Universität Koblenz-Landau

Master Thesis

A feature model for web testing tools

Author:
Jan Stefan Rüther

Supervisor:
Prof. Dr. Ralf Lämmel

A thesis submitted in fulfilment of the requirements
for the degree of Master of Science

in the

Arbeitsgruppe Softwaresprachen
Institut für Softwaretechnik

May 2015

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

(Ort, Datum) (Unterschrift)

"This is what I love, and can't stop loving. . . "

Showtek - FTS

UNIVERSITÄT KOBLENZ-LANDAU

Abstract

Fachbereich 4: Informatik
Institut für Softwaretechnik

Master of Science

A feature model for web testing tools

by Jan Stefan RÜTHER

Web application testing is an active research area. Garousi et al. did a systematic mapping study and classified 79 papers published between 2000-2011 [1]. However, there seems to be a lack of information exchange between the scientific community and tool developers. This thesis systematically analyzes the field of functional, system level web application testing tools. 194 candidate tools were collected in the tool search and screened, with 23 tools being selected as foundation of this thesis. These 23 tools were systematically used to generate a feature model of the domain. The methodology to support this is an additional contribution of this thesis. It processes end user documentation of tools belonging to an examined domain and creates a feature model. The feature model gives an overview over the existing features, their alternatives and their distribution. It can be used to identify trends and problems, extraordinary features, help decision making of tool purchase or guide scientists how to focus research.

Acknowledgements

Ich danke Prof. Ralf Lämmel für die ausführliche und engagierte Unterstützung dieser Arbeit. Weiterhin danke ich Martin Leinberger, meinem Zweitprüfer.

Ich danke meinen Korrektoren Christian und Britta, für ihre konstruktiven und hilfreichen Kommentare.

Ich danke meiner Freundin Lena für ihre moralische Unterstützung.

Ich danke meinen Eltern. Ohne ihre durchgängige, finanzielle Unterstützung wäre es mir nicht möglich gewesen ein Studium in oder teils unter Regelstudienzeit abzuschließen. Außerdem danke ich ihnen für ihr Vertrauen und ihren Rat.

Ich danke den Erfindern und Songschreibern der elektronischen Musik sowie den DJs von Technobase.fm und Hardbase.fm für ihren meist ehrenamtlichen Einsatz. Durch diese aufputschende und aber im Gegenzug durch ihre Gleichmäßigkeit nicht ablenkende Musik habe ich einige lange Tage und Nächte überstanden.

Contents

List of Figures

List of Tables

Abbreviations

ALM	**A**pplication **L**ifecycle **M**anagement
API	**A**pplication **P**rogramming **I**nterface
CSS	**C**ascading **S**tyle **S**heets
DOM	**D**ocument **O**bject **M**odel
DSL	**D**omain **S**pecific **L**anguage
GPL	**G**eneral **P**urpose **L**anguage
GUI	**G**raphical **U**ser **I**nterface
IDE	**I**ntegrated **D**evelpment **E**nvironment
ITE	**I**ntegrated **T**esting **E**nvironment
OCR	**O**ptical **C**haracter **R**ecognition
SUT	**S**ystem **U**nder **T**est
TM	**T**est **M**anagement
URL	**U**niform **R**esource **L**ocator

Chapter 1

Introduction

The web is an ubiquitous part of our society. Up to the present day the importance of the web is growing. Web applications are becoming more complex and increasingly replace the traditional desktop applications [1, 2].

The origins of web application complexity are multifaceted [3, 4]. It often causes the applications to be error-prone and testing to be difficult. Another problem is that traditionally web testing is often done manually [1], which is becoming similarly more complex until the point of impracticality [2].

Web application testing is an active research area. Garousi et al. did a systematic mapping study and classified 79 papers published between 2000-2011 [1]. Among other things, they documented the increasing popularity of web application testing by sorting the publications per year.

Garousi et al. discovered that half of the papers mention an accompanying tool implementation [1]. But only 6 of the 79 papers provide a downloadable tool. Garousi et al. alerted the scientific community to take tool implementations more seriously if they intend to have an impact in industry.
Besides Dobolyi et al. introduced a semantics-based automated oracle comparator for the regression testing of web-based applications [5]. They stated that their approach is 2.5 to 50 times as accurate as current industrial practice. They define what they assume is industrial practice or the state of the art, however they do not substantiate their statement by citation.

Motivation

These examples suggest that there is a need for a comprehensive analysis of state of the art in the area of web application testing tools. There seems to be an information gap between the scientific community and tool developers. Of course there are joint research projects, but these normally include just one software company and one university which cooperate for a limited period of time and there are far too many tools in this domain to get a realistic overview by doing a research project[1].

To the best of my knowledge, until now there has not been an attempt to systematically analyze the web testing tools on the market. This thesis investigates what features the functional web application testing tools on the market exhibit and how these features can be categorized.

Research beneficiaries

The result of this thesis helps researchers to obtain an overview of the capabilities, that the state of the art tools possess. Web testers benefit from this thesis as it gives an overview of the tools on the market as well as the existing features. The provided presentation of features guides the tool selection process as it can disclose the testers' requirements in the tool. Tool developers profit by the thesis' result as they gain an overview of their competitors. Perchance this helps to focus their efforts on the features that distinguish their tools.

1.1 Research context

This section describe the domain of web application testing tools. The focus is set on the terminology of the domain.

The field of web application testing has much interference with the field of web application automation. Also adjoining is of course GUI testing as well as mobile testing and web service testing[2].

[1] This thesis identified 95 actively developed tools in the domain of functional web application testing.
[2] Often known as SOA (Service-oriented architecture) testing [6].

Table 1.1 explains the terminology of web application testing tools. Regression testing is a functional testing approach. In this domain both are often used synonymously. All listed terms are often encountered when reading about web testing tools, with the exception of usability testing.

Functional testing can be categorized according to the granularity of the test. As defined in the literature [14] there are three testing levels: unit, integration and system testing. Tests can cover a module in isolation (unit testing), a few modules and their integration (integration testing) or the whole system (system testing).
Garousi et al. define a modules as a single source-code file or function [1]. Thus an HTML validator is an unit testing approach. An example for integration testing is the communication of client-side JavaScript with server-side JSP or PHP. Testing the application through its GUI is system testing. In contrast to other literature defining testing of a single page through the GUI as unit testing and furthermore testing of multiple pages as integration testing respectively [12], this thesis applies the former definition by Garousi et al.

System-level testing requires testing the application through the GUI. In the case of web applications the GUI is provided by the browser. Thus, system-level testing tools hook into the browser and indirectly control the web application by firing mouse and keyboard events and reading the displayed data.
Another approach is to develop a special browser for the purpose of testing. These browsers do not visualize the page and are called *headless browsers*. They enable much faster testing without the overhead of real browsers. However even the most advanced do not support all features the real browsers do.
A third approach operates the browser by remote control using image recognition. This approach is not web specific as the tools do not emulate or hook into the browser but emulate the mouse and "the users eyes". On the GUI side, the tools do no differentiate between the components of the browser, the web application under test or the operation system GUI elements.

Functional system-level web application testing tools can be categorized in three overlapping categories (Table 1.2).

Term	Description
functional testing approaches	
Regression testing	Regression testing is the process of retesting an application after changes have been made. The test should ensure that no bugs have been introduced, more precisely that the behavior of the application has not unexpectedly changed [7]. Marín et al. observe that Regression testing needs coverage analysis methods that identify the smallest test suite needed to ensure the tests cover the code [8].
Cloud testing	Cloud testing is the combination of web testing and cloud computing. Thus its a business model and not a web testing approach. Cloud testing can provide all functional or non-functional testing approaches [9, 10]. The keyword TaaS (Testing as a service) should be mentioned in that context [11].
non-functional testing approaches [12]	
Performance testing	Performance testing ensures that the systems performances (e.g., response time, service availability) fulfill the specifications. Thousands of simultaneous users that access the page are emulated.
Load testing	Load testing is a specialization of performance testing. The performance is measured with a predefined load level. In this case the time to perform a special task or function is measured. Many of users are simulated and the test fails if the task is not performed within the defined time frame.
Stress testing	Stress testing is a specialization of performance testing, where the system is tested at the limits or beyond the specified requirements. For instance the system should recover after a collapse in a defined time frame.
Security testing	Security testing is the process of securing the system and its data against improper usage and additionally to ensure that regular users have access to its functionality.
Compatibility testing	The role of compatibility testing is to detect failures due to the employment of different platforms, browsers (cross browser testing) or screen resolutions.
Usability testing	Usability testing helps to achieve that the system is easy and intuitive to use. Accessibility testing is a specialization that ensures that the system is accessible with reduced hardware or software on the client-side. In most cases the goal is to verify that handicapped people have access to the application. There are accepted guidelines that define how accessibility can be verified, like the Web Content Accessibility Guidelines [13]

Table 1.1: Web testing terminology

Category	Description
Framework/Library	Frameworks and libraries are used to provide testing of web applications via general-purpose languages. They often benefit from the host language's features that improve the readability and reusability of the testing code.
Integrated testing environment	In a way similar to the popular IDEs like Eclipse or NetBeans, integrated testing environments (ITE) are extensive tool suites that provide an integrated environment for developing tests. They provide different views on the application under test and test debugging support for instance.
Browser driver	Browser drivers are the applications that actually drive or emulate the browser. Every tool of the former two categories either develops its own driver or more likely uses an open source driver. There are two types of browser drivers. First the headless browsers that emulate the behavior of a web browser. And second the browser controllers that hook into an existing browser and take over control.

TABLE 1.2: Categories for functional web testing tools.

1.2 Research question

This thesis systematically reviews the variety and capabilities of the functional web application testings tools currently on the market. The leading question can be put like that.

- Which features can be identified in the field of functional, system-level, web application testing tools?

The previous section pictured the domain of web application testing. This thesis concentrates on functional testing tools. It concentrates on system-level testing excluding unit-level testing tools that validate source code files of the various languages that are used to build web applications. Also excluded are integration-level testing tools, which for instance are used to debug AJAX scripts. The thesis handles web application testing, or more precisely web-specific application testing. Thus tools that primarily work with image recognition are excluded. Table 1.2 introduces categories for the tools of

this thesis. Not included in this thesis is the analysis of browser drivers. Most of them cannot be used independently or only provide minimalistic features. Doing web testing should be preferred with the tools that build on top of them.

The identified features are put into the context of a feature model. As the model is organized hierarchically, the features are to be categorized. This is a necessity for beneficiaries to gain an overview and to compare features.

1.3 Contributions

This thesis has two contributions. A methodology for feature model generation with the aim to picture the overall state of the art within a domain of tools. And the exemplary execution of this methodology in the field of functional web application testing.

The developed methodology has the aim to provide an overview over a domain of tools. The tools have to provide an end user manual to be includable. In the first place preliminarily feature models are constructed for every tool. Afterwards these preliminarily feature models are combined to one feature model. The process in detail is presented in chapter 3.

In this thesis the domain of functional, system-level web application testing tools is analyzed. 23 tools are analyzed and the result is a feature model including 313 features. They are used to discuss for example trends and problems, the distribution of features or the similarity of tool tuples by considering the set of common features in comparison to the all features.

1.4 Structure of the thesis

The thesis is structured as follows: Chapter 2 presents similar approaches and introduces the CaVE information retrieval model that is the foundation of the developed methodology, which is introduced in chapter 3. Chapter 4 describes the application of the methodology. Chapter 5 builds the main part of the thesis. It pictures and explains the feature model and its features. Chapter 6 presents an interpretation of the results and chapter 7 summarizes the thesis and discusses threads to its validity.

Appendix A contains the specific data of all features belonging to each tool.

Chapter 2

Related Work

In this chapter the methodology of this thesis is compared to other approaches in the literature. Additionally the CaVE information retrieval model is introduced.

2.1 Tool comparison approaches

There are different approaches to compare testing tools. The traditional approach is the qualitative analysis. This methodology includes defining criteria first and then later describing and rating the tools with the criteria in mind. It often involves testing the tools manually. Some approaches based on this generel idea are [15], [16] and [17]. Another common approach is to design sample tasks and measure the time needed with the different tools (e.g., [18, 19]). To gain generalizable results that way, it is necessary to conduct an empirical experiment (e.g., [20]). An exceptional testing tool comparison is the approach of Srivastava and Ray [21]. They used and introduced the Fuzzy Analytical Hierarchy Process (FAHP) to compare six system-level GUI automation tools. Their approach relies on experts from CMM[1] level 5 organizations that provide fuzzy numbers in different categories.

All these approaches have a restriction on the number of tools that can be analyzed. Investigation indicates that the limit is about 10 tools. For example Martins et al. analyze 9 tools [16]. The introduced methodology is not restricted to such a limit. Compared to the other approaches this is because it is not primarily to compare tools,

[1] Capability Maturity Model

but to identify the features that are interesting to compare tools. Transmuting the results of this thesis to a complete tool comparison would involve the investigation of all tools for all detected features.

2.2 The CaVE information retrieval model

The CaVE approach (Commonality and Variability Extraction) is a methodology used in the transition process of software product lines. The approach uses the available end user documentations to generate requirement artifacts like features, domains, use cases, requirements and their relations. CaVE was developed by Isabel John at the Fraunhofer IESE[2] in the context of software product lines [22].

The SEI[3] defines software product lines as a set of software systems that share a common, managed set of features and are developed from a set of core assets [23]. The idea of software product lines is software reuse and thus the reduction of costs and time to market as well as the improvement of quality of newly designed products.
The development process is separated into two tasks, the domain engineering and the application engineering. The goal of domain engineering is to develop core assets. Application engineering uses and tweaks these core assets to build specific applications, adding application-specific assets.
Software product lines are in almost every scenario not designed from scratch but introduced from an existing set of software systems. This transition process is supported by the CaVE approach.

In most cases the transition process involves reverseengineering the software requirements, because they are not properly documented or have not been updated to conform to the changes made by maintenance and evolution. Traditionally reengineering involves intensive interviews with domain experts. Isabel John describes that these domain experts are often very busy, thus the goal of CaVE was to lower their workload by extracting artifacts from end user documentation, which is in most cases existing and up-to-date.

[2] Fraunhofer institute for experimental software engineering
[3] http://www.sei.cmu.edu/productlines/

The software product lines requirements are extracted from several software systems. The transition process requires raising the commonality and variability these artifacts have. The CaVE approach handles the extraction of these dependencies for all extracted artifacts.

John defines extraction patterns (2.2.1) and the extraction process belonging to it (2.2.2)).

2.2.1 Extraction Patterns

A key concept of the CaVE approach is to define 38 patterns that formalize the extraction process. Figure 2.1 displays an example pattern. Every pattern is described by a name, a short and a long description, the part of the end user documentation that is processed (input), the type of information that is generated (output), an example and a mathematical definition (query). Furthermore there are estimates for precision and recall for some patterns, that were generated by validating the approach in case studies with three real world product lines of software development companies.

Name	**1**	**Heading-Feature**
Short Description	Headings of sections or subsections typically contain features	
Input	Heading	
Output	Feature	
Query	$\exists q \in d_i \; dca(q) = "Heading"$	
Recall	+	
Precision	++	
Long Description	As features describe functionalities that are of importance for the user, they are found at prominent places in the UD.	
Example	"Send SMS" as a heading of a mobile phone manual is a Feature of the mobile phone	
Related Patterns	2	

FIGURE 2.1: Sample pattern "Heading ->Feature" [22]

2.2.2 Extraction Process

The CaVE process has two actors. A product line engineer that does not need knowledge of the domain and a domain expert. The CaVE process consists of the three phases preparation, analysis and validation. The first two phases are done by the product line engineer self-reliant. The third phase is done by the product line engineer in cooperation with the domain expert.
In the *preparation phase* the user documentation is prepared and the appropriate patterns are selected. This involves:

1. The documentation of all systems related to the product line is collected.
2. The documentation of two or three tools is selected to be analyzed in parallel. The selected tools should be most diverging to cover the variability of the system.
3. If possible, the documentations are split into manageable parts ranging from 3 to 10 pages. The splitting is done with the additional goal that each part should have a corresponding part in each of the other documentations that handles equal topic. Thus the commonality and variability can be analyzed for each tuple.
4. For each tuple of parts, the product line engineer browses through the text to estimate the variability. If the parts differ in more than one third, the parts are compared sequentially otherwise the comparison is done in parallel. Sequential and parallel comparison is described below.
5. From the patterns that output the desired information type, the product line engineer select the patterns that match the text of the current combination of parts.

In the *analysis phase* the documentation is analyzed with the selected pattern. In detail the product line engineer browses through the text, identifies artifacts as described in the pattern and marks them. In the preparation phase the documentations have been split into tuples of documentation parts. Each tuple is analyzed and the artifacts of different parts that share commonality or variability are identified and marked.

In the preparation phase, for each tuple is has been determined, whether the analysis of commonality or variability is to be done in parallel or sequentially.

With *parallel comparison* all parts of the tuple are compared in parallel. The parts' artifacts that fit the selected patterns are marked. Elements with commonality or variability are marked and connected.
The *sequential analysis* strategy instructs to mark all parts on their own. The analysis of commonalities and variabilities is done afterwards.

John advises not to apply all patterns in one step, but to do multiple cycles, e.g., starting with all patterns outputting features.
The next step is to extract the marked elements for the creation of product line artifacts.

The third phase is the *validation phase*. The domain expert and the product line engineer walk through the extracted artifacts, change them, add and delete elements. John states that missing elements could result from upcoming products or software internal features. Both cannot be found in the documentation.

Chapter 3

Methodology

This chapter defines the methodology used in this thesis. Section 3.1 explains the search for functional web testing tools and the screening process that led to the chosen tools. The algorithm to develop the feature model for the chosen tools is presented in Section 3.2.

The methodology can be applied to other tool domains, with only little adaption to the following search process needed.

3.1 Tool Search and Screening

The selection of tools to be analyzed is done in a similar way as done in systematic mapping studies [24]. The scope of the study is derived from the research question as discussed in section 1.2. All available promising sources are considered leading to a long list of candidate tools. These tools are filtered in the screening step to reduce them to the relevant ones. The filter rules grant that the tools fit the research scope, are actively developed and are in a data format processable by the used methodology.

3.1.1 Tool Search

The initial search aims to find all available tools in the area of functional web testing. The search keywords are *web*, *test OR testing*, *functional OR regression* and *tool OR API OR library OR framework*. The search keywords *functional OR regression* can be

omitted if the information source does not provide these keywords. Some webpages do not sort the tools with these keywords, for example. The tool search uses the following sources:

- Wikipedia page: List of web testing tools[1]
- Website: qatestingtools.com[2]
- Website: softwareqatest.com[3]
- Google (first 100 results)

The page *softwareqatest.com* was used by Di Lucca el al. [12] and Arora et. al. [3].

The sources provide links to web pages. These web pages are investigated for candidates. Retrieved candidates consist of a name and the tools URL and if existing the URL of the end user documentation. The documentation is searched on the page of the tool. If this search is not successful a google query is used consisting of the name of the tool and the keywords *documentation OR manual*. The first ten results are investigated.

3.1.2 Screening rules

This section introduces the screening rules a candidate tool has to comply with to be evaluated in this thesis. The rules guarantee that the tools fulfill the precondition of the fact extraction process, receive approval of the community by being maintained or enhanced, and match the thesis' scope.

- *end user documentation provided*: The tool provides end user documentation in pdf or html format. In case of html, the manual should be readable chronologically. A quick start guide or tutorial is not sufficient.
- *actively developed*: The tool is actively developed. At least one of these sources should have been updated in 2013 or 2014: blog post, news, tool version update, github commit, copyright.
- *matching the scope*: The tool should support functional, system-level web application testing. The scope is defined in section 1.2.

All tools matching the criteria above are relevant tools. Their documentation is downloaded and documentation in html-format is converted to pdf-format.

[1] http://en.wikipedia.org/wiki/List_of_web_testing_tools
[2] http://www.qatestingtools.com/taxonomy/term/59/table
[3] http://www.softwareqatest.com/qatweb1.html#FUNC

1. Headings of sections or subsections typically contain features
2. Headings of sections typically are subdomains of the application domain. The subheadings can then be the features for the domain
3. Words or Phrases that are repeated in different parts of the documentation can be domains or subdomains
4. Features can be found in highlighted phrases (bold or italic font) or in extra paragraphs
5. Technical descriptions or short descriptions of a system often contain lists of features
6. Subdomains/Concepts that are found in a function/feature give a hint for classification of this feature into this Subdomain
7. Domain descriptions can be found in overview sections
8. Domain descriptions can be found in the beginning or in the first subsection of chapters describing a certain domain or features in the domain

FIGURE 3.1: These CaVE patterns [22] are used in this thesis as extraction patterns.

3.2 Feature Extraction

The feature extraction process uses a selection of patterns from the CaVE approach (section 2.2.1). All patterns that extract features and domains are used. The domains are used to classify the features into categories. Figure 3.1 lists the patterns.

The extraction process has two phases: The extraction phase and the combination phase.

During the *extraction phase* each documentation is processed from top to bottom. According to the patterns (Figure 3.1) elements in the documentation are marked. Afterwards a feature model is generated, for each tool, from the marked documentation. Additionally for each feature the domains are listed.

In the *combination phase* all feature models are combined to one complete feature model. The creation of the combined feature model is started with the biggest feature model. Afterwards the other feature models are added incrementally. Figure 3.2 displays the algorithm in pseudocode.

```
1 extractionProcess(list<feature model> featureModels)
2   combinedFeatureModel = max(featureModels)
3   for each(featureModel in featureModels) //excluding max(featureModels)
4     combinedFeatureModel.combineWith(featureModel)
```

FIGURE 3.2: Combination phase algorithm

Combining two feature models (A and B) involves analyzing commonality and variability. For each feature F_B from model B there are three possibilities:

1. **Model A contains a feature F_A that is similar:**
   If the two features have a different name, perhaps feature F_A should be renamed so as to better fit the name of feature F_B.

2. **Model A contains a feature F_A that is an alternative for feature F_B:**
   Add feature F_B to model A as an alternative of feature F_A. Doing so the features can be abstracted by feature groups. The naming and the appearance of the feature groups is guided by the classifications.

3. **Neither 1. nor 2. is true:**
   Add feature F_B to model A.

Chapter 4

Study Execution

4.1 Tool Search and Screening

The tool search resulted in a pool of 212 candidate tools. 18 candidates were identified as duplications of other candidates. Thus the remaining list contains 194 candidates. At first the candidates were checked whether they provide end user documentation and are actively developed. In a second step the candidates were tested upon matching the scope. At the end the documentation was downloaded to prepare the remaining candidates for analysis. If there was no pdf version of the documentation, the documentation was downloaded via the "Print - In File - PDF-Format" command of Mozilla Firefox. The pdf parts were concatenated using a command line tool. A few tools had to be excluded because they could not be downloaded, others because their documentation turned out to be to technical. Table 4.1 presents the excluded candidates in numbers ordered by the applied rule[1].

Table 4.2 measures the quality of sources used for screening by means of the remaining and analyzed tools. Each source except wikipedia contributed tools that no other source contained and that are included in the analyzed tools.

23 tools remained and were analyzed (Table 4.3). 17 can be further classified as Integrated testing environments (ITE) and 6 as Framework/Library (See section 1.2). The analyzed tools from the category "Framework/Library" will be further called APIs for

[1]Keeping in mind that often multiple rules apply but only the first one is imposed

Exclusion Rule	Quantity
duplication	18
not actively developed	47
no or wiki-based documentation	61
not matching the scope	52
only technical documentation	8
download not possible	3

TABLE 4.1: Count of candidates that have been eliminated sorted by reason.

Source	Tools	Recall
Wikipedia page: List of web testing tools[2]	7	30%
Website: qatestingtools.com[3]	17	74%
Website: softwareqatest.com[4]	17	74%
Google (first 100 results)	17	74%

TABLE 4.2: The screening sources and the number of tools they provided. Duplications are included.

abbreviation. The ITE tools' documentation were downloaded between the 12th and the 14th of November and the API tools on the 6th of December, both in 2014.

4.2 Feature Model Generation

The process of feature extraction was done separately for the ITEs and the APIs.

4.2.1 Feature Extraction

Each documentation was scanned through two times. At the first time promising keywords and explaining text passages were marked as seen in Figure 4.1 using the extraction patterns. The tool used was Foxit Reader 6[5].

After each marking cycle the documentation was read again and the features were extracted. Thus a feature model instance for each documentation was created. The features were ordered hierarchically and accompanied by explaining text passages and page numbers for easy lookup. Preliminary categories were formed while integrating feature after feature into the existing feature model instances.

[5]http://www.foxitsoftware.com/Secure_PDF_Reader/index.php

Integrated testing environment	documentation format
QF-Test[25]	pdf [26]
QA Wizard Pro[27]	pdf [28]
AppPerfect Web Test[29]	html [30]
iMacros[31]	html [32]
Sahi Pro[33]	html [34]
Robot Framework[35]	html [36]
Application Testing Suite[37]	pdf [38]
WinTask[39]	pdf [40]
TestingWhiz[41]	pdf [42]
CodedUI[43]	html [44]
Rapise[45]	pdf [46]
Testing Anywhere[47]	pdf [48]
Ranorex Test Automation[49]	pdf [50]
Silk Test[51]	pdf [52, 53]
Test Studio[54]	html [55]
RIATest[56]	html [57]
Jubula[58]	pdf [59]
Framework/Library	
Selenium[60]	html [61]
FuncUnit[62]	html [63]
Codeception[64]	html [65]
GEB[66]	html [67]
FluentLenium[68]	html [68]
Arquillian Graphene[69]	html [70]

TABLE 4.3: Tools in this study.

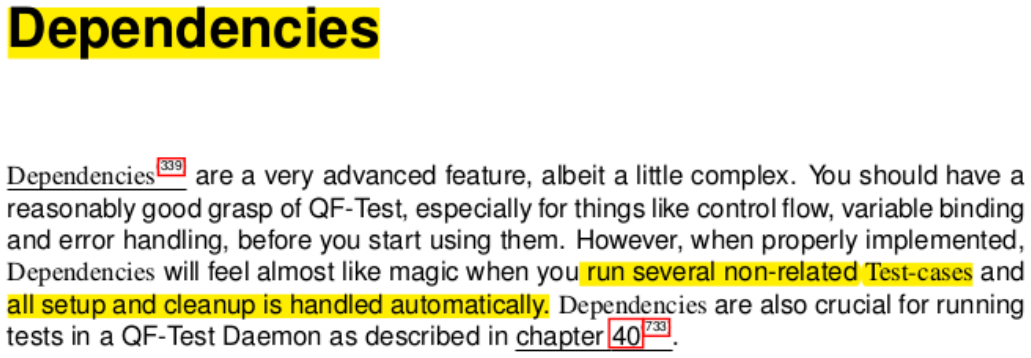


FIGURE 4.1: Example for the marking of features and explaining text passages.

4.2.2 Combination Step

In the combination step the feature model instances with their features ordered by categories were taken and combined to a joint feature model. On the one hand the categories were needed to ensure the feature model is comprehensible and clearly arranged. On the other hand the categories are necessary to ensure the feature model fits on the pages of this thesis.

There had been seven preliminary categories and at the end of the combination step there were ten[6]. As described in the methodology section (3.2) the combination step involves looking at each feature from the currently added feature model instance to identify, if it has been introduced to the feature model yet. And in case it has been introduced, if the names are the same or not. This 'looking at each feature' has been documented to log files for each category. Just as well as the feature model instance files, these files have been designed to be machine processable. Table 4.4 displays the log file notation and table 4.5 lists the statistics of the combination step for the tools of the ITE category.

Keyword	Explanation
m:	feature is missing in the main model and is added; parent *p:* (if not specified, the root node is the parent)
e:	feature is existing in the main model
mo:	feature is moved to another category; moved *to:*
add:	add new feature, discovered during clarifying investigation
ar:	added feature renamed optional *to:*
mr:	main feature renamed optional *to:* , *old:*
ab:	added abstraction feature parent *p:*
mh:	moved in hierarchy new parent *p:*
rref:	Renaming because of refactoring *to:*
xor:	These features exclude each other(XOR group) {*element, elements*}
rerr:	renaming to correct error *to:*
d:	Deleted
in:	Added through induction

TABLE 4.4: Notation of the log files used to document the combination step.

[6]The category "Technology" is printed as two diagrams. The "Technology" diagram with the "HTML" feature collapsed and the sub-diagram "HTML" alone.

Figure 4.2 shows an five lines long excerpt of a log file. The first line adds an abstract feature (*ab:*), the second adds a feature that is not yet in the feature model (*m:*). It is renamed (*ar:*) to *Cookie* and added in the hierarchy as child of the feature *Cleanup between Tests* (*p:*). Line 3 contains a comment (indicated with >>) to explain the feature in line 2. Line 4 moves the feature *Reset Database* to the abstract feature (*mh:*) and line 5 renames the feature (*rref:*).

```
ab: Cleanup between Tests
m:  -Cookie management (92) ar: Cookie p: Cleanup between Tests
 >>clear the browser's cookies at the end of each test method.
mh: Reset Database p:Cleanup between Tests
rref: Reset Database to: Database
```

FIGURE 4.2: Excerpt of a log file belonging to the combination step.

While the activities of the combination step were documented in log files, the feature models[7] were drawn using the Eclipse Modeling Framework[8] which is an eclipse plug-in. The feature models were printed to better spot errors and duplicated features.

Next to each feature the tools which possessed that feature were listed[9]. This was done for two reasons. The first reason is that this was needed to generate the diagrams listed in the next section. The second reason is to enable a technique called "feature induction" in this thesis. Feature induction is used to remove inhomogeneity. The idea is that, if a tool has feature A and feature B is the parent feature of feature A, then the tool must have feature B too. This is a fundamental property of feature models. For example the parent of the feature *Image Click* is the feature *Image Recognition*. A few tools had first feature and the parent feature has been added by induction. Feature induction has also been logged (See table 4.4). When it seemed that more tools should have a specific feature[10] the documentation was investigated again to add the missing feature.

Among the features are abstract features that are only needed to support the hierarchy and to build categories. Abstract features have no impact on the possible variants of a feature model [71]. In this thesis abstract features are excluded from feature induction and statistics.

[7] Its one big feature model, however the feature model is split into categories, which will hence be called the "feature models".

[8] https://projects.eclipse.org/projects/modeling.emft.featuremodel

[9] The tools were referenced by numbers.

[10] E.g. there is a mandatory XOR feature and a tool does not have any of them.

Tool	Features	Unique	Common	Missing	Existing	Refactoring	Renaming	Induction
QF-Test	64	28	36	63	1	4	26	0
QA Wizard Pro	45	9	36	30	15	5	21	0
AppPerfect Web Test	30	13	17	19	11	0	8	2
iMacros	36	20	16	25	11	11	22	1
Sahi Pro	31	10	21	13	18	0	10	0
Robot Framework	31	11	20	14	17	8	19	0
Application Testing Suite	28	5	1	5	1	2	9	0
WinTask	13	0	13	1	12	0	9	0
TestingWhiz	20	3	17	5	15	1	13	0
CodedUI	14	4	10	7	7	1	4	0
Rapise	21	2	19	3	18	0	13	2
Testing Anywhere	33	9	24	13	20	6	17	1
Ranorex Test Automation	30	6	24	8	22	3	10	1
Silk Test	17	2	15	2	15	7	8	1
Test Studio	43	16	27	18	25	2	23	0
RIATest	27	4	23	4	23	6	6	1
Jubula	28	9	19	9	19	9	17	0

TABLE 4.5: Statistics of the combination step (ITEs). Derived from the log files.

Chapter 5

Result

This chapter presents the results of this thesis, a long list of features in their context of the feature model. The feature model of the ITEs is sub-divided into ten categories, which form the sections of this chapter. The features are again ordered by abstract or high-level features. The feature model of the APIs is integrated into the structure.

The key to the feature models in this chapter is pictured in figure 5.1. The notation is changed compared to the traditional feature models in so far that the symbols traditionally representing mandatory and optional features are changed. Figure 5.2 displays how many features occur n times. There are 143 unique features, 32 features that occur in two tools, 15 features that occur in 3 tools and so forth. The features that are supported by seven or more tools are marked in the feature models with the filled circle. The feature descriptions also distinguish the features by means of the number of tools that support them. There is a three level scale, consisting of unique (1), several times (2-6) and frequent (7 or more times).

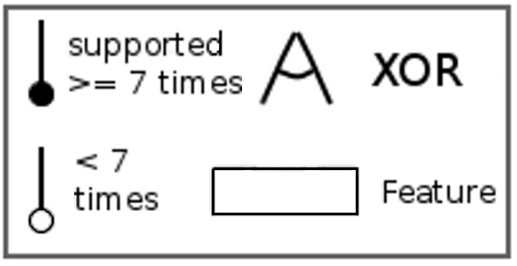


FIGURE 5.1: Featurediagram - Key

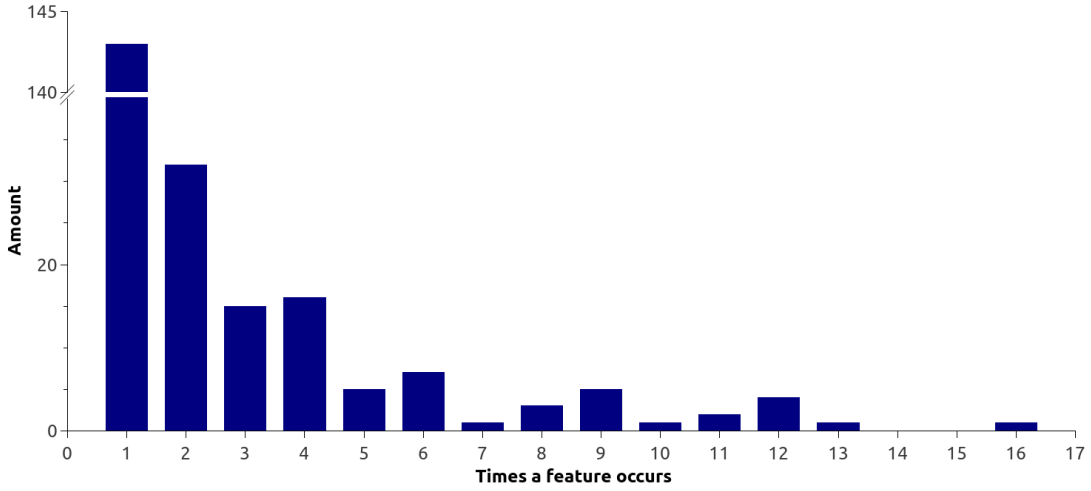


FIGURE 5.2: Distribution of the frequencies of the ITE features.

5.1 Abstraction Supporting Features

The category *Abstraction Supporting Features* contains features that are needed to prevent code duplication following the principle "Don't repeat yourself" [72]. The features *Component Model* and *Page Object* additionally cause abstraction by encapsulating the elements of web pages or parts of web pages.

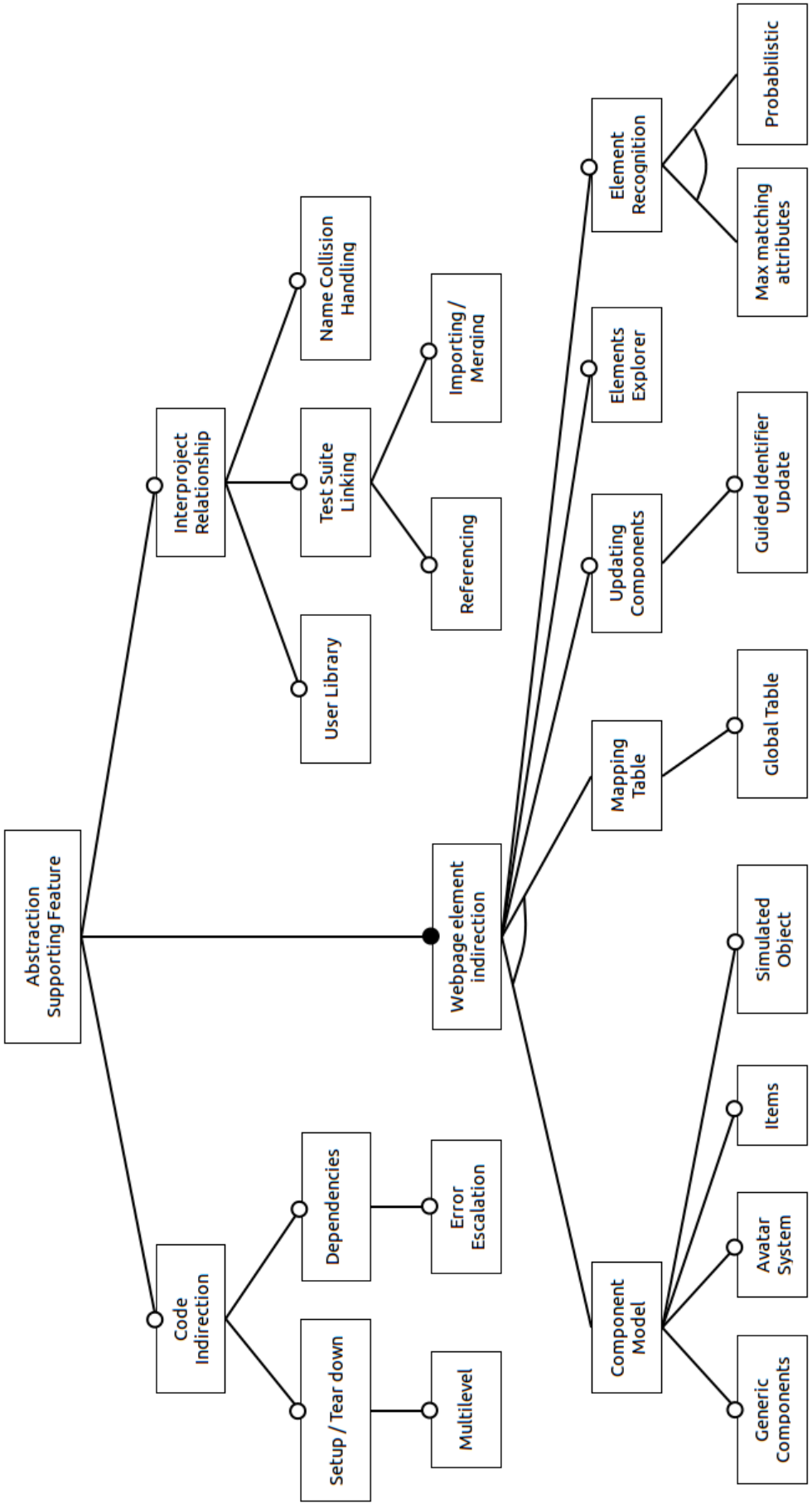


FIGURE 5.3: Feature model (ITE): Abstraction Supporting Features

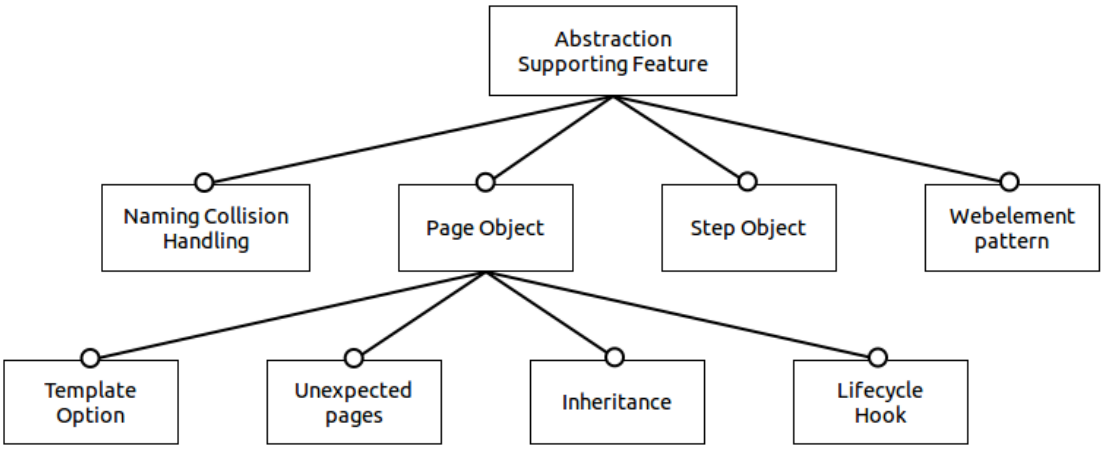


FIGURE 5.4: Feature model (API): Abstraction Supporting Features

5.1.1 Webpage Element Indirection

The general idea is to decouple the HTML structure of the SUT from the test. This can be done by using a *Mapping Table* or a *Component Model*. In the APIs the patterns *Page Object* and *Step Object* are used for this purpose.

5.1.1.1 Mapping Table

ITE SPECIFIC	
Map: key - element identifier	
frequency	several times
also known as	application repository, element repository, object map
specialization	*Global Table*

Encapsulates element identifiers in a map and addresses them in the test via key. Most tools fill the *Mapping Table* automatically when recording tests. Otherwise the user has to fill the table manually before recording. If the *Mapping Table* supports this, it is recommended to nest the identifiers to reduce maintenance even more.

A special version is the *Global Table* feature that ensures that the *Mapping Table* can be shared between multiple users or projects.

5.1.1.2 Updating Components

ITE SPECIFIC	
Semi-automatic element identifier and internal data update to conform alteration	
frequency	several times
specialization	*Guided Identifier Update*

The *Updating Components* feature is used to help with maintenance and error fixing. The changes to the GUI-elements made through development, evolution and maintenance can be adopted to prevent or repair broken identifiers.

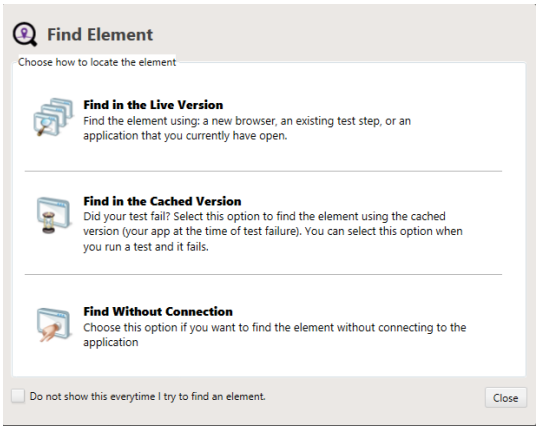


FIGURE 5.5: Guided Identifier Update - locate element [55]

The following work flow is used: At first the tool tries to identify the changed element using the recorded attributes. If that is not possible the user gets involved. Either the user has to adapt the identifiers manually or if the tool supports *Guided Identifier Update* the user identifies the element by clicking on it in the SUT and the ITE changes the identifier (Figure 5.5).

At the end the tool updates the internal representation (attributes) of the element (Figure 5.6). Problems arise if the elements get misidentified, thus its recommended to backup before updating multiple components.

5.1.1.3 Element Explorer

ITE SPECIFIC	
DOM Explorer like overview of the used elements	
frequency	several times

When using a lot of elements via *Mapping Table* or *Component Model* it is easy to lose the overview. The feature *Elements Explorer* gives an overview over all used elements in their context of the DOM by building a sub-tree with the used elements (Figure 5.7).

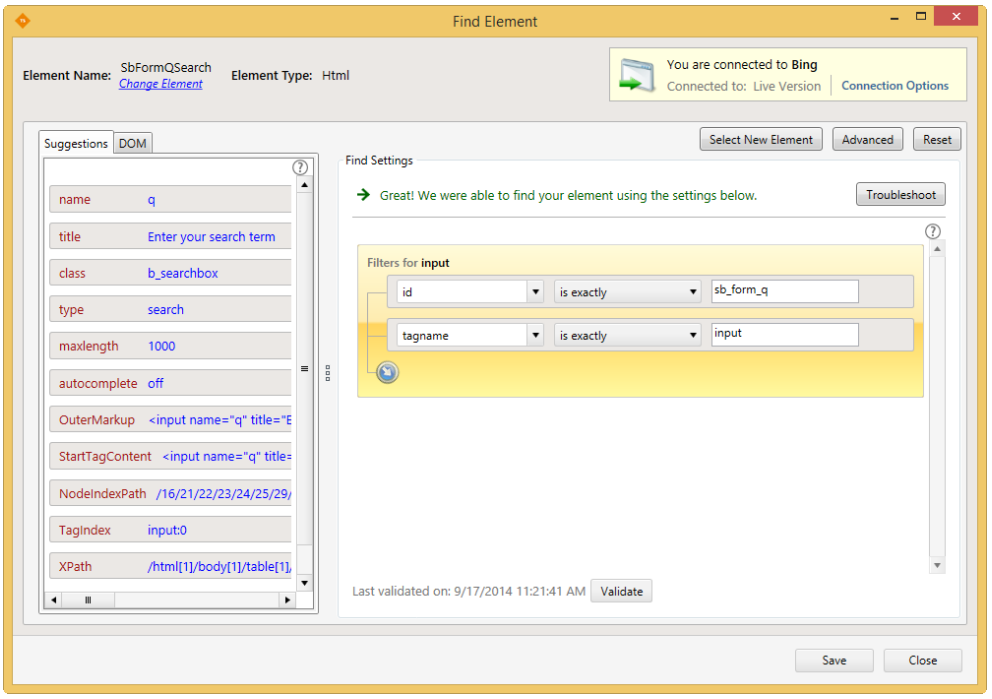


FIGURE 5.6: Guided Identifier Update - element found [55]

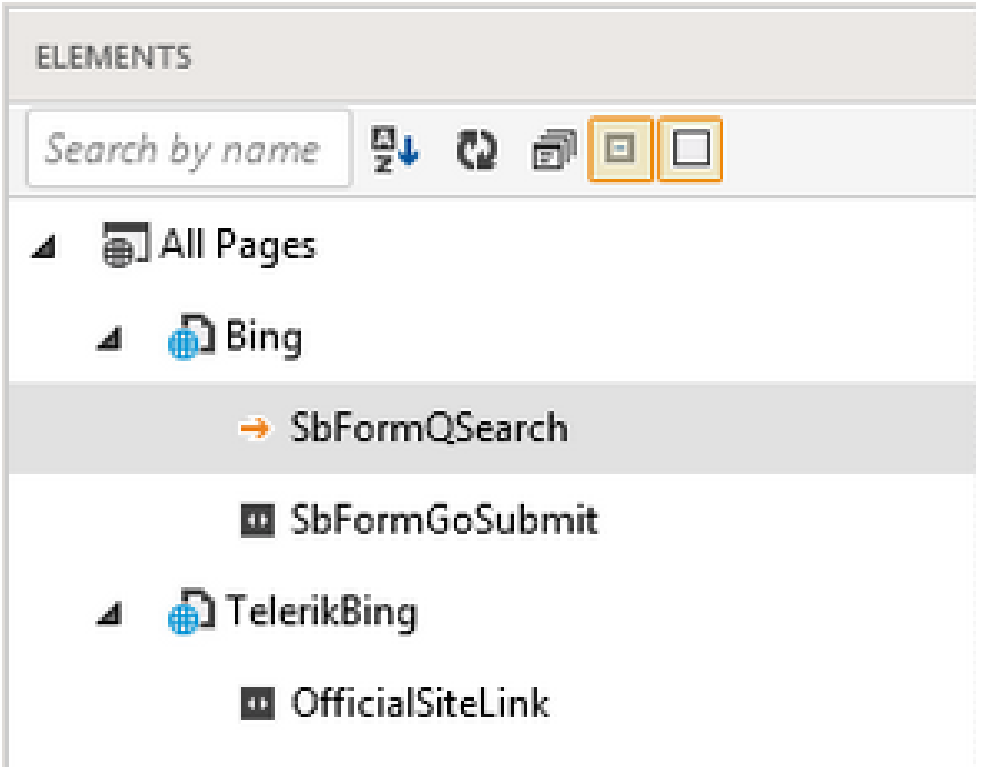


FIGURE 5.7: Elements Explorer [55]

Often the *Elements Explorer* is well integrated into the editor and thus navigation from test to *Elements Explorer* and vice versa is possible.

5.1.1.4 Element Recognition

ITE SPECIFIC	
Recognize elements after application change	
frequency	several times
specialization	*Max matching attributes, Probabilistic*

In the ideal case elements are identified by id attribute. If that is not possible, several tools generate custom identifiers using *Attribute*, *DOM Hierarchy* or *Index* (see Section 5.4). In order to do that they use element recognition algorithms that even enable to recognize elements after changing parts of the identifying information.

The *Max matching attributes* approach uses all available attribute values and selects the elements with the maximum number of elements that have not changed. Similar is the *Probabilistic* approach which is able to include information like the *DOM Hierarchy* or the attributes of the adjacent elements. Each information that has not changed is additionally weighted. In the end the element that has the highest probabilistic value is selected in case it is higher than the applied threshold.

5.1.2 Component Model

ITE SPECIFIC	
Element identifiers grouped by web pages	
frequency	several times
similar to	*Page Object*
specialization	*Avatar System*

The *Component Model* is the extended version of the *Mapping Table*. In addition to the indirection of element identifiers the *Component Model* encapsulates the identifiers using the web page[1] they belong. Often additional information is saved to support for instance *Element Recognition*.

[1] Identified by the URL.

- *Generic Components*: Make use of variables to reuse components over different pages. (Similar to *Step Object*)
- *Items*: Index the sub-items of tables, lists and trees by identifying the root only.
- *Simulated Object*: Identify an element by position or image and use it as a virtual component.

5.1.2.1 Avatar System

ITE SPECIFIC	
Offline testing and test creation.	
frequency	once

The *Avatar system* is a special *Component Model* that saves all component properties on disk. In addition to the indirection of element identification the *Avatar system* enables to execute tests against the saved values of the application. Similarly new tests can be created by recording against the avatar using a running SUT.

5.1.3 Interproject Relationship

When testing huge projects and different testers are involved it is necessary to separate the tests over multiple projects and to provide one project that contains all element identifiers that the other projects need. Another way to to archive code reuse between projects is to develop custom libraries.

These features are only relevant to *Self-made* languages as general purpose languages (GPL) support them out of the box (See section 5.7).

5.1.3.1 Test Suite Linking

ITE SPECIFIC	
Using functionality from another project.	
frequency	several times
specialization	*Referencing, Importing / Merging*

Test Suite Linking enables to using scripts or components across project boundaries.

Referencing lets the user call scripts. Whereas *Importing / Merging* imports the script into the environment of the project and thus enables to merging component model for instance.

5.1.3.2 User Library

ITE SPECIFIC	
Custom libraries for procedures and variables.	
frequency	once
also known as	Resource file

User Libraries enable to encapsulate procedures, depending on the language style also known as user keywords, to be shared between projects via libraries.

5.1.3.3 Name Collision Handling

Solve problems with identical procedure names.	
frequency	once

In case two keywords or procedures have the same name, *Name Collision Handling* determines which version has the highest priority e.g. based on its scope and uses it.

5.1.4 Code Indirection

5.1.4.1 Setup / Tear down

ITE SPECIFIC	
Encapsulates the steps that prepare the SUT for a test case.	
frequency	several times
specialization	*Multilevel*

A test setup is the code that is executed before a test case, and the test tear down is executed after a test case. If the setup fails the tear down is executed and execution continues with the next test case.

Multilevel setup and tear down clusters the test cases and their setup and tear down procedures in hierarchies. To execute a test case, all setups beginning from the bottom are executed, then the test case, and afterwards all tear downs in the reversed order. If test cases are side effect free, it is also possible to execute all adjacent test cases with equal setup without executing tear down and setup in between.

Setup / Tear down is a common testing paradigm and thus often seen when testing with GPLs.

5.1.4.2 Dependencies

ITE SPECIFIC	
Specify dependencies to generate execution order including setup and tear down.	
frequency	once
specialization	*Error escalation*

Each test is annotated with other tests and setup, tear down scripts it depends on. The tool generates an execution order using this information. Additionally the tool handles unexpected behavior, e.g. closing an error dialog, which pops up and blocks your test.

Also *Error escalation* between dependent tests is supported. If for example a setup scripts fails, all dependent tests are skipped.

5.1.5 API specific

5.1.5.1 Page Object

API SPECIFIC	
Grouping of element identifiers and commands belonging to a page	
frequency	several times
similar to	*Component Model*
specialization	*Template Option*

The page object pattern was originally introduced by Selenium [73]. It recommends to group element identifiers and the procedures working with them by the pages they work on. Using Objects the identifiers become properties and the procedures methods respectively. Since the pattern is API independent, the feature *Page Object* describes additional support of the pattern with the use of a DSL for instance.

```
1 import geb.*
2
3 class SomePage extends Page {
4   void onLoad(Page previousPage) {
5     // do some stuff with the previous page
6   }
7 }
```

FIGURE 5.8: Lifecycle Hook [67]

- *Unexpected Pages*: A page object for pages like the 404 error page.
- *Inheritance*: Pages may inherit properties and methods.
- *Lifecycle Hook*: Transfer data between the current and the next page object (Figure 5.8).

Template Option

API SPECIFIC	
Template properties with special semantics	
frequency	several times

The *Template Option* feature enriches the *Page Object* and its properties with extra semantics (Figure 5.9). Some attributes are:

- at: A way to check whether the underlying browser is at the expected page.
- required: The required option controls whether or not the content returned by the definition has to exist or not. When required is set to true, the page is entered and the element does not exist, the application is terminated with an error.
- cache: The cache option controls whether or not the definition is evaluated each time the content is requested.
- to: The to option allows the definition of the page the browser will be sent to if the content is clicked.
- wait: If wait is set to true and an element does not exist but is needed by the application, the program waits for the element to appear.

```
1  import geb.*
2
3  class GoogleHomePage extends Page {
4     static url = "http://google.com/?complete=0"
5     static at = { title == "Google" }
6     static content = {
7        searchField { $("input[name=q]") }
8        searchButton(to: GoogleResultsPage) { $("input[value='Google Search']") }
9     }
10
11    void search(String searchTerm) {
12       searchField.value searchTerm
13       searchButton.click()
14    }
15 }
16
17 class GoogleResultsPage extends Page {
18    static at = { waitFor { title.endsWith("Google Search") } }
19    static content = {
20       results(wait: true) { $("li.g") }
21       result { index -> results[index] }
22       resultLink { index -> result(index).find("a.l") }
23    }
24 }
```

FIGURE 5.9: Template Option - example using the options url, at, to and wait [67].

5.1.5.2 Step Object

API SPECIFIC	
Reusing identifiers and commands over multiple pages	
frequency	several times
also known as	Page Fragment

```
public class PageFragmentExample {

    @Root
    private WebElement optionalRoot;

    @Drone
    private WebDriver browser;

    @FindBy(css="relativeLocatorOfThisPageFragmentPart")
    private WebElement otherPageFragmentPart;

    @FindBy(xpath="relativeLocatorOfThisPageFragmentPart")
    private WebElement alsoPageFragmentPart;

    public void firstServiceEncapsulated() {
        otherPageFragmentPart.click();
    }

    public void secondServciceEncapsulated() {
        alsoPageFragmentPart.clear();
    }

    public void thirdServiceWhichNeedsDirectAccessToRoot() {
        root.click();
    }
```

FIGURE 5.10: Step Object - example [70].

Step Objects are used for modeling GUI components that are used across multiple pages. To reuse them they are combined with *Page Objects*. A *Page Object* can integrate multiple *Step Objects* and that way integrate their properties and methods.

5.1.5.3 Webelement pattern

API SPECIFIC	
Extend existing webelements	
frequency	once

An API that supports the feature *Webelement pattern* enables to integrate the web element classes into the object hierarchy. This makes it possible to add functionality directly to the existing elements (Figure 5.11).

```
public abstract class MyInputFragment implements WebElement {

    @Root
    private WebElement input;

    public String getInputText() {
        return input.getAttribute("value");
    }
```

FIGURE 5.11: Webelement Pattern - add custom method[70].

5.2 Capture

The section capture includes features that support the creation of new test cases. Most important is the *Recording* feature, but a few other features used while creating tests are included too.

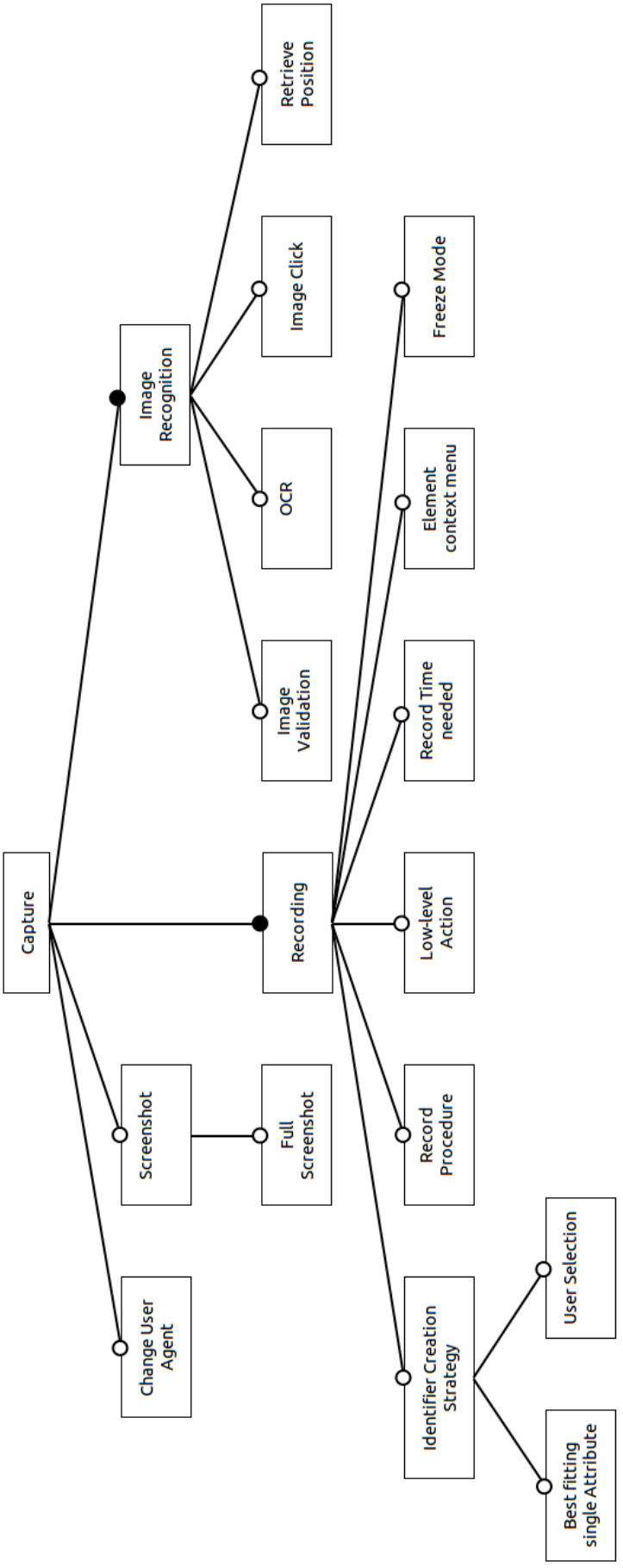


FIGURE 5.12: Feature model : Capture

5.2.1 Image Recognition

ITE SPECIFIC	
Automation and validation of pictures	
frequency	frequent

Image Recognition is the technique of automating web pages by analyzing pictures. There are tools, out of this thesis' scope, that work entirely with *Image Recognition* (cf. Section 1.1). The analyzed tools use *Image Recognition* to verify, that the page layout is correct (*Image Validation*). The complete web page is converted into an image and compared at pixel level.

Another scope of application is the automation of *Widget Toolkits* that are not supported by hook-up. *Image Recognition* is able to automate almost everything. However it is prone to change in the GUI, because the GUI is the only information source for identification. The features needed are *Image Click*, which searches and clicks an image. The alternative is to use *Retrieve Position*. If the tool supports recording of *Image Clicks* it tries to identify the area of the GUI element below the mouse and highlights it.

OCR[2] enables to capture text embedded in an image. This is useful if GUI elements use images with text. If in addition the SUT is internationalized, a test using *Image Clicks* would have to be recorded again for every language. Using *OCR* is an alternative.

5.2.2 Recording

ITE SPECIFIC	
Record the tester using the web page	
also known as	Capture & Replay
frequency	frequent

Recording, also known as Capture and Replay, is the fundamental feature of ITEs (Figure 5.13). It advertises with the dream that test cases do not have to be written by hand,

[2]Optical character recognition

but are automatically generated by recording the tester using the SUT. This technique has different problems, which the advocates of API-based testing mention.

Without any abstraction supporting features (cf. Section 5.1) recorded test cases suffer from a lot of code duplication. Element identifiers, setup, tear down and common functionalities are recorded again for every test case and thus the maintenance effort is very high. The feature *Record procedure* lets the user combine the encapsulation of common functionalities and the time saving of *Recording*.

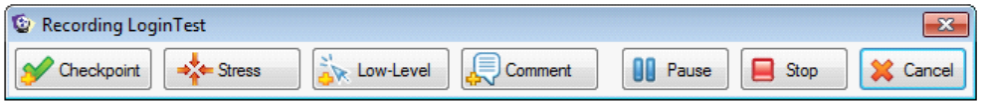


FIGURE 5.13: Recording - toolbar [28].

Almost all tools support *Recording*, but there are a lot differences in the details. One differentiating factor is whether all recorded test cases can be replayed without user adaption. Possible problems are recorded noise (e.g. mouse movement), insufficient waiting for elements to appear or a bad *Identifier Creation Strategy*. Two strategies have been observed, *Best fitting single Attribute* and *User Selection*. The latter is semi-automatic. It captures all attribute values and instructs the test developer to create an identifier. There are more advanced identifier creation strategies (some described in 5.4).

- *Record Time needed*: Embed wait statements so that the replay needs the same time as the recording.
- *Element context menu*: Often the *Recording* feature is accompanied by a context menu opened typically by right clicking. By this means verifications are attached while recording, elements are added to the *Mapping Table* or *Component Model* and elements are located in the DOM.
- *Freeze Mode*: Freeze the GUI of the SUT to stop animations and visual effects that tend to alter their behavior on mouse hover.
- *Low-level Action*: Special recording mode to record “onmouseover” events for example.

5.2.3 Change User Agent

Fake information about used operation system and browser	
frequency	once

When opening a page the web browser sends information about the operation system, the browser and the version it uses. These information are needed because code running on google chrome often does not run on the microsoft internet explorer and vice versa. *Change User Agent* enables to fake these information.

5.2.4 Screenshot

Take a screenshot of the SUT while testing.	
frequency	several times
specialization	*Full Screenshot*

Taking a *Screenshot* is feature provided in automation, but also automatically on error. Some APIs even support it in the debugger. Some tools are able to take a *Full Screenshot*, which means the whole page including the parts that are not visible on the screen.

5.3 Editor Features

The majority of ITE tools provides their own test editor. The APIs often profit from the powerful IDEs their host languages use. This section analyzes, which features the editors of the ITEs possess to compete.

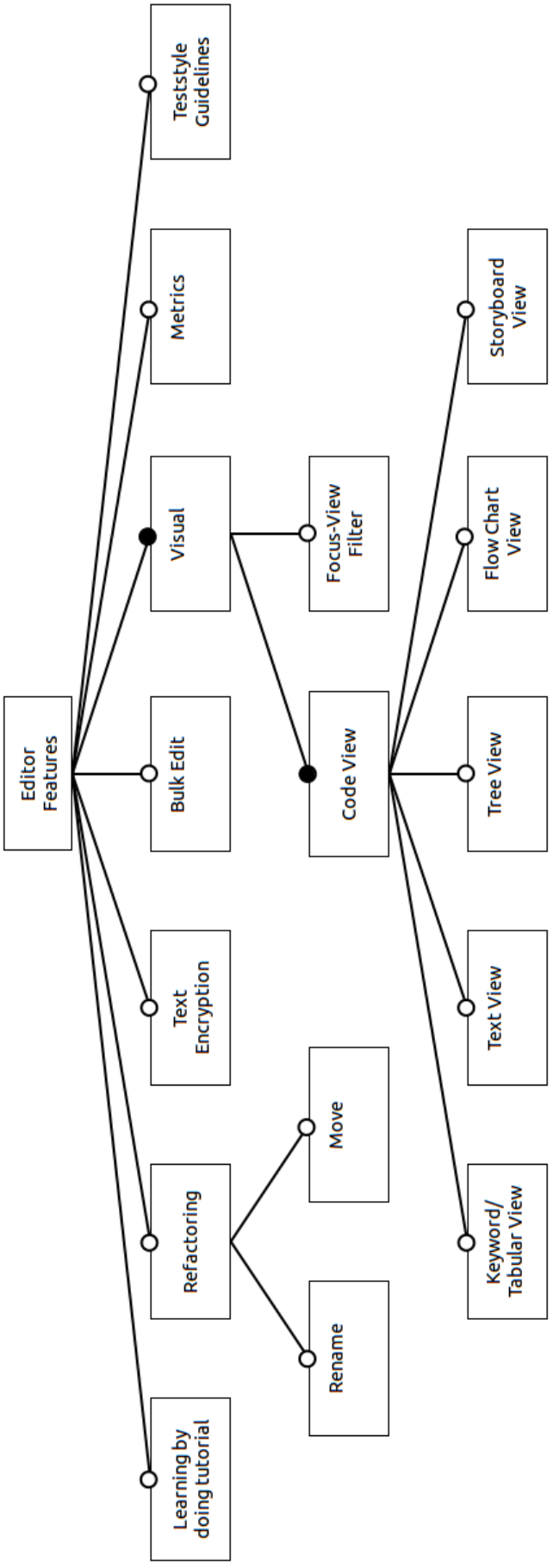


FIGURE 5.14: Feature model : Editor Features

5.3.1 Code View

The abstract feature *Code View* summarizes the different styles of visualizing the testing code to the tester.

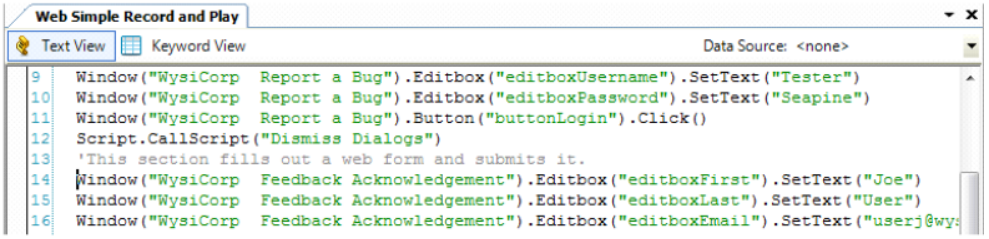


FIGURE 5.15: Text View [28]

5.3.1.1 Text View

ITE SPECIFIC	
Tests as textual code	
frequency	several times
also known as	Code View

The *Text View* is the most ordinary code view. The code is presented with a text editor. Figure 5.15 gives an example. Figure 5.16 presents the same code in *Keyword / Tabular View*.

5.3.1.2 Keyword/Tabular View

ITE SPECIFIC	
Code in a table.	
frequency	several times
also known as	Table View

The *Keyword / Tabular View* is a visualization style, that puts the code in a table (Figure 5.16). In the vertical dimension are code steps. The horizontal dimension is usually starting with the keyword of the code step, following with the remaining properties.

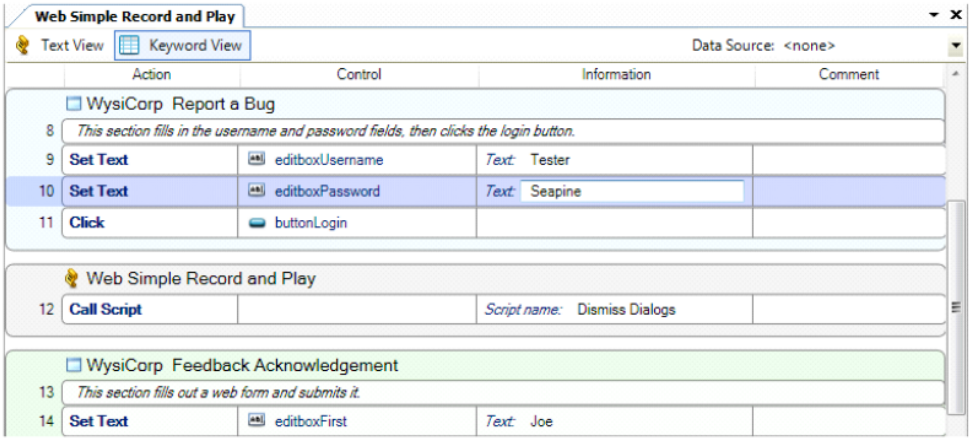


FIGURE 5.16: Keyword/Tabular View [28]

5.3.1.3 Tree View

ITE SPECIFIC
Hierarchical code view
frequency several times

The *Tree View* is presenting the test code in a hierarchical style (Figure 5.17).

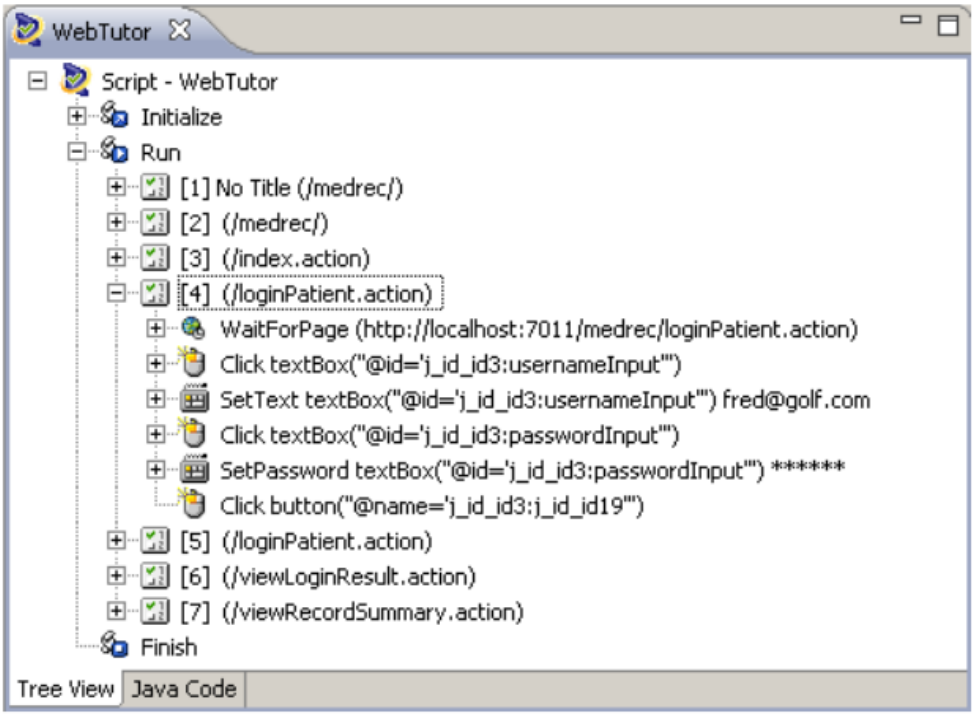


FIGURE 5.17: Tree View [38]

5.3.1.4 Storyboard View

ITE SPECIFIC
Screenshots accompanying to the test steps
frequency once

The *Storyboard View* is a screenshot documentation of the SUT that is created automatically during recording.

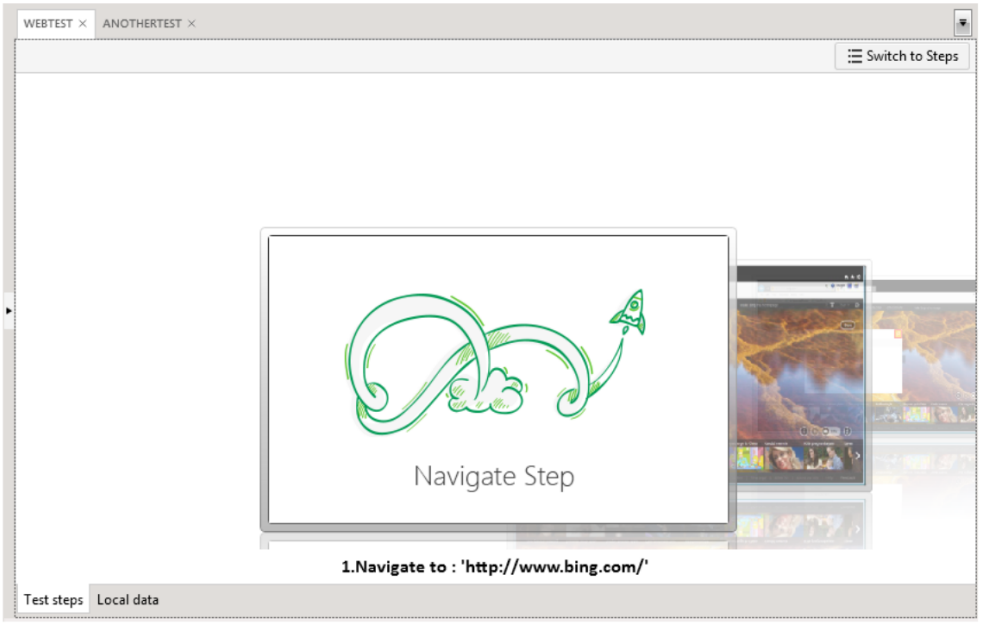


FIGURE 5.18: Storyboard View [55]

5.3.1.5 Flow Chart View

ITE SPECIFIC
Code steps as a flow chart
frequency once

The *Flow Chart View* visualizes the code as a flow chart.

5.3.2 Miscellaneous

Several other features have been observed. All of them appeared unique.

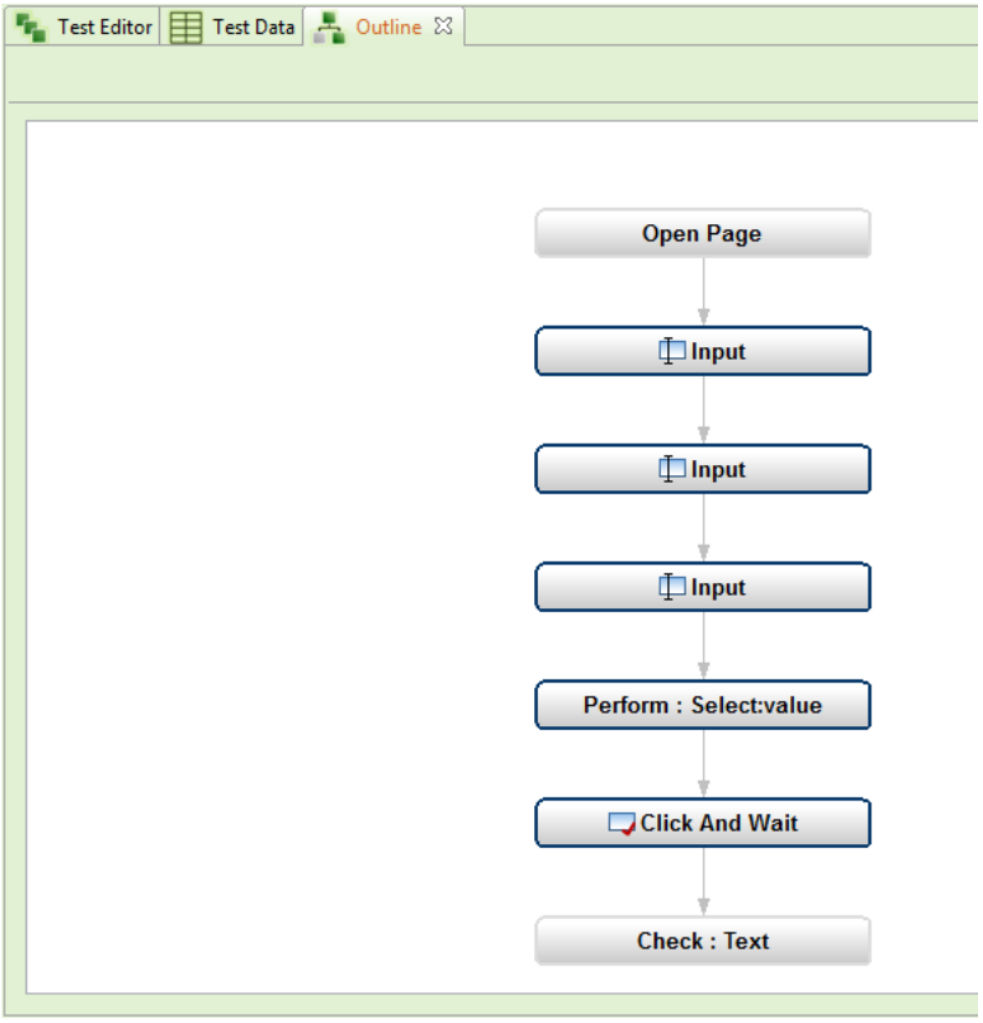


FIGURE 5.19: Flow Chart View [42]

- *Learning by doing tutorial*: An advanced version of the tutorial, that is integrated into the editor and monitors and guides the progress the user does in the tutorial.
- *Refactoring*: Refactoring support is one of the key features that makes IDEs like Eclipse profitable. In the ITEs it is only supported in one tool with the patterns *Rename* and *Move*.
- *Text Encryption*: Prevents sensible information, like passwords, on the screen by masking them.
- *Bulk Edit*: Edit multiple positions in code at one.
- *Teststyle Guidelines*: Performs static analysis of the test cases to check whether they conform to conventions and best practices.

- *Metrics*: Collect information about the test cases by computing metrics.
- *Focus-View Filter*: Include or exclude commands types from the view (e.g. checkpoints, keystrokes, mouseclicks, delays)

5.4 Element Identification

Element identification is the process of describing how to access web elements on the page through testing code. Its comparable to describing a tourist how to reach the town center. The description can be a Path, like Left, two times straight ahead and then left. It can also be task oriented ("If you spot the church. Go in the direction of the church.").

This section is divided into two parts. The *Information* part contains all sources of information that can be used to develop identifiers. The second part (*Method*) collects techniques for using the information to develop or generate an identifier.

This section combines the features of ITEs and APIs and mentions to which domain the examined feature belongs.

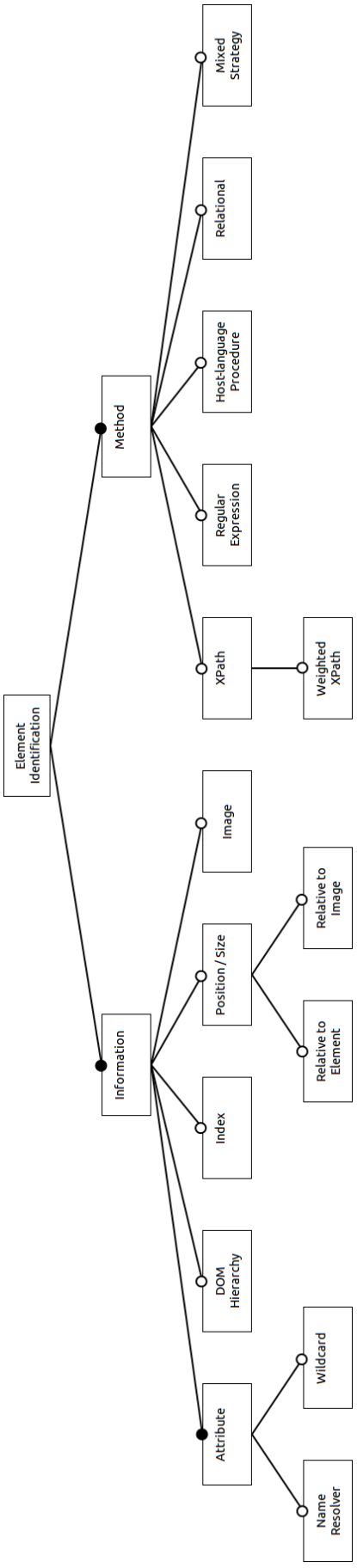


FIGURE 5.20: Feature model (ITE): Element Identification

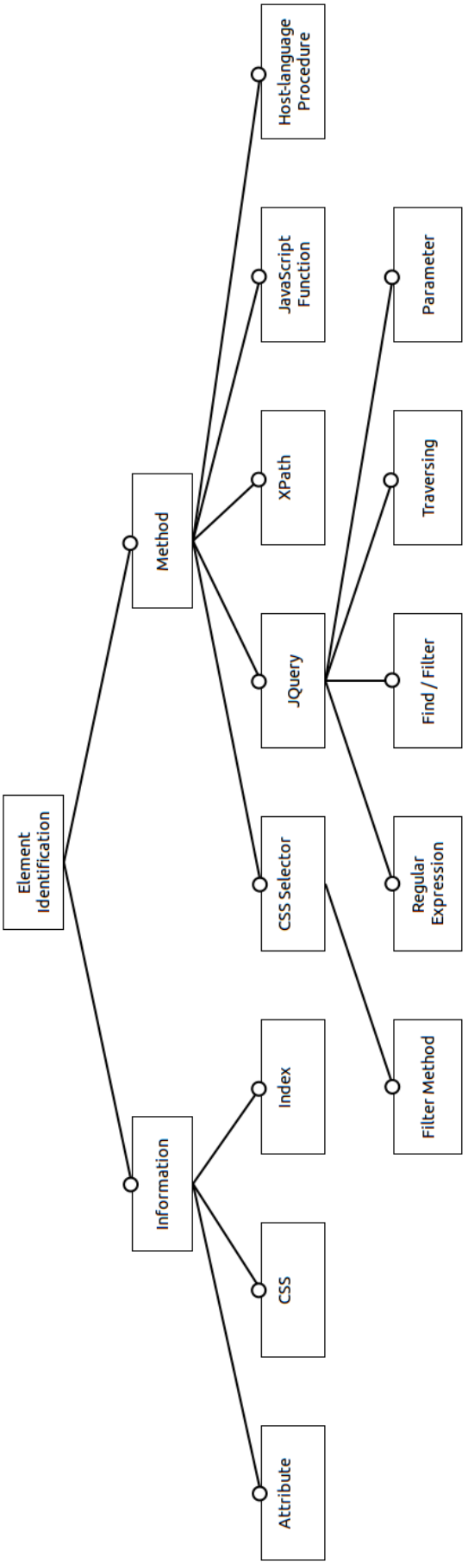


FIGURE 5.21: Feature model (API): Element Identification

5.4.1 Information

The traditional information sources originate from the hook-up of the testing tools into the DOM of the browser. Additional information is gained by *Image Recognition* (Section 5.2.1).

- *Image* (ITE): Position of an Image
- *Index* (ITE, API): The path of objects from the DOM root to the element.
- *DOM Hierarchy* (ITE): The position of the element in the DOM in relation to the surrounding elements.
- *CSS* (API): The CSS properties.

5.4.1.1 Attribute

Properties of web elements like id, name and value	
frequency	several times

Attributes are the most used information sources for identifiers. Using the "id" or another attribute to create unique identifiers is best practice. However this is often impossible, if the tester has no connection to the developer.

The feature *NameResolver* can be used to change or remove attribute values virtually in the test that create problems when writing element identifiers. It is a procedure applied to every attribute.

Another problem with "id" identifiers is that many framework and widget toolkits add random numbers to the ids. Adding *Wildcards* enables to solve this problem.

5.4.1.2 Position / Size

x,y, width and height	
frequency	several times
specialization	*Relative to Element, Relative to Image*

The information about *Position / Size* are technically *Attributes*. Since they are often used as replacement for access to the DOM, they are listed separately. Automating the SUT only with the information of the position[3] is very error prone, as resizing of the browser or moving an element breaks the test case.

Specializations of this feature are relative positions: *Relative to Element* and *Relative to Image*

5.4.2 Method

There are different methods to combine the available information to an element identifier. The ITEs commonly use *XPath* or *Regular Expressions*. The APIs use additionally *JQuery* and *CSS Locators* and sometimes combine techniques like a *JQuery* identifier that works on the results of a *CSS Locator*. Figure 5.22 shows an example with different identifier methods.

```
<?php
$I->click('Log in');
// CSS selector applied
$I->click('#login a');
// XPath
$I->click('//a[@id=login]');
// Using context as second argument
$I->click('Login', '.nav');
?>
```

FIGURE 5.22: Different element identifiers [65]

- *XPath* (ITE, API): often used method that employs *Attribute*, *Index* and partial *DOM Hierarchy* to describe a path to the element. The specialization *Weighted XPath* extends the concept with weight rules.

[3] Click at x=300, y=250

- *Regular Expressions* (ITE, API): ITEs often introduce regular expressions to make the usage of *Attributes* more flexible. On the API side they are provided by the host language.

- *Host-language procedure* (ITE, API): A procedure in the used language, self-made or GPL, that for example gets the DOM-tree and selects an element. Used by ITEs and APIs (Figure 5.23).

```
Function AdvancedSimple(controlData)
    isAdvancedSimple = False
    subType = GetRuntimeValue(controlData, "Subtype")
    hasText = ContainsRuntimeValue(controlData, "Text")
    If hasText and subType = ".NET PushButton" Then
        text = GetRuntimeValue(controlData, "Text")
        If text = "Advanced >>" or text = "Simple <<" Then
            isAdvancedSimple = True
        End If
    End If
    Return isAdvancedSimple
End Function
```

FIGURE 5.23: Host-language procedure [28]

- *Relational* (ITE): This strategy uses relational path descriptors like "_in", "_near", "_under", "_leftOf" to identify elements.

- *Mixed Strategy* (ITE): The ITEs employ several mixed strategies to automatically generate identifiers during recording. An identifier could for instance take attributes of the element and its encapsulating elements.

- *CSS Selector* (API): Expressive technology that uses *Attribute*, *Index* and *DOM Hierarchy* information and is often used in APIs (Figure 5.24). The extension *Filter Method* introduces filters like "containsWord", "notEndsWith", ...

- *JavaScript Function* (API): A procedure similar to *Host-language procedure*. The difference is that this function is executed in the browser and uses the browsers JavaScript.

```
find("input[class=rightForm]")
```

FIGURE 5.24: CSS Selector [68]

```
$("div").has("input", type: "text")
```

FIGURE 5.25: JQuery like locator - selects all div elements that have an input element as descendant that have the value text in the property type [67]

- *JQuery* (API): Expressive competitor of *XPath* and *CSS Selector* (Figure 5.25). Embeds *Regular Expressions*, *Find / Filter* predicates, *Traversing* support like "previous()", "prevAll()", "nextAll()", "parentsUntil()" and *Parameters*. Benefits from the integration into Groovy with language constructs like "range" and the "*-operator", that executes a function on all elements of a collection structure.

5.5 Execution

This section handles all features that affect the execution of test cases, including the monitoring of running tests.

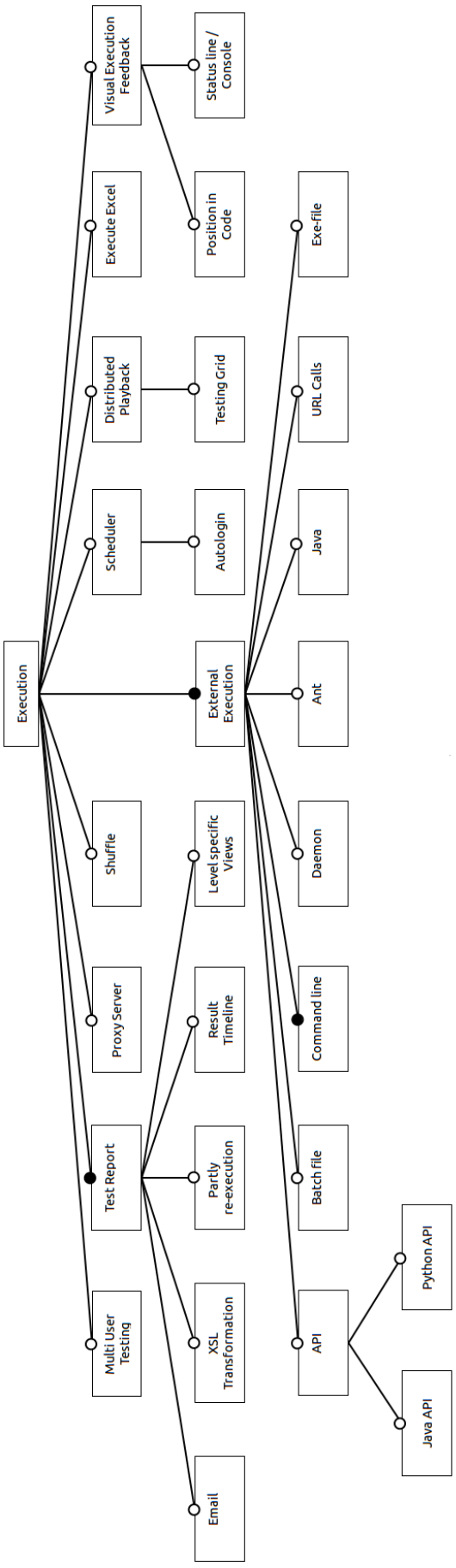


FIGURE 5.26: Feature model (ITE): Execution

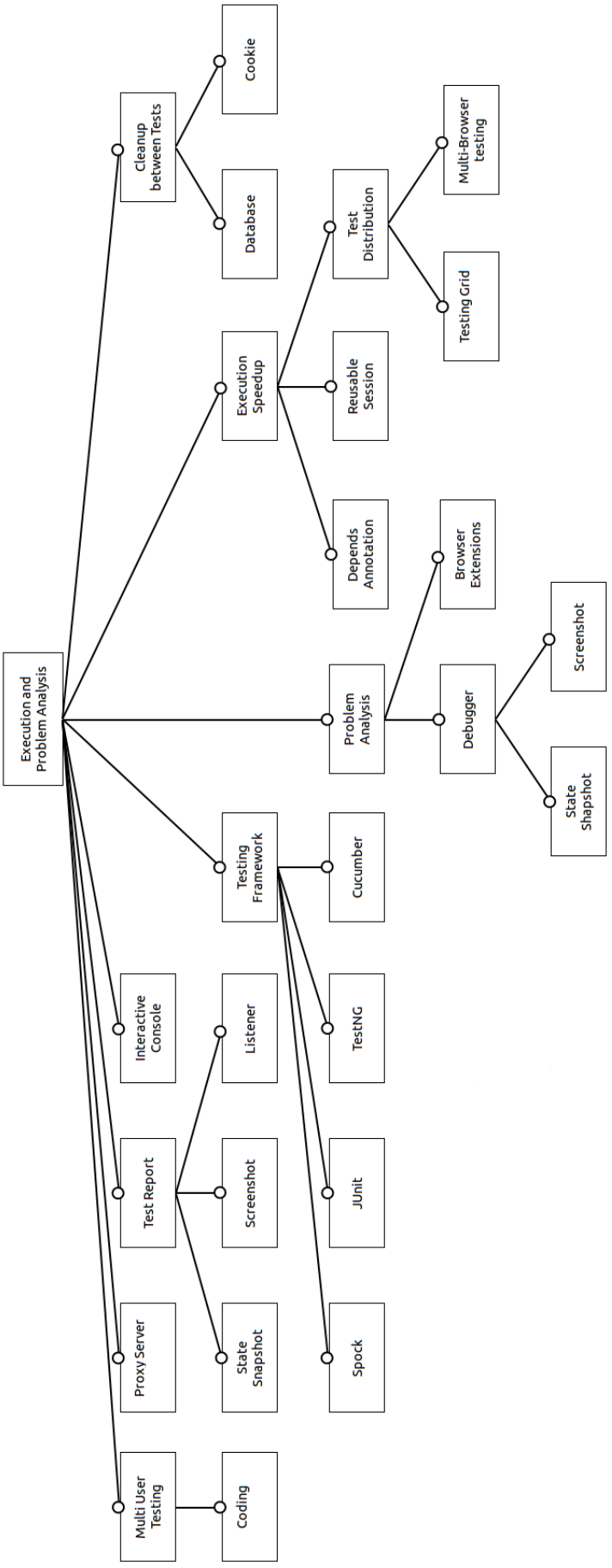


FIGURE 5.27: Feature model (API): Execution and Problem Analysis

5.5.1 Test Report

ITE SPECIFIC	
Present the results of the test execution.	
frequency	frequent

A *Test Report* represents the results of one or more test-runs.

- *Email* (ITE): Receive reports or reports with errors per mail.
- *XSL-Transformation* (ITE): Changing the report format and content using XSL-Transformation
- *Partly re-execution* (ITE): Fix a bug in the application or the test and partly re-execute the test and merge the results.
- *Result Timeline* (ITE): Sort passed and failed tests by date.
- *State Snapshot* (API): Add a snapshot of the state of the browser to the report.
- *Screenshot* (API): Add a screenshot to the report.
- *Listener* (API): Register a listener that processes reports.
- *Level specific Views*: Filter the result view to project, machine, iteration, browser, ...

Additionally the software project *BIRT* can be used to create long-term reports (Section 5.10.2).

5.5.2 External Execution

When creating or debugging tests they are executed from the ITE. In operational state indeed the test has to be executed externally. Most common is the execution via *Command line*. Some tools support *Batch file*, *Ant* or *Exe file*. But there are other options supported individually by one ITE.

- *API*: API support for integration into GPL. *Java* and *Python* are supported.

- *Daemon*: Run as background process.
- *Java*: Provided Java class that starts execution.
- *URL Calls*: Remote execution.

5.5.3 Miscellaneous

- *Proxy Server*: Indirect connections.
- *Shuffle*: Random test execution order.
- *Execute Excel*: Write scripts in Excel.
- *Visual Execution Feedback*: Get visual feedback during script replay that indicates, at which step the test execution is currently. Get feedback via *Status line / Console* or *Position in Code*
- *Cleanup between Tests*: Reset the *Database* or delete *Cookies* between tests.

5.5.3.1 Scheduler

ITE SPECIFIC	
Scheduled execution	
frequency	several times

Scheduling is the process of planning execution for the future, e.g. every Saturday night. There are special schedulers that support execution of tests on locked computers. *Autologin* executes the test and prevents the computer from unauthorized use. Mouse, keyboard and screen are disabled.

5.5.3.2 Multi User Testing

Test the interaction between multiple clients	
frequency	once

Multi User Testing is used to simulate multiple users working with the SUT. Testing real-time messaging between users on site for example. Some APIs offer *Coding* structures to describe multiple actors in one code file (Figure 5.28).

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('try multi session');
$I->amOnPage('/messages');
$nick = $I->haveFriend('nick');
$nick->does(function(AcceptanceTester $I) {
    $I->amOnPage('/messages/new');
    $I->fillFiled('body', 'Hello all!')
    $I->click('Send');
    $I->see('Hello all!', '.message');
});
$I->wait(3);
$I->see('Hello all!', '.message');
?>
```

FIGURE 5.28: Multi User Testing - two actors in one file [65]

5.5.3.3 Execution Speedup

Controlling a full web browser is slow compared to e.g. unit testing, thus there are lot features that try to speedup test execution.

ITEs use *Reusable Sessions* and *Depends Annotations*. The latter links test cases. When a test case fails, all tests that depend on this test case are excluded from execution.

Another chance is to execute tests on other machines (*Distributed Playback*). The climax is the *Testing Grid*, often supported on ITEs and APIs, that runs tests in parallel on different remote machines, potentially on different operation systems and different browsers.

5.5.3.4 Testing Framework

API SPECIFIC
Testing in GPLs is done using a testing framework.
frequency frequent

Testing in GPLs is done via testing frameworks. Traditionally used for unit testing, some APIs' support of testing frameworks blur the difference.

Most tools support *JUnit* and *TestNG*. Additional frameworks are *Spock* and *Cucumber*. These testing frameworks often comprise features like multithreading, *Depends Annotation* or *Data-driven Testing* to name only a few.

5.6 Extra Tools

The category extra tools subsumes components of the ITE that add circumscribable functionality with tool-like character.

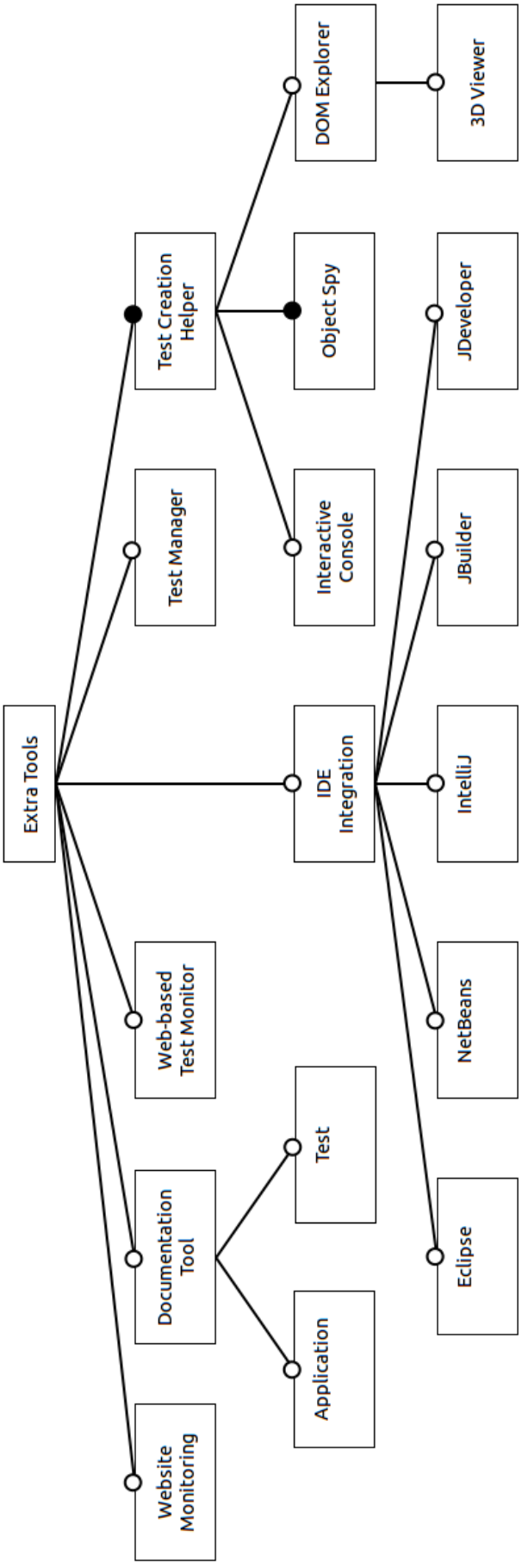


FIGURE 5.29: Feature model : Extra Tools

5.6.1 Test Creation Helper

ITE SPECIFIC	
Helps recording in the browser	
frequency	frequent
also known as	Recording Toolbar
specialization	*Interactive Console, DOM Explorer, Object Spy*

The *Test Creation Helper* helps with the recording and debugging of tests.

An *Interactive Console* is a live testing tool for test steps supported by one ITE and one API. Testing assertions or executing single test steps is possible.

An integrated *DOM Explorer* enables to view the properties of the elements under test. A special 3-dimensional version (*3D viewer*) is pictured in figure 5.30.

Almost every tool provides an *Object Spy*, a tool that shows the properties of the web page element being currently below the mouse.

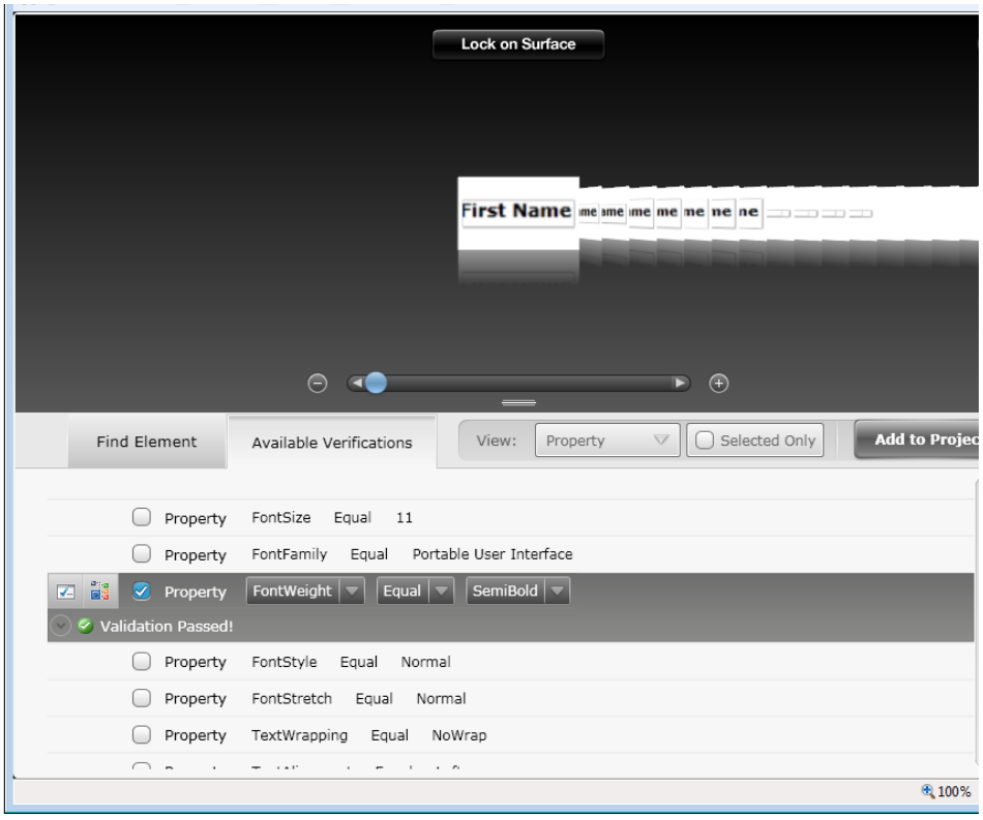


FIGURE 5.30: DOM Explorer - 3D viewer [55]

5.6.2 Miscellaneous

- *Website Monitoring*: Verify that the web page is online.
- *Web-based Test Monitor*: Monitor test execution via browser.
- *Test Manager*: One tool provides a substantial test manager including requirements management, test planning, defect tracking and reporting.

5.6.2.1 Documentation Tool

ITE SPECIFIC
Create documentation of the test cases or the SUT
frequency several times

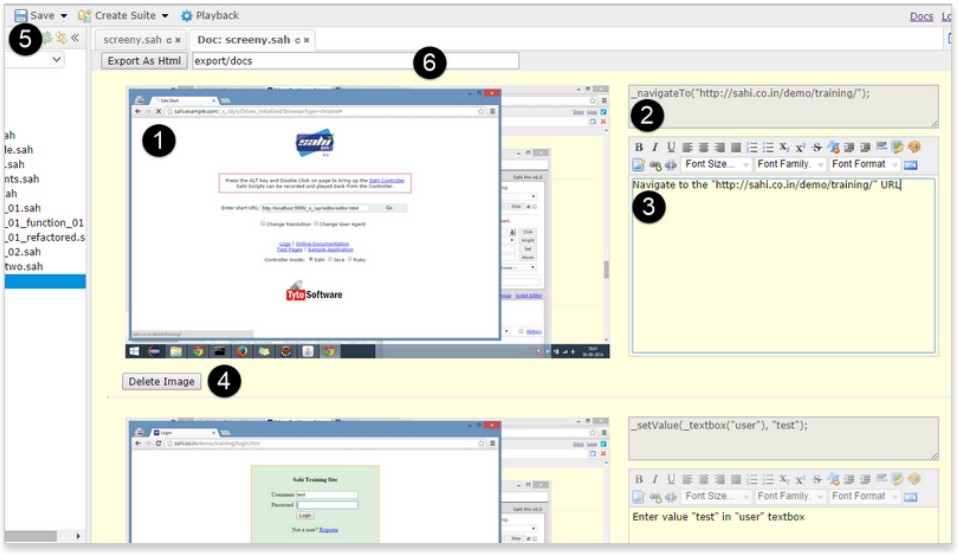
Writing documentation is often disregarded. So much the better is support for generation of documentation. Several tools added support for the documentation of tests.

One tool provides a guided generation of documentation for the SUT. With an approach similar to recording, the user traverses through the application and takes screenshots. Afterwards he adds text to the documentation template (Figure 5.31).

5.6.2.2 IDE Integration

IDE Integration is supported by one ITE. It runs with *Eclipse*, *NetBeans*, *IntelliJ*, *JBuilder* and *JDeveloper*. On the API side one tool supports *IntelliJ* and another *Eclipse*.

5.7 Language



1. The screenshot
2. The script step in a readonly textbox
3. The English version of the step. This can be edited and saved
4. `Delete Image` button lets us delete images which are not very useful to the documentation
5. `Save` button saves modifications to comments and images.
6. Enter a folder/directory path and click `Export as HTML` to export relevant images and HTML files to the specified folder.

FIGURE 5.31: Documentation Tool - create SUT documentation [34]

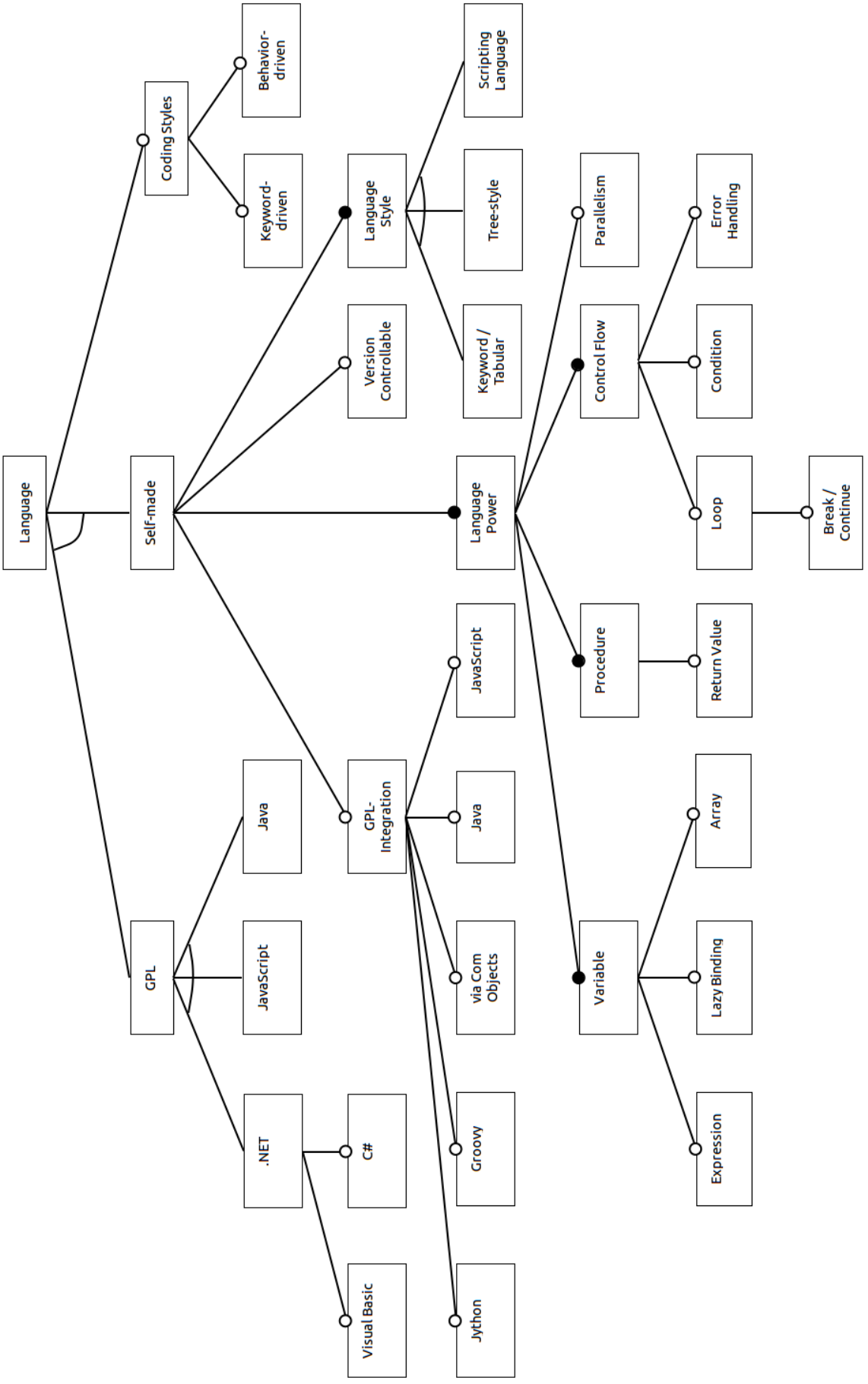


FIGURE 5.32: Feature model (ITE): Language

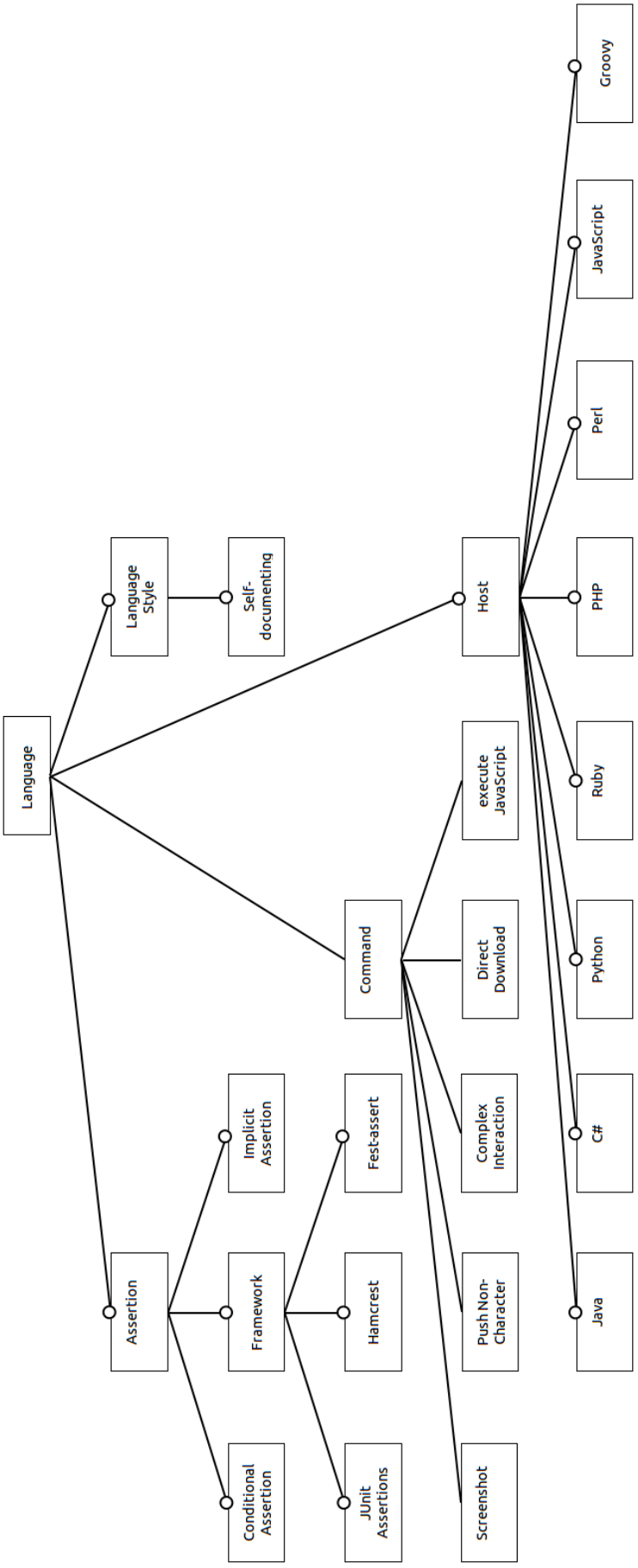


FIGURE 5.33: Feature model (API): Language

5.7.1 Self-made

Thirteen of the seventeen analyzed ITEs use a custom made language. These languages are analyzed in this section.

Several tools state that their tests are *Version Controllable*. There are certainly more tools having this feature than identified. On the other side it must be assumed that there are tools that do not or not properly support version control.

GPL-Integration is used in several tools to overcome the restrictions that the *Self-made* languages create. The following languages are integrated.

- *Jython*
- *Groovy*
- *Java*
- *JavaScript*
- *via COM Objects*: Languages that support Microsofts COM (Component Object Model) Objects, e.g. C++

5.7.1.1 Language Style

The *Self-made* languages are designed with different language styles, which describe noticeable characteristics in the syntax. The basic style is *Scripting Language*, which is the standard case.

The *Keyword/Tabular* style is identifiable by the fixed number of parameters per command. The commands are called keywords. When using a keyword it is followed by the parameters.

Tree-style languages present the code in a hierarchy similar to the file explorers provided by operation systems (Figure 5.34). Each command is displayed with a short description. Each command has a properties page to change it.

The *Self-documenting* style is featured by one API. The syntax of the commands enables to read them as if they were test documentation (Figure 5.35).

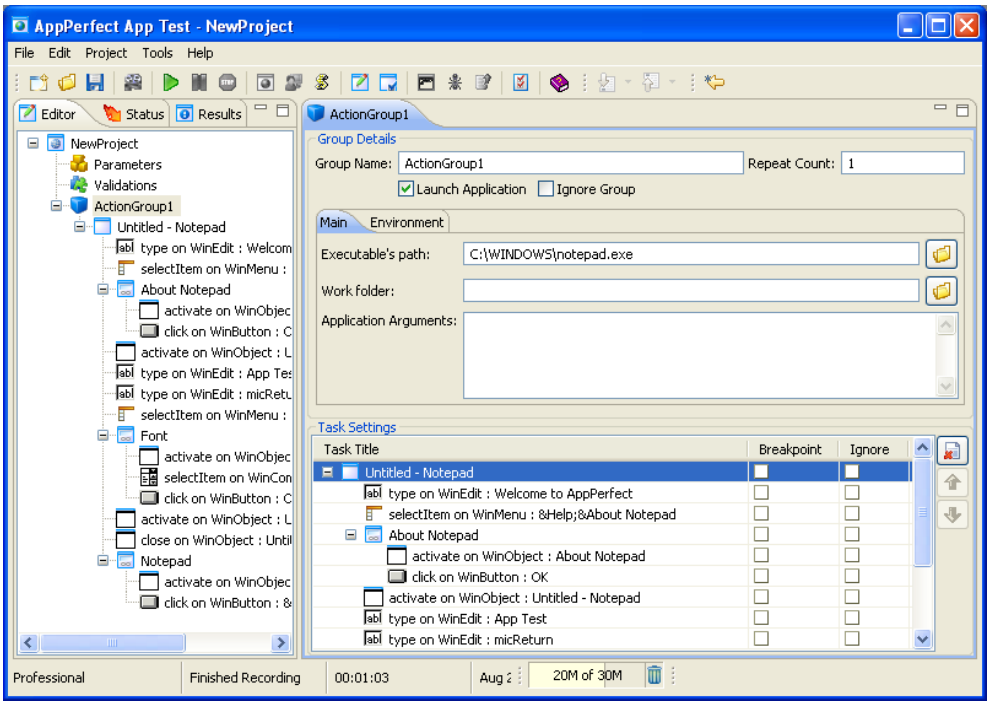


FIGURE 5.34: Example of a tree-style language [30]

```
<?php
$I->amGoingTo('submit user form with invalid
$I->fillField('user[email]', 'miles');
$I->click('Update');
$I->expect('the form is not submitted');
$I->see('Form is filled incorrectly');
?>
```

FIGURE 5.35: Example of a self-documenting language [65]

5.7.1.2 Language Power

This section describes the distribution of language features. All these language features are common in GPLs and if missing in *Self-made* languages they lead to problems in test design.

Frequent features:

- *Variable*
- *Procedure*

- *Control Flow*: (abstract feature)
- *Condition*: If statement

More than once occurring features:

- *Array*
- *Return Value*: Procedure with return value.
- *Loop*: While, for or loop
- *Break / Continue*: Exit points for loop statements.
- *Error Handling*
- *Parallelism*

Unique features:

- *Expression*: Use expressions to define the value of a variable.
- *Lazy Binding*: Variable value is read at the last time as possible.

5.7.2 Coding Styles

Coding Styles are about different ways to write test cases. Test cases written with different *Coding Styles* can describe the same test case, but from different perspectives (e.g. workflow or behavior perspective).

- *Keyword-driven*: Workflow oriented. Take the SUT into the initial state, do something in the SUT and finally verify that the system behaved as expected.
- *Behavior-driven*: Stakeholder oriented. Test cases are written as requirements that also non-technical stakeholders must understand. That can be done using the popular Given-When-Then style known from “Behavior Driven Development” (BDD). Other keywords from this area are “Acceptance Test Driven Development” (ATDD) and “Specification by Example”.

5.7.3 API / GPL

5.7.3.1 Host

On the ITE side five tools use a GPL. Two each support *Java*, *JavaScript* and *.NET*, including *Visual Basic* and *C#*.

Following languages are supported on the API side:

- *Java*
- *C#*
- *Python*
- *Ruby*
- *PHP*
- *Perl*
- *JavaScript*
- *Groovy*

5.7.3.2 Command

- *Screenshot*: see section 5.2.4
- *Push Non-Character*: Support keydown events of non-characters like the delete key.
- *Complex Interaction*: E.g.: Hold the shift key and double click (Figure 5.36).
- *Direct Download*: Download resources
- *execute JavaScript*: Execute JavaScript code in the browser.

```
interact {
    keyDown(Keys.SHIFT)
    doubleClick($('li.clicky'))
    keyUp(Keys.SHIFT)
}
```

FIGURE 5.36: Feature - Complex Interaction [67]

5.7.3.3 Assertion

- *Conditional Assertion*: Assertions that only log failure instead of terminating the execution (Figure 5.37).
- *Framework*: Supported frameworks are *JUnit Assertions*, *Hamcrest* and *Fest-assert*.
- *Implicit Assertion*: Switch expressions automatically into assertions, if needed.

```
<?php
$I->canSeeInCurrentUrl('/user/miles');
$I->canSeeCheckboxIsChecked('#agree');
$I->cantSeeInField('user[name]', 'Miles');
?>
```

FIGURE 5.37: Feature - Conditional Assertion [65]

5.8 Methodology

Features that support a methodological testing principle are described in this category.

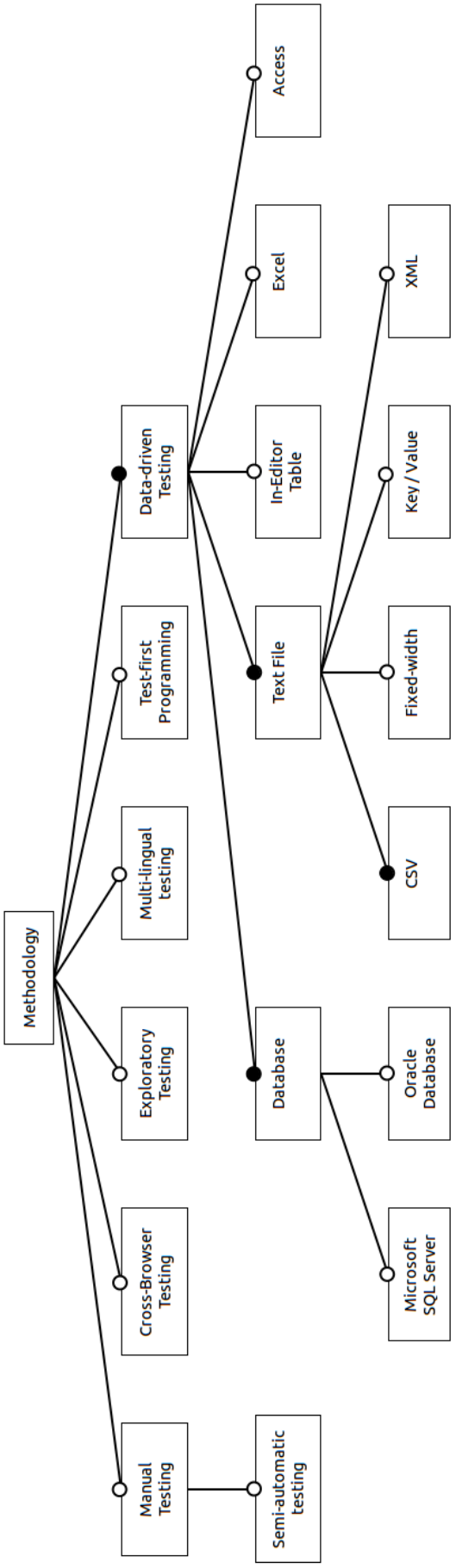


FIGURE 5.38: Feature model : Methodology

5.8.1 Data-driven Testing

ITE SPECIFIC
Execute one test with different sets of test data
frequency frequent

The goal of *Data-driven Testing* is to run a given test multiple times with different sets of input data and expected results. This is done by separating the test code from the test data. Each set of test data consists of a set of input values and a set of result values. The test code is executed with the input values and checks if the result equals the specified result values.

The tools differ in the formats of test data they support. Most tools support *CSV*-files (Comma separated values) or *Databases*. Also prominent is to manage the test data in the ITE (*In-Editor Table*) or as *Excel* spreadsheet. Other unique occurring formats are Microsofts *Access*, text files that use *Fixed-width* to separate values, *Key / Value* structured text files and *XML* files.

5.8.2 Manual Testing

ITE SPECIFIC
Add manual besides automated test cases
frequency several times

In general manual testing is the process of testing without test automation. Thus test automation is the opposite of manual testing. The reason why ITEs support manual testing is, that in general not every test case can be or should be automated. There are always tests where the effort of automation is to high compared to the effort of manual testing.

Several ITEs integrate *Manual Testing* into the testing process. Tracking the results of manual tests from within enables to combine the results of automated and manual testing. Manual tests are created as test cases. On test execution the tester has to inform the ITE on the success or failure of manual test cases and possibly provide error descriptions (Figure 5.39).

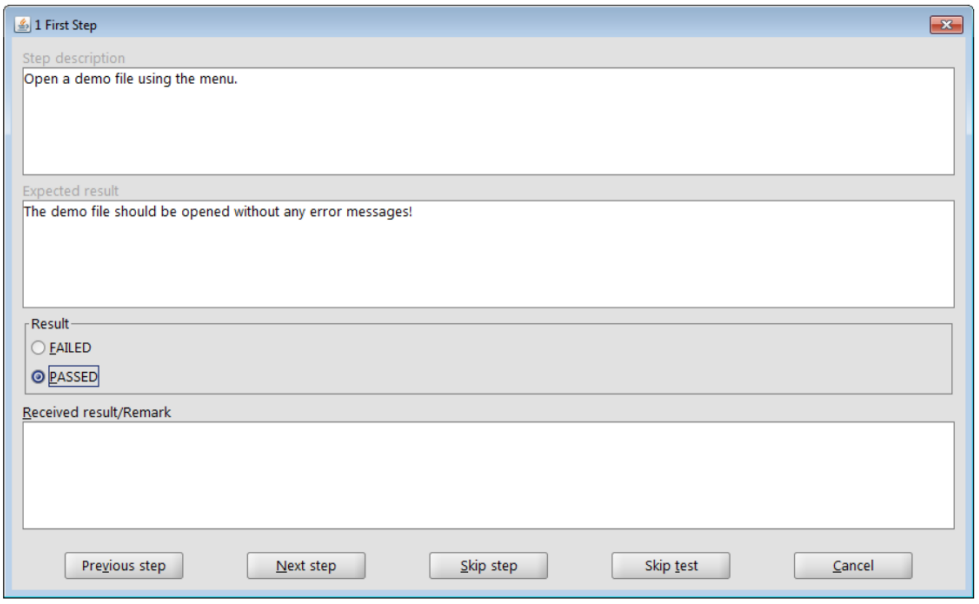


FIGURE 5.39: Manual Testing - Feedback of manual test execution [26]

A special case is *Semi-automatic Testing*, which supports test cases with both automated and manual test steps. This can be a test, which automatically starts the application and navigates it to the point where the tester checks the test condition. It can also be a test where automated and manual tests are mixed freely and the tester actually has to provide data that is needed as variable for subsequent automated steps.

Supporting *Semi-automatic Testing* is additionally important as it is the transition process from manual testing to automated testing. Automating more test steps over time finally leads to a fully automated test case.

5.8.3 Exploratory Testing

ITE SPECIFIC	
Unstructured testing to find errors per chance	
frequency	once

Exploratory Testing is called the process of unstructured testing of the application to observe errors by chance. An ITE is able to support this with screenshot capturing, bug tool connection and documentation of the steps that produced the error.

5.8.4 Multi-lingual Testing

ITE specific	
Execute the same tests for a SUT in different languages	
frequency	once
similar to	*Data-driven Testing*

Multi-lingual Testing helps testing of the application in multiple languages. Essentially this is done by executing all test cases, that are concerned with language specific content (like buttons, messages), once for each language. Similar to *Data-driven Testing* the varying text is substituted for variables and for each variable the values in the different languages are defined. The test is executed each time with a different set of variables.

5.8.5 Test-first Programming

ITE specific	
First write high level test and bind them later to the GUI	
frequency	once

Test-first Programming is the philosophy that demands to start development with writing test cases. Afterwards the application is developed until it passes the tests. *Test-first Programming* has origin in unit testing.

With *Recording* however *Test-first Programming* is problematic, as recording test cases requires the GUI. There is one tool not supporting *Recording* but supporting *Test-first Programming*. The test cases are defined on a high level. When the GUI of the application is far enough developed to be tested, the high level statements are detailed and finally connected with element identifiers.

5.9 Problem Analysis

Problem Analysis is about the features that are needed if an error occurs. Errors occur either because the SUT has a bug that needs to be fixed or because a test case needs to be fixed e.g. be adapted to evolution change in the SUT.

If the SUT is extended it occurs that as yet unique element identifiers break. *Smart Matching* is a technique that tries to keep the identifier unique.

On the API side it is common to use *Browser Extensions* to for example view the DOM. APIs support this by adding the desired *Browser Extensions* to the browser that executes the test case[4].

[4] In general the test browsers are reset every startup and contain no browser extensions.

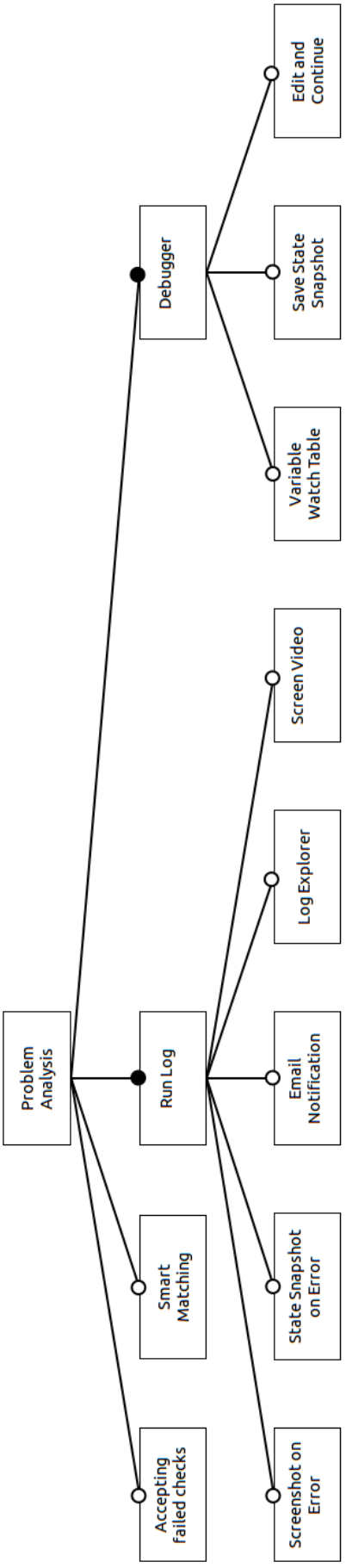


FIGURE 5.40: Feature model : Problem Analysis

Accepting failed checks

ITE specific
Design by current state
frequency once

If an assertion fails, but the tester is certain that the SUT is right and the test case is wrong, *Accepting failed checks* updates the tests assertion with the data from the SUT. This is helpful to adapt change from the evolution of the SUT or if the tester is simply to lazy to write assertions on his own.

5.9.1 Run Log

ITE specific
Test results, warnings, errors and more
frequency frequent

During execution the tests log their progress, warnings and exceptions into one or multiple log files. This *Run Log* is a basic features that can be extended to be more useful.

- *Screenshot on Error*: Identify false negatives or understand right negatives by screenshot.
- *State Snapshot on Error*: Capture the DOM for analysis.
- *Email Notification*: Get notified immediately.
- *Log Explorer*: Integrate the log into the ITE. Link errors to the test step that generated it.
- *Screen Video*: Completely retrace test execution.

5.9.2 Debugger

Step by step error diagnostics
frequency frequent

The *Debugger* is used to debug a test case step by step. It is a frequent feature, but the different *Debuggers* differ in the extensional features they provide.

To have to full overview over the executing test a *Variable Watch Table* is necessary. At least two tools support it. In addition to the state of the variables, the state of the DOM is just as important. There are tools that enable a *State Snapshot* or a *Screenshot*.

Edit and Continue allows to change the test case during debugging and to try the made change immediately. E.g. a failed assertion stops the debugger, the tester changes the assertion and checks if it is valid now.

5.10 Technology

The technology section describes the support of external technologies into the ITEs and APIs.

Two ITEs include *Password Encryption* to protect passwords used in testing. Another tool enables with *Keystroke Encryption* a similar protection. Two tools enable to create test cases using *Authentication* protocols like *HTTPS*.

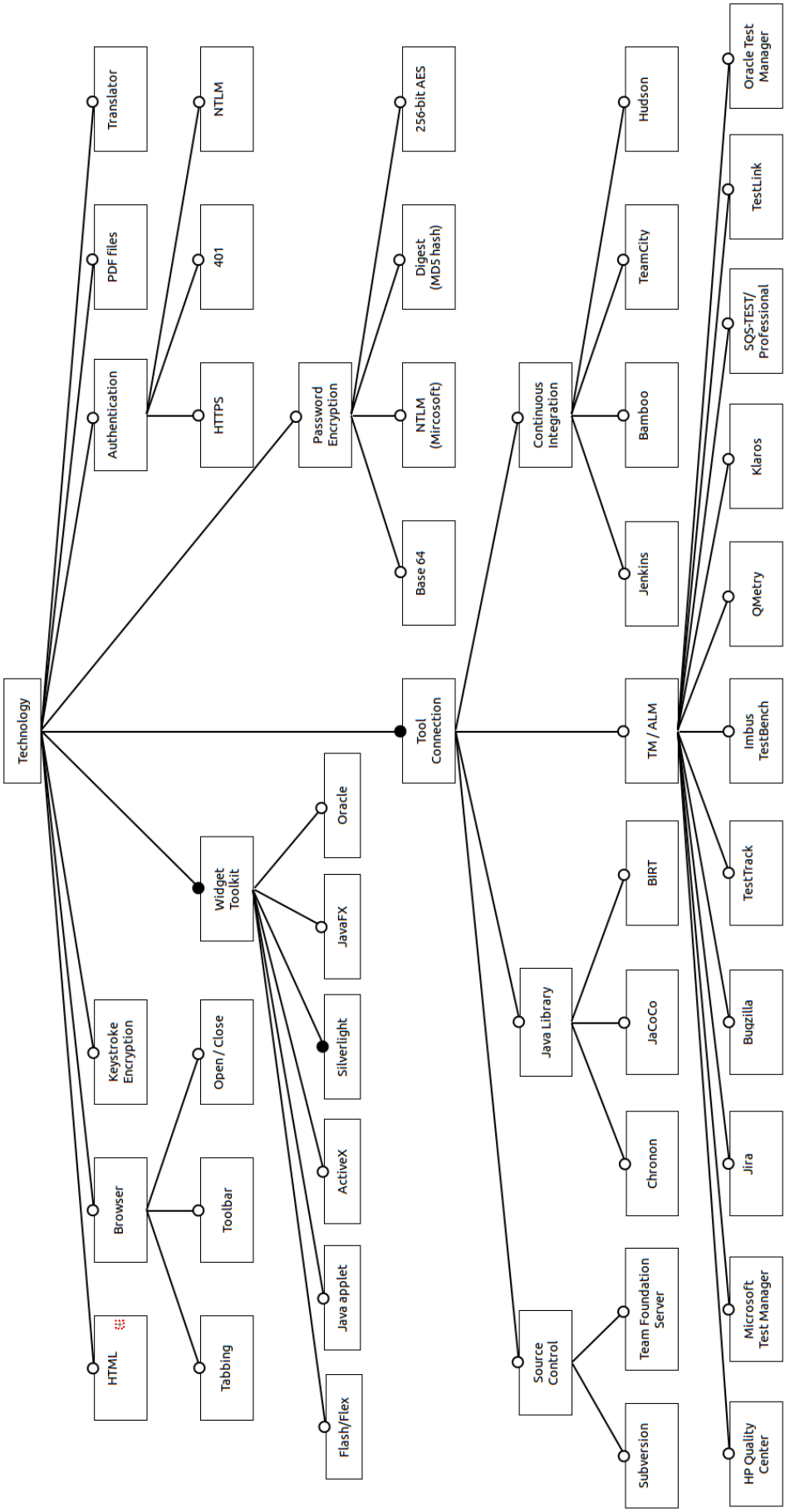


FIGURE 5.41: Feature model (ITE): Technology

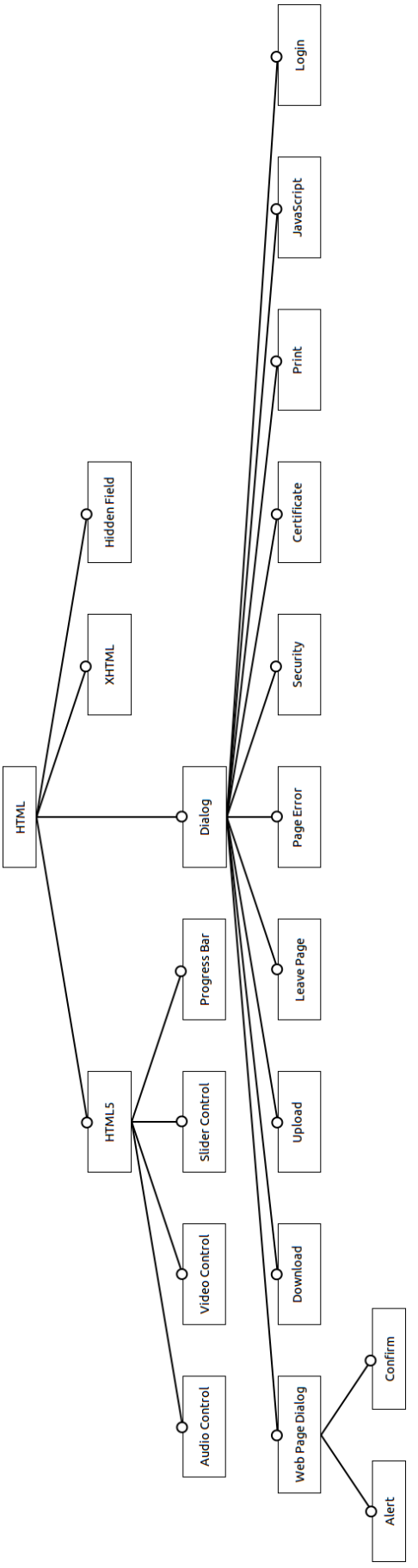


FIGURE 5.42: Feature model (ITE): Technology - HTML

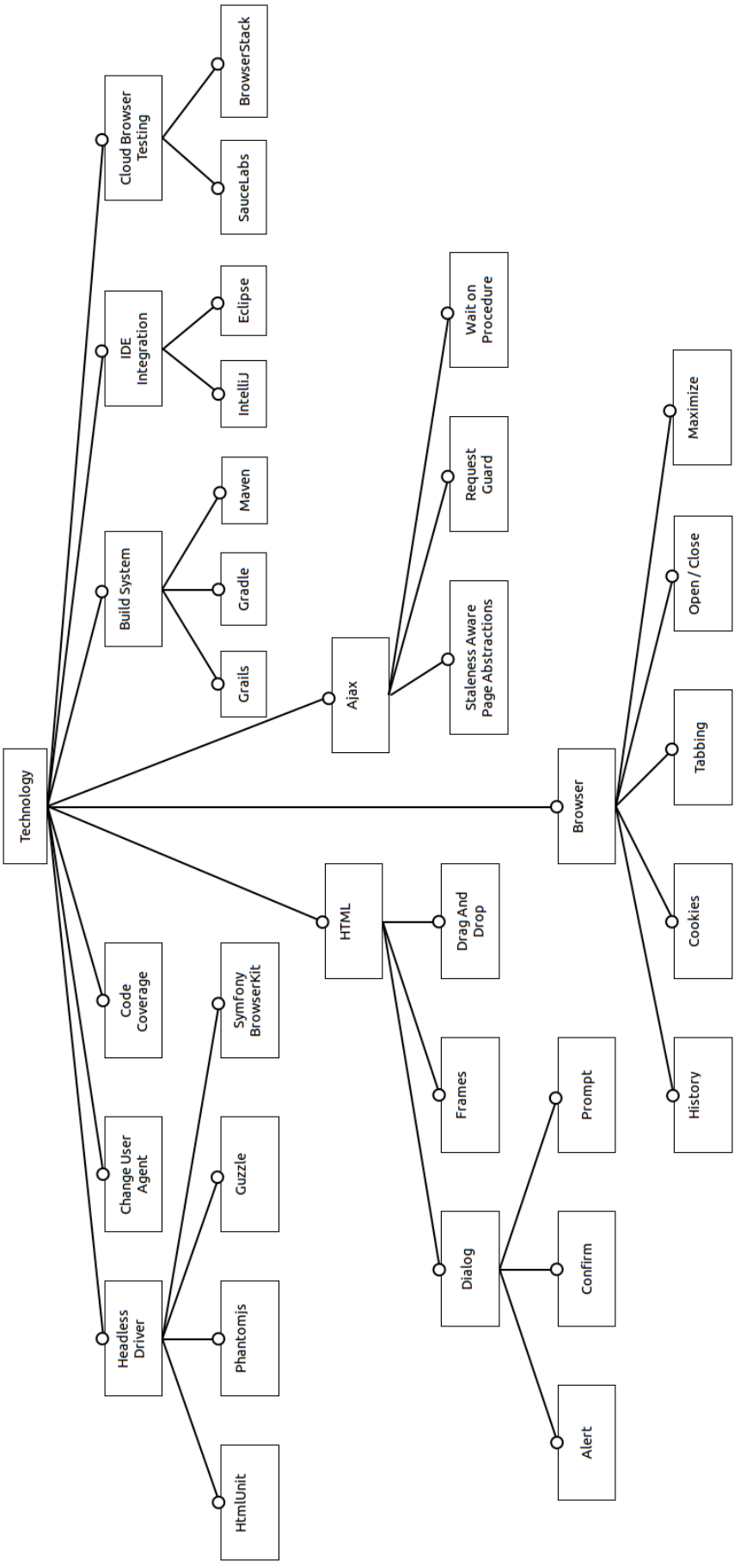


FIGURE 5.43: Feature model (API): Technology

5.10.1 HTML

Since HTML is the essential language for web applications. All ITEs and APIs support HTML. However there are differences e.g. how far new technologies like *HTML5* are supported. *XHTML* is only supported by one tool, since it represents a dead end in the development of HTML [74].

The support of *Dialogs* is also diverging. Since the dialogs differ between browsers, they can be a problem when testing on multiple browsers. One API introduces an approach that transforms dialogs into HTML pages.

5.10.2 Tool Connection

- *Source Control*: Also known as Revision control or version control systems.
- *Java Library*: *JaCoCo* is a code coverage library. *Chronon* records the execution of Java programs and can be used to do post mortem debugging. *BIRT* is a software project used to create data visualizations and reports. It is used to generate long-term reports.
- *TM / ALM*: Test Management and Application Lifecycle Management tools.
- *Continuous Integration*: Several tools support *Jenkins* and *Hudson*.
- *Build System* (API): Also known as build automation tools.

5.10.3 Miscellaneous

Translator

ITE SPECIFIC	
Integrate widget toolkit	
frequency	once

If a specific *Widget Toolkit* is not supported, the tester may develop a *Translator* to integrate it into the ITE. In detail each element type needs a *Translator* to be automatable.

Code Coverage

API SPECIFIC	
Which code is covered by test cases?	
frequency	once
similar to	*JaCoCo*

Code Coverage enables to check which code is and which code is not executed, when the tests are executed.

Cloud Browser Testing

API SPECIFIC	
External execution by a cloud computing provider	
frequency	once
similar to	*Testing Grid*

As described in the research context 1.2, Cloud testing is the combination of web testing and cloud computing. APIs with the feature *Cloud Browser Testing* support external test execution by a cloud computing provider.

To give an example a Gradle plug-in is provided which simplifies declaring the account and the browsers to be tested, as well as configuring a tunnel to allow the cloud provider to access local applications.

5.10.3.1 Browser

Instead of automating the application that is running in the browser, in can be necessary to automate the browser.

- *Tabbing* (ITE, API): Changing the active or creating a new tab or window.
- *Toolbar* (ITE): Activating toolbar elements.

- *Open / Close* (ITE, API): In general the opening and closing of the browser is automatic. However it may be needed to do it manually for example to test code executed when closing or opening the browser.
- *History* (API): Navigate forward and backwards in the browser history.
- *Cookies* (API): Create, read or delete cookies.
- *Maximize* (API): Set the browser to maximum size.

5.10.3.2 Widget Toolkit

Thirteen ITEs support widget toolkits that transcend the fundamental HTML. Frequently supported are *Flash/Flex* and *Silverlight*. Two tools each support *Java applet*, *ActiveX* and *JavaFX*. One tool is supporting several *Oracle* widget toolkits like EBS, Siebel, JD Edwards and Fusion/ADF.

5.10.3.3 Headless Driver

The concept of headless browsers has been introduced in section 1.1. APIs support *Headless Drivers* to provide an remarkably faster alternative to browser drivers.

5.10.3.4 Ajax

Testing Ajax applications is difficult because of the asynchronous behavior. Several APIs feature special support for *Ajax*. Waiting for the results of an Ajax-request to be received enables the feature *Wait on Procedure*. *Request Guards* are used to verify that an Ajax or HTML request is send or that no request is send. The feature *Staleness Aware Page Abstractions* makes sure that observed stale elements are re-initialized.

Chapter 6

Conclusion

This chapter gives an interpretation of the results presented in the last chapter. Section 6.1 presents a qualitative analysis of the trends, problems and differences between ITEs and APIs that have been observed. Section 6.2 presents a quantitative analysis approach.

6.1 Trends, Problems and Differences between ITEs and APIs

Abstraction Supporting Feature

The features in the category *Abstraction Supporting Feature* provide different advantages. *Dependencies*, *Generic Components*, *Interproject Relationship* help to prevent code duplication. The *Avatar System* feature can save time when creating tests. Maintenance cost reduction is achieved by using *Mapping Table* or the more advanced *Component Model*. The changes need to be done manually, but at least at only one position. Furthermore these changes can be automated using *Updating Components*. The *Element Recognition* allows to use the tests without adaption as long as the elements can still be identified. To improve the overview, *Elements Explorer* and *Setup / Tear down* are useful.

The overall idea of abstraction in web page automation is to use object orientation to map the structure of web pages. This leads to a better overview and simplifies maintenance. In addition it completely separates the test case from the element identification. *Page*

Objects benefit from the potentials of object orientation like *Inheritance*. One trend is to standardize the internal structure of page objects. *Template Options* increase the understandability and uniformity of page objects. The “wait” and “required” statements reduce coding afford of commonly used patterns. The statements “to” and “at” simplify the transition between page objects.

The most simple version of a *Step object* is a procedure that enables the reuse of the commands or identifiers it encapsulates. The trend is to integrate *Step Objects* into *Page Objects*. Page objects may include step objects for example to specify properties.

Summing up, ITEs and APIs have different approaches and thus different features. ITEs feature preferably automatic test creation and maintenance, whereas the API approach is to achieve maximum abstraction and overview.

Problematic is that *Webpage element indirection* is not featured by all ITEs. Half the tools do neither support *Mapping Table* nor *Component Model*, whereas on the API side every GPL enables to encapsulate element identifiers using objects. Most tools may enable encapsulating identifiers using *Procedures*, but doing so it is not possible to use any recorded test case without adapting it.

Capture

The *Capture* category is dominated by the *Recording* feature, that all except one ITE support. The *Freeze Mode* is an excitement feature whereas *Record Procedure* is a performance feature that should be supported by more than one tool. This analysis reveals two *Identifier Creation Strategies*, but there are by all means more strategies that need another methodology to be identified.

Image Recognition is multifarious. It is used for text in picture format, to validate the layout and for element identification using *Image Click* or *Retrieve Position*. *Image Recognition* separates the ITEs from the APIs that, at least in this sample, do not support it.

Editor Features

Providing different *Code Views* is an interesting concept. *Code View* is the original view that for example all IDEs on the market use. The *Keyword / Tabular View* is an option for users that are not familiar with programming languages. The *Tree View* illustrates the execution order and increases the overview, but otherwise conceals details behind property windows. The *Flow Chart View* and especially the *Storyboard View* should be emphasized. The latter helps rereading and understanding test cases, because it provides the connection between each test command and the GUI of the SUT in that moment.

Learning by doing tutorial and *Refactoring* are excitement features, but *Refactoring* is only supported by one tool and should be featured more.

Element Identification

In comparison to the APIs the IDEs utilize more information sources, because this is useful for automatic identifier creation (e.g. *Mixed Strategy*). *DOM Hierarchy* is an ITE specific feature used in *Element Recognition*. A typical ITE problem is the handling of random "ids", that is counteracted with *Wildcards* and *Regular Expressions*.

APIs utilize less information sources, but provide advanced identifier methods. Three powerful methods have been observed. In addition to *XPath*, the API also feature *CSS Selectors*, *JQuery* and above all, these methods are combined.

Execution

The category of execution oriented features is dominated by a lot of external execution options. Features that should be highlighted are the *Partly re-execution* of test reports, the *Shuffle* mode and the *Autologin* feature of the scheduler that is needed if there is no separated testing server and the test cases should be executed on the machine of the tester.

On the API side a lot of features are oriented on execution speedup: *Depends Annotation*, *Reusable Session*, *Test Distribution* and in the broader sense *Cleanup between Tests*. *Testing Frameworks* also provide such features.

An excitement feature is *Multi User Testing* from one code fragment (*Coding*). The traditional way involves starting two testing processes that interact asynchronous. Writing the test as one code fragment enables writing it, as if it were synchronous. Thus *Coding* leads to complexity reduction.

Extra Tools

A prominent companion of ITEs is the *Test Creation Helper* including an *Object Spy*. These features exists as conventional browser plug-ins that also feature the *DOM Explorer*. Another feature that should be emphasized is the *Interactive Console* that permits significant test creation and debugging speedup. One ITE and API provide it.

As an alternative to providing a feature rich editor one ITE facilitates *IDE Integration*.

Another interesting feature is the *Documentation Tool*. Especially the version that enables generating end user documentation is a promising idea that reuses the recording functionality.

Language

The majority of ITEs use a self-made language, but it should be noticed that because of lacking *Language Power* features, they often tend to be inexpressive. For compensation they use *GPL-Integration*

An excitement feature is the *Self-documenting* language style, that enables even non technical stakeholders to read test cases. A development focus is on Assertions. *Conditional Assertions* and *Implicit Assertions* do both pursue the goal to help in debugging and deliver “Fail-Fast” if needed, but are flexible to ignore errors if not.

Methodology

The wide propagation of *Data-driven Testing* including the modification *Multi-lingual testing* should be emphasized. Noticeable is that, at least in the occurring setting, *Recording* and *Test-first Programming* are contradictory features.

Full test automation is not always desirable. There are situations where test automation is not possible or the effort of test creation and maintenance is higher than the benefit. Therefore, *Manual Testing* and *Semi-automatic testing* should be emphasized as they integrate and speedup manual testing.

Problem Analysis

To identify the reason for a program error, all available information is acceptable. Both ITEs and APIs use screenshots and state snapshots. Most advanced is the *Screen Video* that enables post mortem analysis.

The *Smart Matching* approach that ensures during recording that identifiers stay unique is a feature that only makes sense in the context of automatic identifier creation.

On the API side the feature *Edit and Continue* should be emphasized. Another excitement feature is *Accepting failed checks*.

Technology

A unique selling point for ITEs is the support of *Widget Toolkits*, that none of the analyzed APIs support. Additionally the ITEs support a wide range of *TM / ALM* and *Continuous Integration* tools.

The APIs on the contrary seem to provide advanced features for *Ajax* testing.

6.2 Equality of Tools

This section provides statistics that illustrate how equal the feature sets of the ITE tools are. The API tools have not been analyzed with this approach[1]. Table 6.1 presents a pairwise comparison of the features. The amount of common features is divided by the higher total number of features. Remarkably are three tool pairs with high percentage of common features. Rapise and Application Testing Suite with 58%, TestingWhiz and

[1]The analysis is inappropriate in that context because the tools are interlaced. In fact all five build on top of Selenium and therefore have a common set of features. However some list the features inherited by Selenium in the documentation and some do not. Hence the results of the equality calculations would be distorted.

WinTask with 48%, and Rapise and Silk Test with 46%. An explanation is that these tools primarily consist of basic features with no or few unique features.

Figure 6.1 presents the distribution of the result of the feature equality matrix (Table 6.1). The climax is in the range between 20% to 24%. This indicates that there are a common sets of basic features. Additionally there are a lot tools on the one side that are equal between 10% and 19% or on the other side between 25% and 34%. That may indicate that there are other feature sets that several tools possess or several other do not.

	QF-Test	QA Wizard Pro	AppPerfect Web Test	iMacros	Sahi Pro	Robot Framework	Application Testing Suite	WinTask	TestingWhiz	CodedUI	Rapise	Testing Anywhere	Ranorex Test Automation	Silk Test	Test Studio	RIATest
QA Wizard Pro	30%															
AppPerfect Web Test	23%	28%														
iMacros	20%	20%	26%													
Sahi Pro	19%	26%	37%	21%												
Robot Framework	17%	26%	16%	17%	24%											
Application Testing Suite	19%	30%	32%	19%	24%	22%										
WinTask	9%	22%	11%	10%	22%	28%	13%									
TestingWhiz	16%	24%	16%	21%	19%	38%	25%	48%								
CodedUI	7%	8%	8%	10%	5%	9%	17%	12%	13%							
Rapise	19%	26%	34%	19%	27%	19%	58%	13%	21%	17%						
Testing Anywhere	17%	36%	21%	24%	22%	29%	21%	26%	32%	6%	24%					
Ranorex Test Automation	25%	32%	32%	26%	30%	23%	37%	17%	29%	9%	31%	31%				
Silk Test	10%	18%	21%	12%	24%	13%	42%	22%	22%	26%	46%	26%	34%			
Test Studio	25%	36%	22%	26%	24%	15%	22%	17%	22%	11%	22%	22%	35%	17%		
RIATest	23%	30%	29%	21%	41%	28%	29%	19%	23%	13%	29%	38%	40%	35%	35%	
Jubula	20%	22%	18%	12%	14%	19%	18%	11%	18%	4%	18%	21%	23%	14%	9%	16%

TABLE 6.1: Feature equality matrix: Pairwise percentage of features that have two tools in common.

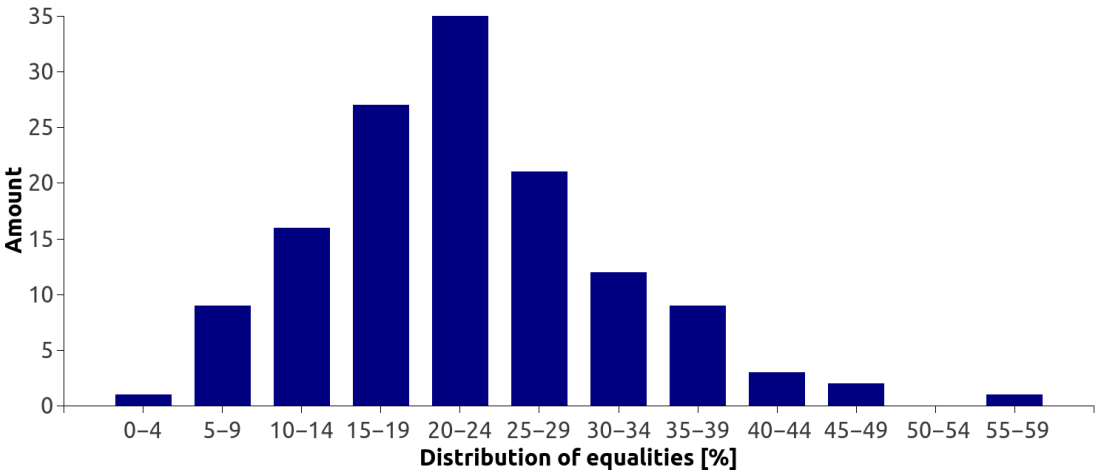


FIGURE 6.1: Distribution of equalities between the ITE tools.

Chapter 7

Summary

7.1 Outlook

This thesis contributed a methodology that enables to analyze a field of tools in order to discover the overall status and trends by using feature modeling.

This thesis delimited the domain of functional, system level web application testing and analyzed it using the methodology. From an initial pool of 212 candidates 23 tools passed the requirements of the methodology and were analyzed. The results are 313 identified features that have been classified into 10 categories, illustrated using 16 feature diagrams and described in detail.

7.2 Threads to Validity

The analysis is done by one researcher. Features can be missed, for example when marking the documentations. Another problem could be variegating granularity in feature identification. A review by another researcher could reduce both issues.

The methodology depends on high quality end user documentation. A problem would be if too many tools are excluded from the analysis because of this restriction. However, tools that do not have a proper documentation are possibly less feature rich and are anyway are rarely used in industry.

Variegating quality of the documentation between the different tools is not a thread, because the methodology does not primarily aim to compare tools.

The methodology misses all features that are not described in the documentation. There are probably features that can be identified via other information sources or by testing the application.

Appendix A

Tool - Feature List

This Appendix presents the data of observed features for each tool. Each tools features are ordered by category. The features are additionally sorted alphabetically. Since the naming of the features is influenced by their position in the feature models' hierarchy, the feature lists should be used having the feature models at hand.

Term	Description
Abstraction Supporting Features	Component Model, Dependencies, Element Recognition, Elements Explorer, Error Escalation, Generic Components, Importing / Merging, Items, Probabilistic, Referencing, Test Suite Linking, Updating Components
Capture	Image Recognition, Image Validation, Record Procedure, Recording
Editor Features	Learning by doing tutorial, Move, Refactoring, Rename, Tree View
Element Identification	Attribute, DOM Hierarchy, Name Resolver, Position / Size, Regular Expression
Execution	Batch file, Command line, Daemon, External Execution, Position in Code, Status line / Console, Test Report, Visual Execution Feedback
Extra Tools	Documentation Tool, Test
Language	Expression, GPL-Integration, GPL-Integration Groovy, GPL-Integration Jython, Lazy Binding, Procedure, Self-made, Tree-style, Variable
Methodology	CSV, Data-driven Testing, Database, Excel, In-Editor Table, Manual Testing
Problem Analysis	Accepting failed checks, Debugger, Run Log, Screenshot on Error, State Snapshot on Error
Technology	Continuous Integration, HP Quality Center, Hidden Field, Hudson, Imbus TestBench, JavaFX, Jenkins, Klaros, QMetry, SQS-TEST/Professional, TM / ALM, TestLink, Widget Toolkit

Table A.1: QF-Test: Feature list

Term	Description
Abstraction Supporting Features	Global Table, Mapping Table, Referencing, Test Suite Linking
Capture	Image Recognition, Image Validation, Low-level Action, OCR, Recording
Editor Features	Keyword/Tabular View, Text Encryption, Text View
Element Identification	Attribute, Host-language Procedure, Regular Expression, XPath
Execution	Batch file, Command line, Distributed Playback, Email, External Execution, Multi User Testing, Test Report
Extra Tools	Web-based Test Monitor
Language	Array, Condition, Error Handling, Loop, Procedure, Return Value, Scripting Language, Self-made, Variable
Methodology	Access, CSV, Data-driven Testing, Database, Excel, Fixed-width, Microsoft SQL Server, Oracle Database
Problem Analysis	Debugger, Smart Matching, Variable Watch Table
Technology	Flash/Flex, Silverlight, TM / ALM, TestTrack, Toolbar, Widget Toolkit

Table A.2: QA Wizard Pro: Feature list

Term	Description
Abstraction Supporting Features	Component Model, Element Recognition, Max matching attributes, Referencing, Test Suite Linking
Capture	Record Time needed, Recording
Editor Features	
Element Identification	Attribute, Index
Execution	Ant, Command line, Distributed Playback, External Execution, Level specific Views, Test Report
Extra Tools	Eclipse, IDE Integration, IntelliJ, JBuilder, JDeveloper, NetBeans, Object Spy, Test Creation Helper
Language	GPL-Integration, GPL-Integration JavaScript, Self-made, Tree-style
Methodology	CSV, Data-driven Testing, Database
Problem Analysis	
Technology	Authentication, Base 64, Digest (MD5 hash), Flash/Flex, HTTPS, NTLM (Mircosoft), Password Encryption, Widget Toolkit

Table A.3: AppPerfect Web Test: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Best fitting single Attribute, Change User Agent, Full Screenshot, Identifier Creation Strategy, Recording, Screenshot, User Selections
Editor Features	
Element Identification	Attribute, Position / Size, Relative to Element, Relative to Image, Wildcard
Execution	Proxy Server, Test Report
Extra Tools	Website Monitoring
Language	GPL-Integration, Keyword / Tabular, Self-made, Variable, via Com Objects
Methodology	CSV, Data-driven Testing, Database, In-Editor Table, Key / Value
Problem Analysis	Run Log, Screenshot on Error
Technology	256-bit AES, Certificate, Dialog, Flash/Flex, Java applet, JavaScript, Login, Page Error, Password Encryption, Print, Security, Tabbing, Web Page Dialog, Widget Toolkit, XHTML

TABLE A.4: iMacros: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Recording
Editor Features	
Element Identification	Relational
Execution	Ant, Batch file, Command line, Distributed Playback, Email, Execute Excel, External Execution, Java, Test Report, URL Calls, XSL Transformation
Extra Tools	Application, Documentation Tool, Interactive Console, Object Spy, Test Creation Helper
Language	Error Handling, GPL-Integration, GPL-Integration Java, Scripting Language, Self-made, Version Controllable
Methodology	
Problem Analysis	Debugger, Run Log
Technology	401, Authentication, Continuous Integration, Dialog, Flash/Flex, HTTPS, Java applet, Jenkins, NTLM, PDF files, Widget Toolkit

Table A.5: Sahi Pro: Feature list

Term	Description
Abstraction Supporting Features	Multilevel, Name Collision Handling, Setup / Tear down, User Library
Capture	Recording
Editor Features	Keyword/Tabular View
Element Identification	
Execution	API, Command line, External Execution, Java API, Partly re-execution, Python API, Shuffle, Test Report
Extra Tools	Documentation Tool, Test
Language	Behavior-driven, Break / Continue, Coding Styles, Condition, Keyword / Tabular, Keyword-driven, Loop, Parallelism, Procedure, Return Value, Self-made, Variable, Version Controllable
Methodology	Data-driven Testing
Problem Analysis	Debugger, Run Log
Technology	

Table A.6: Robot Framework: Feature list

Term	Description
Abstraction Supporting Features	Setup / Tear down
Capture	Recording
Editor Features	Text View, Tree View
Element Identification	Attribute, XPath
Execution	Command line, External Execution, Test Report
Extra Tools	Object Spy, Test Creation Helper, Test Manager
Language	GPL, Java
Methodology	CSV, Data-driven Testing, Database, Excel
Problem Analysis	Debugger
Technology	Flash/Flex, Oracle, Oracle Test Manager, TM / ALM, Widget Toolkit

TABLE A.7: Application Testing Suite: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Image Recognition, OCR, Recording
Editor Features	
Element Identification	
Execution	Scheduler
Extra Tools	Object Spy, Test Creation Helper
Language	Array, Condition, Loop, Procedure, Return Value, Scripting Language, Self-made, Variable, Version Controllable
Methodology	
Problem Analysis	Run Log
Technology	PDF files

TABLE A.8: WinTask: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Image Recognition, Image Validation, Recording, Screenshot
Editor Features	Flow Chart View
Element Identification	
Execution	Email, Scheduler, Test Report, Testing Grid
Extra Tools	Object Spy, Test Creation Helper
Language	Break / Continue, Condition, Keyword / Tabular, Loop, Parallelism, Procedure, Self-made, Variable
Methodology	Data-driven Testing
Problem Analysis	Run Log, Screenshot on Error
Technology	TM / ALM

TABLE A.9: TestingWhiz: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Low-level Action, Recording
Editor Features	
Element Identification	
Execution	
Extra Tools	
Language	.NET, C#, GPL, Visual Basic
Methodology	CSV, Data-driven Testing, Manual Testing
Problem Analysis	Run Log
Technology	Audio Control, HTML5, Progress Bar, Slider Control, Video Control

Table A.10: CodedUI: Feature list

Term	Description
Abstraction Supporting Features	Component Model, Simulated Object
Capture	Recording
Editor Features	
Element Identification	Attribute, DOM Hierarchy, Mixed Strategy, Position / Size
Execution	Command line, Distributed Playback, External Execution, Test Report
Extra Tools	Object Spy, Test Creation Helper
Language	GPL, JavaScript
Methodology	Data-driven Testing, Database, Excel
Problem Analysis	Debugger
Technology	ActiveX, Flash/Flex, HTML5, Silverlight, Widget Toolkit

Table A.11: Rapise: Feature list

Term	Description
Abstraction Supporting Features	Avatar System, Component Model
Capture	Full Screenshot, Image Click, Image Recognition, Image Validation, OCR, Recording, Screenshot
Editor Features	Bulk Edit, Focus-View Filter, Keyword/Tabular View, Text View
Element Identification	Attribute, Image, Index
Execution	Autologin, Batch file, Exe-file, External Execution, Scheduler
Extra Tools	
Language	Condition, Keyword / Tabular, Loop, Self-made, Variable
Methodology	
Problem Analysis	Debugger, Email Notification, Run Log, Variable Watch Table
Technology	Flash/Flex, Keystroke Encryption, Silverlight, Widget Toolkit

Table A.12: Testing Anywhere: Feature list

Term	Description
Abstraction Supporting Features	Mapping Table
Capture	Image Click, Image Recognition, Recording
Editor Features	
Element Identification	Image, Weighted XPath, XPath
Execution	Command line, Exe-file, External Execution, Test Report
Extra Tools	Object Spy, Test Creation Helper
Language	GPL-Integration, Keyword / Tabular, Self-made, Variable
Methodology	CSV, Cross-Browser Testing, Data-driven Testing, Database, Excel, In-Editor Table
Problem Analysis	
Technology	Bamboo, Continuous Integration, Flash/Flex, HP Quality Center, Jenkins, Microsoft Test Manager, Silverlight, Source Control, Subversion, TM / ALM, Team Foundation Server, TeamCity

TABLE A.13: Ranorex Test Automation: Feature list

Term	Description
Abstraction Supporting Features	Mapping Table
Capture	Image Click, Image Recognition, Recording
Editor Features	
Element Identification	Image, Position / Size, XPath
Execution	Ant, Command line, External Execution
Extra Tools	Object Spy, Test Creation Helper
Language	.NET, C#, GPL, Java, Visual Basic
Methodology	
Problem Analysis	Log Explorer, Run Log
Technology	ActiveX, Flash/Flex, Silverlight, Widget Toolkit

TABLE A.14: Silk Test: Feature list

Term	Description
Abstraction Supporting Features	Guided Identifier Update, Mapping Table, Updating Components
Capture	Element context menu, Freeze Mode, Image Recognition, Image Validation, Recording
Editor Features	Storyboard View
Element Identification	Regular Expression, XPath
Execution	Distributed Playback, Result Timeline, Scheduler, Test Report
Extra Tools	3D Viewer, DOM Explorer, Test Creation Helper
Language	Condition, GPL-Integration, Scripting Language, Self-made
Methodology	CSV, Data-driven Testing, Database, Excel, Exploratory Testing, In-Editor Table, Manual Testing, Semi-automatic testing, XML
Problem Analysis	Debugger, Run Log, Save State Snapshot
Technology	Alert, Confirm, Dialog, Download, Leave Page, Login, Silverlight, Source Control, Team Foundation Server, Translator, Upload, Widget Toolkit

Table A.15: Test Studio: Feature list

Term	Description
Abstraction Supporting Features	
Capture	Image Click, Image Recognition, Image Validation, Recording, Retrieve Position
Editor Features	
Element Identification	Image, Index, Regular Expression
Execution	Ant, Command line, External Execution, Test Report
Extra Tools	
Language	Scripting Language, Self-made, Version Controllable
Methodology	CSV, Data-driven Testing
Problem Analysis	Debugger, Edit and Continue, Run Log, Screen Video
Technology	Continuous Integration, Dialog, Download, Flash/Flex, Hudson, Jenkins, Open / Close, Silverlight, Upload, Widget Toolkit

Table A.16: RIATest: Feature list

Term	Description
Abstraction Supporting Features	Element Recognition, Mapping Table, Probabilistic
Capture	
Editor Features	Metrics, Teststyle Guidelines
Element Identification	Attribute, DOM Hierarchy, Index
Execution	Command line, External Execution
Extra Tools	Web-based Test Monitor
Language	Error Handling, Keyword / Tabular, Procedure, Self-made, Variable
Methodology	Manual Testing, Multi-lingual testing, Test-first Programming
Problem Analysis	
Technology	BIRT, Bugzilla, Chronon, HP Quality Center, JaCoCo, JavaFX, Jira, TM / ALM, Widget Toolkit

TABLE A.17: Jubula: Feature list

Term	Description
Abstraction Supporting Features	
Execution and Problem Analysis	Proxy Server, Testing Grid
Element Identification	Attribute, CSS, CSS Selector, JavaScript Function, XPath
Language	C#, Java, JavaScript, PHP, Perl, Python, Ruby, Screenshot
Technology	Alert, Change User Agent, Confirm, Cookies, Dialog, Drag And Drop, Frames, Headless Driver, History, HtmlUnit, Phantomjs, Prompt, Tabbing

Table A.18: Selenium: Feature list

Term	Description
Abstraction Supporting Features	
Execution and Problem Analysis	
Element Identification	Host-language Procedure, Index, JQuery
Language	Java, JavaScript
Technology	Ajax, Wait on Procedure

TABLE A.19: FuncUnit: Feature list

Term	Description
Abstraction Supporting Features	Naming Collision Handling, Page Object, Step Object
Execution and Problem Analysis	Cleanup between Tests, Coding, Database, Debugger, Depends Annotation, Interactive Console, Multi User Testing, Multi-Browser testing, Screenshot, State Shapshot, Test Report, Testing Grid
Element Identification	Attribute, CSS, CSS Selector, XPath
Language	Conditional Assertion, PHP, Self-documenting
Technology	Code Coverage, Cookies, Guzzle, Headless Driver, Symfony BrowserKit

TABLE A.20: Codeception: Feature list

Term	Description
Abstraction Supporting Features	Inheritance, Lifecycle Hook, Page Object, Step Object, Template Option, Unexpected Pages
Execution and Problem Analysis	Cleanup between Tests, Cookie, Cucumber, JUnit, Listener, Screenshot, Spock, State Snapshot, Test Report, TestNG, Testing Framework
Element Identification	Attribute, CSS, CSS Selector, Find / Filter, JQuery, Parameter, Regular Expression, Traversing
Language	Complex Interaction, Direct Download, Groovy, Implicit Assertion, Push Non-Character, execute JavaScript
Technology	BrowserStack, Build System, Cloud Browser Testing, Frames, Gradle, Grails, Headless Driver, HtmlUnit, IDE Integration, IntelliJ, Maven, Open / Close, SauceLabs, Tabbing

TABLE A.21: GEB: Feature list

Term	Description
Abstraction Supporting Features	Page Object, Template Option
Execution and Problem Analysis	JUnit, TestNG, Testing Framework
Element Identification	CSS, CSS Selector, Filter Method
Language	Fest-assert, Framework, Hamcrest, JUnit Assertions, Screenshot, execute JavaScript
Technology	Ajax, Alert, Build System, Confirm, Dialog, Maven, Maximize, Prompt

TABLE A.22: FluentLenium: Feature list

Term	Description
Abstraction Supporting Features	Page Object, Step Object, Webelement pattern
Execution and Problem Analysis	Browser Extensions, JUnit, Multi User Testing, Reusable Session, TestNG, Testing Framework
Element Identification	Attribute, CSS, CSS Selector, JQuery
Language	Java, execute JavaScript
Technology	Ajax, Build System, Eclipse, IDE Integration, Maven, Request Guard, Staleness Aware Page Abstractions, Wait on Procedure

TABLE A.23: Arquillian Graphene: Feature list

Bibliography

[1] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374 – 1396, 2013. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2013.02.006. URL http://www.sciencedirect.com/science/article/pii/S0950584913000396.

[2] Anna Rita Fasolino, Domenico Amalfitano, and Porfirio Tramontana. Web application testing in fifteen years of WSE. In *15th IEEE International Symposium on Web Systems Evolution, WSE 2013, Eindhoven, The Netherlands, September 27, 2013*, pages 35–38. IEEE, 2013. ISBN 978-1-4799-1608-5. doi: 10.1109/WSE.2013.6642414. URL http://dx.doi.org/10.1109/WSE.2013.6642414.

[3] Arora A. and Sinha M. Web application testing: A review on techniques, tools and state of art, 2012.

[4] Arie van Deursen and Ali Mesbah. Research issues in the automated testing of ajax applications. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 2010. ISBN 978-3-642-11265-2. doi: 10.1007/978-3-642-11266-9_2. URL http://dx.doi.org/10.1007/978-3-642-11266-9_2.

[5] Kinga Dobolyi and Westley Weimer. Harnessing web-based application similarities to aid in regression testing. In *ISSRE 2009, 20th International Symposium on Software Reliability Engineering, Mysuru, Karnataka, India, 16-19 November 2009*, pages 71–80. IEEE Computer Society, 2009. ISBN 978-0-7695-3878-5. doi:

10.1109/ISSRE.2009.18. URL http://doi.ieeecomputersociety.org/10.1109/ISSRE.2009.18.

[6] Abbie Barbir, Chris Hobbs, Elisa Bertino, Frederick Hirsch, and Lorenzo Martino. Challenges of testing web services and security in SOA implementations. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 395–440. Springer, 2007. ISBN 978-3-540-72912-9. doi: 10.1007/978-3-540-72912-9_14. URL http://dx.doi.org/10.1007/978-3-540-72912-9_14.

[7] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996. ISSN 0098-5589. doi: 10.1109/32.536955. URL http://dx.doi.org/10.1109/32.536955.

[8] Beatriz Marín, Tanja E. J. Vos, Giovanni Giachetti, Arthur I. Baars, and Paolo Tonella. Towards testing future web applications. In *Proceedings of the Fifth IEEE International Conference on Research Challenges in Information Science, RCIS 2011, Gosier, Guadeloupe, France, 19-21 May, 2011*, pages 1–12. IEEE, 2011. ISBN 978-1-4244-8670-0. doi: 10.1109/RCIS.2011.6006859. URL http://dx.doi.org/10.1109/RCIS.2011.6006859.

[9] Koray Incki, Ismail Ari, and Hasan Sozer. A survey of software testing in the cloud. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, SERE-C '12, pages 18–23, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4743-5. doi: 10.1109/SERE-C.2012.32. URL http://dx.doi.org/10.1109/SERE-C.2012.32.

[10] V. Priyadharshini and A. Malathi. Survey on software testing techniques in cloud computing. *CoRR*, abs/1402.1925, 2014. URL http://arxiv.org/abs/1402.1925.

[11] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. Testing as a service (taas) on clouds. In *Seventh IEEE International Symposium on Service-Oriented System Engineering, SOSE 2013, San Francisco, CA, USA, March 25-28, 2013*, pages 212–223, 2013. doi: 10.1109/SOSE.2013.66. URL http://doi.ieeecomputersociety.org/10.1109/SOSE.2013.66.

[12] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, 48(12):1172–1186,

December 2006. ISSN 0950-5849. doi: 10.1016/j.infsof.2006.06.006. URL http://dx.doi.org/10.1016/j.infsof.2006.06.006.

[13] Web Content Accessibility Guidelines 2.0. http://www.w3.org/TR/WCAG20/, 2008. [accessed October 5, 2014].

[14] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

[15] Matjaz Pancur, Mojca Ciglaric, Matej Trampus, and Tone Vidmar. Comparison of frameworks and tools for test-driven development. In M. H. Hamza, editor, *The 21st IASTED International Multi-Conference on Applied Informatics (AI 2003), February 10-13, 2003, Innsbruck, Austria*, pages 980–985. IASTED/ACTA Press, 2003. ISBN 0-88986-345-8.

[16] Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz. Plagiarism Detection: A Tool Survey and Comparison. In Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 143–158, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-68-2. doi: http://dx.doi.org/10.4230/OASIcs.SLATE.2014.143. URL http://drops.dagstuhl.de/opus/volltexte/2014/4566.

[17] Shuang Wang and Jeff Offutt. Comparison of unit-level automated test generation tools. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pages 210–219. IEEE Computer Society, 2009. ISBN 978-0-7695-3671-2. doi: 10.1109/ICSTW.2009.36. URL http://dx.doi.org/10.1109/ICSTW.2009.36.

[18] Elder Macedo Rodrigues, Rodrigo S. Saad, Flávio M. de Oliveira, Leandro T. Costa, Maicon Bernardino, and Avelino F. Zorzo. Evaluating capture and replay and model-based performance testing tools: an empirical comparison. In Maurizio Morisio, Tore Dybå, and Marco Torchiano, editors, *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 9. ACM, 2014. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652587. URL http://doi.acm.org/10.1145/2652524.2652587.

[19] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 272–281. IEEE, 2013. doi: 10.1109/WCRE.2013.6671302. URL http://doi.ieeecomputersociety.org/10.1109/WCRE.2013.6671302.

[20] Mark Grechanik, Qing Xie, and Chen Fu. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 9–18. IEEE, 2009. doi: 10.1109/ICSM.2009.5306345. URL http://dx.doi.org/10.1109/ICSM.2009.5306345.

[21] Praveen Ranjan Srivastava and Mahesh Prasad Ray. Multi-attribute comparison of automated functional and regression testing tools using fuzzy AHP. In Bhanu Prasad, Pawan Lingras, and Ashwin Ram, editors, *Proceedings of the 4th Indian International Conference on Artificial Intelligence, IICAI 2009, Tumkur, Karnataka, India, December 16-18, 2009*, pages 1030–1043. IICAI, 2009. ISBN 978-0-9727412-7-9.

[22] Isabel John. *Pattern-based documentation analysis for software product lines.* PhD thesis, 2010.

[23] Software Product Lines — Overview. https://www.sei.cmu.edu/productlines/, 2014. [accessed January 31, 2015].

[24] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 68–77, Swinton, UK, UK, 2008. British Computer Society. URL http://dl.acm.org/citation.cfm?id=2227115.2227123.

[25] GUI Test Automation for Java and Web with QF-Test. http://www.qfs.de/en/, 2014. [accessed November 12, 2014].

[26] QF-Test - The Manual. http://www.qfs.de/en/qftest/manual.html, 2014. [accessed November 12, 2014].

[27] Seapine Software - QA Wizard Pro - Features and Benefits. http://www.seapine.com/qawizard.html, 2014. [accessed November 12, 2014].

[28] QA Wizard Pro UserGuide Version2014.1. http://downloads.seapine.com/pub/docs/qawpuserguid 2014. [accessed November 12, 2014].

[29] Web Testing, Load Testing, Java Testing, Server Monitoring. http://www.appperfect.com/, 2014. [accessed November 12, 2014].

[30] AppPerfect^{TM} Web Test v 13.0.0 User Guide. http://www.appperfect.com/support/docs/web-test/index.html, 2014. [accessed November 12, 2014].

[31] Browser Automation, Data Extraction and Web Testing — iMacros Software. http://imacros.net/, 2014. [accessed November 12, 2014].

[32] iMacros. http://wiki.imacros.net/Main_Page, 2014. [accessed November 12, 2014].

[33] Sahi - Web & Browser Automation Testing Tool . http://sahipro.com/, 2014. [accessed November 12, 2014].

[34] Sahi Pro - Introduction. http://sahipro.com/docs/introduction/index.html, 2014. [accessed November 12, 2014].

[35] Robot Framework. http://robotframework.org/, 2014. [accessed November 14, 2014].

[36] Robot Framework User Guide Version 2.8.5. http://robotframework.org/robotframework/#user-guide, 2014. [accessed November 14, 2014].

[37] Application Testing Suite. http://www.oracle.com/technetwork/oem/app-test/etest-101273.html, 2014. [accessed November 14, 2014].

[38] Oracle® Application Testing Suite Getting Started Guide Release 12.4.0.2 E15487-13 July 2014 . http://www.oracle.com/technetwork/oem/downloads/index-084446.html, 2014. [accessed November 14, 2014].

[39] Macro and Data Extraction with WinTask - the automation software for Windows and internet. http://www.wintask.com/index.php, 2014. [accessed November 14, 2014].

[40] WINTASK Develop efficient and reliable Web automation scripts Version 5.1. http://www.wintask.com/wintaskwebbook.pdf, 2014. [accessed November 14, 2014].

[41] Test Automation Tool for Regression, Cross browser and Database. http://www.testing-whiz.com/, 2014. [accessed November 14, 2014].

[42] TestingWhiz User Manual. http://www.testing-whiz.com/documentation, 2014. [accessed November 14, 2014].

[43] Improving Quality with Visual Studio Diagnostic Tools. http://msdn.microsoft.com/en-us/library/dd264943.aspx, 2014. [accessed November 14, 2014].

[44] Verifying Code by Using UI Automation. http://msdn.microsoft.com/en-us/library/dd286726, 2014. [accessed November 14, 2014].

[45] Buy Automated Testing Tools for API & GUI Testing — Inflectra. http://www.inflectra.com/Rapise/, 2014. [accessed November 14, 2014].

[46] Rapise — Online Help Viewer. http://www.inflectra.com/Rapise/HelpViewer.aspx?filename=Rapi
2014. [accessed November 14, 2014].

[47] Automation Testing Tools & Software — Testing Anywhere. https://www.automationanywhere.com/testing/, 2014. [accessed November 14, 2014].

[48] Testing Anywhere Client. https://www.automationanywhere.com/testing/images/manuals/testing
anywhere-usermanual.zip, 2014. [accessed November 14, 2014].

[49] Automated Testing Software — Ranorex - Test Automation. http://www.ranorex.com/, 2014. [accessed November 14, 2014].

[50] Ranorex Test Automation Guide. http://www.ranorex.com/Documentation/Ranorex-Tutorial.pdf, 2014. [accessed November 14, 2014].

[51] Borland Silk Test. http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test, 2014. [accessed November 14, 2014].

[52] Silk Test 15.5 Silk4J User Guide. http://supportline.microfocus.com/Documentation/books/ASQ/
155-help-en.pdf, 2014. [accessed November 14, 2014].

[53] Silk Test 15.5 Silk4NET User Guide. http://supportline.microfocus.com/Documentation/books/AS155-help-en.pdf, 2014. [accessed November 14, 2014].

[54] Software Testing Tools, Automated Testing Software — Telerik. http://www.telerik.com/teststudio, 2014. [accessed November 14, 2014].

[55] Test Studio Overview. http://docs.telerik.com/teststudio/, 2014. [accessed November 14, 2014].

[56] RIATest - Web application automation tool. http://www.cogitek.com/riatest/, 2014. [accessed November 14, 2014].

[57] RIATest User Guide - Cogitek RIATest 6 Documentation. http://www.cogitek.com/riatest/documentation/online.html?v6/RIATest/RIATest.html, 2014. [accessed November 14, 2014].

[58] Welcome to the BREDEX testing resources portal. http://testing.bredex.de/, 2014. [accessed November 14, 2014].

[59] Software and documentation downloads - Home. http://testing.bredex.de/sw-doku-downloads.html, 2014. [accessed November 14, 2014].

[60] Selenium - Web Browser Automation. http://www.seleniumhq.org/, 2014. [accessed December 6, 2014].

[61] Selenium Documentation. http://docs.seleniumhq.org/docs/, 2014. [accessed December 6, 2014].

[62] FuncUnit. http://funcunit.com/, 2014. [accessed December 6, 2014].

[63] Getting Started - FuncUnit Guides. http://funcunit.com/guides/started.html, 2014. [accessed December 6, 2014].

[64] Codeception - BDD-style PHP testing. http://codeception.com/, 2014. [accessed December 6, 2014].

[65] Introduction - Codeception - Documentation. http://codeception.com/docs/01-Introduction, 2014. [accessed December 6, 2014].

[66] Geb - Very Groovy Browser Automation. http://www.gebish.org/, 2014. [accessed December 6, 2014].

[67] The Book Of Geb - Table of Contents - 0.10.0. http://www.gebish.org/manual/current/, 2014. [accessed December 6, 2014].

[68] FluentLenium/FluentLenium · GitHub. https://github.com/FluentLenium/FluentLenium, 2014. [accessed December 6, 2014].

[69] Graphene · Arquillian. http://arquillian.org/modules/graphene-extension/, 2014. [accessed December 6, 2014].

[70] Home - Graphene 2 - Project Documentation Editor. https://docs.jboss.org/author/display/ARQGRA2/Home?_sscc=t, 2014. [accessed December 6, 2014].

[71] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, pages 191–200, Los Alamitos, CA, 8 2011. IEEE Computer Society.

[72] Dont Repeat Yourself. http://c2.com/cgi/wiki?DontRepeatYourself, 2014. [accessed January 31, 2015].

[73] PageObjects - selenium - The Page Object pattern represents the screens of your web app as a series of objects - Browser automation framework - Google Project Hosting. https://code.google.com/p/selenium/wiki/PageObjects, 2014. [accessed January 31, 2015].

[74] XHTML 2 Working Group Expected to Stop Work End of 2009, W3C to Increase Resources on HTML 5. http://www.w3.org/News/2009#entry-6601, 2009. [accessed December 29, 2014].