	subscribe	11266	11289	11376	h,g
36	on	11378	11394	11481	h,g
37	after	11483	11502	11586	h,g
38	detach	11588	11608	11772	k,h

39	unsubscribe	11774	11796	11838	NULL
40	_notify	11840	11863	12027	i,h,f
41	log	12029	12046	12065	g,f
42	fire	12067	12082	12333	NULL
43	fireSimple	12335	12357	12505	f
44	fireComplex	12507	12530	12571	f
45	_procSubs	12573	12598	12759	j,g,f
46	_broadcast	12761	12783	12949	g
47	unsubscribeAll	12951	12976	13021	NULL
48	detachAll	13023	13043	13065	NULL
49	_delete	13067	13086	13178	f
50	e.Subscriber	13181	13209	13286	h,g,f
51	_notify	13312	13335	13561	j,h,i
52	notify	13563	13583	13794	g,i
53	contains	13796	13818	13889	g,f
54	l	14261	14274	14588	m
55	on	14603	14623	15746	r,v,p,w
56	subscribe	11776	11796	11838	NULL
57	detach	15808	15830	16819	p,u,o
58	A	16138	16153	16210	G,F
59	unsubscribe	16822	16843	16885	NULL
60	detachAll	16887	16908	16931	m
61	unsubscribeAll	16934	16958	17003	NULL
62	publish	17005	17026	17341	o,p
63	addTarget	17343	17364	17429	m
64	removeTarget	17431	17455	17496	m
65	fire	17498	17514	17866	p
66	getEvent	17868	17890	18001	n,m
67	after	18003	18022	18211	o,n
68	before	18213	18230	18268	NULL
69	a.EventFacade	18551	18578	18916	g,f
70	this.stopPropagation	18704	18735	18756	NULL
71	this.stopImmediatePropagation	18758	18798	18828	NULL
72	this.preventDefault	18830	18860	18880	NULL
73	this.halt	18882	18903	18914	e
74	b.fireComplex	18918	18943	NULL	h

75	NULL	185	198	2118	NULL
76	NULL	2158	2169	2548	NULL
77	NULL	2588	2599	3288	NULL
78	NULL	5404	5416	6204	g
79	NULL	6308	6320	8319	e
80	NULL	6522	6536	6770	r,i
81	NULL	7742	7756	7846	o,f
82	NULL	7867	7881	7894	o,f
83	NULL	7999	8010	8113	NULL
84	NULL	8201	8212	8315	NULL
85	NULL	8417	8429	18432	e
86	NULL	8464	8475	10156	NULL
87	NULL	13892	13903	18427	NULL
88	NULL	13951	13965	14029	m,n
89	NULL	14042	14056	14258	o,q
90	NULL	14871	14885	15001	x,n
91	NULL	17112	17126	17153	t,s
92	NULL	18492	18504	NULL	a
93	v	18505	18516	NULL	NULL

Table D.8: The Functions results extracted from the profiler for yahoo.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	k	2747	2749	2750
2	j.OnloadCache	3033	3047	3286
3	r	3197	3199	3211
4	k	2747	2749	2750
5	g	6499	6501	6502
6	k	2747	2749	2750
7	q	6509	6511	6512
8	e.Env.evt	8430	8440	8462
9	e.Do	8488	8493	9106
10	this.objs[n]	8838	8851	8852
11	this.before	9180	9192	9193

12	this.after	9195	9206	9207
13	e.EventHandle.prototype	10436	10460	10592
14	this.subscribers	10728	10745	10746
15	this.afters	10748	10760	10761
16	e.CustomEvent.prototype	10847	10871	13179
17	e.Subscriber.prototype	13288	13311	13890
18	l.prototype	14590	14602	18269
19	G	14831	14833	14834
20	m	17099	17101	17102
21	a.Env._eventstack	19099	19117	19188

Table D.9: The Objects results extracted from the profiler for yahoo.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for yahoo.com when evaluating over 20K character size.
No Functions were found for live.com when evaluating over 20K character size.

#	Identifier	Location	Start	End
1	srf_oTemplate	3886	none	none
2	g_DO	10038	none	none
3	srf_oTemplate	16150	none	none

Table D.10: The Objects results extracted from the profiler for live.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for live.com when evaluating over 20K character size.

#	Identifier	Location	Start	End	Input Arguments
1	gaia_onLoginSubmit	6764	6794	6903	NULL
2	gaia_setFocus	12081	12106	12349	NULL
3	window.onload	6373	6401	6662	NULL

4	gaia_loginForm.Email.onkeypress	11993	12038	12076	NULL

Table D.11: The Functions results extracted from the profiler for blogger.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects and event listeners were found for blogger.com when evaluating over 20K character size.

#	Identifier	Location	Start	End	Input Arguments
1	addEV	644	665	767	C,B,A
2	G	768	781	808	A
3	F	1560	1572	1692	NULL
4	H	1693	1708	2431	W,S
5	U	1843	1855	2082	NULL
6	J	2432	2445	2541	S
7	C	2542	2555	2676	T
8	I	2955	2968	3002	C
9	K	3003	3016	3049	C
10	S	3050	3063	3163	C
11	U	3164	3177	3245	C
12	P	3246	3263	3374	G,X,C
13	N	3375	3388	3439	C
14	R	3440	3453	3665	X
15	H	3666	3679	3874	G
16	C	3777	3792	3873	Z,b
17	O	3875	3888	4054	Y
18	L	4055	4068	4278	G
19	C	4411	4424	4531	b
20	G	4532	4545	4639	Y
21	Z	4809	4821	4880	NULL
22	e	4881	4894	5304	n

23	k	5305	5317	5514	NULL
24	l	5515	5527	5547	NULL
25	g	5548	5560	5577	NULL
26	i	5578	5590	5667	NULL
27	b	5668	5681	5733	n
28	G	5734	5747	5793	n
29	n	6642	6654	6716	NULL
30	e	6717	6729	6907	NULL
31	i	6908	6920	6972	NULL
32	G	6973	6985	7009	NULL
33	b	7010	7023	7111	q
34	c	7112	7125	7221	q
35	f	7222	7235	7390	q
36	X	7391	7403	7627	NULL
37	Y	7628	7643	7778	r,q
38	p	7779	7794	7882	q,s
39	j	7883	7895	8347	NULL
40	Z	8348	8360	8454	NULL
41	k	8455	8467	8736	NULL
42	g	8737	8749	9007	NULL
43	G	9401	9413	9514	NULL
44	X	9531	9543	9558	NULL
45	Y	9559	9572	9603	Z
46	X	9803	9816	9919	C
47	Y	9920	9933	9979	C
48	G	10128	10141	10387	Y
49	C	10669	10681	10755	NULL
50	G	10756	10768	10879	NULL
51	a	11050	11062	11141	NULL
52	d	11142	11154	11340	NULL
53	b	11341	11354	11489	e
54	X	11490	11502	11529	NULL
55	G	11530	11542	11998	NULL
56	a[i].onclick	246	269	426	NULL
57	w.onunload	2738	2759	2760	NULL
58	ini	4647	4662	4701	X
59	bdsug.sugkeywatcher.on	5843	5876	5969	NULL

60	bdsug.sugkeywatcher.off	5971	6005	6101	NULL
61	rm	6222	6236	6563	n
62	rm	9022	9036	9351	q
63	rm	9618	9632	9768	Z
64	rm	9994	10008	10090	C
65	rm	10402	10416	10517	Y
66	bdsug.sug	10525	10546	10586	C
67	bdsug.initSug	10588	10612	10636	NULL
68	rm	10894	10908	10974	X
69	rm	12033	12047	12106	e
70	NULL	428	439	516	NULL
71	NULL	874	885	2695	NULL
72	NULL	2002	2013	2044	NULL
73	NULL	2714	2725	2735	NULL
74	NULL	2840	2851	12618	NULL
75	NULL	3291	3303	3332	Y
76	NULL	3310	3321	3331	NULL
77	NULL	4399	4410	4703	NULL
78	NULL	4714	4725	6566	NULL
79	NULL	5381	5392	5422	NULL
80	NULL	6577	6588	9354	NULL
81	NULL	7140	7151	7220	NULL
82	NULL	9365	9376	9771	NULL
83	NULL	9782	9793	10093	NULL
84	NULL	10104	10115	10520	NULL
85	NULL	10657	10668	10977	NULL
86	NULL	10988	10999	12109	NULL

Table D.12: The Functions results extracted from the profiler for baidu.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	bdimeHW	813	821	822
2	window.bdsug	4341	4354	4355
3	bdsug.sug	4357	4367	4368

4	bdsug.sugkeywatcher	4370	4390	4391
5	X.__MSG_QS__	4663	4676	4677
6	G	9798	9800	9801

Table D.13: The Objects results extracted from the profiler for baidu.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	C	736	B	A	addEV at 644 with callees at 2098, 2677, 2700.
2	G	3344	X	C	P at 3246 with callees at 6128, 6147, 6190, 8039, 8058, 8076, 8095, 9515, 11624, 11650.
3	X	5924	keydown	e	None

Table D.14: The Listener results extracted from the profiler for baidu.com when evaluating over 20K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	$	96	109	143	a
2	addLoadEvent	145	169	280	a
3	getLang	281	299	437	NULL
4	convertChinese	713	739	880	a
5	convertZhLinks	881	906	1126	NULL
6	setLang	1157	1176	1416	a
7	NULL	450	461	710	NULL
8	NULL	1429	1440	1720	NULL

Table D.15: The Functions results extracted from the profiler for wikipedia.org when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No Objects were found for wikipedia.org when evaluating over 20K character size.

#	Element	Location	Event	Handler	Factory
1	Document	197	load	a	addLoadEvent at 145 with callees at 438, 1128, 1417

Table D.16: The Listener results extracted from the profiler for wikipedia.org when evaluating over 20K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	fn	948	961	1147	NULL
2	each	3865	3897	3999	a, fn, opt_scope
3	Animate	4515	4548	4745	el, prop, opts
4	HexToR	11725	11744	11800	h
5	HexToG	11808	11827	11883	h
6	HexToB	11891	11910	11966	h
7	getRest	13398	13417	13721	NULL
8	Occasionally	17021	17067	17272	job, decayFn, interval
9	IntervalJob	18983	19026	19180	time, loop, callback
10	onCondition	723	752	861	D,C,A,B
11	window.self.onload	1081	1115	1145	evt
12	Array.prototype.filter	3081	3128	3346	fn, thisObj
13	Array.prototype.indexOf	3396	3442	3593	el, start
14	Animate.canTransition	4857	4892	5054	NULL
15	Animate.prototype._setStyle	5158	5202	5460	val

16	Animate.prototype._animate	5530	5570	6019	NULL
17	Animate.prototype.start	6079	6116	6261	NULL
18	TWTR.Widget	6433	6462	6487	opts
19	matchUrlScheme	6757	6788	6872	url
20	getClassRegEx	7007	7035	7273	c
21	getByClass	7285	7328	7785	c, tag, root, apply
22	browser	7797	7818	7922	NULL
23	byId	7936	7956	8066	id
24	trim	8078	8099	8148	str
25	getViewportHeight	8160	8191	8518	NULL
26	getTarget	8530	8571	8660	e, resolveTextNode
27	resolveTextNode	8672	8703	8862	el
28	getRelatedTarget	8874	8905	9178	e
29	insertAfter	9190	9228	9302	el, reference
30	removeElement	9314	9343	9429	el
31	getFirst	9441	9465	9499	el
32	withinElement	9511	9539	9828	e
33	getStyle	9840	9862	10515	NULL
34	has	10616	10637	10724	el, c
35	add	10734	10755	10872	el, c
36	remove	10882	10906	11062	el, c
37	add	11204	11232	11462	el, type, fn
38	remove	11471	11502	11677	el, type, fn
39	hex_rgb	11695	11716	12068	NULL
40	bool	12162	12180	12227	b
41	def	12237	12254	12306	o
42	number	12316	12336	12397	n
43	string	12406	12426	12472	s
44	fn	12482	12498	12546	f
45	array	12556	12575	12689	a
46	absoluteTime	12849	12876	13781	s
47	hour	13022	13040	13315	NULL
48	timeAgo	13973	14004	15256	dateString
49	link	15436	15458	15808	tweet
50	at	15818	15838	16067	tweet
51	list	16077	16099	16333	tweet
52	hash	16343	16365	16599	tweet

53	clean	16609	16632	16704	tweet
54	start	17411	17429	17485	NULL
55	stop	17592	17609	17719	NULL
56	run	17769	17785	18407	NULL
57	destroy	18528	18548	18626	NULL
58	set	19221	19245	19287	haystack
59	add	19297	19319	19366	needle
60	start	19473	19490	19750	NULL
61	stop	19852	19868	NULL	NULL
62	NULL	30	42	168	g
63	NULL	821	832	855	NULL
64	NULL	1035	1046	1076	NULL
65	NULL	3631	3643	NULL	NULL
66	NULL	6201	6213	6251	NULL
67	NULL	6493	6505	NULL	NULL
68	NULL	9953	9977	10236	el, property
69	NULL	10344	10368	10500	el, property
70	NULL	11379	11391	11442	NULL
71	NULL	11981	11996	12060	hex
72	NULL	15548	15580	15798	link, m1, m2, m3, m4
73	NULL	15899	15922	16057	m, username
74	NULL	16165	16188	16323	m, userlist
75	NULL	16414	16441	16589	m, before, hash
76	NULL	17828	17840	18397	NULL
77	NULL	18291	18304	18353	NULL
78	NULL	19640	19652	19698	NULL

Table D.17: The Functions results extracted from the profiler for twitter.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	page	711	716	717
2	twttr	6544	6552	6553
3	reClassNameCache	6928	6947	6948
4	classes	10598	10608	11068

5	events	11187	11196	11683
6	is	12149	12154	12695
7	ify	15422	15428	16710
8	Occasionally.prototype	17279	17304	18632
9	IntervalJob.prototype	19188	19212	NULL

Table D.18: The Objects results extracted from the profiler for twitter.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Element	Location	Event	Handler	Factory
1	el	11279	type	fn	add at 11204 with a callee at 11143.

Table D.19: The Listener results extracted from the profiler for twitter.com when evaluating over 20K character size. The numbers under the Location column are the number of characters from the start of the analysed text. The Element, Event and Handler columns list the listener's monitoring element, type of event, and its handler respectively.

#	Identifier	Location	Start	End	Input Arguments
1	$	7777	7790	7851	o
2	CreatAjax	7854	7879	8092	a,b,c
3	CreatXmlRequest	8095	8122	8301	NULL
4	parseInfo	8304	8329	8375	a,b,c
5	swTabs	8380	8398	9038	e
6	swLabs	9041	9074	9665	sobject, sid, snum
7	loadNewsMap	9668	9690	9760	NULL
8	tagOver	9763	9782	10001	a
9	tagOut	10004	10022	10055	a
10	getNames	10059	10090	10276	obj,name,tij
11	cplay	10280	10296	10522	NULL

12	formatPageZone	10526	10553	11114	NULL
13	send_request	11118	11146	11665	url
14	loadPage	11669	11690	12352	NULL
15	tail	12355	12372	12982	NULL
16	getExpires	13067	13089	13181	a
17	setdefSkin	13184	13205	13642	NULL
18	setdef	13646	13663	13999	NULL
19	setSkin	16051	16070	16251	n
20	formatSkin	16255	16278	16494	NULL
21	_openSkin	16498	16541	16582	o1, o2, max, min, speed, n
22	_closeSkin	16586	16630	16672	o1, o2, max, min, speed, n
23	openSkin	16696	16738	17315	o1, o2, max, min, speed, n
24	closeSkin	17319	17362	17921	o1, o2, max, min, speed, n
25	loadJS	17924	17950	18379	url, load
26	loadPng	18383	18404	19326	o
27	call_0410	19350	19383	0	et, item, page
28	QosS.isAllLoaded	372	400	609	B
29	QosS.checkLoad	611	636	1519	NULL
30	(QosS.c.onerror	1277	1303	1317	NULL
31	QosS.topSpan	1521	1547	1772	A,B
32	QosS.killTimer	1774	1799	1892	NULL
33	QosS.endCheck	1894	1918	2533	NULL
34	MiniSite.$	2925	2947	3007	s
35	load	3031	3060	3586	sUrl, fCallback
36	_script.onreadystatechange	3335	3372	3466	NULL
37	_script.onload	3506	3531	3553	NULL
38	set	3610	3654	3911	name, value, expires, path, domain
39	get	3915	3933	4075	name
40	clear	4079	4111	4278	name, path, domain
41	insertFlash	4639	4672	5449	elm, url, w, h
42	_print	5469	5494	5524	province
43	print	5528	5544	6348	NULL

44	ok	5552	5565	5596	NULL
45	window.onerror	6482	6511	6529	NULL
46	window.setTimeout	6583	6639	6897	fCallback, nDelay, oObject
47	obj.onreadystatechange	7986	8022	8091	NULL
48	f[i].onclick	10915	10938	10984	NULL
49	f[i].onmouseover	10989	11016	11050	NULL
50	f[i].onmouseout	11052	11078	11112	NULL
51	$("tailorArrow").onclick	11966	12003	12079	NULL
52	$("tailorArrow").onmouse-over	12081	12122	12143	NULL
53	$("tailorArrow").onmouse-out	12146	12186	12206	NULL
54	p[i].onclick	16382	16407	16426	NULL
55	s.onclick	16428	16451	16492	NULL
56	move	16794	16809	17272	NULL
57	$("pro_arrow").onclick	16954	16990	17024	NULL
58	a.onclick	17141	17164	17198	NULL
59	move	17417	17434	17878	NULL
60	$("pro_arrow").onclick	17531	17566	17599	NULL
61	a.onclick	17706	17728	17762	NULL
62	_script.onreadystatechange	18188	18227	18318	NULL
63	_script.onload	18331	18358	18373	NULL
64	img.onerror	19943	19969	19985	NULL
65	null	5758	5769	6177	NULL
66	null	6774	6785	6819	NULL
67	null	9835	9846	9856	NULL
68	null	9967	9978	9993	NULL

Table D.20: The Functions results extracted from the profiler for qq.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

#	Identifier	Location	Start	End
1	QosS.C	81	88	172
2	QosS.G	174	181	370

3	MiniSite.Browser	2656	2673	2922
4	MiniSite.JsLoader	3010	3028	3588
5	MiniSite.Cookie	3591	3607	4280
6	MiniSite.Home	4283	4297	6350
7	QosS	63	none	none
8	MiniSite	2633	none	none

Table D.21: The Objects results extracted from the profiler for qq.com when evaluating over 20K character size. The numbers under the Location, Start and End columns are the number of characters from the start of the analysed text.

No event listeners were found for qq.com when evaluating over 20K character size.

Appendix E

Identification Reference of Tell-Signs

Reference	Tell-Signs Description
TS1-a	Element is identified via Anchor tag pattern.
TS2-r	Element has related Z-Index $>$ 9 element.
TS2-a	Element has Z-Index $>$ 9.
TS2-b	Element ID or style Class found with Z-index $>$ 9 in CSS.
TS3-a	Element position is set in CSS.
TS4-a	Pattern of display = ‘block’ is found!
TS4-b	Pattern of display = ‘none’ is found!
TS5-a	Pattern of visibility = ‘hidden’ is found!
TS5-b	Pattern of visibility = ‘visible’ is found!
TS6	Element is subset of another element.
TS7	Element has click event monitored.
TS8	Element has hover event monitored.
TS9	Element has keyboard events monitored.
TS9-r	Element has related element that has keyboard events monitored.
TS10	Element is monitored for the type of key pressed.
TS11	Element’s Autocomplete property is set to off.
TS12	DOM Content changed automatically.
TS13	Element do not contain any media nodes.
TS14	No content changed found.
TS15	Candidate has sibling element.
TS16	Has similar related source code with sibling element
TS17	Is child of another element.
TS18	Element has tabIndex attribute.
TS19	Has elements in a list form pattern $--$ ul$\vert$ol$\vert$dl $-->$ li$\vert$dt$\vert$dd $-->$ a

TS20-a	Has increment pattern: Pattern $+=$ found!
TS20-b	Has increment pattern: Pattern $+$ or $++$ found!
TS21-a	Has decrement pattern: Pattern $-=$ found!
TS21-b	Has decrement pattern: Pattern $-$ or $--$ found!
TS22	Related to a Popup Content/Collapsible Panel widget.
TS23	Attempt to change the styling position of the element found!
TS24	Element has aria-haspopup attribute.

Table E.1: List of common tell-signs that can be shared among different types of widgets.

<table>
<tr><th>Reference</th><th>Tell-Signs Description</th></tr>
<tr><td>A-TS1</td><td>Keyword: ASL found!</td></tr>
<tr><td>A-TS2</td><td>Keyword: search|query found!</td></tr>
<tr><td>A-TS3</td><td>Keyword: auto|list found!</td></tr>
<tr><td>A-TS4</td><td>Keyword: form found!</td></tr>
<tr><td>A-TS5</td><td>Keyword: suggest found!</td></tr>
</table>

Table E.2: List of tell-signs private to Auto Suggest List (ASL) widget.

<table>
<tr><th>Reference</th><th>Tell-Signs Description</th></tr>
<tr><td>TK-TS1</td><td>ticker|marquee keyword found.</td></tr>
<tr><td>TK-TS2</td><td>scroller|flasher keyword found.</td></tr>
<tr><td>TK-TS3</td><td>Sibling subset element found.</td></tr>
<tr><td>TK-TS4</td><td>Element tag name is Marquee.</td></tr>
</table>

Table E.3: List of tell-signs private to Ticker widget.

<table>
<tr><th>Reference</th><th>Tell-Signs Description</th></tr>
<tr><td>PC-TS1</td><td>Keyword - dialog|float|hover|popup|selector|button|btn found!</td></tr>
<tr><td>PC-TS2</td><td>Keyword - collapsible|drawer|expand|shrink|contract|enlarge found!</td></tr>
<tr><td>PC-TS3</td><td>Keyword - offsetHeight|offsetWidth found!</td></tr>
<tr><td>PC-TS4</td><td>Keyword - expand|shrink|contract|enlarge found!</td></tr>
<tr><td>PC-TS5</td><td>Keyword - collapsible found!</td></tr>
<tr><td>PC-TS6</td><td>Keyword - drawer found!</td></tr>
<tr><td>PC-TS7</td><td>Keyword - selector|button|btn found!</td></tr>
<tr><td>PC-TS8</td><td>Keyword - popup|dialog found!</td></tr>
</table>

<table>
<tr><td>PC-TS9</td><td>Keyword - float|hover found!</td></tr>
<tr><td>PC-TS10</td><td>Keyword - panel|content found!</td></tr>
</table>

Table E.4: List of tell-signs private to Popup Content widget and Collapsible Panel widget.

<table>
<tr><th>Reference</th><th>Tell-Signs Description</th></tr>
<tr><td>T-TS1</td><td>Have keywords – tab|nav</td></tr>
<tr><td>T-TS2</td><td>Have keywords – folder|panel|section</td></tr>
<tr><td>T-TS3</td><td>Has keyword – selected</td></tr>
</table>

Table E.5: List of tell-signs private to Tabs widget.

<table>
<tr><th>Reference</th><th>Tell-Signs Description</th></tr>
<tr><td>CS-TS1</td><td>Keywords – more|next|right|down|increment found.</td></tr>
<tr><td>CS-TS2</td><td>Keywords – carousel|slide found.</td></tr>
<tr><td>CS-TS3</td><td>Keywords – scroll|arrow|btn|button found.</td></tr>
<tr><td>CS-TS4</td><td>Condition pattern to check if it is the end of the list is found.</td></tr>
<tr><td>CS-TS5-a</td><td>Pattern – = start|begin|0|1 found! Determine Next Button process to be loop.</td></tr>
<tr><td>CS-TS5-b</td><td>Keyword – loop found! Determine Next Button process to be loop.</td></tr>
<tr><td>CS-TS5-c</td><td>Keyword – carousel found. Determine Next Button process to be loop.</td></tr>
<tr><td>CS-TS6-a</td><td>No keywords or loop pattern found. Determine Next Button process to be finite.</td></tr>
<tr><td>CS-TS6-b</td><td>Keyword – SlideShow found. Determine Next Button process to be finite.</td></tr>
<tr><td>CS-TS7</td><td>Keywords – less|prev|back|left|up|decrement found!</td></tr>
<tr><td>CS-TS8</td><td>Condition pattern to check if the start of the list found.</td></tr>
<tr><td>CS-TS9-a</td><td>Pattern – = end|last|length|max found. Determine Previous Button process to be loop.</td></tr>
<tr><td>CS-TS9-b</td><td>Keyword – loop found. Determine Previous Button process to be loop.</td></tr>
<tr><td>CS-TS9-c</td><td>Keyword – carousel found. Determine Previous Button process to be loop.</td></tr>
<tr><td>CS-TS10-a</td><td>No keyword or loop pattern found. Determine Previous Button process to be finite.</td></tr>
<tr><td>CS-TS10-b</td><td>Keyword – slideshow found. Determine Previous Button process to be finite.</td></tr>
<tr><td>CS-TS11</td><td>Element is related to another candidate.</td></tr>
</table>

Table E.6: List of tell-signs private to Carousel widget and Slide Show widget.