

FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

Pattern-Based GUI Testing



U.PORTO

Rodrigo Manuel Lopes de Matos Moreira

December 2014

Supervised by

Doctor Ana Cristina Ramada Paiva, FEUP

Doctor Ademar Manuel Teixeira de Aguiar, FEUP

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Engineering by the
Doctoral Programme in Informatics Engineering

Contact Information

Author: Rodrigo Manuel Lopes de Matos Moreira

Publishes as: Rodrigo M. L. M. Moreira

Email: rodrigommoreira@gmail.com

Social Network: pt.linkedin.com/in/rodrigomoreira/



Rodrigo Manuel Lopes de Matos Moreira

Pattern-Based GUI Testing

**Copyright © 2014 by Rodrigo Manuel Lopes de Matos Moreira.
All rights reserved.**

This page was intentionally left mostly blank.

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.



To my parents Mário and Maria Clara

This page was intentionally left mostly blank.

“That’s been one of my mantras – focus and simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it’s worth it in the end because once you get there, you can move mountains.”

Steve Jobs

This page was intentionally left mostly blank.

Abstract

Nowadays, we have noticed the importance of Graphical User Interfaces (GUIs), their impact and dependency on our daily lives. GUIs have become an ideal way of interacting with computer programs, making the software friendlier for its users by offering flexibility in how users perform tasks. The amount of events users are able to perform are not restricted. This flexibility that makes GUIs easy to use also makes testing them for functional correctness notoriously more difficult. Despite all of the advances in automated testing tools and frameworks over the last decade, manual testing still represents the majority of testing effort within most software development organizations. GUI testing in practice remains a huge resource intensive task.

Software systems with a GUI front-end are typically designed and implemented using user interface (UI) patterns. UI patterns describe generic solutions (with multiple possible implementations) for recurrent GUI design problems. Existing testing techniques do not take advantage of this fact, which would allow to test GUIs more efficiently. Testing of GUIs for functional correctness has largely ignored UI patterns.

This dissertation addresses the problem on how to benefit from GUIs' recurrent behavior in order to ensure GUIs functional correctness. The goal is to use test patterns that allow to test the variety of UI patterns implementations, by providing a configurable testing strategy and thus promoting reuse. Therefore, this dissertation presents a Pattern-Based Graphical User Interface (GUI) Testing method (PBGT) for systematizing and automating the GUI testing process, by sampling the input space using *UI Test Patterns* that express generic solutions to test common recurrent GUIs' behavior.

The PBGT methodology is supported by a graphical domain-specific language (DSL) called PARADIGM. This language aims to diminish GUIs' modeling effort and enable reuse of testing strategies. The DSL was built from scratch according to guidelines and best practices. An iterative novel methodology using Alloy is also documented, to find and tune language constraints. Besides UI Test

Patterns, PARADIGM has other elements and connectors that allow building models describing test objectives. Test cases are generated from these models and afterwards are executed over a GUI.

Seven UI Test Patterns are described, according to best practices and guidelines from the field of Pattern Languages, with the goal to concentrate the information necessary to start using the test patterns, when to use them, how to use them, why they should be used and finally what they should be used for. It is intended to provide guidance and to share knowledge for testers and software engineers that want to use the PBGT approach and, therefore, starting benefiting from it in short time.

The PBGT approach is validated via three case studies. The first study had the goal to assess the PBGT general approach, using two fielded web application subjects, in order to ascertain whether it can be used by testers who have not been involved in the development of the approach, and to measure the effort required to obtain benefits from the approach. Findings demonstrate that PBGT approach is both practical and useful as testing teams were able to find real bugs in a reasonable time. The second study provided a deeper analysis by studying software subjects seeded with artificial faults. The findings show that PBGT is more effective than the current state-of-the-art record/playback (capture/replay) techniques. Finally, the third study was designed to compare the modeling efforts of PARADIGM with other modeling approaches, such as Spec# and VAN4GUIM. The results showed that in an universe of 6 subject systems, PARADIGM took less time than VAN4GUIM and even less than Spec#, in terms of crafting and configuring models. Further the study also evaluated the language usefulness, graphical power, and acceptability. Results indicate that PARADIGM language elements are expressive and easy to identify.

The PBGT approach emerges as a new model-based GUI testing approach, and contributes to the improvement of current GUI testing approaches. Although the usage and the concept of patterns is not new, to the best of the author's knowledge, no approach has tried to apply them to GUI testing, until now. This approach also targets to achieve reuse of GUI testing strategies through the usage of test patterns.

Resumo

Hoje em dia, é possível apercebermo-nos da importância das interfaces gráficas com o utilizador, do seu impacto e da sua dependência nas nossas vidas. Elas tornaram-se na forma ideal de interagir com o *software*, tornando-o mais amigável para os utilizadores e garantindo flexibilidade na forma de executar tarefas. Por um lado, esta flexibilidade facilita a interacção com o *software*, mas por outro, dificulta o teste funcional das interfaces gráficas com o utilizador. Apesar de todos os avanços relativamente à automação de ferramentas de teste, muitos testes ainda são efectuados de forma manual. Na prática, os testes de interfaces gráficas com o utilizador continuam a ser uma tarefa complexa e árdua.

Tipicamente as aplicações de *software* que disponibilizam uma interface gráfica com o utilizador, são desenhados e implementados com recurso a padrões de interfaces gráficas. Estes descrevem soluções genéricas (com várias implementações) com o intuito de resolver problemas de desenho recorrentes. As técnicas de teste actuais não tiram partido deste facto, o que possibilitaria testar as interfaces gráficas com o utilizador de forma mais eficiente. O teste funcional de interfaces gráficas com o utilizador tem ignorado os padrões de interfaces gráficas.

Esta dissertação aborda o problema sobre como se beneficiar do comportamento recorrente das interfaces gráficas com o utilizador, com o intuito de assegurar o correcto funcionamento da interface gráfica. O objectivo consiste em utilizar padrões de teste que possibilitem testar uma diversidade de implementações de padrões de interfaces gráficas, fornecendo para isso uma estratégia de teste configurável, e promovendo a reutilização. Neste sentido, esta dissertação apresenta um método (“PBGT”) baseado em padrões, para sistematizar e automatizar o processo de teste, com recurso a padrões de teste de interfaces com o utilizador, que representam soluções genéricas para testar comportamento recorrente presente nas interfaces gráficas com o utilizador.

A metodologia proposta (“PBGT”) é suportada por uma linguagem de domínio específico (“DSL”) denominada “PARADIGM”. Esta linguagem tem como objectivo reduzir o esforço de modelação e promover a reutilização de estratégias de teste.

Esta linguagem foi desenvolvida de raiz, e seguiu um conjunto de boas práticas no seu desenvolvimento. É também apresentada uma abordagem inovadora utilizando uma linguagem de especificação formal “Alloy” para descobrir e afinar as restrições da linguagem “PARADIGM”. Para além de padrões de teste de interfaces gráficas, a linguagem “PARADIGM” é constituída por outros elementos e conectores, que possibilitam a construção de modelos descrevendo objectivos de teste. Os casos de teste são gerados a partir dos modelos para depois serem executados numa interface gráfica com o utilizador.

Sete padrões de teste de interfaces gráficas são descritos de acordo com boas práticas na área de linguagens de padrões, com o objectivo de documentar padrões de teste, indicando quando estes devem ser utilizados, como devem ser utilizados e, finalmente, para que devem ser utilizados.

A abordagem “PBGT” é validada através de três casos de estudo. O primeiro estudo teve como objectivo avaliar a abordagem utilizando para teste duas aplicações *web*, no sentido de se determinar se abordagem pode ser utilizada por *testers* que não estiveram envolvidos no desenvolvimento da abordagem e, para medir o esforço necessário para se conseguir obter benefícios da abordagem. Os resultados do estudo indicam que a abordagem é exequível e vantajosa, uma vez que as equipas de teste conseguiram encontrar *bugs* em tempo útil. O segundo estudo permitiu uma análise mais profunda, verificando aplicações que continham falhas artificiais. Os resultados indicam que a abordagem “PBGT” é mais eficiente do que técnicas de teste “capture/replay”. Por fim, o terceiro estudo pretendeu comparar o esforço de modelação da linguagem “PARADIGM” com outras abordagens, como “Spec#” e “VAN4GUIM”. De um universo de seis aplicações, “PARADIGM” demonstrou demorar menos tempo do que as abordagens restantes, em termos de criação e configuração de modelos. Este estudo também avaliou a utilidade da linguagem, poder gráfico e aceitação, em que os elementos da linguagem demonstraram ser expressivos e fáceis de identificar.

A abordagem “PBGT” surge como uma nova abordagem de testes de interfaces gráficas com o utilizador baseados em modelos e, contribui para a melhoria das abordagens existentes na área de testes de interfaces gráficas com o utilizador. Até à data, apesar de o conceito de padrões não ser novo, e segundo é do conhecimento do autor, nenhuma abordagem tentou aplicar este conceito no âmbito de testes de interfaces gráficas com o utilizador. Esta abordagem pretende garantir a reutilização de estratégias de teste com recurso a padrões de teste.

Glossary

ASE	Automated Software Engineering
CLI	Command-Line Interface
DSL	Domain-Specific Language
EFG	Event Flow Graph
EMF	Eclipse Modeling Framework
ENASE	International Conference on Evaluation of Novel Approaches to Software Engineering
EuroPLOP	European Conference on Pattern Languages of Programs
FSM	Finite State Machines
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPL	General-Purpose Language
GPS	Global Positioning System
GUI	Graphical User Interface
GuiTam	GUI Test Automation Model
GUITAR	GUI Ripping Tool
HPrTN	Hierarchical Predicate Transitions Nets
HTML	HyperText Markup Language
IDE	Integrated Development Environment
iOS	iPhone Operating System
ISSRE	International Symposium on Software Reliability Engineering
LSTS	Labeled State Transition Systems
Mac OS	Macintosh Operating System
MAIDEN	Methodology for the usage of Alloy In DSL Engineering
MBGT	Model-Based Graphical User Interface Testing
MBT	Model-Based Testing
MIT	Massachusetts Institute of Technology

MPS	Meta Programming System
MTI	Motion Tracking Interface
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
NUI	Natural User Interface
OCL	Object Constraint Language
PARADIGM-ME	PARADIGM Modeling Environment component
PARADIGM-RE	PARADIGM Reverse Engineering component
PARADIGM-TE	PARADIGM Test case Execution component
PARADIGM-TG	PARADIGM automated Test case Generation component
PARADIGM	Domain-Specific Language built for Pattern-Based Graphical User Interface Testing
PBGT	Pattern-Based Graphical User Interface Testing
POSA	Pattern-Oriented Software Architecture
SDF	Syntax Definition Formalism
STVR	Software Testing, Verification and Reliability
SUT	System Under Test
TUI	Touch User Interface
UI	User Interface
UML	Unified Modeling Language
V&V	Verification and Validation
VAN4GUIM	Visual Abstract Notation for GUI Modeling and Testing
Web	World Wide Web
WIMP	Windows Icons Menus and Pointers
WUI	Web-based User Interface
WYSIWYG	What You See Is What You Get
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

Table of Contents

Abstract	i
Glossary	v
Acknowledgments	xvii
1 Introduction	1
1.1 Challenges	2
1.2 Research Goals	3
1.3 Research Strategy	3
1.4 Contributions	5
1.5 Results	5
1.6 Dissertation Outline	6
1.6.1 How to Read this Dissertation	7
1.7 Publications	9
2 UI Fundamentals	11
2.1 Types of UI	12
2.1.1 Command Line Interfaces	12
2.1.2 Graphical User Interfaces	13
2.1.3 Natural User Interfaces	14
2.2 UI Qualities	15
2.3 Typical Defects	17
2.4 UI Design and Development Tools	17
2.5 UI Patterns	19
2.5.1 High-Level Patterns	19
2.5.2 Low-Level Patterns	23
2.6 Discussion	26
2.7 Summary	26
3 Graphical User Interface Testing	29

3.1	GUI Testing Challenges	30
3.2	Manual GUI Testing	31
3.3	Automated GUI Testing	31
3.4	GUI Testing Approaches	33
3.4.1	Capture-Replay	33
3.4.2	Random Testing	34
3.4.3	Unit Testing	34
3.4.4	Model-Based Testing	35
3.5	Discussion	36
3.6	Summary	37
4	Research Problem	39
4.1	Research Issues	39
4.2	Research Focus	40
4.3	Thesis Statement	41
4.4	Research Goals	42
4.4.1	Primary Goal	43
4.4.2	Secondary Goals	43
4.5	Validation Methodology	44
4.6	Summary	44
5	Pattern-Based GUI Testing	45
5.1	Overview	46
5.2	Methodology	50
5.2.1	Components	50
5.2.2	Key Principles	52
5.2.3	Process	53
5.2.4	Data Input	54
5.2.5	Levels of Reuse	55
5.3	Tool Support	56
5.4	Application Test Model	57
5.5	Test Case Generation	58
5.6	Summary	61
6	Pattern Library	63
6.1	Pattern Format	64
6.2	Considerations	65
6.3	Input UI Test Pattern	66

6.4	Login UI Test Pattern	69
6.5	Master/Detail UI Test Pattern	71
6.6	Find UI Test Pattern	74
6.7	Sort UI Test Pattern	76
6.8	Call UI Test Pattern	79
6.9	Option Set UI Test Pattern	81
6.10	Summary	83
7	DSL Engineering for GUI Modeling	85
7.1	Domain Specific Language Engineering	86
7.1.1	Benefits	86
7.1.2	Challenges	87
7.2	Tools for Language Implementation	87
7.2.1	Microsoft Domain-Specific Language Tools	88
7.2.2	Eclipse Platform	88
7.2.3	StarUML	88
7.2.4	Open ModelSphere	89
7.2.5	JetBrains MPS	89
7.2.6	SDF/Stratego/Spoofax	89
7.2.7	xText	89
7.3	DSL Engineering Process	90
7.4	Alloy in DSL Engineering	91
7.4.1	Introducing Alloy	92
7.4.2	Proposed Methodology	92
7.5	Development of a GUI Specification Language	94
7.5.1	Language Goals	94
7.5.2	Language Core Model	95
7.5.3	Behavior Specification	102
7.5.4	Concrete Syntax	103
7.5.5	Integration	103
7.5.6	How to Attain Reuse and Evolution	104
7.6	Summary	107
8	Validation	109
8.1	Case Study 1: Assessing PBGT failure detection capability	110
8.1.1	Metrics	110
8.1.2	Subject Systems	110
8.1.3	Testing Goals	111

8.1.4	Setup	112
8.1.5	Procedure	112
8.1.6	Results	113
8.1.7	Findings	119
8.1.8	Threats to validity	120
8.2	Case Study 2: Comparison between PBGT and Capture-Replay . . .	120
8.2.1	Metrics	121
8.2.2	Subject Systems	121
8.2.3	Testing Goals	122
8.2.4	Setup	123
8.2.5	Procedure	124
8.2.6	Results	125
8.2.7	Findings	126
8.2.8	Threats to Validity	127
8.3	Case Study 3: Analysis of Modeling Effort	128
8.3.1	Metrics	129
8.3.2	Subject Systems	129
8.3.3	Setup	130
8.3.4	Procedure	130
8.3.5	Results and Findings	131
8.3.6	Threats to Validity	132
8.4	Discussion	133
8.5	Summary	138
9	Conclusions	139
9.1	Summary of Contributions	140
9.2	Future Research	141
	Bibliography	142

List of Figures

Figure 1.1: Structure of the Dissertation.	8
Figure 2.1: Overview on existing types of user interface.	12
Figure 2.2: Model-View-Controller (MVC) Pattern.	20
Figure 2.3: Model-View-Presenter (MVP) Passive View Pattern.	21
Figure 2.4: Model-View-Presenter (MVP) Supervising Controller Pattern.	22
Figure 2.5: Model View ViewModel (MVVM) Pattern.	22
Figure 2.6: Input prompt example (source from Twitter [DWSG14] welcome page).	23
Figure 2.7: Calendar Picker example (source from [Gro14a]).	23
Figure 2.8: Account Registration example (source from [Nin14a]). . .	24
Figure 2.9: Breadcrumbs example (source from [Gro14b]).	24
Figure 2.10: Auto-complete example (source from [Int14]).	25
Figure 2.11: Sort example (source from [Ama14]).	25
Figure 4.1: Research focus of this research work, identifying the re- search domains and the contributions of this work directed towards a new GUI testing approach.	41
Figure 5.1: iTunes (top) and Finder (bottom) examples (sources from <i>www.apple.com</i>).	47
Figure 5.2: Different implementations and outcomes of the Login UI Pattern.	49
Figure 5.3: Pattern-Based GUI Testing Components.	51
Figure 5.4: Pattern-Based GUI Testing Process.	53
Figure 5.5: PBGT Tool featuring the PARADIGM-ME component.	56
Figure 5.6: PARADIGM model of a simple web-application.	57
Figure 5.7: Configurations for the Login UI Test Pattern (number 1) from Figure 5.6.	58
Figure 5.8: Description of a test case in XML format.	61

Figure 6.1: UI Part featuring the input functionality from Stattrek – Online Statistical Table [Sta14].	68
Figure 6.2: UI Part featuring the authentication functionality from Facebook[Fac14]	70
Figure 6.3: UI Part of Mobile.de[mob14] that features the Master/Detail area.	73
Figure 6.4: UI Part of GPUupdate search news functionality[GPU14]. . .	75
Figure 6.5: UI Part of Telerik[Tel14] that features a sort functionality. .	78
Figure 6.6: UI Part of Mobile.de [mob14] that features an action control.	81
Figure 6.7: UI Part of Mobile.de [mob14] that features a control with multiple options.	83
Figure 7.1: Language model-driven DSL engineering process (source from [SZ09]).	90
Figure 7.2: A Methodology for the usage of Alloy in DSL Engineering (MAIDEN).	93
Figure 7.3: PARADIGM Language metamodel.	95
Figure 7.4: PARADIGM Alloy Model – First iteration.	97
Figure 7.5: Generated instance of the model from Figure 7.4.	97
Figure 7.6: New added constraints to the PARADIGM Alloy Model (from Figure 7.4) – Second iteration.	98
Figure 7.7: Generated instance of the model from Figure 7.4 including the new statements from Figure 7.6.	98
Figure 7.8: New added constraints to the PARADIGM Alloy Model – Third iteration.	99
Figure 7.9: Generated instance of the model including the new statements from Figure 7.8.	100
Figure 7.10: New added constraints to the PARADIGM Alloy Model – Fourth iteration.	100
Figure 7.11: Generated instance of the model including the new statements from Figure 7.10.	101
Figure 7.12: New added constraints to the PARADIGM Alloy Model – Fifth iteration.	101
Figure 7.13: Generated instance of the model including the new statements from Figure 7.12.	102
Figure 7.14: PARADIGM concrete syntax.	103
Figure 7.15: High-Order UI Test Pattern – LIMIT.	105
Figure 7.16: LIMIT UI Test Pattern – Inner configuration details.	106

Figure 7.17: LIMIT UI Test Pattern – configuration for Login UI Test Pattern.	106
Figure 8.1: Australian-charts model (first level) written in PARADIGM.	113
Figure 8.2: Australian-charts featuring the modeling of the registration form.	115
Figure 8.3: Australian-charts featuring the modeling of the search form.	117
Figure 8.4: Tudu Lists model in PARADIGM.	125
Figure 8.5: TaskFreak GUI featuring project properties.	134
Figure 8.6: TaskFreak GUI featuring the details of a project.	134

List of Tables

Table 8.1: Defined testing goals for <i>mobile.de</i> (SS1.1).	111
Table 8.2: Defined testing goals for <i>australian-charts.com</i> (SS1.2).	112
Table 8.3: Configuration details for UI Test Patterns featured in the model from Figure 8.1.	115
Table 8.4: Configuration details for UI Test Patterns featured in registration form from Figure 8.2.	116
Table 8.5: Configuration details for UI Test Patterns featured in Search Form from Figure 8.3.	117
Table 8.6: Part of the generated test cases from the model from Figure 8.1.	118
Table 8.7: Case study findings concerning the modeling and configuration efforts and failures found.	119
Table 8.8: Defined testing goals for <i>Tudu Lists</i> (SS2.1).	122
Table 8.9: Defined testing goals for <i>Task Freak</i> (SS2.2).	123
Table 8.10: Defined testing goals for <i>iAddressBook</i> (SS2.3).	123
Table 8.11: Configuration details for the model from Figure 8.4.	126
Table 8.12: Efforts required to build and configure models/scripts for each subject system.	127
Table 8.13: Maximum number of mutants killed for each subject system.	127
Table 8.14: Number of mutants killed in each PBGT test strategy.	128
Table 8.15: Case study findings concerning the modeling and configuration efforts for all approaches.	132
Table 8.16: PARADIGM and VAN4GUIM findings concerning Moody's quality criteria, with input collected from the teams.	132

Acknowledgments

I would like to express my gratitude to my parents, Mário and Maria Clara, for their constant support, encouragement and caring. Thank you for standing by me through the many trials and decisions I have faced. I express a deep appreciation for their efforts to help me obtain, what I have achieved today.

I am also in gratitude to Siemens, specially to Eng. David Rocha (Technical Manager at Siemens Healthcare Portugal), Eng. João Seabra (General Manager at Siemens Healthcare Portugal; Division Cluster Lead, Imaging & Therapy – Southwest Europe and; Head of Healthcare Region SWE at Siemens Healthcare), and my colleagues, for all the support provided.

I want to thank to both my supervisors Professor Ana Paiva and Professor Ademar Aguiar for their mentoring, patience and flexibility throughout this PhD. I would like to express my appreciation for their guidance, continuous feedback, suggestions, support and invaluable assistance. I am also grateful to Professor Ana Paiva for all the support provided for the conferences.

My gratitude is extended to Professor Atif Memon, who I had the privilege to work with, for the guidance, focus and precise feedback. It was great and insightful to exchange ideas with him. I am grateful to Professor Eugénio Oliveira and Professor Augusto Sousa, for the support provided.

1

Introduction

In recent decades Graphical User Interfaces (GUIs) have been gaining importance and popularity. They assist us in our daily lives to facilitate tasks, and can be seen in several different contexts (smartphones, tablets, personal computers, televisions, tablets, GPSs, and so on). The graphical impact, ease of interaction (usability concerns) and the absence of defects, are crucial aspects to the end-users' acceptance of software systems that feature GUIs. GUIs have been growing in functionality and complexity. Therefore, it is increasingly necessary to test GUIs for their functional correctness.

User Interface (UI) patterns facilitate the design of today's software [BZ14a, BZ14b]. UI patterns embody frequently recurring solutions that solve common GUI design problems. For example, the *Sortable Table* pattern that is used to “*show the data in a table, and let the user sort the table rows according to the cell values in a selected column*” [Tid11] or the *Springboard* pattern, a UI pattern that can be seen on smartphones, is described as “*a landing screen with options that act as launch points into the application*” [Nei14]. There are several UI patterns that can be found on numerous software applications [Yah12, Pat14, Tox14, vW08, IDT08]. UI patterns can be implemented in different ways, and one example is the previously described *Springboard* pattern, that depending on the platform (iOS, Android and Windows Phone), has different implementations to fulfill the same goal. The

platform is not the single cause to contribute to having different implementation of UI patterns. The context of use, design and interactivity goals, and usability represent one possible set of concerns that lead to the demand of having such a wide-range of implementations. Despite these facts, UI patterns share one common property: they have recurrent behavior. This behavior is observed across their multiple implementations.

From a testing perspective, it is not beneficial and not practical to test all the different implementations of a given UI pattern. Instead, an approach is missing to allow to test these different implementations by focusing on the GUIs' recurrent behavior. Testing of GUIs for functional correctness has been disregarding UI patterns. This dissertation focus on the problem that existing testing techniques have not yet benefited from UI patterns recurring nature, having the intent to approximate the design and quality assurance of GUIs.

This chapter gives a general introduction to the main topics of this dissertation. It discusses the importance of GUIs. The chapter describes the main challenges related with GUI testing and then presents the goals of this research work, along with the strategy that was defined. Further, it describes the contributions and results of this research work, and enumerates the list of publications achieved during its course.

1.1 Challenges

Due to the broad usage of GUIs in numerous software systems, GUI testing has become an area of increasing importance to software development. GUI testing is often perceived as a challenging task. It is difficult, extremely time-consuming, and costly, with scarce tools and techniques available to assist the testing process.

End-users are able to perform tasks through GUIs (via events) in diverse ways. This constitutes a challenge since testers will have to decide to either check all sequence of events or only a subset. With automated software testing, it is possible to reduce the effort required to test GUIs. Nevertheless, despite the progress made in automated testing tools over the last decade, manual testing still represents the majority of testing efforts. Manual testing is a laborious activity, specifically error-prone and tedious. In addition, testers are required to be patient, perceptive, creative and skillful. As GUIs grow in functionality and, consequently become more complex, they make even further manual GUI testing a defiant task to comply.

However, there are several popular testing techniques, such as random testing

and capture/replay that can be applied to test GUIs. Some are only able to find certain type of errors, while others are not able to automate test case generation. Other techniques are more useful to perform regression testing, for instance, capture/replay. Other testing techniques are able to generate test cases from models. These are model-based testing techniques (MBT). However, most of the time, the testing techniques are not developed with reuse concerns as a way to diminish the testing effort when similar behavior is present in different GUIs or even within the same GUI under test. Testers start from scratch and do not have a mechanism to support reuse besides copy & paste.

1.2 Research Goals

The main goal of this research is to define an approach to improve current model-based GUI testing methods, by promoting reuse of GUI testing strategies and increasing models' abstraction, bringing closer the design and test phases, and thus ensuring the build of higher quality GUIs.

In addition, to support the main goal, two secondary goals have also been defined: (1) develop a set of patterns to facilitate the modeling and testing of UIs that feature UI patterns; and (2) develop a DSL to be used in the context of this new proposed approach.

All research goals are further detailed in Chapter 4 (p. 39).

1.3 Research Strategy

The research methodology for empirical software engineering has not fully matured [Sch05]. The most adequate research for Software Engineering still remains under discussion. Research methodologies can be categorized as scientific, engineering, empirical and analytical [ZW98]. In the scientific method, a theory is formulated in order to explain a phenomenon. An engineering method aims at formulating a hypothesis and tries to develop and test a proposed solution. An empirical method uses statistical methods to validate a given hypothesis. Finally, the analytical method develops a formal theory. These research methodologies can be applied to science in general, but for the software engineering domain, other specific approaches may be combined.

Zelkowitz et al [ZW98], introduce twelve different software engineering validation models. These models can be combined into three categories: (i)

observational methods, (ii) historical methods, and (iii) controlled methods. Observational methods collect relevant data during the development of the project. There are four types: *project monitoring*; *case study*; *assertion*; and *field study*. Historical methods gather and analyze data from projects that have already been completed. There are four methods: *literature search*; *legacy data*; *lessons learned*; and *static analysis*. Controlled methods are classical methods of experimental design used in other scientific disciplines and gather information from different instances of an observation for statistical validity of the results. There are four types: *replicated experiment*; *synthetic environment experiment*; *dynamic analysis*; and *simulation*.

The scientific area of this research work is Software Engineering. The strategy for this research work consisted of four phases:

- **Information gathering.** This was the starting phase of this research work. It was dedicated to collect, study, and synthesize information on the topics most relevant for the problem defined. The goals for this phase were: (1) to overview the main topics related to the research area and their open issues, and (2) narrow the directions of the research. The main important topics related with this research were: software patterns; software reuse; UI patterns gathering and evaluation; GUI modeling approaches; GUI testing approaches; Model-Based Testing; Model-Based GUI Testing approaches; and Domain-Specific Language Engineering. The methods used were historical methods as it is the case of literature search. The obtained results from this phase are included in Part I - State of the Art (Chapters 2 and 3).
- **Hypothesis definition.** Based on the information collected and studied, an hypothesis was formulated (Chapter 4, Section 4.3). The solution is based on a new approach for model-based GUI testing, called Pattern-Based GUI Testing.
- **Approach development.** After formulating the hypothesis, the approach was developed. The approach (Chapter 5) consisted in the specification of UI Test Patterns to promote reuse of testing strategies (Chapter 6); development of a domain-specific language to be used in the context of PBGT, called PARADIGM; development of a method to model and test GUIs using PARADIGM DSL (Chapter 7); and development of a tool to support the proposed methodology. The research methods used in this phase were observational ones.

- **Approach evaluation.** Once constructed the proposed solution for the identified problem, observational methods, like case studies, and controlled methods, such as replicated experiments, were conducted to validate the solution (Chapter 8). The results obtained during the research work were presented and discussed in international conferences after being approved in its reviewing processes.

1.4 Contributions

The main contributions of this dissertation are:

- **A new model-based GUI testing paradigm (PBGT) for GUI modeling and testing**, that aims to promote reuse of GUI testing strategies and benefit from GUIs recurrent behavior.
- **The design of an innovative testing process** – the PBGT process – that testers should follow during their testing activities.
- **A bridge to address the gap between the design and quality assurance of GUIs**, by leveraging UI patterns that have been used for GUI design, towards the notion of UI Test Patterns.
- **A collection of UI Test Patterns** with the purpose of providing guidance for practitioners that want to use the PBGT approach.
- **A domain-specific language (DSL)** developed specifically for PBGT, that was engineered according to a set of guidelines and best practices on the field, including a detailed description of process that was followed in its construction.
- **A novel approach using Alloy in DSL engineering**, describing a methodology to find and tune DSL language constraints using Alloy. A practical example is provided to support the proposed methodology.

1.5 Results

This dissertation presents the following results:

- A set of case studies to assess the modeling efforts required to craft models with PARADIGM, plus PBGT failure detection capability and a comparison between PBGT and Capture-Replay.

- A low maintenance and evolutionary language. PBGT provides the capability to extend the initial set of UI Test Patterns, to adjust current test strategies (or create new ones) to fulfill new UI trends. The provided set of Base UI Test Patterns, can be used to test a wide range of user interfaces.
- An easy to use and learn approach, since testers without any knowledge of PBGT approach are able to use and obtain the benefits from the approach, immediately after a 2-hour training session.
- An integrated testing environment that provides functionalities for modeling (either manually or automatically), configuration, automated test case generation, automated test case execution and test coverage analysis. The tool is able to model and test web and mobile based applications. Moreover, it does not require access to the source code of the system under test (SUT) to create or generate GUI models from them.
- The usage of the PBGT tool in an industrial environment.

1.6 Dissertation Outline

The remaining of this dissertation is logically organized into three parts, according to the following structure:

Part I: State of the Art. The first part provides background on different kinds of user interfaces and their typical failures, and performs a literature review on GUI testing. Additionally, it focus on several aspects concerning Specification-Based GUI Testing.

- Chapter 2, “UI Fundamentals” (p. 11), focuses on the development and quality of UIs. The chapter describes the different types of UIs, including the latest trends, tools to support the design and implementation of UIs, and desired qualities and common defects. It also introduces two strands of patterns for designing GUIs.
- Chapter 3, “Graphical User Interface Testing” (p. 29), discusses manual and automated GUI testing. Further presents a state of art review on the topic of GUI testing, by describing several approaches and identifying their flaws.

Part II: Problem and Solution. The second part identifies the problem and states the proposed approach. Thus, it presents: the PBGT approach; a collection of test

patterns to be used in the context of PBGT; and the process that was followed in the development of a domain-specific language, designed for PBGT.

- Chapter 4, “Research Problem” (p. 39), considers the state of the art, formulates the thesis statement, overviews the solution proposed, the methodology that was followed and draws the validation strategy.
- Chapter 5, “Pattern-Based GUI Testing” (p. 45), presents the solution proposed describing a new approach for Model-Based GUI Testing.
- Chapter 6, “Pattern Library” (p. 63), describes a set of UI Test Patterns to be used in the context of PBGT. This chapter intends to act as guidance on how to effectively use UI Test Patterns for GUI Testing, and to facilitate a common understanding on their usage.
- Chapter 7, “DSL Engineering for GUI Modeling” (p. 85), describes the process that was followed to develop a DSL to be used in the context of PBGT. The chapter also proposes a novel approach — a methodology for the usage of Alloy in DSL Engineering — in order to facilitate the finding and tuning of languages constraints.

Part III: Validation and Conclusions. The third, and last part, presents case studies, validates the thesis, and presents the conclusions of the dissertation.

- Chapter 8, “Validation” (p. 109), presents and discusses the findings and results of three case studies used to evaluate and validate the approach proposed.
- Chapter 9, “Conclusions” (p. 139), reviews the dissertation and points future work directions.

Figure 1.1 gives an overview of the chapters and their relations.

1.6.1 How to Read this Dissertation

For a comprehensive understanding of this dissertation, all the parts should be read in the same order as they are provided. Those who want go directly to the *core* of the approach proposed, may skip the first part and go directly to Chapter 5 (p. 45), and read onwards. Yet, those interested in pattern languages, may also skip the first part and go directly to Chapter 6 (p. 63). Further, those interested in the PBGT approach and wanting to see how it can actually be used, can go directly to Chapter 5 (p. 45) and then Chapter 8 (p. 109).

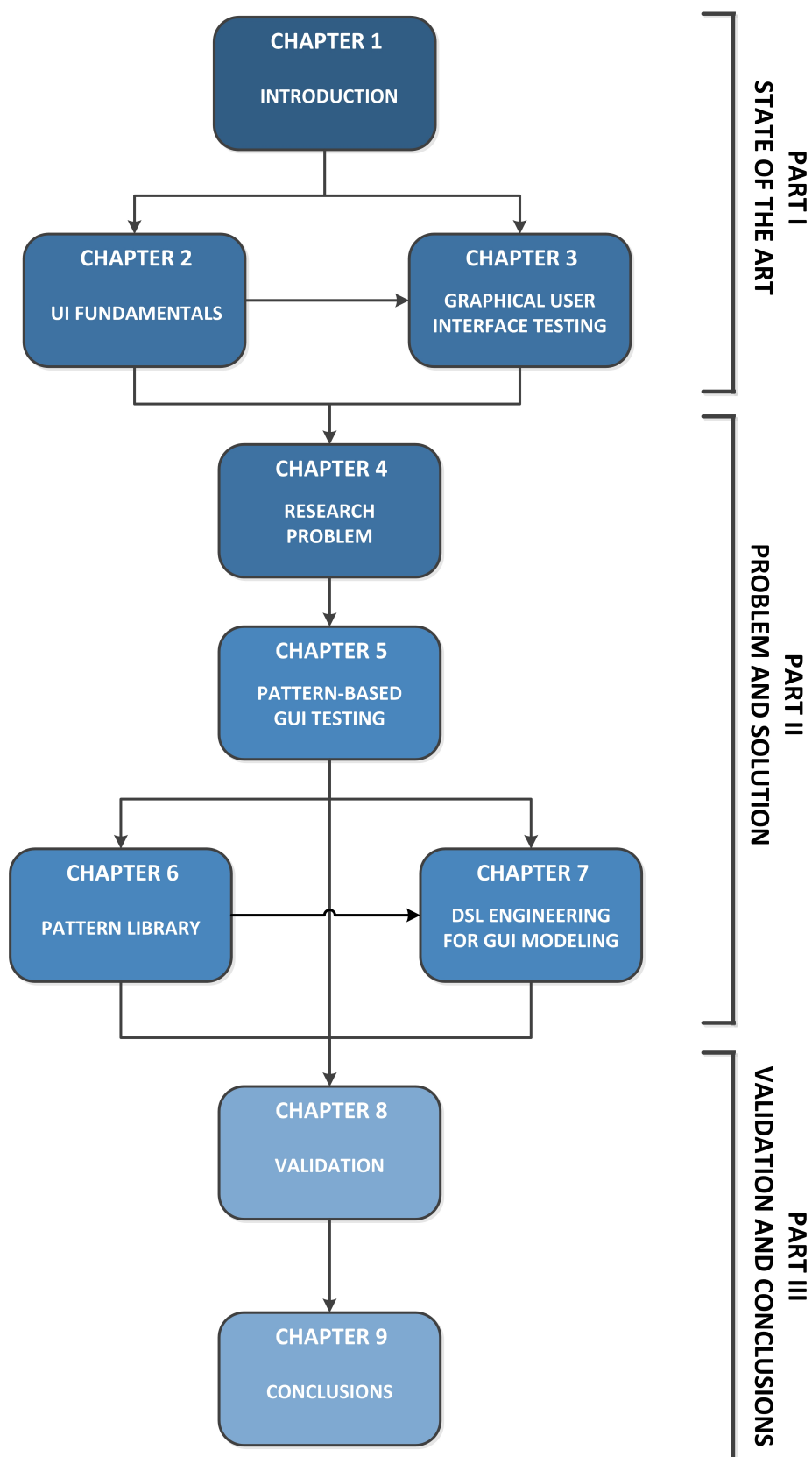


Figure 1.1: Structure of the Dissertation.

1.7 Publications

This dissertation integrates the work performed during the author's PhD research. Preliminary versions of portions of this dissertation have been chronologically published as:

- **A Pattern-Based Approach for GUI Modeling and Testing**, 24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013), Pasadena, California, USA.
- **A GUI Modeling DSL for Pattern-Based GUI Testing – PARADIGM**, 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'14), Lisbon, Portugal.
- **Towards a Pattern Language for Model-Based GUI Testing**, 19th European Conference on Pattern Languages of Programs (EuroPLOP 2014), Kloster Irsee, Bavaria, Germany.
- **PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-Based GUI Testing**, 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014), Västerås, Sweden.
- **A Novel Approach using Alloy in Domain-Specific Language Engineering**, 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), ESEO, Angers, Loire Valley, France.

A manuscript entitled **Pattern-Based GUI Testing: Bridging the Gap Between Design & Quality Assurance** has also been submitted for reviewing to the *Software Testing, Verification and Reliability* (STVR) journal. Until the moment of writing this dissertation, positive feedback was obtained, where reviewers suggested a major review for resubmission. A second version of the manuscript was submitted to the journal.

2

UI Fundamentals

User interfaces (UIs) have an important role in the communication between end-users and the underlying software system. They are the system’s entry point, upon which users interact by performing tasks in the pursuit of attaining a goal (or a set of goals). Their usability, interactivity, aesthetics, consistency, responsiveness and trustworthiness are some of the characteristics/factors that contribute to the success of the software and to its adoption by users. Minor details in design can greatly affect users’ perception on a software system. The UI should be designed to fulfill end-users’ expectations. In nowadays applications, this is even more perceptible. A UI is designed considering its context of use.

Jef Raskin [Ras00] mentions that a user interface represents “*the way that you accomplish tasks with a product — what you do and how it responds — that’s the interface*”. The UI provides means for gathering *input*, allowing users to control the system, and consequently means to observe its *outputs* by providing feedback to inform the users of their actions.

Developers and designers can use several interface design tools in order to assist and speed up the UI design and implementation process. In addition, a UI can have different styles of interaction.

This chapter presents an overview over the topic of user interfaces. It provides understanding concerning the several types of user interfaces, the desired qualities

that a user interface should have, the common defects that appear in UIs and some development tools that are used in the creation of UIs.

2.1 Types of UI

There are three main types of user interfaces (displayed in Figure 2.1): command-line interfaces (CLI); graphical user interfaces (GUI); and natural user interfaces (NUI).

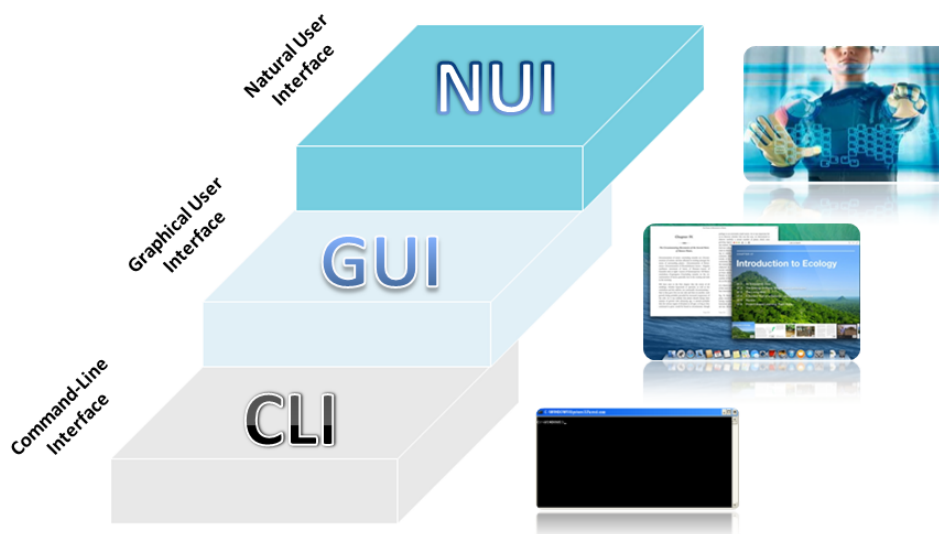


Figure 2.1: Overview on existing types of user interface.

The CLIs emerged as one of the first user interfaces. Then came the era of graphical user interfaces (GUIs), which from the beginning have gained great popularity and are widely used today. In the same sense that GUIs were seen as the next stage in evolution from CLIs, NUIs are following the same steps from GUIs. A more detailed description of the different types of interfaces is presented next.

2.1.1 Command Line Interfaces

A command-line interface (CLI) is a text-based system that enables users to type commands in terminals or console windows. The system waits for input from the user to process commands that will be executed once the user hits the *Enter* key. Then, the user will be notified with a response from the system.

CLIs have a high learning curve. The set of commands provided by the CLIs are enormous. Users are required to have knowledge, time and patience to learn a variety of commands, so they become proficient to use the system.

When compared with other user interfaces such as GUIs, CLIs are not user-friendly. They are however faster than other UIs. CLIs tend not to change their appearance, providing a certain degree of expertise to the user. Further, CLIs also save computer resources in a way they do not require graphical power or high memory consumption.

2.1.2 Graphical User Interfaces

A large segment of computer programs feature a GUI. GUIs have become a popular way of interacting with computer programs, providing user-friendly interfaces to its end-users. Occasionally GUIs are denoted as the WIMP interface: **Windows, Icons, Menus, and Pointers**.

GUIs interaction is performed with a pointing device and a keyboard. GUIs feature a collection of objects, enabling tasks to be performed by the user. The order upon users perform tasks on GUIs is arbitrary. Users have the possibility to interrupt a task in order to interact with another window or dialog.

GUIs aim to “*reduce the memory requirements imposed on the user and dramatically reduce system learning requirements*” [Gal07]. Galitz indicates a set of factors that contribute to the success of GUIs, to name a few: **faster learning** as graphical representation enables learning; **easier remembering** that due to simplicity, users are able to remember basic concepts; **increased feeling of control** since users obtain a perception of control that boosts confidence; **more attractive** as GUIs provide a new level of “entertainment” and are more appealing; and **smooth transition from CLIs**, while it is comfortable to move from CLIs to GUIs, the reverse is yet to be proven.

Nevertheless, GUIs are perceived to be slower than CLIs in terms of interaction. As an example, to perform a certain task in a GUI, users may have to go through several windows and dialogs to attain their goal. A GUI, due to its powerful graphical capabilities, will require more system resources.

Web-Based User Interfaces

The fast expansion and popularity of the Internet led to the rise of web user interfaces (WUIs) [Mem01]. WUI design can be summarized as design of navigation and presentation of information [Gal07]. WUIs are interactive as they

accept input and provide output by generating web pages that are transmitted via the internet or intranet. Users are able to view and interact with the generated web pages using a browser.

WUIs share GUIs' characteristics [Mem01]: (1) **event-driven nature**, as the input is driven by events causing changes in the UI state; (2) **graphical output**, with a similar layout as GUIs; (3) **graphical objects** with properties and; (4) **hierarchical structure**, featuring pages, frames and relationships among them.

2.1.3 Natural User Interfaces

NUIs provide a new way of interfacing [CRS⁺13, WW11, LIRC12]. They stand as the next evolutionary step from GUIs. Instead of using the mouse to interact with the system, by moving, pointing and clicking, the user can use his hand(s) *natural* movement(s)/gesture(s), without using the mouse physical device, to achieve the same purpose. Alternatively, he can also interact through voice commands. The idea is to go beyond mouse and keyboard, using gestures and/or voice to interact with the interface.

NUIs can leverage the trend for the development of a new UI, that feels more natural to use, and change the way people interact with technology [WW11]. Besides natural physical gestures and verbal speaking behaviors, other behaviors, like motion, non-verbal speaking, facial features and eye tracking, are also considered to be associated with natural interfaces. There are several domains for NUIs, such as virtual reality, healthcare, gaming industry, mobile industry, among others.

Touch User Interfaces

A touch user interface (TUI) provides a type of interaction different from GUIs, consisting of a pointing technology that is based upon the sense of touch. A TUI features a set of virtual interface elements such as virtual scroll wheels, slider bars, keyboards, dials, menus, and others. These controls appear *floating* on the top of content and applications. They can be accessed via specific touch gestures, depending on the implementation.

A *touchscreen* is a graphical display that features a TUI, upon which users interact by applying pressure and using gestures (with one or more fingers) or by using a pen/stylus. The touchscreen acts as both input and output.

TUIs are easy to learn, facilitating user interaction, enabling a quick form of interaction. Users are able to rapidly perform actions, such as dragging and

selecting objects in the UI. TUIs are also flexible as the UI can be arranged and customized for each action to be performed [SPS91]. Users have the option to select the type of keyboard layouts in order to meet their preferences. Smartphones, tablets, touch-screen desktops and smartwatches are some examples of TUIs.

Motion Tracking Interfaces

A motion tracking interface (MTI) consists in monitoring the user's body movements with the purpose to translate them into commands, providing the ability of pointing, selecting and manipulate objects on a screen. Through gesture recognition it becomes possible to interpret user's gestures and identify their meaning. Mitra describes gestures as "*expressive, meaningful body motions involving physical movements of the fingers, hands, arms, head, face, or body with the intent of conveying meaningful information or interacting with the environment*" [MA07].

Gesture recognition can be used in different contexts, such as [MA07]: sign language recognition; monitor patients' emotional states or stress levels; lie detection; navigating and/or manipulating in virtual environments; among others. In addition, MTIs have gained popularity in the gaming market. They can be found on Nintendo Wii [Nin14b], Microsoft Kinect [Mic14a], and Playstation Move [Son11].

2.2 UI Qualities

In terms of quality UIs can be evaluated from two different perspectives [GC96]: *internal quality* and *external quality*. External quality refers to properties of the software as a product that users can experience (user's perspective). Internal quality refers to properties of software as seen by developers that are desirable to facilitate the process of creating a good product (engineering perspective). Regarding external quality, this perspective can be classified in terms of functionality, reliability, usability, efficiency, flexibility and simplicity. These are characteristics that can be perceived by using the UI. In terms of development efforts, there are characteristics that developers care about, such as maintainability, reusability, performance and testability.

As noted by [Fad09], "*all great user interfaces share 8 qualities or characteristics*":

- **Clarity.** The UI should be clear in terms of language, flow and descriptions for visual elements. Clear UIs ensure that users do not make mistakes while using them and that they are not required to use manuals.
- **Concision.** An overpopulated UI with several visual elements will lead to confusion and monotonous use. The goal is to make the UI concise and clear at the same time.
- **Familiarity.** When using a UI for the first time, there are elements that users are familiarized with. For instance, tabs are used for navigation on several sites and applications. This is also related with experience. If users have already seen and worked with UIs that feature certain elements, then when using other applications, no matter the design, they will know how to interact with them.
- **Responsiveness.** Responsiveness is about the speed to respond to user's requests. A fast UI inspires user confidence. The UI should provide rapid feedback for all user actions and it can be visual, textual or even auditory.
- **Consistency.** Design consistency is key since it allows users to use skills learned in a particular situation to be transferred to another similar to it. As users learn how parts of the UI work, they become able to use this knowledge to other areas or features, provided by the UI.
- **Aesthetics.** An appealing design makes a UI attractive to the eye. If a UI looks inviting it will make the users' experience more enjoyable. UIs that lack design aesthetics, turn out "*disorienting, obscuring the intent and meaning, and slow down and confuse the user*" [Gal07].
- **Efficiency.** A UI should have a good design and shortcuts to simplify user's tasks. Tasks should be accomplished in a quick manner and with low efforts.
- **Forgiveness.** Sooner or later, people will make mistakes while using the UI. Hence, common and inevitable mistakes should be tolerated. Errors should be prevented from occurring and should be anticipated. Users should be allowed to correct their actions whenever required. When errors occur clear instructions should be presented on how to correct them [Gal07].

2.3 Typical Defects

A GUI can originate/led to defects in the software. A defect refers to issues with the program, including its external and internal behaviors. There is an inherent confusion in respects to the description of what is a failure, error and fault, as they evoke many different answers. It is important to examine some of the fundamentals concerning these terminologies. These terms are all related, but they do have different meanings. A **failure** occurs when the external behavior of the system does not conform to the system specification [NT11]. In addition, Chillarege [Chi10] provides two perspectives: a failure occurs “*when something does not happen as it should*” or, when “*something happens that should not happen*”. A **fault** is the cause that led to the failure, due to “missing or incorrect code” [Bin00]. It is an incorrect step that causes the program to perform in an unplanned/unanticipated manner. An **error** is a “human action that produces a fault” [Bin00]. A detailed description about this terminology can be found in [Chi96].

Typical errors that occur in GUIs can be categorized in three groups [Pai07]: usability errors, functionality errors and performance errors. Usability errors refer to the difficulties that users have to surpass when interacting with the system. These errors are related, for instance, with the lack or absence of feedback, and with communication issues between the system and its users, due to missing information or spelling errors. Functionality errors are related to problems with the behavior of the system. These errors are associated to unexpected behavior, either due to missing implementation of certain functionalities or due to incorrect implementation. For instance, the system behaves differently from expected when [Ger97]: does not ensure data validation; provides incorrect field defaults; does not force mandatory fields; and when wrong fields are retrieved from queries. Performance errors are termed as non-functional errors. They are related with the efficiency of the system. The time spent to execute a task and the amount of consumed resources are some examples of the measurements that can be obtained for the overall efficiency of the system.

2.4 UI Design and Development Tools

GUI design and development tools assist design and development teams in reducing hard work in the construction of UIs. They enhance teams’ productivity, by reducing construction efforts in the design and development tasks required

in the creation of GUIs. Some tools are specific to design, while others provide support for both design and behavior of UIs. These tools also contribute to the rapid conclusion of projects and also unleash creativity, enabling the creation of compelling UIs.

Developers often spend considerable amount of time writing code for developing GUIs. This is due to two reasons: the acceptance of the importance of GUIs in creating user-friendly software and; to facilitate users to accomplish tasks. According to literature, back in 2001, a GUI was already constituted by 50 up to 60% of the total software code [Mem02]. Nowadays, due to the increasing functionality on GUIs, these numbers are even higher. Developers normally use integrated development environments (IDEs) to assist their tasks. An IDE typically consists of a code editor, a compiler, a debugger, and a GUI builder. IDEs provide a user-friendly framework for many programming languages, such as C#, Java, Visual Basic, Visual C++, among others. The most popular IDEs are Microsoft Visual Studio, Microsoft Expression Studio, IBM Eclipse and Netbeans. IDEs have evolved over the years, to include several tools with the purpose to make developers' work more efficient. They have advanced to include extension mechanisms/plugin to support the addition of more tools.

User interface builders aim at reducing the learning curve, letting designers and developers create user-friendly applications faster. UI builders provide a list of standard UI components, menus to configure such controls and a direct-manipulation editor for placing them in a window. In addition, they also allow creating custom controls, providing more degree of liberty and creativity to users. These tools are WYSIWYG (What You See Is What You Get) and they can either be standalone, or they can appear integrated with a particular IDE. However, dynamic changes in GUIs must be manually coded.

Prototypes are visual representations of software systems. Several different approaches exist for prototyping, which vary from simple to rich, fully interactive GUIs. A prototype can also have a variety of audiences: potential or existing customers, senior management, UI reviewers, developers, documentation teams, and marketing staff. Prototypes can be classified in three main categories: wireframes/paper prototypes; visual prototypes; and interactive prototypes. Wireframes/paper prototypes are limited, non-interactive early-stage techniques. Though often simplistic, this style of prototyping is useful because prototypes can be quickly created and do not require technical expertise. Visual prototypes, on the other hand, are screen mock-ups that offer the look and feel of a system design. In addition, they are practical to demonstrate potential

appearances and layouts. Last, the interactive prototypes, are costly and require time to be created. Further, they aim to model a system design more realistically, and represent the system's expected behavior. These prototypes are the richest, most useful types of prototype, although the slowest to create.

Quite often developers and designers tend to use toolkits. Toolkits provide a set of components that enable development teams to build and deploy sophisticated UIs. Additionally, toolkits allow developers to combine UI parts, in an interactive environment, in order to build the layout of a given interface.

2.5 UI Patterns

In the context of graphical user interface design, there are certain sets of patterns that can be applied in their development. These set of patterns are typically referred as UI patterns. UI patterns represent commonly recurring solutions that solve common GUI design problems. Because of the high-level nature of this definition, there are different types of UI patterns at multiple levels of abstraction. We can have two levels of UI patterns: *high-level* and *low-level* ones. The former defines how to separate concerns on GUIs, while the latter describe specific parts of GUIs, how they should behave on user interaction, and how they should be implemented.

2.5.1 High-Level Patterns

There are mainly three high-level patterns that Software Architects and Software Engineers are most familiar with: Model-View-Controller (MVC); Model-View-Presenter (MVP); and to some extent more recently, Model-View-ViewModel (MVVM). The main *pillar* of these patterns, is the concept of *Separated Presentation* [Fow06]. As pointed by Martin Fowler, the goal is to have a clear defined division between domain objects that model the real world, and presentation objects that are the GUI elements.

Model-View-Controller (MVC)

MVC is probably one of the most widely quoted pattern. It emerged in the 1970's to build user interfaces in Smalltalk-80 [KP⁺88]. It is a composite pattern consisting of the Observer, Strategy and Composite patterns [FRBS04]. MVC consists on three kinds of objects:

- **Model** – contains the core functionality, data and state. It also referred as the application object. The model is oblivious to the view and controller.
- **View** – represents the displayed information to the user, i.e., provides the presentation of the model. It is a composite of GUI controls. The view usually gets notifications about the state and data it requires to display directly from the model, reflecting the state of the model.
- **Controller** – handles user input, defining the way the UI reacts to user input.

MVC decouples the above objects in order to increase flexibility and reuse [GHJV94]. There are several versions and implementations of the MVC pattern. The one that will be described next is the one presented in POSA [BMR⁺96] and discussed by Martin Fowler [Fow02]. A graphical representation of the MVC pattern can be seen in Figure 2.2.

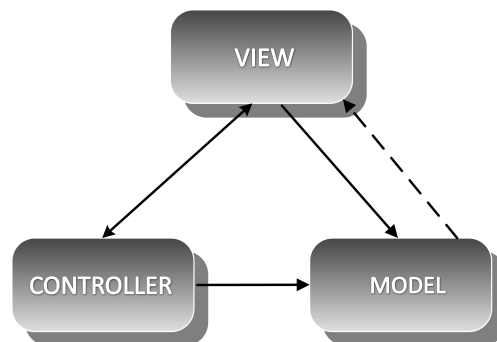


Figure 2.2: Model-View-Controller (MVC) Pattern.

Upon user interaction with the *view*, the latter collects the user actions and communicates them to the *controller* for processing. The *controller* handles the user actions and interprets them. The *controller* will identify the user actions and will determine how the *model* should be manipulated based on the performed action. Depending on the scenario, the *controller* may inform the *view* to change as a result. Whenever the *model* changes, the *model* notifies *view* that its state has changed. The *view* gets the state from the *model* and updates itself.

Model-View-Presenter (MVP)

Model-View-Presenter (MVP) is an evolution from MVC. It appeared at IBM and gained more visibility at Taligent during the 1990's [Pot96]. The most noticeable change is the replacement of the *controller* with the *presenter*. To put it short, the *presenter* is responsible for binding the *model* to the *view*. The *view* is unaware of the *model*.

The *view* consists of UI controls and forwards user actions to the *presenter*. The *presenter* manages the user actions, updates the model and alters the state of the *view*. The *view* defines an interface that will be used by the *presenter* to communicate with the *view*.

There are two variations of the MVP Pattern: *Passive View* (Figure 2.3) and *Supervising Controller* (Figure 2.4). The main difference between the patterns is that *Supervising Controller* (can also be named *Supervising Presenter*) promotes coupling between the *view* and the *model* while *Passive View* does not allow it.

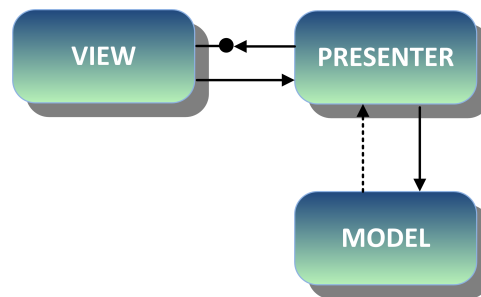


Figure 2.3: Model-View-Presenter (MVP) Passive View Pattern.

In the *Passive View* (Figure 2.3) variation, the *presenter* is the only object who is able to interact with the *model* and retrieve the data required for performing business logic. The *view* is not aware of the *model*. In this variation, the *view* responsibility is to display the data provided by the *presenter*. The purpose of the *presenter* is to ensure communication between the *view* and the *model*. The *presenter* understands how to reach the *model*, retrieving the necessary data from it, performing additional processing (when required) and returning this data to the *view*. This variation promotes the usage of building views that contain few logic as possible, and provides a bigger role for the *presenter*, since it will be responsible (mediator) for all the interactions.

In the *Supervising Controller* (Figure 2.4) variation, the *view* interacts directly with the *model* without using the *presenter*. This variation promotes data binding even further, since the *view* binds to the *model* using this process. Data binding represents the process that establishes a connection between the application UI and business logic. Whenever the data changes its value, the elements bound to the data reflect changes automatically.

Model-View-ViewModel (MVVM)

The MVVM pattern has emerged to overcome some of the flaws of MVC and MVP patterns [Vic12]. MVVM is tailored to be used for modern UI development

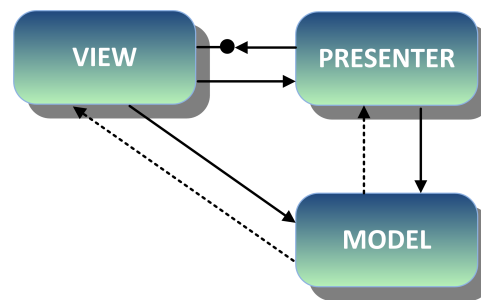


Figure 2.4: Model-View-Presenter (MVP) Supervising Controller Pattern.

platforms [Gos05]. The MVVM pattern is illustrated in Figure 2.5.

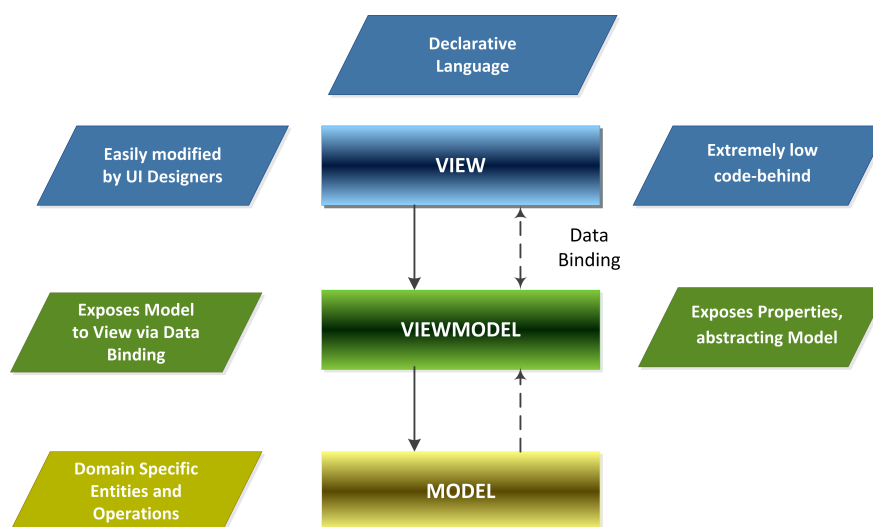


Figure 2.5: Model View ViewModel (MVVM) Pattern.

In MVVM the *view* is the responsibility of a designer (since it requires a certain graphical design skills) and not much of a developer. The design is done in a declarative language such as HTML or XAML [Mic14f]. The *view* uses a binding system for its communications with the *viewmodel* and typically contains very few code-behind.

The *viewmodel* acts as a mediator between the *view* and the *model*. It moves data from the *model* to the *view* and communicates user actions from the *view* to the *model*. Unlike the *presenter* object in MVP, the *viewmodel* does not require a reference to a *view*. The *view* binds to properties on a *viewmodel*, which are exposed with data from the *model*. The *viewmodel* is responsible for implementing *view* logic. The *model* has the same role as the one in MVP.

MVVM ensures extremely loosely coupled design, since the *view* is unaware of the *model*, while the *viewmodel* and *model* are unaware of the *view*. Even the *model* is oblivious of the existence of the *viewmodel* and the *view*.

2.5.2 Low-Level Patterns

This category of UI patterns represents generic graphical implementations of a given functionality. Although certain functionalities, can have different layouts, the aspect that is particular relevant to mention is the common behavior (which is repeated over time) described by these common implementations. Low-level UI patterns have a key part in the foundations of this research work.

A large number of UI patterns can be found on the web [Ras14, vW08, IDT08, Pat14, Yah12, Tox14] and on books [Tid11, Nei14]. Thus, the objective is to describe next part of the most common UI patterns that can be seen practically in the majority of UIs (GUIs, WUIs, and TUIs).

Input Prompt

This pattern (Figure 2.6) provides a way for the user to enter data in the system. It can feature a label (“Phone, email or username” in Figure below), in order to explain what the input field refers to.



Figure 2.6: Input prompt example (source from Twitter [DWSG14] welcome page).

Some implementations do not feature the label and the description of the input field appears blank.

Calendar Picker

Calendar Picker (Figure 2.7) aims to simplify the selection of a date or a date range with the purpose of submit, sort or filter data [Tox14].

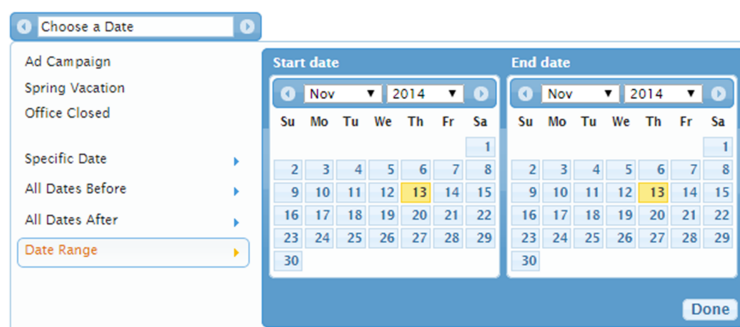
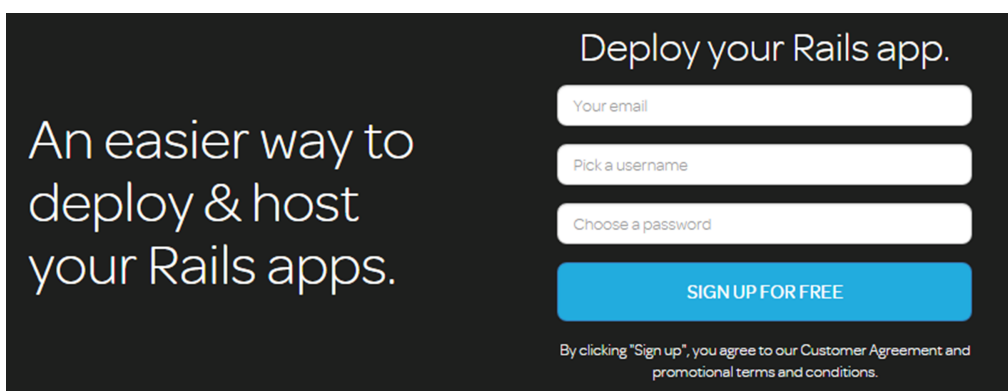


Figure 2.7: Calendar Picker example (source from [Gro14a]).

Instead of having to manually insert date, which could result in several errors, users have the capability of graphically select and navigate through dates.

Account Registration

This pattern enables users to create an account for a specific web site. This way the system will store the information about the user and will be able to display personalized information for the user. Further, it will provide access to restricted areas that were not available before. One implementation of this UI pattern is illustrated in Figure 2.8.



The image shows a registration form on a dark background. On the left, white text reads: "An easier way to deploy & host your Rails apps." On the right, the heading "Deploy your Rails app." is followed by three white input fields: "Your email", "Pick a username", and "Choose a password". Below these is a blue button labeled "SIGN UP FOR FREE". At the bottom right, small white text states: "By clicking 'Sign up', you agree to our Customer Agreement and promotional terms and conditions."

Figure 2.8: Account Registration example (source from [Nin14a]).

There are several implementations for this pattern. They normally contain at least three textboxes (desired *username*, desired *password* and *password confirmation*) and a *register* button to submit the data that was entered.

Breadcrumbs

Breadcrumbs are used to display the user's current location in relation to the site's hierarchy. They are displayed in a horizontal list that reflect the location of the current page and allowing the user to navigate up. This UI pattern is illustrated in Figure 2.9.



Figure 2.9: Breadcrumbs example (source from [Gro14b]).

Auto-Complete

Auto-Complete (Figure 2.10) is a rather popular pattern. It can be found in the Google search engine, for instance. While the user types a word, a list of possible

matches appear and then the user selects the desired item. It enables a faster and more accurate user's decision in order to select information from a larger dataset. It removes ambiguity, avoiding mistyped information, and narrowing the results.

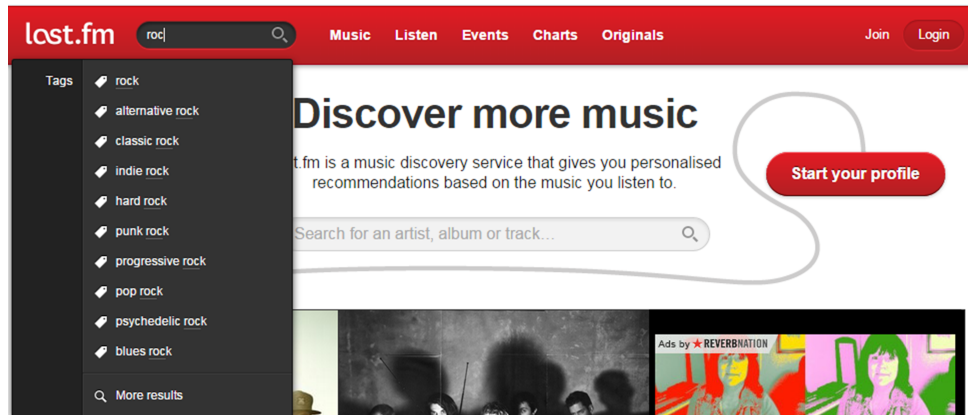


Figure 2.10: Auto-complete example (source from [Int14]).

Sort

The sort UI pattern has the goal to order displayed data according to specific criteria. Figure below illustrates one implementation of this pattern.

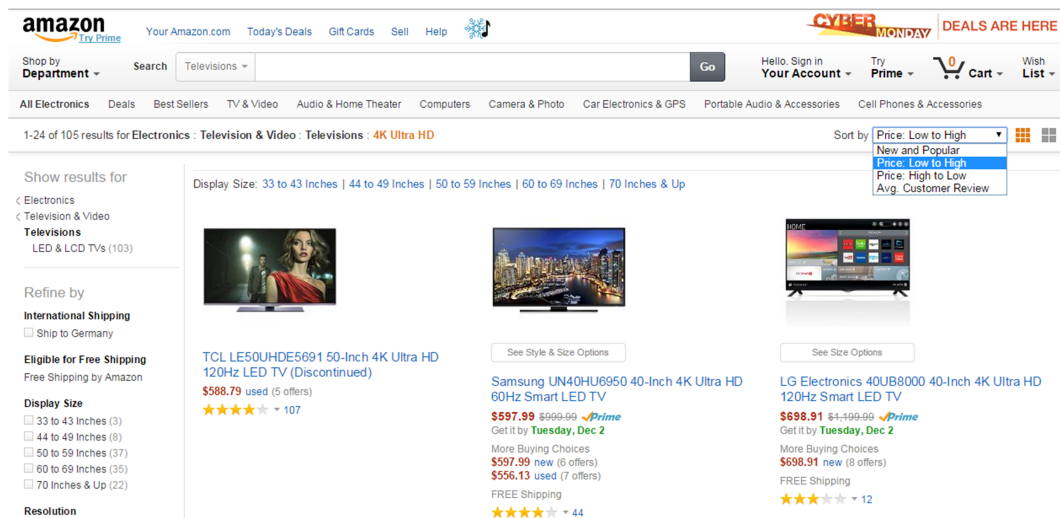


Figure 2.11: Sort example (source from [Ama14]).

In Figure 2.11, the data is being displayed in multiple columns, and the end-user is able to sort the displayed data according to: “New and Popular”; “Price: Low to High”; “Price: High to Low” and; “Avg. Customer Review”.

2.6 Discussion

User interfaces have been evolving over the years, and their importance in today's software systems has become apparent. Developers use specific tools to assist the creation of such UIs. The majority of such tools provide a useful contribution to assist and enhance teams' productivity in the construction of UIs.

From a testing perspective, these tools do not provide any relationship between the design of GUIs and its verification. As an example, a developer uses an IDE for the construction of a certain GUI. Normally, the developer would use a set of controls to assist the design and implementation of the GUI. These controls only contain the necessary code for the design and the functional behavior of the application to be constructed. In this phase, these UI controls do not contain any code that could be used for testing purposes.

In an ideal setting, the UI controls could have the associated generated test code. The developers could use these UI controls, as they normally would, but they were not required to having to change it. Then, in a later phase, the tester would be able to modify the test code, according to his desired needs for testing purposes.

2.7 Summary

This chapter provided an overview on UI basics, describing the several main types of UIs, their desired qualities, typical defects that can be found on UIs and a description of the tools that can be used to assist their creation. Moreover, the concept and usage of UI patterns was also introduced in this chapter.

There are three main types of UIs: command-line user interfaces (CLIs), graphical user interfaces (GUIs) and natural user interfaces (NUIs). GUIs are popular UIs and are widely used. Web-user interfaces have emerged due to the popularity of the internet. They share the same characteristics of GUIs: event-driven nature; graphical output; hierarchical structure and graphical objects with properties. NUIs are an evolution from GUIs, where the idea is to interact with the UI using gestures/movements, voice commands or eye-tracking features, replacing the physical mouse and keyboard. Touch user interfaces (which can be seen on smartphones) and motion tracking interfaces (featured on Nintendo Wii, for instance) are two examples of NUIs.

Several UIs are developed using UI patterns. UI patterns are recurring solutions that solve common GUI design problems. They can be classified in two levels of

abstraction: high-level and low-level. High-level UI patterns are often referred as GUI architectures like MVC, MVP and MVVM. Low-level UI patterns describe specific parts of GUIs, how they should behave on user interaction, how they should be implemented and, when and what they should be used for. The focus of this dissertation is on low-level UI patterns.

UIs should be built having in mind a collection of principles, which act as guidance for producing high quality UIs. Software is prone to have defects. GUIs can have defects too. The next chapter will review the state of the art concerning GUI testing.

3

Graphical User Interface Testing

During the past decade, the importance of GUIs has become recognized. GUIs represent a popular way of interacting with computer programs. They provide flexibility in the way that users are able to perform tasks. For instance, a user may format a paragraph in Microsoft's Word software, in several ways. Users perform tasks by interacting with GUI widgets via events. The sequences of events that they can execute are not limited.

Unfortunately, this flexibility that makes GUIs easy to use also makes testing them for functional correctness notoriously more difficult. The tester has to check whether all, or a reasonable subset, of the possible sequences of events that an end-user can execute for a task execute correctly. This task is compounded by the fact that despite all of the advances in automated testing tools and frameworks over the last decade, manual testing still represents the majority of testing effort within most software development organizations [MPM13]. GUI testing in practice remains a huge resource intensive task.

Due to the extensive usage of GUIs in various software systems, GUI testing has become an area of increasing importance to software development. Furthermore, software companies have the best of interests on finding defects on their products before their costumers' do, not only to meet user demands and therefore increase confidence in relation to their software, but also to induce correctness and

commitment with them.

This chapter discusses manual and automated GUI testing. It also provides background and related work on the topic of GUI testing approaches.

3.1 GUI Testing Challenges

GUI applications diverge from typical software to such a degree that complicate traditional testing tools and techniques [Mem02]. This is due to the event-driven nature of GUIs. Several possible combinations of commands can be executed on the GUI. Hence, testing all possible interactions is not practical, since a GUI containing “n” controls requires a factorial number of test cases to test all possible combinations [Whi96]. Instead, testers attempt to limit the number of test cases that need to be executed [BM07].

Coverage criteria are sets of rules that are used as a guideline for determining testing adequacy. Nevertheless, traditional coverage criteria are not suited to GUIs: (i) GUIs are usually developed using precompiled components (libraries) that might not be available for coverage evaluation; and (ii) GUIs’ inputs consist of a sequence of events, which due to the large number of permutations may lead to outsized number of GUI states. For adequate testing, a GUI event may need to be tested in a large number of these states [MSP01].

GUI objects might have synchronization and dependency characteristics implemented. In many situations, these characteristics between objects might not be restricted to objects in the same window. For instance, a value in a particular window can be linked to a value in a separate window. Thus, finding all these links can be impractical [LHRM07].

Performing modifications in GUIs can introduce new errors into already tested code portions. Both inputs and outputs to a GUI depend on the layout of graphical elements. Consequently, changes on the layout can modify the input/output mapping turning old test cases ineffective [Mem02]. Another challenge relates with the response time of the GUI. If the tester sets the maximum response time to a particular value, and the GUI exceeds that time, it will generate an error, even though it can be related with external sources, like a non-responsive database.

There are several approaches for GUI testing, which will be described in this chapter, that feature some drawbacks and therefore, also contribute to the difficulty in testing GUIs.

3.2 Manual GUI Testing

In the real world, manual tasks (tests) may provide more sense of control, since they represent something that is human-controllable. However, manual testing is a laborious activity, specifically error-prone and tedious. In addition, testers are required to be patient, perceptive, creative and skillful. Manual GUI tests are suitable for initial/exploratory testing [Pai07]. They can be performed by end-users, in order to check basic functionality of the software system.

GUIs are becoming more complex, making manual GUI testing a difficult task to comply. Current practices in GUI testing are still a manual activity. Moreover, errors found in manual testing often rely on the skills of the tester. Furthermore, manual tests are difficult to reproduce when software is updated, since regression testing is not supported. Despite all of the advances in automated testing tools and frameworks over the last decade, manual testing still represents the majority of testing effort within most software development organizations.

3.3 Automated GUI Testing

Automating software testing can considerably reduce the effort required for adequate testing. Furthermore, it helps to greatly reduce the time and the cost spent on software testing throughout the entire development life cycle. Automation ensures that tests are performed regularly and consistently, resulting in early error detection that leads to enhanced quality and shorter time to market. With a proper automated testing process, more testing cases can be conducted and more faults can be found within a short period of time.

Automated tests are repeatable, using exactly the same inputs in the same sequence time and again, something that cannot be guaranteed with manual testing. Automated testing enables even the smallest of maintenance changes to be fully tested with minimal effort. Test automation enables testing tasks to be performed more efficiently and systematically. Furthermore, test automation provides the following benefits [GF99]:

- **Regression testing.** Existing (regression) tests can be run on a new software version. Since the tests have been already created and automated to run on an earlier version of the software, it should be possible to select the tests and initiate their execution in short time.
- **Run further tests more often.** A comprehensible advantage on automation

is the ability to run more tests in less time and hence providing the ability to run them more frequently.

- **Run tests that would be difficult/impossible to run manually.** To perform a live test of a system for 600 users is impractical. This input can be simulated using automated tests. Another advantage is that since the tests can be replayed automatically, the user scenario tests can be run by technical staff, even if they do not understand the details of the SUT.
- **Better use of resources.** There are a whole set of tedious tasks, like repetitively entering the same test inputs, for instance. Automating this process will provide a better accuracy and will definitely boost testers' mood. Furthermore, skilled testers can be put into use for more demanding test cases.
- **Consistency and repeatability of tests.** Tests that are repeated automatically will be repeated without modifications every time. This provides a level of consistency to the tests, something that is very difficult to achieve manually. The same tests can be executed on different hardware configurations, using different operating systems, or using different databases.
- **Increased confidence.** The confidence in the system is considerably boosted when a set of automated tests has run successfully.

Despite the several advantages of automated testing, there are some problems, such as [GF99]:

- **Maintenance of automated tests.** Typically, when the system changes it will be necessary to update the tests. This effort can be demanding if the tests require continuously maintenance. Further, if the effort to maintain the tests is higher than the effort to re-run tests manually, it can lead automated tests to be discarded.
- **False sense of security.** Although using automated tests can provide more confidence in finding defects, it does not necessarily represent that the SUT is clear from them. Tests scripts can be incomplete and may contain defects. When this happens, it could be disastrous for the project, since the tests scripts were not correct and the results from the execution would be erroneous.

- **High expectations.** By using automated tests it is often expected that it will be possible to find more new defects. Yet, new defects will not be found if a test has already passed when running the same test again under the same conditions.

3.4 GUI Testing Approaches

Current GUI testing methods still require considerable manual efforts. However, endeavors have been made in order to automate the GUI testing process. Automation ensures that tests are performed regularly and consistently, resulting in early error detection that leads to enhanced quality. With a proper automated GUI testing process, more test cases can be conducted and more faults can be found within less time.

3.4.1 Capture-Replay

Capture-Replay (CR) is a popular GUI testing approach that attempts to automate test execution. This approach is supported by a set of tools [Ran14, Sel14, IBM12], that provide the capability to record user actions, such as mouse motions, mouse clicks, and keyboard inputs, in test scripts, while interacting with the GUI. Based on the test scripts, the user actions can later be replayed, when required, with the goal to verify if the reaction of the system matches the expected result [ABPS12]. The test scripts are specified in a particular scripting language provided by the capture-replay tools.

One of the drawbacks of this approach is the cost of script maintenance. If the GUI changes, it will cause a large number of previously recorded tests to break. This means that tests will need to be manually edited and later rerun. Such difficulty to adapt to small changes, indicates a maintenance problem, that often leads capture-replay method to be discarded after several software releases.

Yet, there are advantages to using capture/replay tools. Testers can use them as fast means to capture the first test and therefore get early feedback. Furthermore, testers can use this method on early prototypes at design time. Since the GUI's functionality is not yet implemented, there are no bugs that interrupt the capturing of test scripts [LW06]. Moreover, the recorded test scripts are suitable for regression testing.

3.4.2 Random Testing

Random testing is also recognized as *stochastic* testing or *monkey* testing. The term *monkey* refers to any form of automated testing performed randomly without any typical user bias [Nym00].

The idea is to have *someone* (*i.e.*, *a monkey*) who does not know how to use the application, interacting with the latter, in order to perform mouse clicks and keyboard strokes, arbitrarily. Microsoft reported that 10 to 20% of bugs are found via test monkeys [Nym02].

This method distinguishes three types of monkeys. *Dumb* monkeys do not have any idea about the system, nor its state. In addition, they are not aware of which inputs are legal or illegal. The downside is that these monkeys are unable to recognize a bug when they face one. Their key goal is to try to crash the SUT. Other monkeys, referred as *semi-smart* monkeys, are able to recognize a bug when they encounter one. *Smart* monkeys have certain awareness about the application they are testing. They obtain their knowledge from a state table or model of the SUT. They traverse the state model and choose the legal steps that allow to move forward in each state. Nevertheless, *smart* monkeys are most costly to develop.

3.4.3 Unit Testing

Unit testing is a popular testing technique that “has proven its value in that a large percentage of defects are identified during its use” [Mic14e]. It is the process of “testing program components, such as methods or object classes and involves verifying that each unit meets its specification” [Som10]. NUnit [Ham04], JUnit [M+04] and Pex [TDH08] are popular frameworks for unit testing.

Unit testing can be applied in GUI testing. It can be used to code test cases manually, but testing GUI behavior is rather difficult. Unit testing frameworks can prove valuable in organizing and executing test cases (specifically for API testing), apart from their generation [Pai07]. Using unit testing frameworks, testers write code to reproduce user actions with the SUT’s GUI, while observing the output, determining whether the result obtained matches the one expected.

However, these frameworks often require considerable coding efforts, in order for GUI testing become effective. Numerous bugs are only found if a particular order of actions is followed. Further, many of these sequences might only be exposed by exercising the GUI intensively. Since unit tests are small sequences of actions, the possibility of letting these errors pass undetected is high.

3.4.4 Model-Based Testing

Model-based Testing (MBT) has been gaining considerable interest in software testing. MBT has been used successfully to testing GUIs [TKH11, Pai07, PFTV05, NRBM14, AFT⁺14, Xie06, XM06, KMPK06a]. A lot of research has been conducted since the past decade. MBT is a software testing technique upon which test cases are derived from a model that describes functional aspects of the SUT [UL07]. It allows checking the conformity between the implementation and the model of the SUT, introducing more systematization and automation into the testing process. The test generation phase is based on algorithms that traverse the model and produce tests as desired.

One popular approach uses Event Flow Graphs (EFGs) to create a GUI model [MSP01]. The idea is to capture the flow of events, featuring all possible event interactions in the UI. The EFG is comprised by nodes and edges. An edge from one node to another means that the second node can be executed immediately after the first one. EFG edges are not labeled, nor do EFGs have states. Furthermore, models are generated directly from an executable by means of a GUI Ripping Tool (GUITAR [NRBM14]).

Other approach uses Labeled State Transition Systems (LSTSs), action words and keywords, with the goal to describe a test model as a LTS, where its transactions correspond to action words [KMPK06b]. A LTS consists of a set of states and a set of transitions among those states. Action words describe user events with a high level of abstraction and keywords correspond to key presses and menu navigation. The idea is to provide simple means for the domain experts so that they can design test cases with action words even before the system implementation. This simplifies the work of testers since programming skills will not be required. According to the authors, by varying the order of events it becomes possible to find previously undetected events.

Finite State Machines (FSMs) ([SS97]) are one of the most popular used models for software design and software testing. Miao et al. [MY10] proposed an FSM based approach (called GUI Test Automation Model – GuiTam), with the goal to automate all the tests of the EFG based approach, aiming to provide more automation concerns, and thus solving limitations of the EFG approach. They indicate that EFGs are not able to model certain scenarios where GUI objects are modified dynamically: (i) changing the visibility of a GUI object, which depends on another objects' state (checkbox), via code will lead to an event being undefined; (ii) toggling of the visibility property value of a panel can cause several events to be uncovered or concealed. These events cannot be modeled and therefore cannot be

tested. Additionally, state-charts' models can also be used for both modeling and generating test cases for GUIs [ROM08]. The use of state-charts in GUI testing has not yet received great interest due to the lack of approaches and contributions to this matter.

Petri nets can also be used in GUI testing. Reza et al. [REG07], propose a new model-based approach to test the structural representation of GUIs using Hierarchical Predicate Transitions Nets (HPrTNs). In their terms, HPrTNs are high classes of Petri nets. They justify their usage due to the capability of HPrTNs to recognize and treat events (desirable and undesirable) and also states (desirable and undesirable behaviors). With HPrTNs it is possible to model the planned behavior of GUIs and generate test cases from the model. Furthermore, authors also propose new coverage criteria for GUIs using HPrTNs.

GUIs can be modeled with Spec# [Mic14d] describing actions performed by the end-user and the expected outcome of each user action. Hence, GUI Spec# models are the input to Spec Explorer [VCG⁺08] with a model-based testing tool (GUI Mapping Tool [PFTV05]) that is able to generate and execute test cases automatically. The effort in writing these models in Spec# is high. Therefore, to diminish this effort and to hide the formalism details from testers, Moreira et al. [MP08], developed a graphical visual notation entitled VAN4GUIM (Visual Abstract Notation for GUI Modeling and Testing), providing tool support. In addition, in this approach, models are translated automatically to Spec#, according to a set of rules. However, the model does not feature all GUIs behavior. Additional behavior needs to be included manually in the generated Spec#. After the Spec# model is completed it will be used as input to Spec Explorer with GUI Mapping Tool.

3.5 Discussion

Automated GUI testing involves performing a set of tasks automatically and then comparing their result with the expected output, providing the ability to repeat the same set of tasks multiple times with different input data, maintaining the same level of accuracy. Furthermore, it has become tremendously important as GUIs become progressively more complex and popular. Test automation allows performing a set of testing tasks, without user intervention, with possible different test input data, which may increase the code coverage without significant additional effort. Current software testing tools and techniques suffer from lack of generic applicability and scalability. Despite the term *automated testing*, the

majority of the approaches still require considerable manual efforts.

Finite state machines suffer from the enormous number of states and transitions. In addition, an increase on the complexity of the system would cause an explosion in the number of states and transitions. State-charts also suffer from the state explosion problem. FSMs cannot represent concurrency, since only one state can be active at a time.

EFGs are one of the most successful model-based techniques. Yet, the generated graphs are too large, and thus have a large testing suite. Further, it is complicated to test specific UI parts and to handle prioritization [EMNM10]. For rather complex applications GUI ripping will have difficulties in generating the full model.

Petri nets provide a different perspective on the field of GUI modeling and testing. However, authors [REG07] have not proven the feasibility of their approach, nor provided a case study to support their view. Petri nets can be pointed as a good resource to model GUIs with the goal to represent concurrency. However, classic petri nets do not support hierarchy [ROM08].

Capture-replay although popular, faces some drawbacks: test data is hard-coded in the scripts; by default it is not possible to compare expected output with the obtained output; scripts are hard to manage; the re-capture takes the same time as the original capture and; the scripts are required to be modified upon small changes in the user interface.

Unit testing frameworks can provide support for executing test cases that were previously programmed by testers. In this case, GUI testing is treated like API testing. However, manual test case construction, with short written sequence of actions, might leave some parts of the application untested.

When analyzing existing approaches it is possible to identify the need to increase models abstraction and promote reuse in order to significantly reduce the effort in building models and testing software applications.

3.6 Summary

Testing is a central quality assurance activity in the software development process. The popularity of GUIs and their widespread use has increased the demand for GUI testing. Current GUI testing techniques still require considerable manual efforts. Manual GUI testing is challenging and requires skills from the tester. Automated testing aims at reducing the manual testing efforts required for testing. There are several techniques that have been developed to address several automation

aspects in the GUI testing process. Yet, they are far from being ideal since each has drawbacks. One possible ideal scenario would be to combine both MBT and CR approaches. Existing GUI testing approaches lack of models' abstraction and do not encourage reuse.

This chapter has reviewed existing GUI testing approaches and discussed their limitations/challenges. The next chapter focuses on identifying the research problem, by analyzing open issues on the topic of GUI testing and proposes a new approach to address some of the issues identified earlier.

4

Research Problem

Current GUIs have a considerable set of features. The diversity and ease with which an end-user can perform a task (via events) in a GUI is high. However, these events tend to converge to achieve the same objective. The effort required to test all the variability is costly. The testing effort could be reduced if current testing techniques would take into account the concept of reuse, specifically when resembling behavior is identified and observed on different GUIs, or within the same GUI under test.

In this chapter, several research issues are described allowing to focus on the path that led to the rising of a new GUI testing approach. Then the thesis statement of this dissertation is defined and extensively explained. The chapter concludes by presenting the methodology devised to validate the thesis.

4.1 Research Issues

From the state of the art described in the previous chapters, a number of research issues have emerged and will be discussed next. They have been filtered in order to provide a better focus over the context of this dissertation.

Model-based Testing (MBT) has been gaining considerable interest in software testing. MBT has been used successfully to testing GUIs (MBGT) [Mem01,

PFV03, KMPK06b, ROM08, REG07, NRBM14, ABPS12]. Yet, MBGT has three important drawbacks worth mentioning: firstly, is the effort required to build the specification/model of the SUT, which is high; secondly, these models are typically tightly coupled to the SUT implementation, i.e., they are often built using a language that does not allow to define elements in an abstract manner, and thus not allowing reuse; and finally the state space explosion problem, since the system can originate an unmanageable number of states to test.

GUIs are developed using a set of tools specific for this purpose. However, these tools are only able to support the initial phases of the GUI development process, i.e., they do not provide means to support verification phases. UI patterns comprise these preceding claims. UI patterns represent generic graphical implementations of a given functionality. During design and development phases, GUIs are built using UI patterns. UI patterns have two particular characteristics: (1) each UI pattern can be realized using different implementations and (2) they have recurrent behavior. GUI testing has largely ignored UI patterns, and existing testing techniques have not yet benefited from the UI patterns recurring nature.

4.2 Research Focus

Due to the extensive use of graphical user interfaces in various software systems, the demand for GUI testing has also increased. The correctness of GUIs can affect a user's overall perception [BRB14] and acceptance of a software system. User's confidence in the software can be greatly reduced when defects are found in the GUI. GUI testing has become an important area that requires the usage of verification and validation (V&V) processes. The goal of verification is to check if the software conforms to the stated functional and non-functional requirements. Validation ensures that the software meets the customer's expectations [Som10].

The context of this dissertation emerges in the area of GUI testing. Therefore, it addresses the research issues identified in the previous section, to summarize:

- the effort required to build models and the absence of abstract models to facilitate reuse.
- the disparity between *design* and *test* phases.
- the absence of generic solutions to test GUIs' recurrent behavior (UI patterns) across their multiple various implementations.
- the state space explosion problem.

Figure 4.1 aims to facilitate a better understanding regarding the focus of this research.

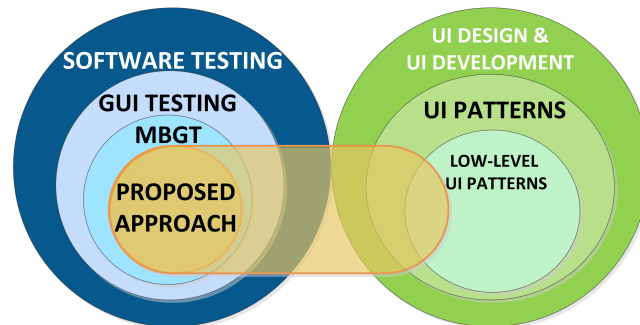


Figure 4.1: Research focus of this research work, identifying the research domains and the contributions of this work directed towards a new GUI testing approach.

The proposed approach positions itself in the sphere of MBGT. It also targets to approximate the design and quality assurance phases.

4.3 Thesis Statement

After analyzing the research issues described in section 4.1 and the state of the art review, the thesis statement is defined as:

“A pattern-based approach for GUI testing can provide reuse of common testing strategies, for GUIs featuring UI patterns, allowing to test their recurrent behavior, across their various implementations, and thus diminishing the modeling and testing effort required when similar behavior is present on these GUIs.”

Since the above statement may raise questions, it is important to explain it and provide further discussion:

- **What are “UI patterns”?**

As described in Chapter 2 “UI Fundamentals”, Section 2.5 (p. 19), in the context of this work, UI patterns are described in a more *low-level* definition, i.e., they refer to specific parts of GUIs, describing how they should behave on user interaction, how they should be implemented, when and why they should be used for. For instance, one popular UI pattern is known as *Login*. It is normally comprised by two input fields – username and password – and a submit button. This pattern describes that it should be used when there is the

necessity of restricting user access to further functionality in the system. The pattern has behavior associated with it. The user is able to fill the username and password fields and then press the submit button. Upon submit the user gets authenticated in the system or the authentication may fail.

- **What is meant by “*pattern-based approach*”?**

GUIs are typically implemented using UI patterns. The natural and perhaps, the logical way of test them, is to also use patterns, i.e., test patterns. Such test patterns can be used to test the recurrent behavior of UI patterns. Test patterns can be the key to facilitate the practice of testing generic behaviors featured in GUIs. Hence, the proposed approach is based on test patterns where the latest goal is to achieve commonality.

- **What do “*common testing strategies for GUIs*” refer to?**

UI patterns can be implemented in different ways. Independently of their design, the aspect that is relevant to notice is their recurring behavior across their multiple implementations. Using a generic strategy can allow to test this common behavior featured on such UI patterns. A test pattern is able to provide a generic testing strategy to test a GUI, that was implemented using a given UI pattern or a set of UI patterns. Further, since a UI pattern may be realized using different implementations, the configurations in a test pattern enables to test different implementations by setting different parameters to the testing strategy. The definition of testing strategies will allow to give more flexibility in testing these type of behaviors.

- **What is the purpose of “*reuse*”?**

Given that GUIs often feature the same UI patterns, yet with different implementations, it would be beneficial to provide a way to test these dynamic range of implementations, in order to achieve reuse. This will allow to save testing costs and reduce the time required to test GUIs that have similar behavior.

4.4 Research Goals

This research is focused on the following research questions:

- How to test multiple implementations of a given UI pattern?
- How to benefit from UI patterns’ recurrent behavior, but from a testing perspective?

- How to diminish the effort required to build test models?
- How to increase the test models' level of abstraction?
- How to make the state explosion problem less prone to occur?

4.4.1 Primary Goal

The primary goal of this research is to *define an innovative approach to improve current model-based GUI testing methods*, by promoting reuse of GUI testing strategies and increasing models' abstraction, bringing closer the design and test phases, and thus ensuring the built of higher quality GUIs. The main requirements of such approach are:

- ***easy of use and understand*** without requiring high efforts in learning the approach and starting to benefit from it in a short time.
- ***goal oriented*** to provide the ability to know upfront what to test, and therefore allowing testers to be more precise.
- ***platform independent*** so several GUIs can be tested without concerns about platform restrictions, augmenting the range of applications that can be tested.
- ***facilitate acceptance*** by reducing testing efforts and thus saving costs, aiming to achieve industry adoption.

4.4.2 Secondary Goals

One secondary goal is to *design a test pattern library* to facilitate the modeling and testing of UIs that feature UI patterns. In order to do so, it is required to first collect a set of UI patterns, study them, analyze them, and identify their recurrent behavior. Then, *other type* of patterns can be reused in order to test a set of UI patterns. The pattern library should be developed considering the following requirements:

- ***adequate***, to ensure a whole set of test strategies to be reused across different implementations for a wide-range of functionalities.
- ***evolutionary***, in order to cope with latest UI trends, where new elements/patterns may be added.

The last secondary goal is to *develop a DSL* to be used in the context of this new proposed approach. The key requirements for the DSL are:

- ***high level of abstraction***, to facilitate the modeling of GUIs and enable reuse of testing strategies.
- ***flexibility***, by providing mechanisms to adapt to the needs of GUIs' evolution.
- ***simplicity***, to facilitate the use and understanding of the DSL elements, during the modeling process.

4.5 Validation Methodology

This research work was validated using 3 empirical studies to compare it with other approaches, and to prove the approach feasibility. Further, the results obtained during the research work have been presented and discussed in international conferences, after being approved in its reviewing processes.

4.6 Summary

Despite the fact MBT has been successfully applied to GUI testing, it still faces some drawbacks, namely: the effort required to build models; lack of models' abstraction and reuse; and the state explosion problem, caused by the uncontrollable generated number of states to test. GUIs are developed using a set of tools that although supporting the initial phases of the development process, lack means to support the verification phase. A specific tool that embodies the latter observations is the UI pattern. UI patterns represent commonly recurring solutions that solve common GUI design problems. Each UI pattern can be realized using different implementations. GUI testing has largely ignored UI patterns, and existing testing techniques have not yet benefited from the UI patterns recurring nature.

The primary goal of this research work is to define an innovative approach to improve current model-based GUI testing methods. The secondary goals refer to the design of a pattern library and to the development of a DSL to be used in the context of the proposed approach. The next chapter presents the proposed approach.

5

Pattern-Based GUI Testing

Software systems with a Graphical User Interface (GUI) front-end are typically designed and implemented using user interface (UI) patterns. UI patterns describe generic solutions (with multiple possible implementations) for recurrent GUI design problems. Existing testing techniques do not take advantage of this fact, which would allow to test GUIs more efficiently.

This chapter focuses on the primary goal of this dissertation, which is to “*to define an innovative approach to improve current model-based GUI testing methods, by promoting reuse of GUI testing strategies and increasing models’ abstraction, bringing closer the design and test phases, and thus ensuring the built of higher quality GUIs*”. Hence, this chapter presents a new modeling and testing approach that leverages UI patterns that have traditionally been used for GUI design. It starts by providing an overall insight on the approach, contextualizing UI patterns and their usage on software applications, and explaining the journey that led to the *original* concept of *UI Test Patterns*. An example on how to use UI Test Patterns and how to take advantage of its use is given on this chapter. In addition, the PBGT methodology is presented and the chapter also describes the tool to support this methodology. An overview on how test case generation is performed in PBGT, is also provided.

5.1 Overview

Graphical applications are developed using sets of windows and widgets. A widget represents a graphical element that is responsible for describing a specific behavior and functionality. These widgets can be implemented in different ways, and despite the fact that their layout may vary in implementation, their behavior across different implementations is typically the same. These widgets are denoted as *UI patterns*. UI patterns, in this dissertation, by definition, represent generic graphical implementations of a given functionality. Since certain functionalities, can have different layouts, the aspect that is particular relevant to notice is the common behavior (that is repeated over time) described by these common implementations.

The definition of a UI pattern described previously, is of a high-level nature, as there are different types of UI patterns at multiple levels of abstraction. Perhaps the most widely quoted high-level UI pattern is the Model View Controller (MVC) that advocates a clear division between domain objects that model the real world, and presentation objects that are the GUI elements. What is relevant for this dissertation, are more concrete low-level UI patterns that describe specific parts of GUIs, how they should behave on user interaction, and how they should be implemented.

One UI pattern that is often seen in several software applications is the *Master/Detail UI Pattern*. This UI pattern allows the user to navigate through items, while visualizing the whole UI. It consists of two areas: master and detail. The content of the detail area changes according to the selection of a specific item from the master area. iTunes and Finder (Figure 5.1), from the Mac OS, are two examples that feature the *Master/Detail UI pattern* (a tutorial on how to implement this UI pattern can be found in Mac Developer Library [App14]).

The iTunes application (<http://www.apple.com/itunes>) features multiple optional selections for songs, albums, artists, genres, playlists, among others. These items represent the master area. Upon selection, the area below, i.e., the detail, changes in respects to the item from the master. For instance, if the user selects “Albums” the detail will display the albums covers, titles and artists, in mosaic. However, if the user selects “Artists” the layout and structure for the detail will be modified. The same behavior happens with Finder. The user is able to navigate through the UI, staying on the same screen, accessing the folders contents. This application features a number of instantiations of the Master/Detail UI Pattern arranged in a hierarchy.

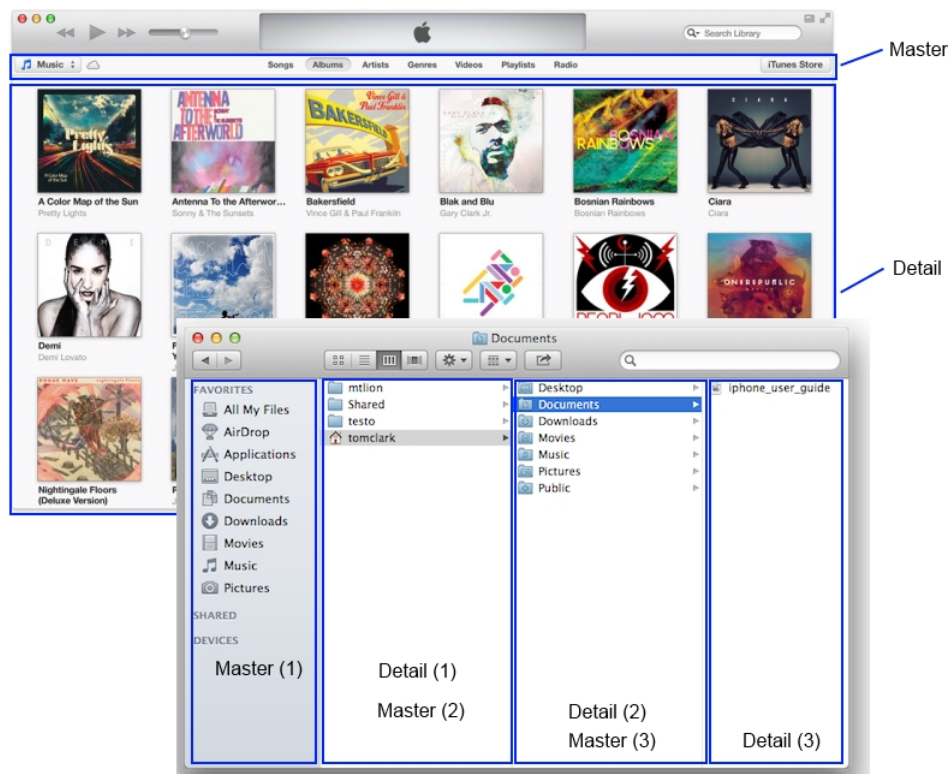


Figure 5.1: iTunes (top) and Finder (bottom) examples (sources from www.apple.com).

Despite the abundance of patterns in the GUI design field, model-based testing of GUIs has unnoticed UI patterns.

From UI Patterns to UI Test Patterns

In the same way that developers and designers make use and are assisted by UI Patterns during GUIs development, it makes sense to give support to testers during their activity too, which, in this case, can be done by providing generic test strategies to test those UI patterns. A new approach emerged, described in this dissertation, called PBGT, which goal is to provide generic test strategies with different configurations in order to allow testing different implementations of a given UI pattern, and therefore promoting reuse. Consequently, a new notion of *UI Test Patterns* was defined. A UI Test Pattern provides a way to test UI patterns across its several implementations and particular demands.

Definition 1. A UI Test Pattern describes a generic test strategy, formally defined by a set of test goals, for later configuration, denoted as $\langle Goal, V, A, C, P \rangle$ where:

1. **Goal** is the ID of the test;

2. V is a set of pairs $\{[variable, inputData]\}$ relating test input data with the variables involved in the test;
3. A is the sequence of actions to perform during test case execution;
4. C is the set of possible checks to perform during test case execution and;
5. P is a Boolean expression (precondition) defining the conditions over variables that determine when it is possible to execute the test.

Commonly, *Goal* is the *name* of the test goal; A and the variables in V describe *what* to do and *how* to execute the test. C describes the final purpose (or *why*) the test should be executed. P defines *when* the corresponding test strategy can be executed. During the modeling phase, the tester needs to configure each UI Test Pattern within the model. The tester has to select the test Goals and, for each Goal, provide the test input data (V), select the checks (C) to perform, and define the precondition (P). The tester can select the same test Goal multiple times for a UI Test Pattern providing different configurations.

As an example, a UI Test Pattern defining a test strategy for the Master/Detail UI pattern, would have a testing goal (change master) which tests if changing the master, the detail updates accordingly. The sequence of actions to perform would be “select master”. Regarding the checks to perform could be: “detail has value X”; “detail does not have value X” and “detail is empty”. For this testing goal, the tester needs to define the input data for master and for detail. Then, the tester selects the checks to perform during test case execution and defines the corresponding precondition. A tester can use multiple times the same testing goal providing different configurations, i.e., different input data and checks to perform.

GUIs feature UI patterns that can be implemented in a multitude of ways. UI Test Patterns are able to test UI patterns across their several different implementations. Within PBGT, UI Test Patterns are classified as **Base UI Test Patterns** or **High-Order UI Test Patterns**. The Base UI Test Patterns represent the “starter kit” of UI Test Patterns that testers are able to use to test GUIs. The High-Order UI Test Patterns are seen as extensions of the Base UI Test Patterns. The higher-order UI Test Patterns are composed by two or more patterns. They allow the definition of a new set of UI Test Patterns that can be reused on different models.

Example on using UI Test Patterns

To better understand the usage of UI Test Patterns and their benefits, a practical example featuring real web applications is provided next. One recurring problem in web applications is authentication, ensuring or restricting access to certain functions or data. In order to access these protected features, users have to successfully authenticate themselves. To accomplish this, software systems feature at least two textboxes (username and password) and one submit button. The latter is embodied in a UI pattern typically referred as Login [Tox14]. This UI pattern can be implemented in different ways, in terms of layout and outcomes (Figure 5.2).

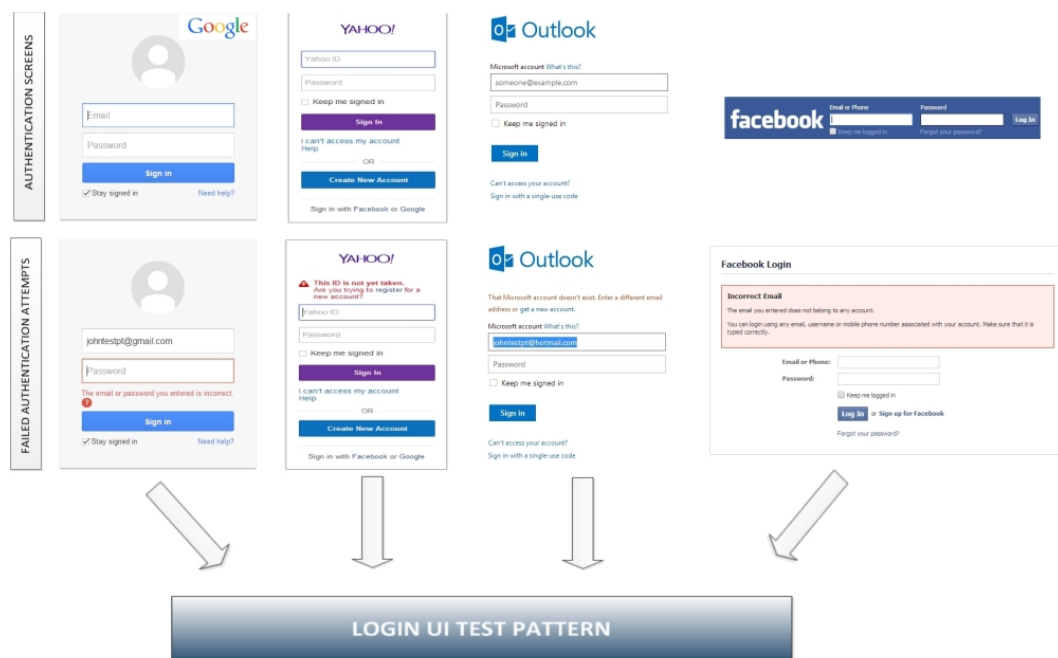


Figure 5.2: Different implementations and outcomes of the Login UI Pattern.

Upon a failed authentication the UI can display an error, indicating the reason (although) not much detailed, in the same screen or it can redirect users to a different area. The first row of Figure 5.2 displays the different implementations of the authentication mechanism, of four web applications: Gmail [Goo14], Yahoo! Mail [Yah14], Outlook [Mic14c] and Facebook [Fac14]. They have all been implemented differently. For instance, upon unsuccessful login, Gmail indicates, in the same page: “The email or password you entered is incorrect”. The remaining applications provide more detail about the reason of such failure. They indicate either login is incorrect or the provided username does not exist. Moreover, they all display the reason in the same page (one above username textbox, and others

below the password textbox), while Facebook redirects the user to a different page.

The Login UI Test Pattern is able to test the different implementations of the Login UI pattern, by providing generic test strategies with possible different configurations in order to allow testing different implementations of the Login UI pattern promoting reuse.

5.2 Methodology

Pattern Based GUI Testing (PBGT) is a new emerging model-based GUI testing approach that aims at systematizing and automating the GUI testing process, by benefiting from UI Test Patterns and therefore promoting reuse of GUI testing strategies. A UI Test Pattern provides a configurable test strategy to test a GUI that was implemented using a specific UI pattern or a set of UI patterns. Because a UI pattern may be realized using different implementations, the configurations in a UI Test Pattern provide the ability to test different implementations by setting different parameters to the testing strategy.

This methodology defines two roles: the tester and the (PBGT) developer. The tester is responsible for crafting models, featuring UI Test Patterns, to fulfill the designed testing goals for a given application. The developer is in charge of implementing new UI Test Patterns, extending the framework with additional UI Test Patterns. Moreover, the developer can also implement additional test case generation algorithms to the framework.

5.2.1 Components

The PBGT approach is supported by five components, which are illustrated in (Figure 5.3).

PARADIGM

PARADIGM is a domain specific language (DSL) for building GUI test models based on UI Test Patterns [MP14a]. The language along with the process that was followed for the development of this DSL, according to best practices is later detailed in Chapter 7, “DSL Engineering for GUI Modeling” (p. 85).

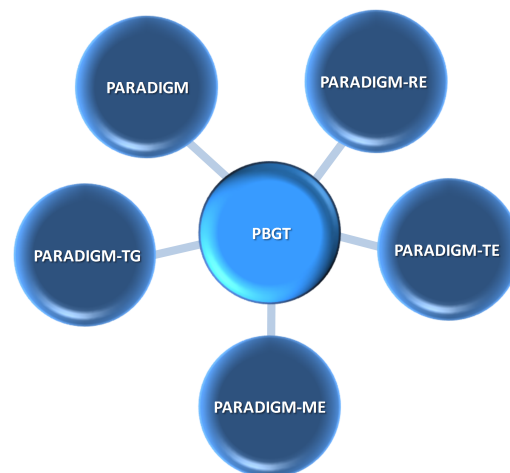


Figure 5.3: Pattern-Based GUI Testing Components.

PARADIGM-RE

This component represents a reverse engineering component whose purpose is to automatically identify UI Patterns featured on GUIs, and then extract them to facilitate the building of models for PBGT [SP14, NPF14]. These UI Patterns correspond to the language elements defined in PARADIGM. The PARADIGM-RE component does not require access to the source code. After models are generated, the tester has the ability to manually modify, adapt and tune them to fit the desired purpose.

PARADIGM-TG

The PARADIGM-TG component stands as an automated test case generation component that builds test cases from PARADIGM models [NP14]. This component provides three different test case generation techniques, which will be described in Section 5.5, “Test Case Generation” (p. 58).

PARADIGM-TE

The PARADIGM-TE is a test case execution component that executes the tests, analyzes their coverage [VP14] and returns detailed execution reports.

PARADIGM-ME

PARADIGM-ME is a modeling environment that supports the building and configuration of test models [MP13, MP14c]. Models are crafted using the elements defined in PARADIGM. The modeling environment checks for integrity errors in

the model and allows to save models in XML format. In addition, this component also provides features to extend current language elements. The latter will be explained in Section 5.3, “Tool Support” (p. 56).

5.2.2 Key Principles

PBGT is a new model-based GUI testing approach. Given this fact, it is normal that some questions emerge concerning the approach. In this regard, it is necessary to explain what are the principles of the approach along with its design decisions. Thus, the PBGT approach is founded on the following set of principles:

- PBGT is designed to leverage the concept of patterns by applying them to the test of graphical user interfaces, and thus promoting reuse.
- Each UI Test Pattern is able to test a wide range of UI Patterns.
- The UI Test Patterns were designed considering their evolution, i.e., they may be improved over time in order to address specific testing needs.
- PBGT enables free choice in the selection of test techniques to be used for gathering the input data provided by the tester.
- PBGT aims to be platform independent and does not require access to the source code of the SUT.
- The PBGT approach can be used at any phase of the software development process.
- Testers do not require programming skills in order to use the PBGT approach.
- Testers can create their own set of UI Test Patterns and promote other testers' collaboration by sharing their own customized set of patterns.
- With PBGT the testing focus is directed towards *testing goals* instead of exercising the full scale of the system. The testing goals are provided *prior* to the beginning of testing of the SUT.
- The UI Test Patterns are generic *per se* so that their name “UI Test Patterns” refer to user interfaces (like WUIs and TUIs) and not just GUIs.

5.2.3 Process

The PBGT process (illustrated in Figure 5.4) sets a total of eight main steps: (1) modeling; (2) configuration; (3) test path generation; (4) test case generation; (5) mapping; (6) test case execution; (7) results analysis and (8) model update (when required). Test path generation, test case generation and execution are fully automated steps.

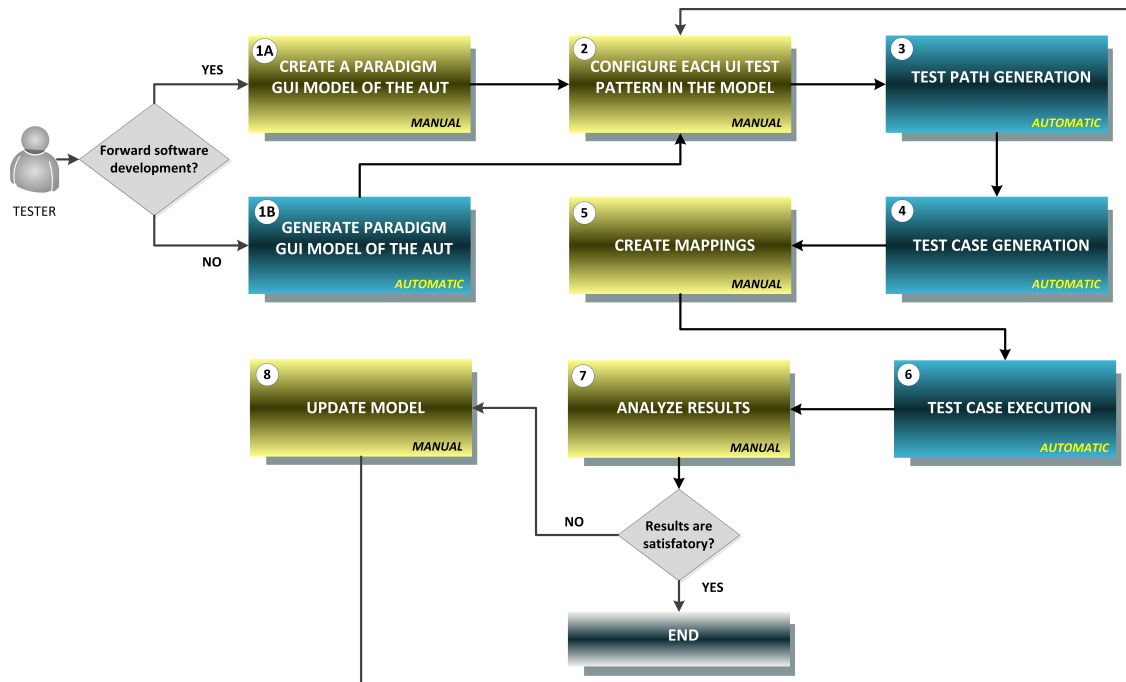


Figure 5.4: Pattern-Based GUI Testing Process.

The modeling phase can be performed manually from scratch (step 1A), in a forward software development process, or it can be performed automatically (step 1B), to obtain part of the model by a reverse engineering process from an existing software system. This is done by the PARADIGM-RE component that explores the SUT and infers the set of existing UI patterns, building, afterwards, a PARADIGM model with the UI Test Patterns that are appropriate to test them. In a forward development process, the tester may start building the application test model during requirements elicitation and evolve such model along the development.

Then, the tester configures each UI Test Pattern (step 2) with the required data (input data, preconditions, and checks to perform during test execution). Afterwards, the PARADIGM-TG component generates the test cases calculating all the paths (step 3) within the model. From those paths, test cases are generated considering the configurations provided previously by the tester (step 4).

To execute the tests, it is necessary to establish previously a mapping between

the model UI Test Patterns and the UI patterns of the SUT which will allow to identify the web elements that correspond to a certain UI Test Pattern to interact with during test case execution.

The mapping (step 5) is established by the tester through a pointing and click process. During this process the information of the web elements is saved by Selenium Web Driver [Sel14]. Besides some properties, the PBGT tool saves also images of the elements through Sikuli [SL14] and their area coordinates to cope with situations where it is not possible to identify those elements through their properties. For instance, for mapping the Login UI pattern of a certain application model, the tester must relate the login (and password) of the model with the login (and password) textbox within the SUT by clicking on it. The PBGT tool will save the information relevant to the mapping from the tester clicks: (i) textboxes IDs using Selenium Web Driver; (ii) images, in this case the image of the authentication form is saved through Sikuli (Sikuli is used when the tool is not able to identify the object by its ID); and (iii) area coordinates of the object are stored when the two previous methods fail.

Once the mapping is established, PARADIGM-TE executes the generated test cases over the SUT and produces reports with the results of the tests (step 6). When a failure is found, the tester needs to analyze (step 7) if its origin is in the SUT or in the model, and decide which updates (step 8) are necessary in order to generate and execute test cases again. In this case, when the model requires an update, some manual steps can be skipped, for instance, the mapping (step 5). Moreover, there may be situations where it is only necessary to change a mapping, skipping step 2 (configuration). If the results obtained are satisfactory to the tester (e.g., no more failures are found), the process ends.

5.2.4 Data Input

By definition, UI Test Patterns do not impose any particular technique concerning data input, so the tester has the freedom to use/select the technique he desires. For instance, one technique can be *equivalence class partition* [Mat08], in which the input domain is divided in classes selecting one value from each class, based on the principle that the behavior of the SUT will be the same for every value within the same class. Other technique can be *boundary value analysis* [CJ02], which targets to select the values that are at the border of equivalence classes, because they are expected to have more probability of finding bugs.

5.2.5 Levels of Reuse

Testing is particularly demanding, costly, difficult and requires knowledge. It is plausibly the most time consuming activity in the software development process [KA04]. Progressively the need arises to reduce the testing efforts and therefore testing costs. As pointed earlier, UI patterns have recurring behavior. To test this recurring behavior will be a very repetitive task. The latter can be captured into UI Test Patterns, which not only would allow to simplify the testing tasks, but also to test their various implementations. Such perception of using patterns for testing is, to some extent, similar to the usage of patterns in software design.

One way to profit from these recurring tasks and to reduce the testing efforts, can be achieved through reuse. The vision of software reuse is not new. It was introduced by Naur and Randell [NR68] back in 1968, in order to “*reduce the time and effort required to build and maintain high-quality software systems*” [Agu12]. Software reuse is the “*process of creating software systems from existing software rather than building software from scratch*” [Kru92]. The same idea can also be applied to testing. Testing activities that have repetitive tasks should not be required to be recreated every time from the start. Abstraction is a key part in software reuse. According to Wegner “*abstraction and reusability are two sides of the same coin*” [Weg83]. An abstraction represents a concise description suppressing the details that are irrelevant and emphasizing on the information that is important. For instance, as explained before, UI patterns have different implementations, but the testing focus is based on their recurrent behavior.

In PBGT, testing reuse can be achieved on:

- **UI Test Patterns *without* provided data for configuration.** Each UI Test Pattern has its own configuration. There might be situations where the data referred to the configuration of a given UI Test Pattern is not important for reuse. Base UI Test Patterns can be reused *per se* and/or to *produce* High-Order UI Test Patterns. These *new* High-Order UI Test Patterns can also be reused enabling, when applicable, a larger-scale reuse of the test pattern.
- **UI Test Patterns *with* provided data for configuration.** The concept is similar to what was stated before, but in this case, the UI Test Pattern can contain an already provided *generic* data configuration that can be useful to suit a more broader reuse. Alternatively, it can contain a more *specific* data configuration to test a more precise scenario.
- **PARADIGM Models.** An already crafted and configured model containing

a set of UI Test Patterns, can be reused to test other several software applications.

A more precise representation and detailed information on how to achieve the above *levels of reuse* will be explained in Chapter 7 “DSL Engineering for GUI Modeling” (p. 85).

5.3 Tool Support

The PBGT tool provides an integrated modeling and testing environment that supports the crafting of test models based on UI Test Patterns, using a GUI modeling DSL – PARADIGM. The tool is developed on top of the Eclipse Modeling Framework and is available as an Eclipse plugin.

PBGT is supported by this tool. The tool incorporates the five components described early: PARADIGM, PARADIGM-RE, PARADIGM-TG, PARADIGM-TE and PARADIGM-ME. The tool displaying the PARADIGM-ME component appears illustrated in Figure 5.5.

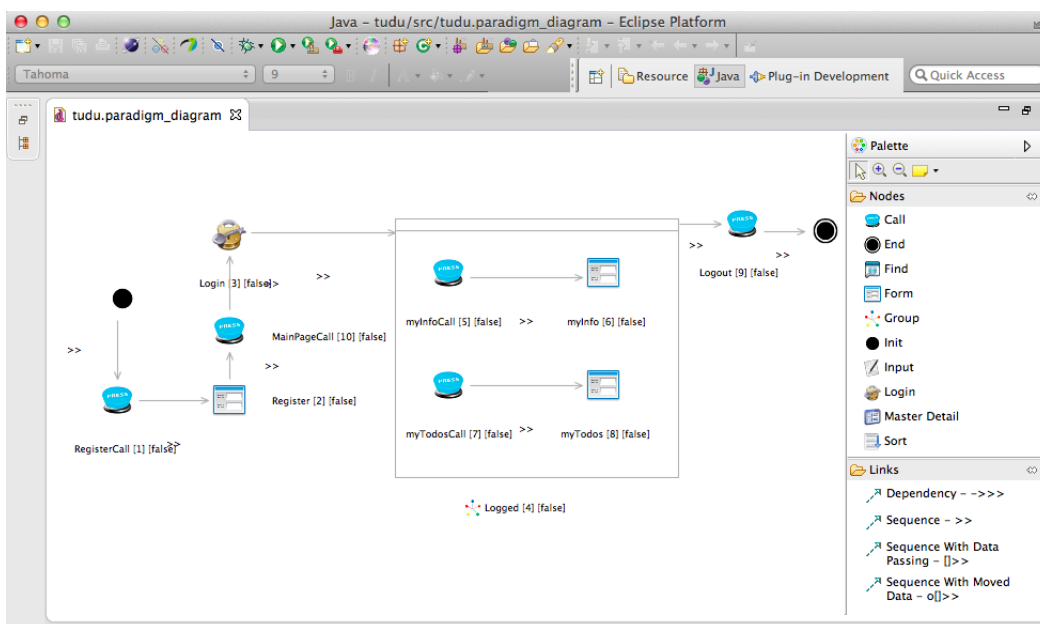


Figure 5.5: PBGT Tool featuring the PARADIGM-ME component.

The center of the Figure 5.5 shows an area where the modeling of the SUT takes place. On the right there is a *palette* that features *nodes* and *links*. The *nodes* area contains a collection of elements (described in the DSL) and the *links* area is comprised by the language connectors. The tester is able to *drag-and-drop* these elements on the area at the center. As described before, after defining a model

the tester needs to configure each node, i.e., each UI Test Pattern featured in the model. Each UI Test Pattern has its own configuration and the tester has to provide test input data via a dialog. The environment imposes a set of rules to be followed to ensure that models are correctly built.

Extensibility

Sommerville mentions that “*software development does not stop when a system is delivered but continues throughout the lifetime of the system*” [Som10]. Software evolution is always prone to happen, so as software is modified and enhanced, the testing also needs to follow the same path. The PBGT tool also provides extension points that allow its components to be updated. For instance, to incorporate a new language element – new UI Test Pattern – in PARADIGM-ME, the following process should be followed: (1) add a new element in the Domain Model; (2) create the graphical design for the element; (3) create a tool for the element and (4) specify the relationship among domain, graphical and tooling elements. Afterwards, the new element becomes available in the PARADIGM-ME component.

5.4 Application Test Model

In the PBGT approach, models are written in PARADIGM. These models reflect the testing goals that were defined for the SUTs, and are configured by testers using the PBGT tool. A model can be seen in the form of a graph (as illustrated in Figure 5.6), consisting of a set of nodes (elements) and linked by arrows (connectors).

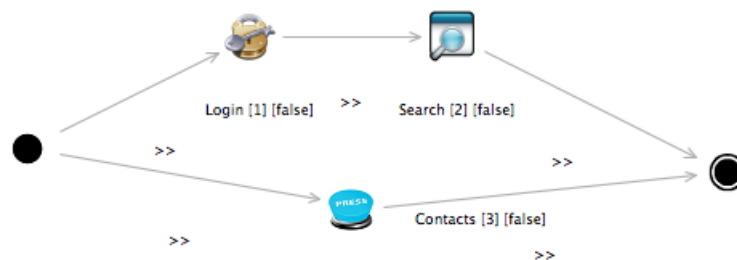


Figure 5.6: PARADIGM model of a simple web-application.

A model written in PARADIGM is always composed by two types of nodes: an *Init* node to mark the start of the model and an *End* node to indicate where the model terminates. These are mandatory nodes. Between these nodes, it is possible

to have other type of elements, such as UI Test Patterns, *Groups* and *Forms*. These elements can be mandatory or optional. Each UI Test Pattern contains a specific ID, which is a number that is used to identify the element in the model. *Forms* and *Groups* allow to structure a model. A *Group* contains a set of elements that can be performed in any order and can also be optional. A *Form* contains another PARADIGM model inside. This type of element is mainly used for structural purposes in order to simplify the viewing and modeling of the SUT.

Figure 5.6 illustrates a simple model of a web application. This model is comprised by three UI Test Patterns that have the purpose of testing the functional correctness of a given SUT. The first element, the Login UI Test Pattern (referred as number 1 in Figure 5.6) aims to verify user authentication. If valid credentials are provided, the system will redirect the end-user to another page, enabling the feature to perform a search. However, if the login credentials are invalid then the end-user will remain in the same page. The configuration for this UI Test Pattern, is illustrated in Figure 5.7. The behavior of the search can be tested using another UI Test Pattern (Find UI Test Pattern represented with number 2 in Figure 5.6). Alternatively, the end-user can go directly to the page “Contacts” (represented by the Call UI Test Pattern number 3 in Figure 5.6), without having to provide credentials.

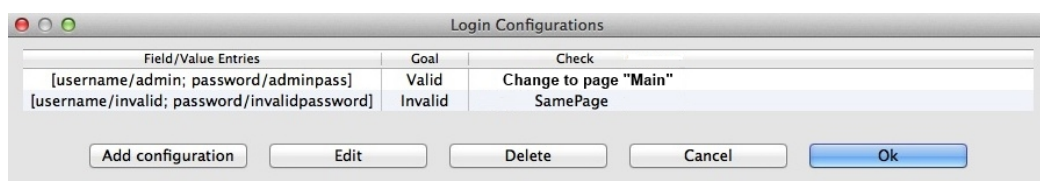


Figure 5.7: Configurations for the Login UI Test Pattern (number 1) from Figure 5.6.

The whole collection of UI Test Patterns (including the ones mentioned above) will be described and analyzed in more detail in Chapter 6 “Pattern Library” (p. 63).

5.5 Test Case Generation

The generation of test cases from models is considered a challenge as it represents a considerable impact on the effectiveness of the overall test. In the context of PBGT, test cases are generated automatically from a PARADIGM model. In order to obtain these test cases, a few steps are required to be followed, as will be described next. The first step, consists in extracting all UI Test Patterns, by recursively

expanding all structural elements (*Forms* and *Groups*) from the model, leaving only UI Test Patterns and connectors. Afterwards, the PARADIGM-TG component [NP14] will generate all possible paths that traverse the model, starting from the *Init* element to the *End* element. A test path represents a sequence of UI Test Patterns. Considering the model from Figure 5.6, the test paths are [1, 2] and [3], where 1, 2 and 3 are the IDs of the UI Test Patterns within the model.

There are some characteristics within PARADIGM models that should be discussed. As mentioned earlier, there are elements that can be optional. Given this case, the PARADIGM-TG component will generate test paths that include those elements as well as test paths that do not include them. Further, these elements can belong to *Groups*, and in this case, these elements can be executed arbitrarily. The component will generate several test paths that cover different permutations of those elements. In respects to the *Group* element, the maximum number of test paths is given according to the following formulas [MPM13].

Definition 2. A Group with N loose mandatory elements, will generate at most the following test paths.

$$N!$$

Definition 3. A Group with M loose optional elements, will generate at most the following test paths.

$$\sum_{i=M-1}^M i! \cdot {}^M C_i + 1$$

Definition 4. A Group with MT mandatory and OP optional loose elements will generate at most the following test paths.

$$MT! + \sum_{i=OP-1}^{OP} (MT + i)! \cdot {}^{OP} C_i$$

Definition 5. Two connected Groups g_1 and g_2 with t_1 and t_2 number of test paths each, will generate at most (if pre-condition of internal elements is True) the following test paths.

$$t_1 \cdot t_2$$

Definition 6. Overall, a set of M test paths, where some of them (N) have a Group with T number of test paths will generate at most the following test paths.

$$(M - N) + N \cdot T$$

Considering for all permutations of elements within groups and further permutations for samples with different lengths when that group has optional

elements gives rise to many test paths. For example, in the case of N optional elements within a Group, it could be possible to generate test cases for samples with 2 elements, 3 elements, .. , $N-1$ elements. In the formulas above, only permutations over samples of $N-1$ elements were considered. Yet it is observable that, even for a small model and in case of preconditions true, the number of test cases generated is enormous.

Either due to the specificity of the testing goals or to the size of model (which can also be caused by the testing goals), the number of test paths can be considerably high. There will be situations where the tester will only require to work with a smaller set of test paths in order to pursue a particular testing goal. To facilitate this demand, it is possible to define an upper limit for the number of test paths generated. If the model has cycles, it is possible to limit the maximum number of times that the cycle is executed. More detailed information about test case generation for PBGT can be found in [NP14].

In situations where the number of generated test cases is massive, causing impact on the generation and execution of the test case, the PARADIGM-TG component provides a collection of test case generation algorithms [NP14]:

- **Default.** Only generates N test cases per test path, where N is defined by the tester;
- **Invalid Configurations.** Only generates test cases that traverse invalid configurations, for example, it will contain test cases that check the behavior of invalid authentications within Login UI Test Pattern;
- **Random.** Generates test cases arbitrarily selecting the configurations of the UI Test Patterns within randomly selected test paths. The algorithm stops after a predetermined time interval.

Furthermore, it is also possible to set some parameters to adapt the test case generation algorithm to the intended test goals, such as [NP14]:

- **Cycles.** When the model has cycles, the number of test paths generated may be infinite. In that case the tester can define the upper limit of times a cycle can be executed.
- **Mandatory and Exclusion Elements.** In case the tester wants to test a certain feature or a certain workflow that can only be reached after crossing a certain path, he can explicitly define the mandatory and exclusion elements. By excluding one element (or a sequence of elements), every Test Path that contains that element will not be generated.

- **Specific UI Test Patterns configurations.** When the tester wants to select a subset of configurations to include in the generated test cases. For example, when only a part of the functionality of the application is implemented. All the other configurations will be ignored.
- **Number of test paths and test cases.** The tester defines the maximum number of test paths and test cases to generate from the model.

After the selection of a test case generation algorithm or after tuning the test case generation algorithm, it is possible to generate test cases and execute them over a GUI under test. The generated test cases are stored in a XML file. Considering the model described in Figure 5.6, the generated test case for the test path [1, 2] is illustrated in Figure 5.8. This path checks a valid authentication for the Login UI Test Pattern (element number 1 in Figure 5.6), and then checks the number of results obtained by searching for “Computer Science” (element number 2 in Figure 5.6).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Script>
<Path value="[1, 2]">
<Init/>
<Login flag="false" name="Login" number="1" optional="false" validity="Valid">
  <Check check="ChangePage" result=""/>
  <Value field="username_" fieldName="username" value="admin"/>
  <Value field="password_" fieldName="password" value="adminpass"/>
  <Button field="button_1"/>
</Login>
<Find flag="false" name="Search" number="2" optional="false">
  <Check check="NumberOfResults_more_than" position="1" result="50"/>
  <Value field="Name_2" fieldName="Name" value="Computer Science"/>
</Find>
</Path>
</Script>
```

Figure 5.8: Description of a test case in XML format.

5.6 Summary

GUIs feature UI patterns. UI patterns represent generic graphical implementations of a given functionality and they can have multiple implementations. GUIs that are similar in design, i.e., based on the same UI pattern, should share a common testing strategy. A UI test pattern provides a configurable test strategy to test a GUI that was implemented using a specific UI pattern or a set of UI patterns. PBGT is a new model-based GUI testing paradigm that aims at promoting reuse of GUI testing strategies, by systematizing and automating the GUI testing process. The approach does not require access to the source-code of the SUT and is platform independent. The PBGT approach is supported by a total of five components, namely: a DSL; a reverse engineering component; an automated test

case generation component; a test case execution component and; a modeling environment. All components are embedded in an integrated environment called PBGT tool. The PBGT process defines a total of eight steps, where three are fully automated steps. The PBGT approach can be used at any phase of the software development process. Reuse can be achieved on several levels in the PBGT approach.

6

Pattern Library

For more than a decade, patterns have influenced software architects and developers in creating computing systems [BHS07]. Several systems have been developed with the conscious awareness of patterns. The notion of patterns was introduced in the field of civil architecture by Christopher Alexander et al [AIS77]. They explained the nature of patterns as follows:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

The above definition works perfectly for software as well. In addition, patterns can be applied to selectively address specific challenges and problems with a recurrent nature. Thus, patterns are useful and express instances of good design practices while embodying high-level principles and strategies.

A key part of patterns is that they are rooted in practice. In addition, the usage of patterns provides several benefits. First, they provide proven design solutions and guidance for their use. Patterns explain how a problem can be solved and why the solution is appropriate for a particular situation. Second, patterns grant process improvement. Identifying patterns and cataloging them, can increase

productivity by reducing the time spent on “reinventing the wheel”. Furthermore, if user interface components are built for patterns, designs can be developed, tested and iterated rapidly. Third, patterns share a common language. They help support and improve communication among practitioners from diverse disciplines, by developing a vocabulary when explaining solutions. Finally, patterns are able to increase confidence in software systems. Patterns are based on a history of successful usage and implementation guidance.

This chapter describes a collection of UI Test Patterns to be used in the context of PBGT. The UI Test Patterns have been designed after researching generic testing strategies for the most common UI patterns, from various sources [Tid11, Nei14, BZ14a, BZ14b, Yah12, Pat14, Tox14, vW08, IDT08].

6.1 Pattern Format

Patterns can be described in several styles [AIS77, GHJV94, Mar03, MD96]. The adoption of a given style depends on the subject and desired purpose. The description of Base UI Test Patterns that are going to be introduced next, has the goal to concentrate the information necessary to start using the patterns, *when* to use them, *how* to use them, *why* they should be used and finally *what* they should be used for. Therefore, the patterns will be presented having in mind simplicity and comprehension, according to the following structure, with guidance from [MD96] and an adaptation (according to feedback gathered from experts on the field of patterns at EuroPLoP 2014 [MP14b]):

- **Pattern Name:** unique identifier to shortly refer the pattern;
- **Context:** situation where the problem occurs;
- **Problem:** description of the problem addressed by the pattern;
- **Forces:** reflections to consider when choosing a solution to the problem;
- **Solution:** description of the proposed solution for the pattern;
- **Consequences:** positive and negative consequences that arise from the solution;
- **Application Candidates:** real conditions (UI Patterns) where the (UI Test) patterns can be applied;

- **Known Uses:** applications (web and mobile) where the patterns have been successfully applied;
- **Example:** concrete example of the applicability of the pattern.

6.2 Considerations

This pattern language aims to be platform independent. It is very flexible, since it can be customized/configured to model and test any UI. The patterns that will be detailed below, are instances of the formal definition (described in Chapter “Pattern-Based GUI Testing”, Definition 1, on p. 47). There are forces and also consequences that will be similar among patterns. For this reason, the contents of which are prone to be similar will be described in the following paragraphs and not individually for each pattern.

There are several **forces** that influence the problem of testing GUIs that feature UI patterns that can be implemented in different ways, and actually contribute to making the problem harder to solve. One refers to the capability/criteria of identifying GUIs recurrent behavior, along with the UI Patterns recognition featured in the GUI. Each UI Pattern can have several different implementations, and therefore the set of checks to perform and cover/verify can be high. The quality of the tester contributes to the correct pattern recognition, usage and expected results.

The formal definition of the UI Test Pattern is a solution to the problem stated before. Thus, several **consequences** emerge from this solution. UI Test Patterns can be reused, during the modeling and testing process. Due to this high reusability, modeling and testing efforts are reduced allowing to save costs in GUI testing. The input data is not restricted, since the tester has the freedom to select the technique he desires, such as equivalence class partition, among others. The focus of testing is directed towards goals, while with other MBT approaches, the focus is on modeling the behavior of the application, which can often lead to the well known *state explosion problem* [EFW01].

6.3 Input UI Test Pattern

Context

GUIs often feature input fields to gather data from the user. Input can be obtained via different ways, depending on the implementation, for instance: dialog windows, textboxes, drop-down menus and window prompts.

GUIs vary in implementation in terms of validating input data. Some GUIs display a message when the input is not valid, while others do not allow invalid data input. For instance, in some situations, selecting dates can be performed using timepickers. In this case, it is not possible to have an invalid data since we are selecting one specific date (from a set of options) instead of having to write it manually. However, the tester can check both situations (check for valid and check for invalid data) or check if the error message is displayed. In generic terms, testers aim to verify the behavior of input fields, by providing valid and invalid data, to check how the GUI responds to such input variations. Since the input can be gathered in several ways, the tester should have a generic, unique and reusable way to provide input data, in order to test the behavior of input fields, across their different implementations.

Problem

How to define a test strategy that can be reused for testing the input functionality over its different possible implementations?

Forces

- One single field can include numeric, text and alphanumeric types combined at once
- Wide range of error messages upon invalid entries
- Validation rules can be applied to more than one field (group) at once
- Conversion errors might occur upon invalid entries

Solution

Provide a generic test strategy for GUIs that feature input fields or drop-down menus, across their different implementations. The test strategy consists in the following:

1. Test **Goals**: “Valid data” (INP_VD) and “Invalid data” (INP_ID);
2. Set of variables **V**: {input};
3. Sequence of actions **A**: [provide *input*].
4. Set of checks **C**: {“message box X”, “label Y”, “error provider Z”} where X, Y and Z correspond to the text to be displayed;

During configuration, the user has to provide valid input data for INP_VD (and invalid input data for INP_ID), select the checks to perform and define the precondition.

Consequences

- No additional validation rules are required to be written (just need to be selected).
- A wide range of implementations can be tested within a simple configuration step.

Application Candidates

- Input Prompt [Tox14, Pat14, Tid11]
- Forgiving Format [Tox14, Tid11]
- Input Controls [Pat14]

Known Uses

- Mobile.de [mob14]
- iAddressBook [Wac14]
- Australian-charts.com [Hun14a]

- Italian charts portal [Hun14b]
- Professional Calendar/Agenda [Men14]

Example

A tester aims to verify the behavior of input fields (Figure 6.1) for valid and invalid input data. In this case, with the Input UI Test Pattern, the tester is able to verify the correctness of the GUI.

Degrees of freedom

t score

Cumulative probability: $P(T \leq t)$

Figure 6.1: UI Part featuring the input functionality from Stattek – Online Statistical Table [Sta14].

Possible configurations for the Input UI Test Pattern could be:

Goal – Valid data

V : { [Probability, 0.5] }

A : [Provide *Probability*]

C : { }

P : True

In this configuration, the tester provides a valid input, since the probability is expressed within the range 0 to 1, and the tester provides value 0.5.

Goal – Invalid data

V : { [Probability, 1.4] }

A : [Provide *Probability*]

C : { Label “ERROR: Probability must be between 0 and 1.” }

P : True

The tester wants to verify if an error is displayed (in a label above the textbox) when providing invalid data – value 1.4 for the probability field.

6.4 Login UI Test Pattern

Context

Numerous software systems have restricted functionalities that are only available after a successful authentication. Typically, users have to provide a username and a password.

Testers target to verify the behavior of the authentication functionality, where the objective is to check if it is possible to authenticate in the system with a valid username/password and check if it is not possible to authenticate otherwise.

Problem

How to define a test strategy that can be reused for testing authentication functionality over its different possible implementations?

Forces

- The implementation of the authentication functionality varies significantly.
- High number of checks to cover.
- The need to provide combinations of data input to enable checking the expected behavior (for valid and invalid login).

Solution

Provide a generic test strategy for GUIs that feature authentication mechanisms. The test strategy consists in the following:

1. Test **Goals**: “Valid login” and “Invalid login”;
2. Set of variables **V**: {username, password};
3. Sequence of actions **A**: [provide *username*; provide *password*; press *submit*];
4. Set of checks **C**: {“change to page X”, “pop-up error Y”, “same page”, “label K”}, where X is the target page and Y, K the message to be displayed.

During configuration the user has to provide valid username/password input data for “Valid Login” (and invalid username/password for “Invalid Login”), select the checks to perform and define the precondition.

Consequences

- After a simple configuration step it is possible to quickly verify the expected behavior.
- A wide range of software applications featuring an authentication mechanism can be verified in short time.

Application Candidates

- Log In [Tox14] (referred as *Login* in [Ras14]).

Known Uses

- TaskFreak [Ozi14]
- Mobile.de [mob14]
- Tudu Lists [Dub13]
- Australian-charts.com [Hun14a]
- Italian charts portal [Hun14b]

Example

A tester aims to verify the functional correctness of the GUI, that features an authentication mechanism (Log In UI Pattern as displayed in Figure 6.2). In this case, with the Login UI Test Pattern the tester is able to verify the correct behavior of the GUI. Thus, the testing goals are: test the authentication for (1) valid and (2) invalid Username/Password.

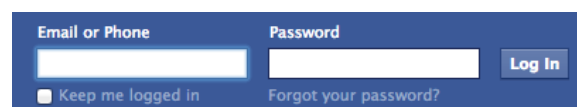


Figure 6.2: UI Part featuring the authentication functionality from Facebook[Fac14]

Possible configurations for the Login UI Test Pattern could be:

Goal – Valid login

V : { [EmailOrPhone, "JohnTestPT@gmail.com"], [Password, "john56532"] };

A : [Provide EmailOrPhone, Provide Password, Press Log In];

C : {Change to page "Welcome"};

P : True.

In this configuration, the tester provides a valid username and valid password. The tester wants to verify if upon successful authentication, the application redirects the user to another page.

Goal – Invalid login

V : {[Login, "JohnTestPT@gmail.com"], [Password, "test"]};

A : [Provide Login, Provide Password, Press Submit];

C : {Label "Please re-enter your password. The password you entered is incorrect. Please try again (make sure your caps lock is off)."};

P : True.

This configuration is different from the previous one, since the tester provides a valid username but an invalid password. The tester desires to verify the behavior for an invalid login, i.e, if the user is redirected to a different page upon unsuccessful login.

6.5 Master/Detail UI Test Pattern

Context

Depending on the situation, GUIs have a set of items to be displayed (they represent an area called *master*), where each item has associated content (detail). The detail is only visible upon selection of items from the master.

Testers want to verify the behavior of two related objects in a GUI (master and detail), where the idea is to check if changing the master's value correctly updates the contents of the detail.

Problem

How to define a test strategy that can be reused for testing Master/Detail behavior over its different possible implementations?

Forces

- The Master/Detail behavior might not be easy to identify within a GUI.
- There are implementations with Master/Details arranged in hierarchy.

Solution

Provide a generic test strategy for GUIs that feature Master/Detail implementations (two related objects). The test strategy consists in the following:

1. Test **Goal**: “Change master” (MD);
2. Set of variables **V**: {master, [detail]};
3. Sequence of actions **A**: [select *master*];
4. Set of checks **C**: {“detail has value(s) *X*”, “detail does not have value(s) *X*”, “detail is empty”, “detail has *N* elements”}.

During configuration the user has to provide master input data for MD, select the checks to perform and define the precondition.

Consequences

- A simple and fast configuration allows to verify the expected behavior.
- Possibility to check Master/Detail behavior structured in hierarchy.

Application Candidates

- Two-panel selector [Tid11];
- Breadcrumbs [Ras14, Tox14, Tid11, Pat14];
- Tab Bars [Ras14] (referred as *Tabs* in [Yah12]; referred as *Navigation Tabs* in [Tox14]);

- Sequence Map [Tid11];
- Accordion [Tid11, Yah12, Tox14];
- Collapsible Panels [vW08];
- Details on Demand [vW08].

Known Uses

- Australian-charts.com [Hun14a]
- Mobile.de [mob14]
- iAddressBook [Wac14]

Example

A tester aims to verify the functional correctness of the GUI, that features a Master/Detail area (Figure 6.3). The tester will use the Master/Detail UI Test Pattern to verify the correctness of the GUI. The aim is to check if changing the value of Make, the Model set of values changes accordingly.

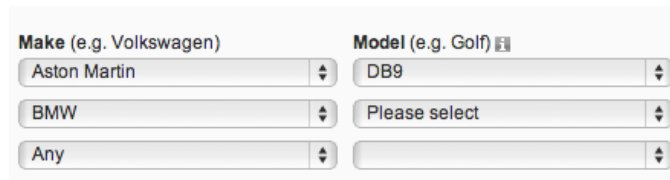


Figure 6.3: UI Part of Mobile.de[mob14] that features the Master/Detail area.

One possible configuration that the tester could do:

Goal – Change Master

V : { [Make, “Aston Martin”], [Model, “DB9”] };

A : [Select Make];

C : {“detail has value DB9” };

P : True.

For the above configuration the result would be evaluated to True, since DB9 is one model from Aston Martin.

6.6 Find UI Test Pattern

Context

GUIs provide search engines to find information. Some implementations show the result obtained while the user is typing. Others present the result only at the end of the input after submission.

Testers target to verify the behavior of the search functionality, in order to check if the result of a search is as expected, i.e., if it find the right set of values.

Problem

How to define a test strategy that can be reused for testing the search functionality over its different possible implementations?

Forces

- Search can include special characters.
- Search can have a minimum and maximum characters limit.
- In some implementations blank spaces are ignored except if they appear between characters.

Solution

Provide a generic test strategy for GUIs that feature search functionalities. The test strategy consists in the following:

1. Test **Goals**: “Value found” (FND_VF) and “Value not found” (FND_NF);
2. Set of variables **V**: $\{v_1, \dots, v_N\}$ where N is defined during configuration time by the user/tester;
3. Sequence of actions **A**: [provide v_1, \dots , provide v_N];
4. Set of checks **C**: {“result is an empty set”, “result has X elements”, “results do not have element X ”, “result has element X in line Y ”, “results have more than X elements”, “results have less than X elements”}.

During configuration, the user has to define the value for N , provide input data for variables v_1, \dots, v_N , select the check to perform and define the precondition.

Consequences

- No additional rules/logic (including string parsing) are required to be written (checks just need to be selected).
- Provides a simple, fast and practical way to verify the expected behavior.

Application Candidates

- Auto Complete [Tox14, Pat14, vW08] (referred as *Auto Completion* in [Tid11]);
- Search [Ras14, Tid11];
- Fill In The Blanks [Tox14, Tid11];
- Table Filter [Tox14];
- Advanced Search [vW08];
- Search Box [vW08];
- Search Area [vW08].

Known Uses

- Australian-charts.com [Hun14a]
- Mobile.de [mob14]
- iAddressBook [Wac14]

Example

A tester aims to verify the functional correctness of the GUI, that features a search box (Figure 6.4). The tester will use the Find UI Test Pattern to verify the correctness of the GUI. He aims to check to find it is possible to find the right set of news in the news' archive. Hence, the testing goals are: test the search for (1) value found and (2) value not found.



Figure 6.4: UI Part of GUpdate search news functionality[GPU14].

Possible configurations that the tester could perform:

Goal – Value found

V : { [SearchFor, “Ferrari”] }

A : [Provide *data*]

C : {“result has 12103 elements”}

P : True

With this configuration, the tester aims to check if there are 12103 news related with “Ferrari” cars. The test passes because it is exactly the number of news obtained.

Goal – Value not found

V : { [SearchFor, “Monaco”] }

A : [Provide *data*]

C : {“result has 3412 elements”}

P : True

In this configuration the tester wants to verify if there are 3412 news related with “Monaco”. The test fails because the result set has 3277 news.

6.7 Sort UI Test Pattern

Context

When displaying a set of results, the user is able to sort them according to a given criteria (ascending or descending).

The tester’s target is to verify the behavior of the sort functionality, in order to check if the result (of a sort action) is ordered accordingly to the chosen sort criterion. In some implementations the ordering is only performed over a specific field. Yet, other implementations allow to define several fields for ordering.

Problem

How to define a test strategy that can be reused for testing sort functionality over its different possible implementations?

Forces

- Wide range of sort implementations (by field or set of fields, for instance)
- Searching for elements within a sorted list can be realized using specific algorithms such as Binary Search, Quicksort, among others

Solution

Provide a generic test strategy for GUIs that feature sort mechanisms. The test strategy consists in the following:

1. Test **Goals**: “ascending” (SRT_ASC) and “descending” (SRT_DESC);
2. Set of variables **V**: $\{(v_1, c_1), \dots, (v_N, c_N)\}$ where N is defined by the user/tester during configuration time and “ c_i ” represents the criteria defined for the given variable “ v_i ”;
3. Sequence of actions **A**: [provide $(v_1, c_1), \dots$, provide (v_N, c_1)];
4. Set of checks **C**: {“element from field X in position Y has value Z ”, “elements (v) with a given criteria (c) are sorted accordingly”, “elements with value X is before element with value Y ”}.

During configuration, the user has to define the value for N , provide input data for the pair variables and criteria $(v_1, c_1), \dots, (v_N, c_N)$, select the check to perform and define the precondition.

Consequences

- The tester has only to indicate the elements and the related sort criteria to test the sort behavior
- The criteria can be easily modified, only requiring to select a different sort criterion.
- The tester does not need to implement the code that verifies the position of a given element, or the check if a list is sorted.

Application Candidates

- Sort by Column [Tox14] (referred as Sortable Columns in [IDT08])
- Table Sorter [Pat14, vW08] (referred as Sortable Table in [Tid11])

Known Uses

- Mobile.de [mob14]
- TaskFreak [Ozi14]

Example

A tester aims to verify the functional correctness of the GUI, that features a sort functionality (Figure 6.5). The tester will use the Sort UI Test Pattern to verify the correctness of the GUI. Thus, the testing goals are: “Ascending” and “Descending”.

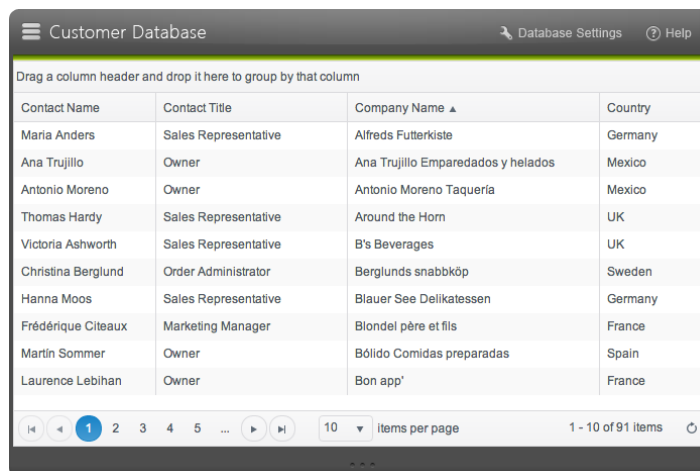


Figure 6.5: UI Part of Telerik[Tel14] that features a sort functionality.

One possible configuration that the tester could do:

Goal – Ascending

V : { (“Company Name”, ascending) };

A : [Provide (*Company Name*, *ascending*)];

C : {“element from field *Company Name* in position 4 has value *Alfreds Futterkiste*”};

P : True.

The previous configuration returns the result False. When ascending the column “Company Name” the value “Alfreds Futterkiste” does not appear in position 4.

Goal – Descending

V : { (“Company Name”, descending)};

A : [Provide (*Company Name*, *ascending*)];

C : {“element from field *Company Name* in position 2 has value *Alfreds Futterkiste*”};

P : True.

In this configuration the goal is to verify if elements are ordered in descending criteria. The value “Alfreds Futterkiste” does not appear in position 2. This configuration is evaluated to False.

6.8 Call UI Test Pattern

Context

Users trigger actions in GUIs. For instance, to register an account, users have to provide input and then, to complete the process, they have to submit the data to the system (press submit button).

Testers aim to check the functionality of a corresponding invocation.

Problem

How to define a test strategy that can be reused for testing the invocation/call functionality over its different possible implementations?

Forces

- Should facilitate checking the expected behavior for a call action.
- Should provide an intuitive and simple configuration (call succeeded and call failed).

Solution

The test strategy consists in the following:

1. Test **Goal**: “Action invoked” (CL_AS);
2. Set of variables **V**: no input data is involved;
3. Sequence of actions **A**: [press];
4. Set of checks **C**: {“pop-up message”, “stay in the same page”, “change to page X”}.

During configuration, the user has only to select the check to perform and define the precondition.

Consequences

This is an auxiliary pattern to assist the invocation of a given action.

Application Candidates

Due to the *atomic* nature of this UI Test Pattern, it can be applied practically everywhere.

Known Uses

- Mobile.de [mob14]
- TaskFreak [Ozi14]
- Australian-charts.com [Hun14a]
- Tudu Lists [Dub13]
- iAddressBook [Wac14]
- Professional Calendar/Agenda [Men14]
- The Address Book [HH13]
- zkCalendar [Cor14]

Example

A tester aims to verify the functional correctness of the GUI, by checking the behavior of pressing the *Show results* button (Figure 6.6). The tester will use the Call UI Test Pattern to verify the correctness of the GUI. Thus, the testing goal is: “Action invoked”.

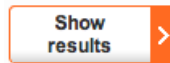


Figure 6.6: UI Part of Mobile.de [mob14] that features an action control.

One possible configuration for this UI Test Pattern could be:

Goal – Action Invoked

V : { };

A : [press];

C : {“change to page My car search”};

P : True.

In this configuration, the tester wants to verify if upon action invoked, causes a change to a different page.

6.9 Option Set UI Test Pattern

Context

Given a set of choices, the users may select multiple elements in a GUI. Testers aim to check the functionality corresponding to multiple selections.

Problem

How to define a test strategy that can be reused for testing the multiple selection functionality over its different possible implementations?

Forces

- Should facilitate checking the expected behavior for multiple selections (selection succeeded and selection failed);
- Should provide an intuitive and simple configuration (for selection succeeded and selection failed).

Solution

The test strategy consists in the following:

1. Test **Goal**: “Selection invoked” (OST_OK);
2. Set of variables **V**: $\{v_1, \dots, v_K\}$ belonging to the set of possible selections;
3. Sequence of actions **A**: [select v_1, \dots , select v_K];
4. Set of checks **C**: {“pop-up message”, “stay in the same page”, “change to page X ”}.

During configuration, the user has to define the number of options (k), provide the k values to select, select the checks to perform and define precondition.

Consequences

- Possibility to check multi-selection expected behavior in a single step.

Application Candidates

- Live Filter [Tox14];
- Array of N checkboxes [Tid11];
- Array of N toggle buttons [Tid11];
- Multiple-selection list or table [Tid11];
- List with checkbox items [Tid11].

Known Uses

- Tudu Lists [Dub13]
- iAddressBook [Wac14]

Example

The tester aims to check the behavior after selecting a subset of the options displayed in Figure 6.7. The tester will use the Option Set UI Test Pattern to verify the correctness of the GUI. Thus, the testing goal is: “Selection invoked”.

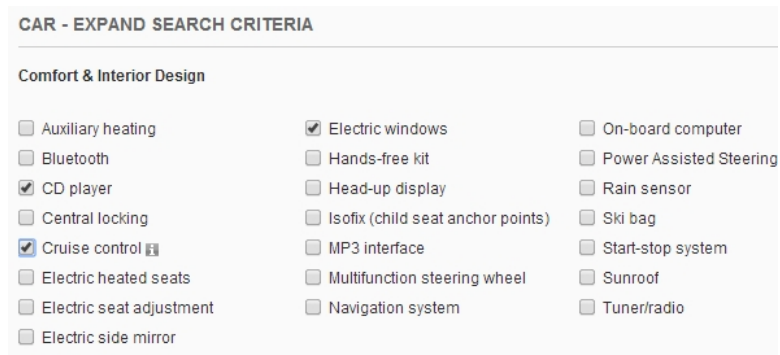


Figure 6.7: UI Part of Mobile.de [mob14] that features a control with multiple options.

One possible configuration for the Option Set UI Test Pattern could be:

Goal – Selection Invoked

V : { CD Player, Cruise Control, Electric Windows }

A : [select *CD Player*, select *Cruise Control*, select *Electric Windows*, press *Show results*]

C : {stay in the same page}

P : True

In this configuration, the tester provides CD Player, Cruise Control and Electric Windows. He wants to verify if the application remains in the same page. This configuration will be evaluated to True, since the application remains in the same page.

6.10 Summary

This chapter presented a collection of seven patterns, i.e., UI Test Patterns to be used in the context of PBGT. These patterns describe when they should be used, how to use them, why they should be used and what they should be used for. The

notion of UI Test Pattern is formally defined and describes a configurable generic test strategy with the purpose of testing multiple implementations of given UI patterns. Some of UI Test Patterns have already been presented and evaluated by the patterns community at EuroPloP 2014. Their feedback was incorporated in the overall pattern language. The UI Test Patterns represent the behavioral elements in PARADIGM (DSL). The next chapter describes the activities and guidelines that were put into practice for the development of the DSL. Moreover, it also introduces a novel approach for finding and tuning language constraints.

7

DSL Engineering for GUI Modeling

In general, domain-specific languages are high-level languages exclusively tailored to specific tasks. They are designed to make their users effective in a specific domain. They are important because they represent a more natural, robust, precise, and maintainable way of capturing the essence of a given problem, rather than merely being expressed in a general-purpose language (GPL).

The need to express specific domain features, along with the desire to express them using paradigms familiar to the domain experts, leads to the demand for DSLs. DSLs are languages tailored to a specific application domain [MHS05]. However, DSLs are, in general, difficult to design, implement and maintain. Developing DSLs from scratch represent a laborous task. Moreover, when/if a DSL grows in complexity and size, its design becomes more error-prone. Yet, the DSL development process is supported by a set of tools that are increasingly improving, and hence reducing the effort required to build such DSLs.

This chapter describes PARADIGM, a DSL to be used in the context of PBGT. It describes the activities followed during its development, according to best practices and a set of guidelines. Further, it describes tools to assist the development of DSLs and introduces a novel approach concerning the usage of Alloy in DSL engineering as means to find and tune language constraints. The chapter also discusses examples on how to attain reuse using the PBGT approach.

7.1 Domain Specific Language Engineering

DSL have been gaining importance in Software Engineering. Quite often DSLs are compared to general-purpose languages (GPLs). Within a limited domain, DSLs provide more expressiveness but lose generality. DSLs provide notations that suit a particular domain that offer more expressiveness and ease of use when compared to GPLs [MHS05]. Hence, they boost productivity and reduced maintenance costs. Also, the need for domain expertise is greatly reduced. Therefore, DSLs have a more broader openness to software developers compared to GPLs.

DSLs can be classified in two categories: *internal* and *external* DSLs. An *internal* DSL is represented within the syntax of a general-purpose language (GPL) [Fow10]. It uses the syntactic elements and can access all features of the host language. The set of tools – editor, debugger and compiler – included in the host language or platform can be directly used. An *external* DSL is a domain-specific language that is defined in a different format than the intended target language [SZ09]. This language can use all kinds of syntax, independently of other languages. External DSLs provide flexibility to DSL designers as they have the freedom to define the syntax that best fits their purposes. The syntax can either be textual or graphical. External DSLs are not bound to specific platforms or host languages. They can be targeted to different platforms through transformation mechanisms.

7.1.1 Benefits

The use of DSL can provide several profits, such as [VBD⁺13]:

- **Productivity.** Once the language is in place to handle a specific task, the work becomes more efficient, as the manual work is no longer required.
- **Quality.** Due to the reduced bugs, better architectural conformance and increased maintainability, the quality of the resulting product increases. If the DSL is engineered the right way, there will be avoidance of duplicated code and better automation of repetitive work. The language will only allow the construction of correct programs.
- **Validation and verification.** DSLs gather domain abstractions without mixing with implementation concerns. Verification is facilitated and error messages are more expressive as they use domain concepts. Manual review

and validation is more efficient, since the domain-specific aspects are well-defined and clear.

- **Data Longevity.** Models are expressed in a particular level of abstraction of a given domain. They are independent of implementation techniques. Models can be migrated to other representations.
- **Improved Communication.** DSLs allow to remove complexity focusing on what is essential, simplifying the team communication. The crafting of the language can improve understanding of the domain for which the DSL is being built.

7.1.2 Challenges

There are factors that complicate the decision to develop a new DSL. Initially, it is complex to realize that a DSL might be useful or that developing a new one might be the right track. Voelter et al [VBD⁺13] point a few perspectives:

- **Development Efforts.** When DSLs are required to be developed within a project, the effort of creating a DSL will need to be considered along with the costs associated. However, DSLs have potential to be reused, so the investment can be justified. Modern tools aim to reduce the effort required to build DSLs, which facilitates its usage in other projects.
- **Skills and expertise.** DSL development is hard. To build a DSL requires knowledge and in a certain way, experience. Also, it requires both domain and language development expertise [MHS05]. If the DSL is designed and developed from scratch, if not designed carefully, it can lead to incoherent designs.
- **Evolution and maintenance.** People, cost, time and skills need to be addressed for the maintenance phase. If the language is not maintained and if it does not evolve, it will become outdated and eventually forgotten.
- **Tool lock.** DSLs will eventually get *locked* into a tool. The investments in DSL implementation are specific to a given tool.

7.2 Tools for Language Implementation

Numerous tools have the capability to generate graphical editors from a meta-model of a DSL language. Such tools differ on the: (i) implementation to

attain concrete graphical syntaxes; (ii) implementation complexity; and (iii) maintenance efforts.

7.2.1 Microsoft Domain-Specific Language Tools

Microsoft Domain-Specific Language Tools [CJKW07] allow to create a representation of the model of a language, to specify their relations, to describe the semantics of the language and to define the DSL graphical representation. Furthermore, Microsoft DSL Tools offer a solid and seamless implementation, nonetheless depending on the implementation goals, the fact of being platform specific, can be considered as a restriction.

7.2.2 Eclipse Platform

The Eclipse platform contains a set of modeling frameworks, such as Eclipse Modeling Framework (EMF) [SBPM09], Graphical Editing Framework (GEF) [RWC11] and Graphical Modeling Framework (GMF) [GB14]. EMF is a modeling framework that facilitates building tools to support modeling languages. GEF provides the graphical support needed for building a model/language editor on the top of the EMF framework. In addition, GMF is a more advanced and generative framework for developing graphical editors for providing support for DSL development, by leveraging EMF and GEF. These frameworks are popular, straightforward to use and maintain. Moreover, they have a strong community support.

7.2.3 StarUML

StarUML [Sta13] is an open source project having in mind flexibility, extensibility, and fast development. StarUML can be extended to provide further functionality over the tool by the development of new modules. The tool acts as modeling platform to provide functionality for various platform technologies and external tools. This modeling software offers great extensibility, but it requires a deep knowledge over the core extension mechanisms of the tool. Furthermore, this tool only runs on win32 systems and its development has stagnated.

7.2.4 Open ModelSphere

Open ModelSphere [Gra14] is a platform independent modeling tool. This tool is supported by an experienced community and can be freely modified as desired. The downside is the complexity to cope with the addition of new language elements. In addition, it also requires a deep knowledge over the ModelSphere layered model.

7.2.5 JetBrains MPS

The Meta Programming System (MPS) is a language workbench, that does not include either a grammar or a parser [VBD⁺13]. It is an open-source project developed by JetBrains. Along with its several features, the one that is unique is the projectional editor, which supports mixed notations, such as textual, symbolic and tabular notations [EvdSV⁺13]. In addition, it also supports a wide range of language composition features [VBD⁺13].

7.2.6 SDF/Stratego/Spoofax

Spoofax [KV10] is a language workbench for developing textual DSLs, that is built on top of the Eclipse Framework. Spoofax is comprised by a set of tools that support grammar definition and DSL transformation to the desired targeted language. Grammars are specified using a syntax definition formalism (SDF). Another important part is the Stratego transformation language, which allows to describe the semantics of the language [BKVV08].

Spoofax uses the Eclipse IDE Meta-tooling Platform IMP. By doing so, it provides several benefits such as code outline, code completion, syntax highlighting, error checking and also offers the possibility to export the complete project as a stand-alone Eclipse plugin. However, as indicated in [SKK⁺14], Spoofax/IMP lacks stability.

7.2.7 xText

xText is a language development framework to assist the creation of DSLs and programming languages. It includes a parser, a code generator/interpreter, and facilitates Eclipse IDE integration [Bet13]. DSLs are specified using xText's grammar language. In order to validate a DSL, validators are required to be implemented to perform additional constraint checks [Bet13].

xText is a mature project with a strong supporting community [VBD⁺13]. Also, the surrounding ecosystem provides a huge number of add-ons that support the construction of sophisticated DSL environments [VBD⁺13].

7.3 DSL Engineering Process

Strembeck and Zdun [SZ09] describe a systematic approach for the development of DSLs. They present a set of activities that should be addressed when developing DSLs. Moreover, they incorporate and document these activities within a DSL engineering process.

A language model driven process represents a type of DSL engineering process. This type of process indicates that the language model definition drives the DSL development and the process is incremental and iterative. Figure 7.1 describes this DSL engineering process.

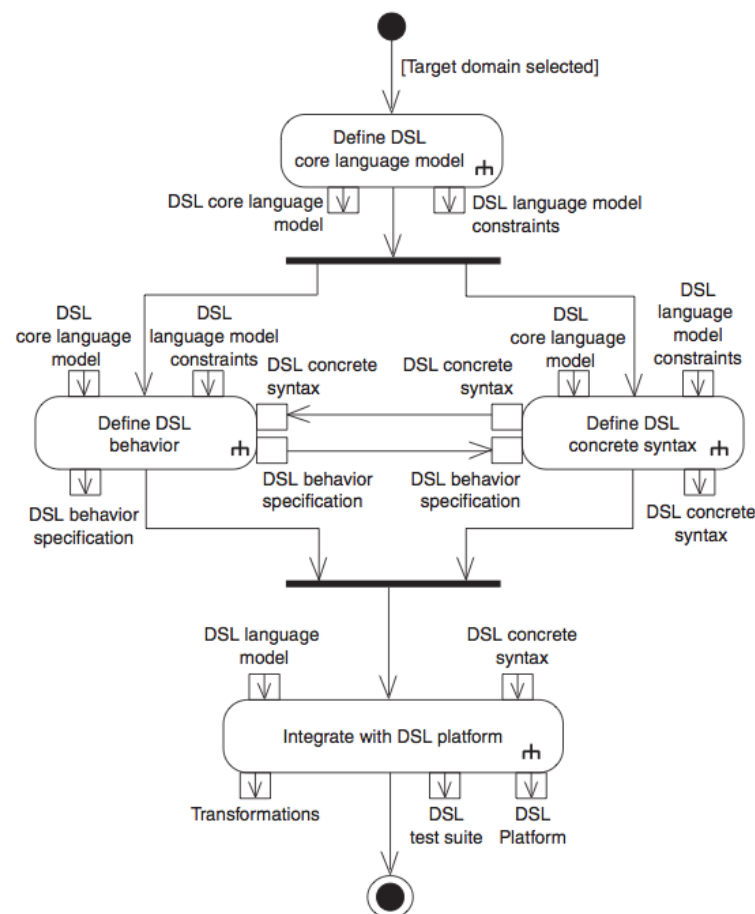


Figure 7.1: Language model-driven DSL engineering process (source from [SZ09]).

The process defines four main activities and several artifacts. The most crucial ones will be highlighted next. The *core language model* is responsible for capturing all relevant domain abstractions (elements of the DSL target domain) and their relations. It can be defined using a modeling language, for instance UML. The *language model constraints* express invariants on elements of the core language model and/or on relations between those elements and thereby define semantics that cannot be expressed directly in the (graphical) core language model. In situations where the core language model is described in UML, these constraints could be defined in OCL. However, other languages can be used to describe these restrictions. The *DSL behavior specification* defines the (behavioral) effects that result from using a DSL language element. Moreover, it defines how the DSL language elements can interact at runtime. Each DSL needs a *concrete syntax* to use the DSL in a certain system environment. A concrete syntax represents the abstractions defined through the DSL's abstract syntax, and each DSL can have multiple concrete syntaxes, for instance a graphical syntax and a textual syntax. In addition, the concrete syntax should address concerns such as *readability*, so it can be read efficiently and *learnability*, as it can be useful to novices to understand and to learn, among others. The last activity, *platform integration*, consists in mapping the DSL to the desired platform on which the DSL is expected to run.

7.4 Alloy in DSL Engineering

Alloy [Jac11] is a lightweight, simple yet powerful formal language that supports structural and behavioral modeling. It can be used at early stages of software development allowing to discover the correct software abstractions. Moreover, it allows to express structural and behavior constraints. The main benefit of using Alloy during DSL development is its capability to support an iterative process to define and tune language constraints. This process terminates when the constraints necessary to ensure the construction of well-formed model are found. Alloy is supported by the Alloy Analyzer tool, a friendly SAT (satisfiability) based tool that enables automatic model V&V (verification and validation). When models are specified using Alloy, it becomes possible to generate instances that should represent valid models (according to specific rules), and thus analyzing them in order to find/define and tune language constraints.

In the context of this dissertation, Alloy is used to support the development of a DSL from scratch. From the Alloy model of the DSL, it is possible to generate instances (that should represent possible valid GUI test models) and analyze those

instances in order to define and tune language constraints. This is an iterative process that ends when the constraints necessary to ensure the construction of well-formed PARADIGM test models are found.

7.4.1 Introducing Alloy

Alloy is a declarative specification language developed by the Design Group at MIT since 1997. It is based on first-order logic, for expressing complex structural constraints and behavior [Jac11]. It is designed to accurately describe the specification and the modeling of a system. Alloy is appropriate for early stages of software development, allowing to discover the correct software abstractions. Alloy is based on relational logic that combines the quantifiers of first order logic with the operators of the relational calculus.

Alloy has simple but powerful syntax that is easy to read and write. Alloy is supported by a verification tool called Alloy Analyzer, which can be used to automatically analyze the alloy specifications. This tool performs bounded verification using SAT solvers to answer verification queries.

An Alloy specification (model) needs to be built textually. After building the alloy model by the analyzer, the model can be represented by the graphical part and the textual part [He06]. An Alloy specification is defined by a module, which includes a set of imports with paragraphs. A paragraph can be a signature, a fact, a predicate, a function, an assertion or a command. A signature introduces a typed set of atoms and may have fields. Facts are constraints on relations that always hold. A predicate is a named constraint with zero or more arguments. Functions define reusable expressions. Assertions are properties that must hold from facts of the model. Commands are instructions that allow to perform *check* and *run* analysis. *Run* instructs the analyzer to search for an instance of a given predicate. *Check* instructs the analyzer to search for a counter-example of a given assertion.

7.4.2 Proposed Methodology

Given a domain that requires a DSL to be created for that specific purpose, the proposed approach defines a total of five steps that domain experts should comply with, in order to design the DSL using Alloy. One important aspect to mention is that this is always an iterative process. This process should be concluded, only after defining, tuning and analyzing the expected behavior of the DSL language elements. The proposed methodology [MP15] is illustrated in Figure 7.2.

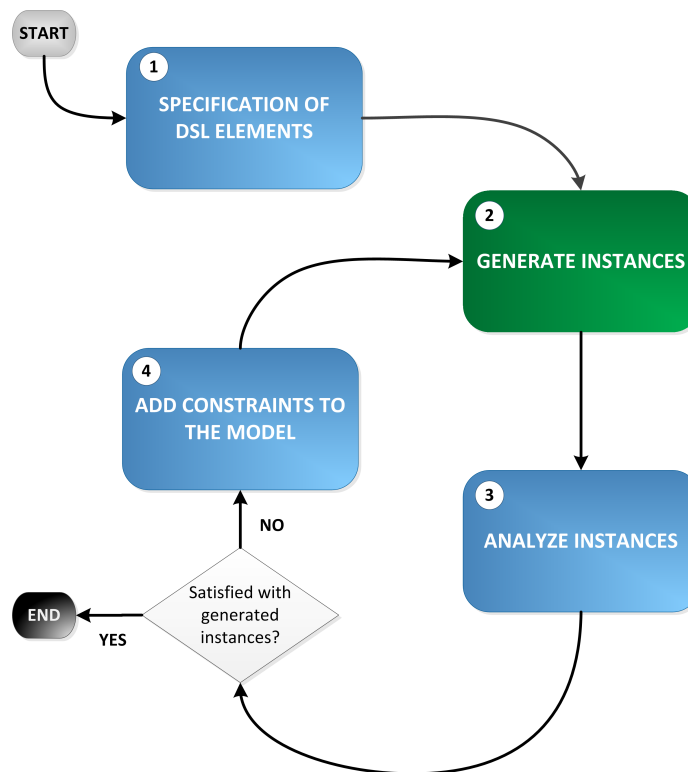


Figure 7.2: A Methodology for the usage of Alloy in DSL Engineering (MAIDEN).

The steps for proposed approach (MAIDEN) are described as follows:

1. **Specification of DSL elements** – The first step is the definition of the structure of the DSL with its elements. These elements are defined as signatures in Alloy;
2. **Generate and analyze instances** – The Alloy model is executed through the Alloy Analyzer tool to generate instances that should correspond to valid models written in the DSL being defined. To generate those instances, the developer should write run commands with proper bounds, for example, “run {} for 3” means that the instances generated will have a maximum of 3 elements of each signature. Typically, in the first iteration it is not necessary to define high bounds, since issues will be found in small instances. In the forthcoming iterations there will be the need to increase the bound of runs in order to keep aiming towards the growth of confidence in the quality of the model;
3. **Add constraints to the model** – If problems are identified within instances generated in the previous step, constraints should be added to the model;

4. **Repeat the process from step 2** – The process should be repeated until no problems are found in the generated instances of the model.

7.5 Development of a GUI Specification Language

The PBGT methodology requires a GUI model to be crafted specifically for this domain. Thus, a language is required to assist in the creation of models particularly tailored for PBGT. With the latter in mind, several DSLs for GUI modeling have been evaluated along with tools to support their development [MP14a]. However, to the best of the author’s knowledge, there is no DSL built from the right beginning of its development with reusability concerns, providing UI Test Patterns that can be adapted for testing different implementations after a small configuration step. The *envisioned* language promotes reusability either by reusing existing elements or extending them to be reused by others. In addition, this *new* DSL allows building a model describing the test goals instead of describing the expected behavior.

The development of the GUI specification language called PARADIGM, followed the process indicated in Section 7.3 (p. 90) and will be described in the current Section. In addition, the finding and tuning of language constraints was driven by the proposed methodology MAIDEN (Section 7.4.2).

7.5.1 Language Goals

PARADIGM is a DSL that has been created, from scratch, for PBGT. The main goals of this language are to:

- provide models at a higher level of abstraction.
- provide generic test strategies for testing GUIs.
- provide language flexibility and promote reuse.
- provide extension mechanisms to cope with future UI Patterns trends.
- be simple to use, evolve and maintain.
- reduce the effort in building models.

7.5.2 Language Core Model

A language core model captures all relevant domain abstractions and specifies relations among them. Abstractions refer to concepts within the context of PBGT. In PARADIGM, these abstractions are behavioral elements (UI Test Patterns), structural elements and language connectors. The PARADIGM language metamodel specified in UML is illustrated in Figure 7.3.

Elements

An *Element* is an abstract entity that represents the concepts within PBGT domain. A model written in PARADIGM starts with the *Init* element and finishes with the *End*. Both *Init* and *End* are specializations of the abstract entity *Element*.

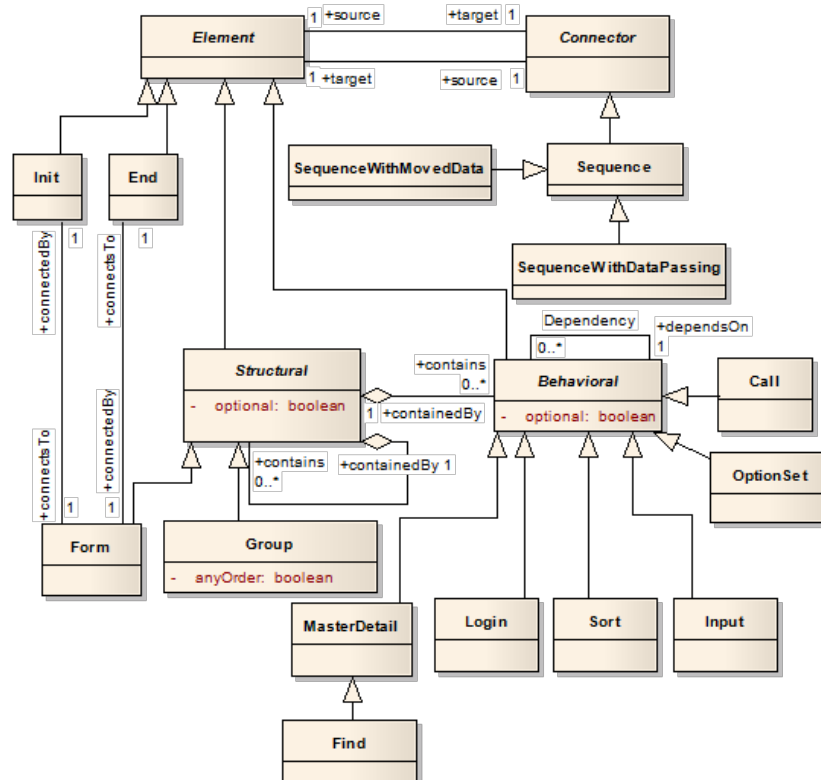


Figure 7.3: PARADIGM Language metamodel.

Models can be structured in different levels of abstraction. For instance, as models grow it is possible to use structuring techniques, to handle different hierarchical levels, so that a *model A* can “live” inside a *model B*. *Structural Form* elements were created specifically for this purpose. A *Form* (*Structural* element) embodies a model (or sub-model), with an *Init* and *End* elements. *Groups* are also structural elements but they are used to gather elements that may be executed in

any order. *Behavioral* elements represent the UI Test Patterns (described in Chapter 6, “Pattern Library”, p. 63) that define strategies for testing the UI Patterns.

Connectors

ConcurTaskTrees (CTT) [PMM97] are a popular, expressive and powerful graphical notation for task representation. It defines a set of temporal operators to combine tasks. The connectors within the PARADIGM language are inspired by CTT.

PARADIGM defines three connectors: “*Sequence*”; “*SequenceWithDataPassing*”; and “*SequenceWithMovedData*”. The “*Sequence*” connector indicates that the target element cannot start before the source element has completed. The “*SequenceWithDataPassing*” connector has the same behavior as “*Sequence*” and, additionally, indicates that the target element receives data from the source element. “*SequenceWithMovedData*” has a similar meaning to the “*SequenceWithDataPassing*” connector, however, the source element transfers data to the target, so the source loses the data that was transferred. In addition, there is another kind of relation among elements – “*Dependency*” – indicating that the target element depends on the properties of a set of source elements, for instance, when it is the result of a calculation.

The Alloy specification of the connectors and their specific constraints will be detailed next.

Language Constraints

The model of the PARADIGM’s language contains additional constraints that cannot be expressed directly in the UML language model. During DSL development some initial thoughts regarding language constraints tend to evolve in time and need to be enhanced as the language model increases and progresses.

The PARADIGM UML metamodel (Figure 7.3) was translated to Alloy in order to generate instances (that should represent possible valid GUI test models) and, afterwards, analyze them to think about the properties that such models should have and tune the language constraints. This is an iterative process, defined in Section 7.4.2, that ends when the instances obtained from the Alloy model correspond to valid well-formed PARADIGM models. This is when the correct set of constraints is found.

As such, the starting point of this process was to define the structure of PARADIGM models as they may be structured in different levels of abstraction.

First iteration

The first iteration started by creating an Alloy model for PARADIGM. The model is illustrated in Figure 7.4. The command “run {} for 4 but exactly 1 Form” means that instances generated will have at most 4 elements of each signature but exactly one instance of the *Form* signature.

```

open util/boolean

abstract sig Element{
  connector: lone Element
}
sig Init extends Element{}
sig End extends Element{}
abstract sig Behavior extends Element{}
abstract sig Structural extends Element{
  innerStructs: set Structural,
  innerBehavior: some Behavior
}
sig Group extends Structural {
  {#innerBehavior > 1}
}
sig Form extends Structural{
  init: one Init,
  end: one End,
}

run {} for 4 but exactly 1 Form

```

Figure 7.4: PARADIGM Alloy Model – First iteration.

Then, the model (from Figure 7.4) was executed (using the Alloy Analyzer tool), which allowed to analyze the generated instances. One of those instances is displayed in Figure 7.5.

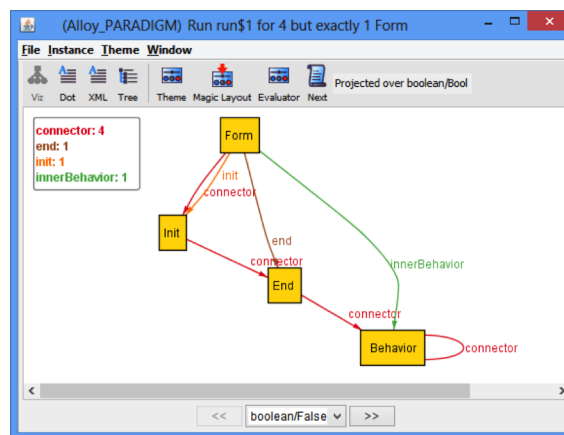


Figure 7.5: Generated instance of the model from Figure 7.4.

First iteration issues

The instance (Figure 7.5) faces some issues. There is a link from a *Form* element to the *Init* element (the *Init* element is an internal element inside the *Form*). This

should not happen since these elements belong to different hierarchical levels. The other issue is related with links from an element to itself. This must not happen. Yet, in the generated instance, it can be seen that there is a connector from *Behavior* to *Behavior*. Also, the *Init* element should always be the first element in the model and the *End* element should be the last one. There is a link from the *Init* to the *End* and from the *End* to the *Behavior* element. This is not correct. Further, the *Init* element should not link directly to the *End* element.

Second iteration

The study proceeded for the second iteration in the pursuit of finding the expected behavior of PARADIGM language elements. In this iteration, new statements (constraints) were added in the model. These constraints are displayed in Figure 7.6.

```
fact {
  //no connection to the same element
  all e:Element | no e.connector & connector.e
  //Init and End cannot be directly connected
  all i:Init | no i.connector & End
  all e:End | no connector.e & Init
  //End (Init) cannot be the origin (destination) of a connector
  no End.connector
  no connector.Init
}

run {} for 5 but exactly 2 Form
```

Figure 7.6: New added constraints to the PARADIGM Alloy Model (from Figure 7.4) – Second iteration.

One of the generated instances of the model (from Figure 7.4 including the new statements from Figure 7.6) is illustrated in Figure 7.7.

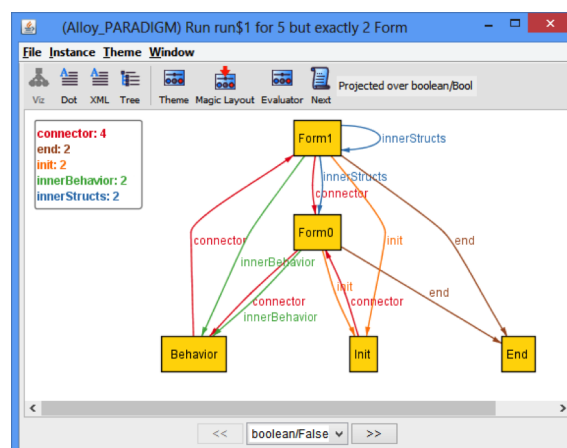


Figure 7.7: Generated instance of the model from Figure 7.4 including the new statements from Figure 7.6.

Second iteration issues

This iteration still faces some issues. The *innerStructs* relation cannot allow one element to “live” inside itself (there is a *Form1* with another *Form1* inside). Moreover, it cannot be possible to have elements from different levels (depth levels) connected. However, *Behavior1* is connected to *Form1*. *Form1* has *Init* and *End* elements inside. *Form0* has the same *Init* and *End* elements inside. Yet, elements *Init* and *End* cannot belong to more than one *Form* (level). Each level (*Form*) must have an *Init* and an *End* element but they cannot belong to more than one *Form* (level).

Third iteration

In order to solve the issues identified in previous iteration, the number of constraints were continuously being increased. The added constraints for this third iteration are displayed in Figure 7.8.

```

fun Parent[e:Element] : Structural{
  innerStructs.e + innerBehavior.e + inite + end.e
}

fact {
  // init and End belong just to one Form
  all i:Init | one init.i
  all e:End | one end.e
  // relation innerStructs is acyclic
  all e:Element | e not in e.∧innerStructs
  // no connectors between elements in different levels of abstraction
  all e:Element | one e.connector => Parent[e] = Parent[e.connector]
}

run {} for 7 but exactly 2 Form

```

Figure 7.8: New added constraints to the PARADIGM Alloy Model – Third iteration.

The obtained instance from executing the model (including its previous iterations) with the new constraints from Figure 7.8, is illustrated in Figure 7.9.

Third iteration issues

By analyzing the executed model, it was possible to become aware of the following issues. The *Behavior* element is inside *Form0* and *Form1* (it belongs to two different abstraction levels, i.e, it has two parents). This is not allowed. In addition, PBGT approach generates test cases from PARADIGM models and for that it calculates all the paths from *Init* to *End* elements. This means that the elements inside PARADIGM models must be connected. Inside *Form1* there are not connectors

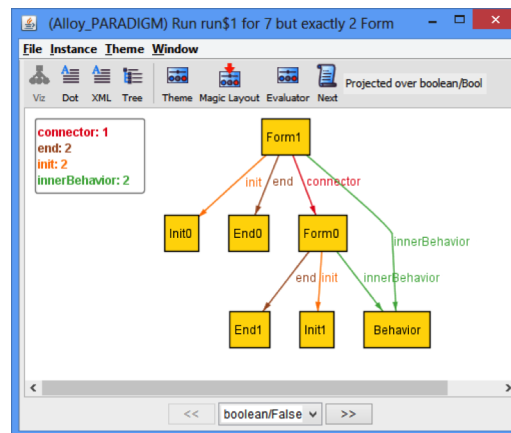


Figure 7.9: Generated instance of the model including the new statements from Figure 7.8.

that link *Init0* to *Form0* and *End0*. Therefore, there is no path. It is required to ensure that there must be connectors from *Init* to the *End* elements.

Fourth iteration

In this iteration, further constraints were added to the model (Figure 7.10).

```
fact {
  // all elements but the inicial Form has a Parent
  all e:Element - {f:Form | no Parent[f]} | one Parent[e]
  // there is a path from all elements inside a Form until the End element of that form
  all f:Form | f.end+f.innerBehavior+f.innerStructs in f.init^(connector)
}

run {} for 8 but exactly 2 Form
```

Figure 7.10: New added constraints to the PARADIGM Alloy Model – Fourth iteration.

Then, the generated instances were analyzed by executing the model (including its previous iterations) with the new constraints from Figure 7.10. One of those instances is displayed in Figure 7.11.

Fourth iteration issues

In this iteration only one issue was encountered. This instance shows two separated models and thus, another constraint is required, in order to ensure that there is only one *Form* without *parent*, i.e., the main model.

Fifth iteration

This iteration consisted in adding a constraint to ensure only one *Form* with no *parent* and the bound of the run command (Figure 7.12) was increased, to check

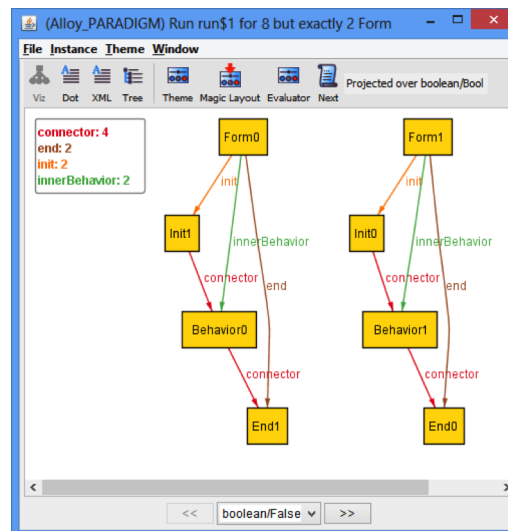


Figure 7.11: Generated instance of the model including the new statements from Figure 7.10.

if the generated model instances were valid, i.e., correspond to correct models written in PARADIGM.

```
fact {
  //There is one Form without parent
  one f:Form | no Parent[f]
}

run {} for 11 but exactly 2 Form, exactly 1 Group
```

Figure 7.12: New added constraints to the PARADIGM Alloy Model – Fifth iteration.

Fifth iteration issues

In this iteration, for the instance from Figure 7.13, no issues were found. All restrictions have been encountered, and this led to the completion of the process.

Final constraints

After five iterations, the obtained results were satisfactory. Therefore, the final language constraints for PARADIGM were:

LC1: A *Connector* cannot connect an element to itself.

LC2: A *Connector* cannot have *Init* as destination neither *End* as source.

LC3: An *Init* element cannot connect directly to an *End* element.

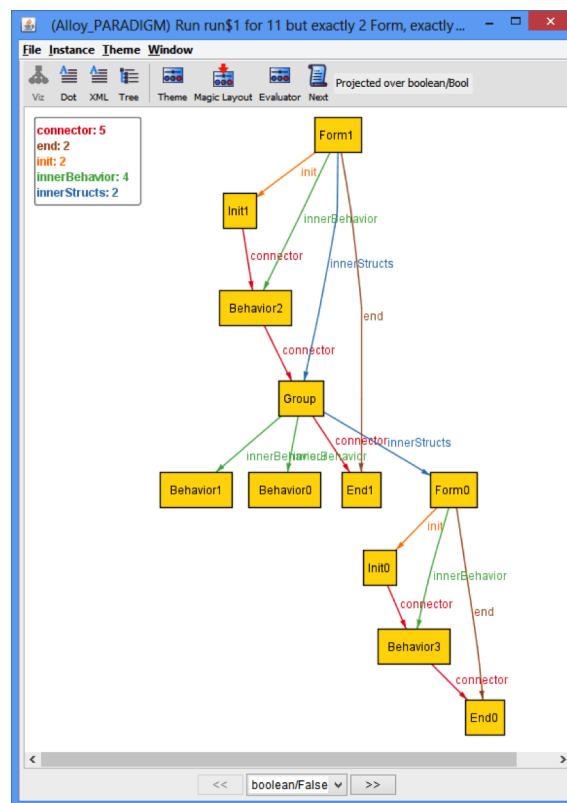


Figure 7.13: Generated instance of the model including the new statements from Figure 7.12.

- LC4:** Two elements cannot be connected more than once by connectors of the same type.
- LC5:** Two Elements can only be connected if they belong to the same *Structural Element* (*Model*, *Form* and *Group*).
- LC6:** Elements inside a *Form* (but not inside *Groups* of that *Form*) cannot be loose, i.e., for all elements within a *Form*, there is at least one path from the *Init* to the *End* that traverses that element;
- LC7:** The model must be a *Form* without a parent.

7.5.3 Behavior Specification

The DSL's behavior specification, also noted as dynamic semantics, stands as a part of the language model and defines the behavior concerning the usage of a given DSL language element. PARADIGM's behavioral semantics establishes the instructions on how to use the language and defines the interactions among language elements.

To some extent, the DSL language constraints define the behavior of the PARADIGM language. In addition, the UI Test Patterns configuration represent how UI Test Patterns behave at runtime, during test execution. The configuration for the UI Test Patterns is described in Chapter 6, “Pattern Library” (p. 63).

7.5.4 Concrete Syntax

The next step in the DSL development is the definition of its concrete syntax (Figure 7.14). Hence, the graphical symbols for the language connectors and for the language elements were defined. The reasons for selecting a graphical notation over a textual one, was based on the following aspects: (i) easier to understand; (ii) simpler to navigate and; (iii) intuitive and faster learning for end-users.

All PARADIGM’s elements and connectors are defined by: (i) an icon/figure to represent the element graphically; (ii) a short label to name the element; (iii) a number (between square brackets) to identify the element, according to its context; and (iv) a Boolean value, also placed between square brackets, to indicate if the element is optional.

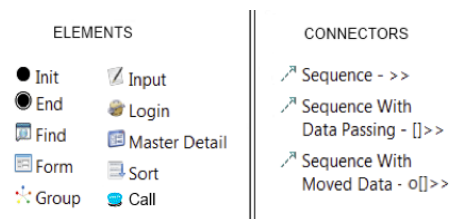


Figure 7.14: PARADIGM concrete syntax.

7.5.5 Integration

Eclipse was the selected target platform for PARADIGM. EMF was the selected framework for the development of a specific tool, called PBGT Tool [MP14c], targeted to assist the construction of PARADIGM models in the context of PBGT. However, several tools for language implementation have also been evaluated, but they failed in simplicity in the development, maintenance and support aspects.

The PBGT tool development incorporated all the previous definitions of the language: elements and their properties, connectors, graphical representation and constraints. These constraints were implemented in OCL [CG12]. In addition, new widgets were defined just for improving end-user interaction (menus, toolbars and buttons). This tool provides support for modeling GUIs using PARADIGM and further functionality: UI Test Pattern configuration; test cases generation from

models; test case execution and; test coverage information. Since PARADIGM aims to be an expandable language (to cope with latest UI trends), new elements and connectors may be added. This means the language can be expanded during implementation phase (development) and/or afterwards (modeling phase). For instance, the tester is able to create his own set of behavioral elements by reusing and combining the existent ones within *Forms*. The set of those *Forms* can be seen as a library of new elements that can be reused within other model.

7.5.6 How to Attain Reuse and Evolution

DSLs can be seen as enablers of reuse [MHS05]. Reuse is one of the important perspectives that substantiate DSLs' value. Thus, the main contribution of DSLs is to enable reuse of software artifacts [Big98]. In this context, PARADIGM also enables reuse. In section "Levels of Reuse" (p. 55), the levels of reuse that can be attained are described. In order to have a more graphical perspective on how to achieve reuse, a detailed description is provided next. In addition, details on how to achieve evolution, i.e., by extending current Base UI Test Patterns into High-Order UI Test Patterns is also described.

High-Order UI Test Patterns

This type of UI Test Patterns extend the Base UI Test Patterns to facilitate the building of new UI Test Patterns. This way, it is possible to promote even further the concept of reuse, and also represents a way to evolve from the Base UI Test Patterns. Besides allowing to structure the models in different levels of abstraction, the *Form* element can be seen has an extension mechanism of the language that supports the definition of High-Order Patterns. High-Order UI Test Patterns can be created by testers and are always comprised by at least two other (either Base UI Test Patterns or High-Order UI Test Patterns) patterns combined by the existing connectors.

The main benefit of the language extension possibility, is that each tester may have his own set of UI Test Patterns that represent a kind of test library that can evolve as desired according to the specific characteristics of the GUIs they test. As an example, considering a GUI that features an authentication mechanism. Upon successful login redirects the user to another area of the GUI that features a master/detail area. The user is able to interact with that area, where the detail contains a sortable grid. This is populated after the selection from one master element from the master/detail area. By analyzing this GUI from a testing point of

view, the tester would require one Login UI Test Pattern, one Master/Detail UI Test Pattern and a Sort UI Test Pattern. The tester realizes that this particular flow can be reused to test other GUIs as well. The tester wants to combine these three Base UI Test Patterns into a new UI Test Pattern called LIMIT (LogIn; Master/Detail and SorT). The LIMIT UI Test Pattern will be classified as an High-Order UI Test Pattern. Figure 7.15 illustrates how the LIMIT UI Test Pattern can be arranged.

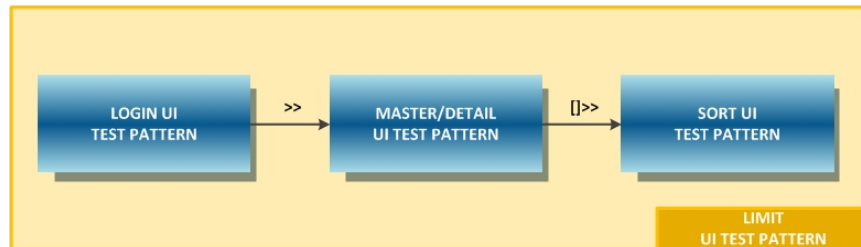


Figure 7.15: High-Order UI Test Pattern – LIMIT.

Figure above is comprised by three Base UI Test Patterns with connectors between them. From the Login UI Test Pattern no data is sent to the next UI Test Pattern. Therefore the connector that is used is a “Sequence” one. Master/Detail UI Test Pattern transfers data to the Sort UI Test Pattern. For this reason it is used the “SequenceWithDataPassing” connector. All these elements (UI Test Patterns and connectors) are gathered into a *Form* allowing to create this new High-Order UI Test Pattern.

UI Test Patterns without provided data for configuration

According to the nature of UI Test Patterns, each UI Test Pattern can be reused. This happens because each UI Test Pattern was designed to test a wider-range of UIs that feature UI Patterns. Whenever a given UI Pattern is identified in the UI, from a testing perspective, there is a “testing match” that allows to test the recurrent behavior of such UI Pattern. UI Test Patterns have configuration concerns built-in. For instance, the Login UI Test Pattern has its own configuration and the Master/Detail UI Test Pattern has another one. The tester is responsible for providing data input for the instances of the UI Test Patterns featured in the PARADIGM models. Independently of the configuration data, UI Test Patterns can be reused across multiple implementation of a given UI Pattern. Also, from another view, Base UI Test Patterns are also enablers of reuse as they are required to be combined in order to create new sets of UI Test Patterns, given by the classification of High-Order UI Test Patterns.

UI Test Patterns with provided data for configuration

Both Base and High-Order UI Test Patterns may contain an already entered data for configuration by the tester. Depending on the goal of the tester, these configurations can also be reused. As an example, considering the LIMIT UI Test Pattern, this pattern is comprised by other three UI Test Patterns. In this case, two options emerge: (1) the tester may want to reuse the configuration from all Base UI Test Patterns or (2) the tester may only desire to use configurations from specific Base UI Test Patterns. Figure 7.16 details the first option.

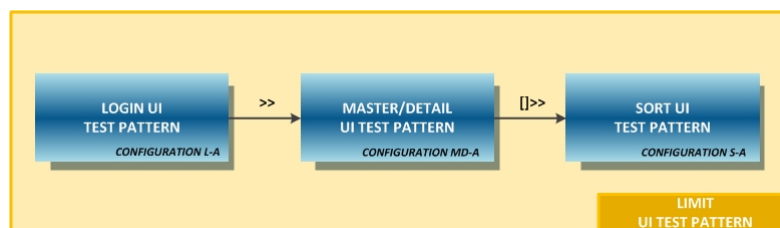


Figure 7.16: LIMIT UI Test Pattern – Inner configuration details.

In Figure 7.16, each UI Test Pattern contains its own data for configuration. Login UI Test Pattern (first element in Figure 7.16) contains the data described as “CONFIGURATION L (for Login) - A”. The data for configuration for the Master/Detail UI Test Pattern is referred to as “CONFIGURATION MD-A” and the data for Sort as “CONFIGURATION S-A”.

As mentioned earlier, it is also possible for the tester to want to reuse the LIMIT UI Test Pattern with already provided data for configuration, but the tester only wants to reuse one specific configuration of a particular UI Test Pattern. For example, the Login UI Test Pattern has two goals: Login Valid and Login Invalid. If the tester does not want to repeat the same configuration for Login Invalid, which can be blank password only (not matter the username), he can reuse it later. Therefore, this scenario is illustrated for the LIMIT UI Test Pattern in Figure 7.17.

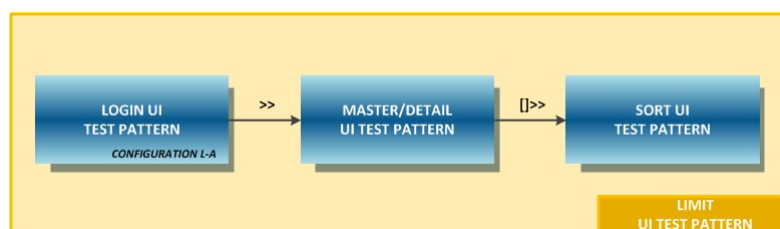


Figure 7.17: LIMIT UI Test Pattern – configuration for Login UI Test Pattern.

7.6 Summary

PARADIGM is a DSL that was built to be used in the PBGT approach. The DSL was developed from scratch, following a systematic approach for DSL engineering. This approach defines a set of guidelines and best practices that should be taken in consideration when developing a DSL. Alloy was used in the development of PARADIGM to define and tune language constraints. This led to the raise of a new approach, entitled MAIDEN (a methodology for the usage of Alloy in DSL engineering). MAIDEN defines an iterative process that consists of four steps. The process is concluded when the definition, tuning and the expected behavior of the DSL language elements are attained. PARADIGM enables reuse at different levels of abstraction, and provides means to support evolution of its language elements. The next chapter addresses the validation of the thesis.

8

Validation

To evaluate the PBGT approach, including the methodology along with processes defined for design and development of the approach, three case studies were conducted following Yin's guidelines [Yin09]. These guidelines cover all aspects of a case study research method. Thus, when conducting a case study, there are six major activities to be followed: (1) Plan – consists in identifying the research questions and idealize the case study; (2) Design – define data to be collected as well as procedures to maintain the case study quality (external validity and internal validity) and identify the criteria to interpret the findings; (3) Prepare – identifying teams to be part of the case study and define procedures for data collection; (4) Collect – case study execution with data gathering; (5) Analyze – consists in examining, categorizing, tabulating data to draw empirically based conclusions; and (6) Share – report the case study demonstrating its findings and results.

The first case study aims to assess the PBGT general approach, using two fielded web application subjects, in order to ascertain whether it can be used by testers who have not been involved in the development of the approach, and to measure the effort required to obtain benefits from the approach. The second case study provides a deeper analysis by studying software subjects seeded with artificial faults. Finally, the third case study, compares the modeling efforts of

PARADIGM with other modeling approaches. In addition, the language usefulness, graphical power, and acceptability is also evaluated. This study also presents an evaluation at industrial level.

8.1 Case Study 1: Assessing PBGT failure detection capability

The main goal of *case study 1* is to measure the overall effectiveness of the PBGT approach. The idea is to determine whether testers are able to effectively employ PBGT to detect failures. In addition, it is also desired to measure the effort required to use PBGT. In particular, this study is designed to answer the following research questions:

RQ1.1 What is the effort required to start using the PBGT approach?

RQ1.2 What is the proficiency of the PBGT approach in finding failures?

8.1.1 Metrics

To address the research questions above, a set of metrics were defined in order to be gathered during the execution of the study. The metric (**M1.1**) was related to the research question **RQ1.1** and kept track of the time required to present the PBGT approach, the PARADIGM language and the modeling environment to the teams; and the time the teams spent in building and configuring the necessary models to test the subject systems. The second metric (**M1.2**) related to research question **RQ1.2** and stored information about the number of failures that the teams were able to find in fielded applications.

8.1.2 Subject Systems

To prepare the case study, a set of subject systems was selected, according to the following criteria: (1) the subject systems were required to have a variety of functionalities so they could provide the ability to apply and fulfill at large scale, the different PBGT testing strategies; and (2) due to the increasing usage of web applications in recent days, the choice went to public web-based applications. Therefore, the selected subject systems were:

- **SS1.1** – mobile.de [mob14]. This is a german marketplace for buying and selling new and used vehicles. It allows to search vehicles according to various parameters (make, model, kilometers, year, etc.), sort the search result according to a set of parameters and refine the search. The website also allows registration for users who, after authentication, can insert vehicles for sale, among other functionalities.
- **SS1.2** – australian-charts [Hun14a]. This is an australian site, that features the best selling music singles and albums in Australia. It displays the current top 50 and the best of all time singles and albums; allows searching for songs, albums, compilations and DVDs; shows a score based on reviews; allows new user registration and, subsequently, authentication; provides support for forums; among other functionalities. Moreover, this site has another functionality that allows access sites for the same purpose (listing the best selling music) from other countries, such as, Finland, Germany, etc.

8.1.3 Testing Goals

A set of testing goals were defined for this study, having in mind to test some of the functionalities of the web sites, i.e., SS1.1 and SS1.2. The defined testing goals for SS1.1 are described in Table 8.1 and the ones for SS1.2 are represented in Table 8.2. The tables feature the ID that will be used to refer to the testing goal in question and the description of the testing goal.

Table 8.1: Defined testing goals for *mobile.de* (SS1.1).

ID	Description
TG1.1	Test search for models of a particular automotive brand
TG1.2	Test if the number of the cars within the result set is correct and if it contains the elements it is supposed to have
TG1.3	Test the possibility to sort the results providing different criteria
TG1.4	Test the search by refining its contents
TG1.5	Test if the items selected to limit the search are transferred to the search criteria area and vice versa.
TG1.6	Test the relationship between make and model
TG1.7	Test if providing input in the adjust search affects the results
TG1.8	Test the limit search and verify if it changes the results set

Table 8.2: Defined testing goals for *australian-charts.com* (SS1.2).

ID	Description
TG1.9	Test the authentication functionality in several moments, for instance, after performing a search
TG1.10	Test new user registration
TG1.11	Test the dependency between the selection of a different country and the contents of a list with different options
TG1.12	Test the dependency between the selection of an element within a list of options and the content displayed in the middle panel
TG1.13	Test the simple search
TG1.14	Test the extended search
TG1.15	Test the top search

8.1.4 Setup

In order to start the case study, two teams composed by students, who were not familiar with the PBGT approach, were assigned to embrace the role of testers. The first team (to be referred as **Team 1.1**), was assigned **SS1.1**. The second team (referred as **Team 1.2**) was assigned **SS1.2**. To answer **RQ1.1**, the time required to craft and configure models was registered. Concerning **RQ1.2** it was collected the number of failures detected by each team for each subject system.

To run the study, the URL of the PBGT tool was provided to the teams so they could be able to download the tool. Once they installed the tool, on their machines, the URL for each subject system was provided to them. The procedure that was followed to execute this study is described next.

8.1.5 Procedure

The study started by providing a two-hour training session to the teams about the PBGT approach. This session introduced the PBGT overall approach, describing the concepts within PARADIGM language and modeling environment PARADIGM-ME (where they were able to create and configure models). Furthermore, all their doubts were clarified, and modeling, configuration and testing examples were also presented. Then, the subject systems to be tested in the case study was introduced, so they could become familiar with them. Afterwards, the testing goals to achieve in each subject system were presented. Each member of each team had to fulfill, individually, the testing goals proposed for the assigned subject system.

By using the PBGT tool, each member of each team started building the models in order to fulfill the testing goals. During the study execution, they recorded the

time spent in building the models, and also the time spent in their configuration. Afterwards, they generated test cases and start executing the tests, in the PBGT tool. Then they analyzed the test results and recorded the failures found, and thus concluding the case study.

8.1.6 Results

When the study execution finished, it became possible to collect the models created by the teams, the configurations applied to the models, the generated test cases and a report listing the failures found. Due to the high amount of data obtained from this study, the author decided to show and explain only one of the models built by one of the team members, as well as its configuration, the generated test cases and failures found. However, all models, their configurations, generated test cases and reports are documented in a public repository that was created to store the data that was collected from this study. The public repository can be found in [PBG14].

An example of a model (consisting of two levels) created by one member from Team 1.1 regarding SS1.2 is illustrated in Figure 8.1.

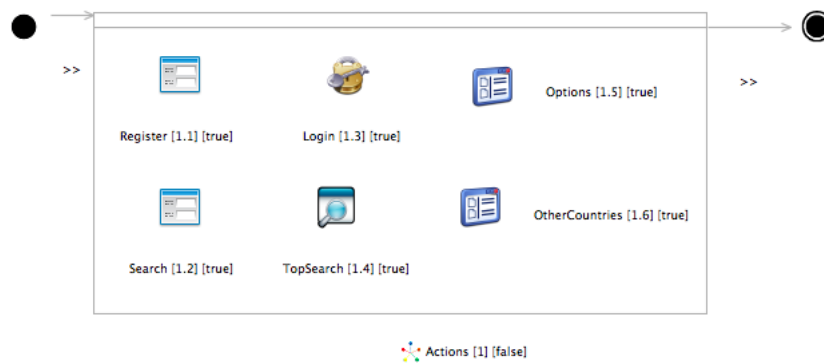


Figure 8.1: Australian-charts model (first level) written in PARADIGM.

The tester created the model from Figure 8.1 in order to achieve the testing goals from Table 8.2. In SS1.1 the end user is able, without any forced order, to register, login, search for songs/albums/artists, select countries according to his preference, and select several elements from the left menu (referred as Options for now on). Therefore, in PARADIGM, this arbitrary access to functionality maps to a *Group* element (labeled Actions, element number 1). This *Group* element is comprised by several elements: a Login UI Test Pattern (labeled “Login”, element number 1.3), that the tester selected in order to check TG1.9; a *Form* element called “Register” (element number 1.1) to check TG1.10 to test the

user registration; another *Form* element that refers to several searches (element number 1.2, labeled “Search”); a Find UI Test Pattern (labeled “TopSearch”, element number 1.4) to address TG1.15; and two Master/Detail UI Test Patterns, where the first element labeled “Options” number 1.5 fulfills TG1.12 and the second element labeled “OtherCountries” number 1.6 addresses TG1.11.

The configuration performed for the above model, is represented in Table 8.3. This table contains the details in respect to the configuration for the UI Test Patterns that are featured in the model. It indicates the name of the element referred in the model, the type of the UI Test Pattern and the different configurations (valid and invalid), variables to be used and the checks performed for each UI Test Pattern featured in the model. Each row contains the configuration details for each UI Test Pattern from Figure 8.1. For instance, the first row displays the configuration for the Login UI Test Pattern (element number 1.3 in Figure 8.1). The definition for the Login UI Test Pattern is described in Chapter 6, Section 6.4 (p. 69). Therefore, according to the pattern definition, the tester needs to provide valid username/password for LG_VAL (login valid) and invalid username/password for LG_INV (login invalid) and select the checks to perform. In this case, for LG_VAL, the tester defined “pbgt” for the username, “test” for the password and the selected check was to verify if the user changes to page “Main”. For LG_INV, the tester chose “pbgt” for the username, but provided an invalid password, i.e., “testtest”. The selected check for LG_INV was to verify if upon unsuccessful login the system would display in a label the “Login failed. Unknown user or wrong password (...)” message.

The model from Figure 8.1 contains a *Form* element that refers to the registration functionality. The contents of this *Form* are displayed in Figure 8.2. The registration *Form* contains a Call UI Test Pattern (element number 1.1.1 labeled “RegisterButton”) that will trigger the access to the registration fields. These fields are Input UI Test Patterns, allowing the user to provide data such as username (element number 1.1.2.1 labeled “User”), password (element number 1.1.2.2 labeled “Password”), confirm password (element number 1.1.2.3 labeled “CPass”) and email (element number 1.1.2.4 labeled “Email”). All these elements are structured within a *Group* element (number 1.1.2 labeled “RegisterFields”), since they can be filled in arbitrary order. After providing the necessary data to make a registration, the user needs to press the “SignOn” button, in order to submit the entered data. The latter is modeled using a Call UI Test Pattern (element number 1.1.3 labeled “SignOn”).

Table 8.4 holds the configuration of the elements (UI Test Patterns) from the

Table 8.3: Configuration details for UI Test Patterns featured in the model from Figure 8.1.

Name	Type	Validity	Variables	Check
Login	Login	Valid	{[username, "pbgt"]; [password, "test"]}	{Change to page "Main"}
		Invalid	{[username, "pbgt"]; [password, "testtest"]}	{Label "Login failed. Unknown user or wrong password (...)"}
OtherCountries	MD	NA	{[Master, "United Kingdom"], [Detail, ["UK Charts", "Search"]]} }	{Detail has 2 elements}
Options	MD	NA	{[Master, "Top 50 singles"], [Detail, "Of Monsters and Men – Little Talks"]}	{Detail has "Of Monsters and Men – Little Talks" element}
TopSearch	Find	Valid	{ [SearchForum, ""] }	{Result has element "show all topics" in line 1}
		Invalid	{ [SearchForum, "metallica"] }	{Result is an empty set}

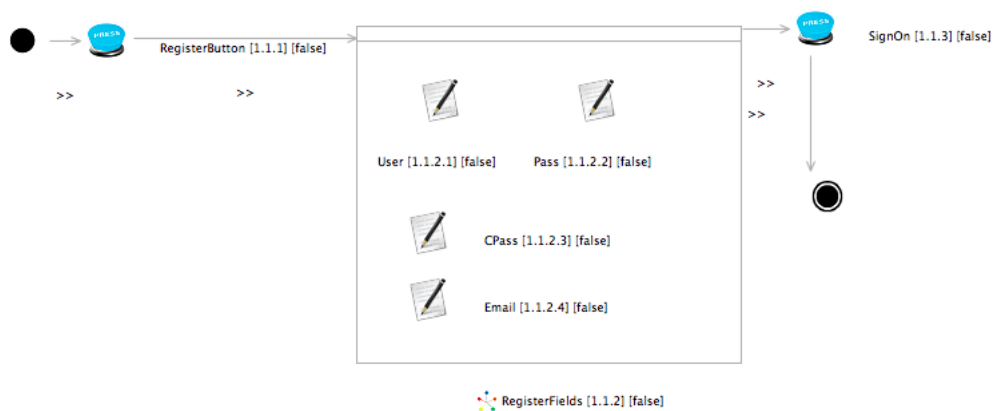


Figure 8.2: Australian-charts featuring the modeling of the registration form.

model from Figure 8.2. For example, the configuration for the Input UI Test Pattern that refers to element number 1.1.2.1 labeled “User” from Figure 8.2 is displayed in the second row of this table. According to the pattern definition (in Chapter 6, Section 6.3, p. 66), the tester needs to provide valid input data (INP_VD), invalid input data (INP_ID) and select the checks to perform. For INP_VD the

tester provided “PBGT” as the input for the username field without any check to be performed. Yet, for INP_ID he chose a blank username and the check refers to a label containing the text “Please fill out all mandatory fields: Username”.

Table 8.4: Configuration details for UI Test Patterns featured in registration form from Figure 8.2.

Name	Type	Validity	Variables	Check
RegisterButton	Call	NA	{}	{Change to Page “Sign On”}
User	Input	Valid	{[Username, “PBGT”]}	{}
		Invalid	{[Username, “”]}	{label “Please fill out all mandatory fields: Username”}
Password	Input	Valid	{[Password, “PBGTpass”]}	{}
		Invalid	{[Username, “”]}	{label “Please fill out all mandatory fields: Password”}
CPass	Input	Valid	{[ConfirmPassword, “PBGTpass”]}	{}
		Invalid	{[Password, “”]}	{label “Please fill out all mandatory fields: Confirm Password”}
Email	Input	Valid	{[e-Mail, “PBGT@gmail.com”]}	{}
		Invalid	{[e-Mail, “PBGTmail”]}	{label “An error occurred on the server when processing the URL. Please contact the system administrator”}
SignOn	Call	NA	{}	{Change to Page “Main”}

The model in Figure 8.1 contains a second *Form* (element 1.2, labeled “Search”) that refers to the advanced search functionality. The contents of this *Form* are illustrated in Figure 8.3. This *Form* contains a SearchButton, modeled as a Call UI Test Pattern, that will provide access to the simple and extended search functionalities. The simple (element number 1.2.2.1 labeled “SimpleSearch”) and extended (element number 1.2.2.2 labeled “ExtendedSearch”) searches are modeled as Find UI Test Patterns. They are structured within a *Group* (element number 1.2.2 labeled “SearchTypes”), since they can be used arbitrarily.

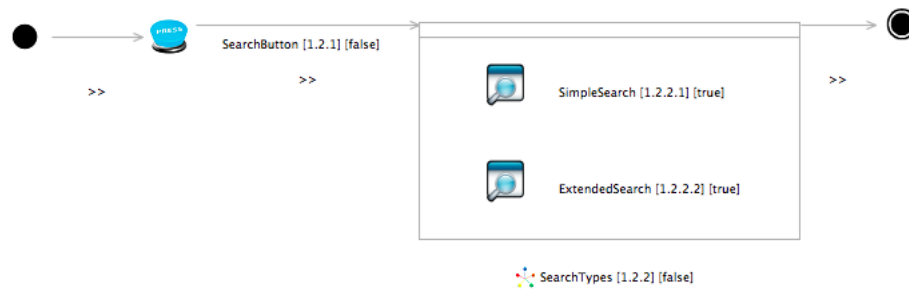


Figure 8.3: Australian-charts featuring the modeling of the search form.

The configuration for the model from Figure 8.3 is displayed in Table 8.5. This table describes the configuration for the UI Test Patterns (one Call and two Find UI Test Patterns) featured in the *Form*. Considering the configuration for the Call UI Test Pattern (first row in Table 8.5), and according to the pattern definition (in Chapter 6, Section 6.8, p. 79) the tester has to select the check to perform. For this element (number 1.2.1 labeled “SearchButton”) the tester selected the check “Change to Page Search”, which means that he wants to verify that upon pressing the Search button, the system will redirect the user to another page (Page Search).

Table 8.5: Configuration details for UI Test Patterns featured in Search Form from Figure 8.3.

Name	Type	Validity	Variables	Check
SearchButton	Call	NA	{}	{Change to Page “Search”}
SimpleSearch	Find	Valid	{[SearchSong, “nothing”]}	{result has element “Alesha Dixon” in line 1}
		Invalid	{[SearchSong, “nothing”]}	{result has element “Sinead O’ Connor” in line 5}
ExtendedSearch	Find	Valid	{[SearchAlbum, “justice for all”]}	{result has element “Metallica” in line 1}
		Invalid	{[SearchAlbum, “justice for all”]}	{result has element “Metallica” in line 25}

In PBGT, test cases are automatically generated from PARADIGM models. Due to high number of test cases only the ones that led to encountering failures are displayed. Table 8.6 displays part of the generated test cases from the model from Figure 8.1. It contains the ID that refers to the identifier of the test case in question, the test path and the test case described in XML format. The first row in

this table refers to test case 1.1 (TC1.1). The path that led to the failure was 1.3 (refers to element number 1.3 in Figure 8.1, Login UI Test Pattern) then 1.4 (refers to element number 1.4 – Find UI Test Pattern) and 1.6 (refers to element number 1.6 – Master/Detail UI Test Pattern). The last column displays the test case in XML format. It contains all the configurations for the elements within the path (second column) that led to the failure.

Table 8.6: Part of the generated test cases from the model from Figure 8.1.

ID	Test Path	XML Test Case
TC1.1	[1.3, 1.4, 1.6]	<pre> <Script> <Path value="[1.3, 1.4, 1.6]"> <Init/> <Login flag="false" name="Login" number="1.3" optional="true" validity="Valid"> <Check check="change to page" result="main"/> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="test"/> <Button field="button_1.3"/></Login> <Find flag="false" name="TopSearch" number="1.4" optional="true"> <Check check="Result has element" position="1" result="show all topics" /> <Value field="" fieldName="searchforum" value="" /> </Find> <MasterDetail flag="false" name="OtherCountries" number="1.6" optional="true"> <Check N="2" check="ContainsN" /> <Master field="master_1.6" value="United Kingdom"/> <Detail field="detail_1.6"> <Value name="UK Charts"/> <Value name="Search"/> </Detail> </MasterDetail> </Path> </Script> </pre>
TC1.2	[1.4, 1.3]	<pre> <Script> <Path value="[1.4, 1.3]"> <Init/> <Find flag="false" name="TopSearch" number="1.4" optional="true"> <Check check="Result has element" position="1" result="show all topics" /> <Value field="" fieldName="searchforum" value="" /> </Find> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="Login failed. Unknown user or wrong password(_)" /> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/></Login> </Path> </Script> </pre>
TC1.3	[1.5, 1.3]	<pre> <Script> <Path value="[1.5, 1.3]"> <Init/> <MasterDetail flag="false" name="Options" number="1.5" optional="true"> <Check N="2" check="ContainsN" /> <Master field="master_1.5" value="Top 50 Singles"/> <Detail field="detail_1.5" value="Of Monsters and Men - Little Talks"/> </MasterDetail> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="Login failed. Unknown user or wrong password(_)" /> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/></Login> </Path> </Script> </pre>
TC1.4	[1.2.1, 1.2.2.1, 1.3]	<pre> <Script> <Path value="[1.2.1, 1.2.2.1, 1.3]"> <Init/> <Call flag="true" name="SearchButton" number="1.2.1" optional="false"> <Check check="change to page" result="search"/> <Field name="call_1.2.1"/> </Call> <Find flag="false" name="Simplesearch" number="1.2.2.1" optional="true"> <Check check="Result has element" position="1" result="Alesha Dixon" /> <Value field="" fieldName="SearchSong" value="nothing" /> </Find> <Login flag="false" name="Login" number="1.3" optional="true" validity="Invalid"> <Check check="Label" result="login failed. Unknown user or wrong password(_)" /> <Value field="" fieldName="username" value="pbgt"/> <Value field="" fieldName="password" value="testtest"/> <Button field="button_1.3"/> </Login> </Path> </Script> </pre>

8.1.7 Findings

All the teams were able to build the model and fulfilled the expected testing goals. All the models created by all members from the teams, were validated and analyzed manually. Further, a comparison among the models was performed. From this analysis it was possible to verify that the models are similar, only differing in their structure (depth levels). Some used *Forms* to feature further detail while other members opted not to structure elements within *Forms* and displayed all elements in the first level model.

Another important concern of this study was related to the measurement of the time required to create and configure the models to fulfill the testing goals. Moreover, the intention was to know PBGT capability in finding failures. Thus, the results of this case study are represented in Table 8.7 (RQ1.1 and RQ1.2).

Table 8.7: Case study findings concerning the modeling and configuration efforts and failures found.

Subject System	Modeling and Configuration Efforts (minutes)	Revealed failures
SS1.1	M: 30 C: 26	1
SS1.2	M: 44 C: 31	4

One failure was reported in SS1.1. This failure was related with the difference in the number of elements displayed in two different pages, after performing a search on the first page to later be submitted to the second page. The model and its configuration, and the generated test cases can be found in [PBG14].

A total of 4 failures were found in SS1.2. The first failure was caused by selecting “Top 50 Singles” and afterwards performing an invalid login (TC1.3). This resulted in the display of an empty white page. The second failure was triggered after performing a top search looking for forum and afterwards performing an invalid login (TC1.2). This displayed an error message: “Microsoft OLE DB Provider for SQL Server error...”. The third failure was also related with the sequence of actions performed in the UI, namely: performing a simple (or extended) search looking for songs with name “nothing” and performing an invalid login afterwards, does not show the invalid login message as expected (TC1.4). Finally, in the context of the latter, changing the country to “United Kingdom” does not show “Search” neither “UK Charts” in the related area as what happens with other countries (TC1.1). In summary, the failures encountered in this case study were related with: (i) sequence of actions performed in the UI, such as selecting elements after providing invalid logins; and (ii) with dependencies

between related UI elements.

The teams reported some complaints regarding the usability of the modeling environment. They were directed towards the auto-alignment and placement of the connectors between elements, since it was not perfect, in visual terms, which required extra effort to correct.

Furthermore, all teams agreed that the time spent in training (2 hours) was sufficient to understand the PBGT approach and how to use the PBGT tool to model applications (via PARADIGM-ME) and start finding failures.

8.1.8 Threats to validity

The limited number of subject systems in this evaluation can be seen as *external threat to validity*. However, to minimize this problem, several various public web sites ranging in complexity and functionality were used in this case study. Therefore, it was chosen a diverse set of applications that are representative of current web applications featuring a wide range of functionalities.

In terms of *internal validity*, the threat is related with the measurement concerning the modeling and configuration efforts. All teams were leveraged, to start from scratch in the way that they were not familiarized and did not have any experience with PBGT. If the study was repeated but to test other subject systems, the efforts concerning modeling and configuration would probably be lower, since the teams would already have some experience with PBGT.

8.2 Case Study 2: Comparison between PBGT and Capture-Replay

The goal of the second case study was to compare PBGT with capture/replay approaches regarding their ability to find failures and the time spent in testing activities prior to running the tests. The research questions designed for this study were:

RQ2.1 How many failures are detected by the different PBGT test case generation techniques and Capture/Replay?

RQ2.2 What are the manual efforts required by PBGT and Capture/Replay?

This case study occurred 6 months after the completion of case study 1, described previously. During this time interval the PBGT tool (along with its supporting components) was improved, having as basis the feedback obtained from case study 1. Further, a new UI Test Pattern (Option Set UI Test Pattern described in Chapter 6, Section 6.9, p. 81) was added to the pattern-language. A revision on current UI Test Patterns was also performed, and as result additional checks have been added and their configurations have also improved.

8.2.1 Metrics

To address the research questions above, some metrics were collected during the execution of the study. The first metric (**M2.1**) (related to question **RQ2.1**), keeps track of the amount of failures found. The second metric (**M2.2**) (related to question **RQ2.2**) saves information about the effort required to build test models or test scripts. The third metric (**M2.3**) (related to question **RQ2.2**) saves information about the effort required by PBGT during the configuration step. This is an information gathered only for PBGT since capture/replay does not contain this step in the testing process.

8.2.2 Subject Systems

In order to answer the research questions posed, a set of applications available on the web have been selected, with source code available, and with a wide set of features. The reason for the selection of existing web applications was to maintain independence in the performed study and also as a way of emphasize that PBGT approach can be applied to any web application and not just some specific software that had been created by the author for this purpose. Moreover, the selection was about applications with source code available in order to facilitate the usage of fault seeding techniques to compare which of the approaches could find more failures. Thirdly, the applications had to contain numerous windows and widgets, and a reasonable set of features, in order to have a broader assessment of the capabilities of both testing tools. Therefore, the following subject systems were selected:

- **SS2.1** – Tudu Lists [Dub13]. This is a simple application that manages todo lists. These lists can be accessed, edited and shared. It is possible to create, edit, delete lists and also todos. Further, todos can be filtered according to criteria, and Tudu also allows to manage user information. Moreover, these

functionalities are only available upon successful authentication.

- **SS2.2** – Task Freak [Ozi14]. This is a web based task manager. It allows to create, edit and delete projects and also tasks. It is possible to filter them by several contexts (work, document, meeting, among others) and also to sort. It features an authentication mechanism (username and password). It also provides the ability to add comments to projects and tasks.
- **SS2.3** – iAddressBook [Wac14]. This is a web based application to manage contacts. It allows to create, edit and delete contacts; delete multiple contacts; create and delete categories for the contacts; add/remove contacts to/from categories; filter contacts and; search for contacts.

8.2.3 Testing Goals

The defined testing goals for all subject systems are listed in different tables, namely: Table 8.8 displays testing goals for SS2.1; Table 8.9 for SS2.2; and Table 8.10 displays the testing goals for SS2.3. These tables contain IDs that will be used to refer the testing goal in question and its description.

Table 8.8: Defined testing goals for *Tudu Lists* (SS2.1).

ID	Description
TG2.1	Test the add new list functionality
TG2.2	Test the edit current list functionality
TG2.3	Test the delete current list functionality
TG2.4	Test the add quick Todo functionality
TG2.5	Test the add advanced Todo functionality
TG2.6	Test the edit Todo functionality
TG2.7	Test the delete Todo functionality
TG2.8	Test the complete Todo functionality
TG2.9	Test the show other Todos functionality
TG2.10	Test the delete completed Todos functionality
TG2.11	Test the “Next 4 days” filter
TG2.12	Test the “Assigned to me” filter
TG2.13	Test the authentication functionality
TG2.14	Test the edit user data functionality
TG2.15	Test the logout functionality

Table 8.9: Defined testing goals for *Task Freak* (SS2.2).

ID	Description
TG2.16	Test the login functionality
TG2.17	Test the logout functionality
TG2.18	Test the access user data page functionality
TG2.19	Test the delete user (as admin only) functionality
TG2.20	Test the see project functionality
TG2.21	Test create new project (as admin only) functionality
TG2.22	Test the edit project (as admin only) functionality
TG2.23	Test the delete project (as admin only) functionality
TG2.24	Test the add new task (by table and by menu) functionality
TG2.25	Test the edit task (by icon and by table entry) functionality
TG2.26	Test the delete task functionality
TG2.27	Test the change task completion status functionality
TG2.28	Test the add new comment functionality
TG2.29	Test the edit comment functionality
TG2.30	Test the sort tasks functionality
TG2.31	Test the filter by user functionality
TG2.32	Test the filter by context functionality

Table 8.10: Defined testing goals for *iAddressBook* (SS2.3).

ID	Description
TG2.33	Test the select contact functionality
TG2.34	Test the add new contact functionality
TG2.35	Test the edit contact functionality
TG2.36	Test the delete contact functionality
TG2.37	Test the delete several contacts (bulk delete on contact list) functionality
TG2.38	Test the add to category functionality
TG2.39	Test the delete from category functionality
TG2.40	Test the create category functionality
TG2.41	Test the delete category functionality
TG2.42	Test the contact's initials filter functionality
TG2.43	Test the category filter functionality
TG2.44	Test the search functionality

8.2.4 Setup

This study was performed by two separate teams consisting of master students. The first team (to be referred to as Team 2.1) used the PBGT approach according to the following steps: construction of test models based on the defined testing goals, configuration of those models, selection/configuration of the test case generation

algorithm, generation of test cases and execution of test cases. The second team (to be referred to as Team 2.2) applied capture/replay techniques to test the subject systems. This team used the Selenium IDE tool for constructing the test scripts according to the defined testing goals and for the execution of those tests. Selenium IDE is a plugin that is installed in a web browser and allows the tester to capture all his actions in a website and replicate them later on. These teams have not been in contact with each other.

Selenium was chosen because the PBGT tool uses Selenium libraries in its core, and so it makes sense to see what advantages this approach could bring over regular Selenium testing. Both of the tools are black-box testing methods, as none of them has access to the application source code. They both perform the tests by following through a sequence of actions previously defined (a PARADIGM model in the case of PBGT and a test script in the case of Selenium). This comparison featured all test case generation algorithms used by PBGT (*Default*, *Random* and *Invalids*).

To answer the research question **RQ2.1**, each web application was seeded with faults and it was measured how many failures each approach, PBGT and capture/replay, could find. In addition, test case generation techniques provided by PBGT (described in 5.5) were also used to compare these techniques. To address **RQ2.2** the time was measured, in minutes, concerning the effort required by the teams to create and configure the models (for the PBGT approach) and also the time spent in creating scripts (for the capture-replay approach).

To run the study, the URL of the original version of each system and also the URL of the corresponding versions with the seeded faults was provided to both teams. The procedure followed to perform this study is described below.

8.2.5 Procedure

Firstly, a preparation of the environment for the execution of the study was required: altered versions of the original applications were created by means of a custom developed tool with the same behavior as *muJava* [MOK05] using mutation operators. Each mutant was made available in a different URL. Then, the applications to test were presented to both teams in separate sessions. The URL of the original and altered versions of the three different applications was also provided.

The first team (PBG) created models to fulfill the testing goals. Then, they configured the models based on each original version of the subject system, and

then they executed the test cases in each mutated version of the subject systems.

The second team created a set of scripts, based on the testing goals, using Selenium IDE for each unaltered subject system. While navigating through the subject systems, Selenium IDE would automatically record each action they performed. After the scripts were recorded, they were replayed in each mutated version of the subject system.

During the realization of the study, both teams, gathered information about the time to build the model, and the time to execute the test cases.

8.2.6 Results

One example of a model written in PARADIGM for PBGT, regarding SS2.1 is illustrated in Figure 8.4. In Tudu Lists the user is able to make a new user registration. In order to achieve the latter, the user needs to access the registration area. This is modeled with a Call UI Test Pattern (element number 1 labeled “Register Call”). Then, the user is redirected to a registration form (element number 2, labeled “Register”) where he provides the necessary data to register a new user. In order to login, the user needs to select a link (called “Welcome”) that will lead to the login screen. This is modeled with a Call UI Test Pattern (element number 10, labeled “MainPageCall”). The login screen is modeled as a Login UI Test Pattern (element number 3, labeled “Login”). After a successful login the user is able to access, without any specific order (element number 4 labeled “Logged”) to his todos (element number 8, labeled “myTodos”) or to access his user information (element number 6, labeled “myInfo”). The user is also able to perform logout (element number 9, labeled “Logout”). More details about the models, configurations and generated test cases can be found in [PBG14].

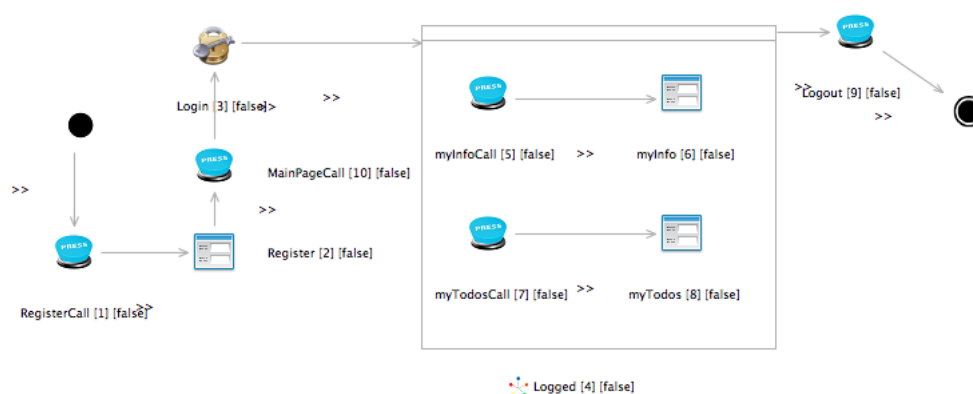


Figure 8.4: Tudu Lists model in PARADIGM.

Models written in PARADIGM for PBGT require configuration. The configuration occurs in the PBGT tool and it is related with the UI Test Patterns featured in the models. For each UI Test Pattern, the tester defines the input data and selects the checks to perform. A part of the configuration for the model displayed in Figure 8.4, is represented in Table 8.11.

Table 8.11: Configuration details for the model from Figure 8.4.

Name	Type	Validity	Variables	Check
RegisterCall	Call	NA	{}	{Change to Page "New user registration"}
MainPageCall	Call	NA	{}	{Change to Page "Welcome to Tudu Lists"}
Login	Login	Valid	{[username, "john doe"]; [password, "pass"]}	{}
		Invalid	{[username, "test"]; [password, "test"]}	{Label "Welcome to Tudu Lists!"}
myInfoCall	Call	NA	{}	{Change to Page "Manage user information"}
myTodosCall	Call	NA	{}	{Change to Page "Actions"}
Logout	Call	NA	{}	{Change to Page "You have left Tudu Lists."}

8.2.7 Findings

The teams had no major problems during the execution of the case study. They were able to fulfill all testing goals. The efforts required to build and configure models with PBGT and CR are displayed in Table 8.12 (RQ2.2).

Creating scripts for CR took less time than crafting and configure models for PBGT. On average, from the universe of 3 subject systems, CR took 46 minutes less than PBGT. In detail, for SS2.1 (Tudu Lists), PBGT took 13 extra minutes compared to CR. For SS2.2 (TaskFreak), PBGT registered additional 63 minutes and for SS2.3 (iAddressBook) 62 minutes.

The findings regarding the maximum of mutants killed for each subject system concerning PBGT and CR approaches, are represented in Table 8.13 (RQ1). For

Table 8.12: Efforts required to build and configure models/scripts for each subject system.

Subject System	GUI Testing Approach	Time (minutes)	Difference (PBGT – CR)
Tudu Lists	PBGT	73	+13 minutes
	Capture-Replay	60	
TaskFreak	PBGT	164	+63 minutes
	Capture-Replay	101	
iAddressBook	PBGT	137	+62 minutes
	Capture-Replay	75	

SS2.1 (Tudu Lists), 35 faults were seeded, for SS2.2 (TaskFreak) 82 faults and for SS2.3 (iAddressBook) 73 faults.

Table 8.13: Maximum number of mutants killed for each subject system.

Subject System	GUI Testing Approach	Seeded Faults	Mutants Killed	PBGT – CR
Tudu Lists	PBGT	35	32	+16
	Capture-Replay		16	
TaskFreak	PBGT	82	73	+20
	Capture-Replay		53	
iAddressBook	PBGT	73	68	+12
	Capture-Replay		56	

For instance, for SS2.1, the PBGT approach was able to kill a maximum of 32 mutants, while the CR approach killed 16. PBGT was able to kill additional 16 mutants when compared with CR. Overall, PBGT was able to kill more mutants than CR, making an average of 16 additional mutants killed.

The results regarding the comparison of different PBGT test strategies (**RQ2.1**) can be seen in Table 8.14. The table displays how many faults were seeded into the SUTs, how many mutants were killed, the average number of Test Cases (TC) required to kill one mutant and the average time (in minutes) to do so. For example, considering SS2.1, 35 faults were seeded, and the *Default* strategy was able to kill 24 mutants. In addition, the average number of test cases required to kill the latter mutants was 1.3, making an average of 9.1 minutes to achieve it.

8.2.8 Threats to Validity

In terms of *external validity*, the threats are related with the number of subject systems used, and the number of GUI testing approaches that are compared

Table 8.14: Number of mutants killed in each PBGT test strategy.

Subject System	Strategy	Faults	Mutants Killed	TC/mut. (avg)	Time/mut. (avg)
Tudu Lists	Default	35	24	1.3	9.1
	Random		32	5.2	26.5
	Invalids		27	3.7	18.5
TaskFreak	Default	82	59	2.3	18.1
	Random		73	6.1	51.2
	Invalids		69	4.0	33.8
iAddressBook	Default	73	60	1.2	8.8
	Random		68	5.7	41.2
	Invalids		67	3.4	25.7

with PBGT. The number of subject systems is limited, but are representative of modern open-source web applications, with different functionalities, and were implemented in different programming languages. In addition, the results gathered were obtained by applying test cases over 190 mutants which requires analysis time. In terms of *internal validity*, the threat can be related to the measurement of modeling effort. Nevertheless, the work was done by different teams unfamiliar with the testing approaches used (PBG and Selenium) and therefore were teams who started the work from scratch. If the study is repeated with teams with some prior knowledge of the different testing approach, the time spent in modeling may be different but in that situation it will be possible to argue that the prior knowledge may be different and the results may not be valuable either.

8.3 Case Study 3: Analysis of Modeling Effort

The goal of this case study is to measure the feasibility of the PBGT approach, starting from the modeling phase until the generated test cases output. More specifically, to assess the effort required to model a GUI, to configure the UI Test Patterns and the capability to model common UIs. Furthermore, it is designed to measure the acceptance level of PARADIGM, among GUI testers. This evaluation addresses the following research questions:

- **RQ3.1:** What are the overall modeling efforts required to craft a model with PARADIGM, when compared with other Model-Based GUI Testing approaches?

- **RQ3.2:** Are current language elements and connectors adequate and effective to model UIs, with the goal to finding errors?
- **RQ3.3:** What is the graphical effectiveness and power of PARADIGM?

8.3.1 Metrics

In order to address the research questions above, some metrics were collected during the execution of the study. The first metric (M3.1) (related to question RQ3.1), keeps track of the time in minutes concerning the modeling efforts. The second metric (M3.2) (related to question RQ3.2) registers information concerning RQ3.2. The third metric (M3.3), related to question RQ3.3, saves information about the feedback gathered from testers.

8.3.2 Subject Systems

A collection of subject systems was selected to prepare this study. They were selected based on the diversity of functionalities. The selected subject systems were:

- **SS3.1** – Italian charts portal [Hun14b]. This is a public site that reflects the best selling music singles and albums in Italy.
- **SS3.2** – The Address Book [HH13]. The Address Book is an open source address management system, that in short, allows to create unlimited number of addresses and associated information.
- **SS3.3** – Professional Calendar/Agenda [Men14]. This application is written in C#, and provides a calendar view for creating appointments and day events.
- **SS3.4** – zkCalendar [Cor14]. This tool features scheduling functionality, with daily, weekly and monthly views.
- **SS3.5 and SS3.6** – Industrial Applications. Two industrial applications that due to confidentiality terms cannot be revealed.

8.3.3 Setup

Prior to start the study, two different teams were selected: one comprised by research students and the other comprised by professionals working in a software company. The first team was assigned SS3.1, SS3.2, SS3.3 and SS3.4. The second team was assigned SS3.5 and SS3.6.

To address **RQ3.1** three Model-Based GUI Testing approaches have been selected, namely: Spec#, VAN4GUIM, and PARADIGM. With these approaches, models have to be crafted manually.

In order to address **RQ3.1** it was measured the time (in minutes) required to create and configure the models using the 3 different approaches mentioned before. To address **RQ3.2** and **RQ3.3** feedback was gathered from the teams to verify if they were able to model the proposed UIs and to assess language acceptability, graphical effectiveness and communicative aptitude. Thus, PARADIGM was evaluated according to Moody's criteria [Moo07], in respects to 8 dimensions: (1) Discriminability – consists in verifying if language elements are easy to differentiate and are easy to see; (2) Perceptual and Cognitive limits – level of perceptiveness concerning the amount of information (elements) displayed in the model; (3) Emphasis – level of relevance of elements, in order to capture end-users attention; (4) Perceptual directness – are “*representations whose interpretation is spontaneous or natural*” [Moo07], i.e., icons resemble the objects they characterize; (5) Structure – refers to organizing elements into perceptual groups; (6) Identification – models, language elements and links should be labeled; (7) Expressiveness – number of different visual variables used to encode information (shape, colors, size, value, orientation and texture); and (8) Simplicity – the number of different graphical conventions should be limited.

The URL to download the PBGT tool was provided to both teams. After they installed the tool on their machines, they analyzed the subject systems. The procedure that was followed to execute this study is described next.

8.3.4 Procedure

Since all elements from both teams had not been in contact with PARADIGM developers neither had previous knowledge about PBGT, Spec# and VAN4GUIM, it was necessary to provide additional training about these three approaches. Two different sessions took place, during which they had the opportunity to clarify doubts in order to be prepared to start the experiments.

After the sessions, a collection of testing goals, for each subject system, was

hand-out to both teams. The teams started the crafting of models with the aim to fulfill the proposed testing goals. Once the teams concluded the construction of the models, a questionnaire was delivered to them, featuring the eight dimensions according to Moody's (as described early) criterion. For each dimension, the teams were able to select only one value of the following scale (in terms of perception): High – 3; Medium – 2; Low – 1; Do not know/blank – 0. In addition, VAN4GUIM was also evaluated according to the same criteria, in order to perform a comparison between PARADIGM and VAN4GUIM. Spec# was not included in this comparison, since it is not a graphical language. After completing the questionnaires, all elements of the teams handed over them.

8.3.5 Results and Findings

After completion of the study, the following data was obtained. Table 8.15 registered the data to address RQ3.1. This table displays the time measurements, in minutes, concerning the modeling (displayed as **M** in the table cells) and configuration (displayed as **C** in the table cells) efforts that were required for the subject systems. For instance, focusing on subject system SS3.1, the configuration efforts for the Spec# approach were 303 minutes for modeling and 46 minutes for configuration. Further, the configuration efforts using VAN4GUIM for the same subject system, were 181 minutes for modeling and 41 for configuration. Still for subject system SS3.1, the modeling efforts estimated in 62 minutes and configuration efforts in 20 minutes.

Overall and by analyzing the data from Table 8.15, concerning the crafting of the models, Spec# took on average 343 minutes, VAN4GUIM registered an average of 243 minutes and PARADIGM an average of 64 minutes. In respects to configuration concerns, on average, Spec# took 48 minutes, VAN4GUIM 45 minutes and PARADIGM 31 minutes.

The obtained data regarding the evaluation of PARADIGM and VAN4GUIM, according to Moody's quality criteria, is display in Table 8.16. This table displays the data to fulfill RQ3.2 and RQ3.3.

Each table cell shows the average value for each dimension of each method. The total represents the sum of the cells dividing by all dimensions (8). For instance, for the *discriminability* dimension PARADIGM registered an average of 2.2 and VAN4GUIM an average of 2.0. The average rating (concerning all dimensions) for PARADIGM according to Moody's criteria, was 2.7, while VAN4GUIM registered a rating of 2.6.

Table 8.15: Case study findings concerning the modeling and configuration efforts for all approaches.

Subject System	Method		
	Spec#	VAN4GUIM	PARADIGM
SS3.1	M:303 C:46	M:181 C:41	M:62 C:20
SS3.2	M:210 C:34	M:121 C:31	M:32 C:14
SS3.3	M:177 C:32	M:110 C:27	M:26 C:12
SS3.4	M:183 C:40	M:125 C:38	M:30 C:22
SS3.5	M:546 C:61	M:421 C:57	M:104 C:50
SS3.6	M:671 C:76	M:498 C:75	M:131 C:68

Table 8.16: PARADIGM and VAN4GUIM findings concerning Moody's quality criteria, with input collected from the teams.

Quality Criteria	PARADIGM	VAN4GUIM
Discriminability	2.2	2.0
Perceptual and cognitive limits	2.8	2.5
Emphasis	2.9	2.8
Perceptual directness	2.6	2.5
Structure	2.8	2.8
Identification	2.7	2.5
Expressiveness	2.8	2.6
Simplicity	2.7	2.7
Total	21.5 / 8 \simeq 2.7	20.4 / 8 \simeq 2.6

8.3.6 Threats to Validity

In respects to *external validity*, more subject systems could have been selected in order to support the attained results. Yet, the selection of five subject systems, in which two of them were industrial systems with considerable amount of features, can prove expressive to the intents of the study.

As to *internal validity* is concerned, the threat can be directed towards the measurement of modeling and configuration efforts. All teams started with the same level of knowledge concerning all approaches. If the study was repeated with the same testers, with the aim of testing different subject systems the results would be different and lower, since they already had knowledge about all approaches.

Regarding Moody's quality criteria, the perception of each dimension is variable for each tester. This means that if the study was repeated with different teams, the results could be different.

8.4 Discussion

At the end of the case studies it was possible to analyze the results and findings in order to answer the research questions.

Case Study 1

After *case study 1*, it became apparent that testers without any knowledge of PBGT approach could begin to use and obtain the benefits from its use immediately after a training session of 2 hours. This also indicates that the approach is easy to learn. Furthermore, during this study, testers were able to successfully accomplish the testing goals and were able, by themselves, to create and configure models. In addition, testers were also able to find failures in fielded applications.

Case Study 2

At the end of *case study 2*, it was possible to notice that the effort to build PARADIGM test models is greater than to create test scripts in Selenium. The extra effort required to model the application in PBGT (as opposed to the *crawling* nature of Selenium, in which the testers simply interact with the application and their steps are recorded), compensates in the number of mutants killed.

There were some mutants killed by PBGT that were not killed by Selenium-IDE. For instance, in the TaskFreak application (SS 2.2), there is a specific mutant that allows a user to create tasks for a project that does not belong to that specific user (Figure 8.5). Typically, the tester first selects the user and then selects the project. If that specific user does not belong to the project, the user will be automatically changed to a default one. This mutant prevents the change. Since the capture-replay scripts follow a linear approach, filling the fields in order, led to this mutant not being killed. In addition, given that these fields are marked as "AnyOrder" in the PARADIGM model, there are several test cases in which the user field is altered first than the project field. Some mutants affected web elements that Selenium-IDE could not find in the web page. Considering TaskFreak (SS 2.2), when the user modifies the status (percentage of completion) of a task, color images appear

Priority: Context:

Deadline:

Project: > new project?

Title:

Description:

User: public internal private

Status:

Figure 8.5: TaskFreak GUI featuring project properties.

(as illustrated in Figure 8.6, featuring the colored squares on the right). In this figure, the “teste” task is at 40% (each square located at the most right of the Figure 8.6 corresponds to 20%). These images cannot be detected by Selenium, and as a response, PBGT uses Sikuli to verify this behavior. Therefore, the mutants that change the behavior of the color images cannot be killed by Selenium. Even



Figure 8.6: TaskFreak GUI featuring the details of a project.

though PBGT uses Selenium libraries to identify web elements, it can also use Sikuli to identify and interact with web elements via image recognition. Sikuli takes a picture of the element, during the mapping phase, to facilitate the finding of the element during the test execution. This is mostly useful for applications that work within their own black-box technology, such as Adobe Flash [Ado14] or Microsoft Silverlight [Mic14b] based applications. In addition, for pop-up windows/dialogs, if Selenium-IDE is unable to interact with them, PBGT will be able to do so using Sikuli.

Sikuli is able to interact with several controls allowing to test further behavior of the application. Yet, this is not the only advantage of using Sikuli, and it is not the reason why PBGT is able to kill more mutants. An important issue is undoubtedly the set of generated test cases by PBGT as opposed to the number of scripts built with Selenium. While the Selenium team created an average of 5 to 7 scripts to fulfill the test goals defined for each web application, the PBGT tool was able to generate hundreds of different test cases from each model (for Tudu Lists alone, 245 test cases were generated). This allowed to test sequences of actions that were not tested by Selenium scripts.

An additional important remark is related with the set of checks included in the PBGT models and Selenium scripts. After the construction of the PARADIGM

models, the PBGT team configured each UI Test Pattern, which included the checks to run. Regarding Selenium, the checks were defined along with the construction of the test scripts. It was possible to notice that the PBGT models (and consequently the test cases generated from them) had more checks than the Selenium scripts. It seems that having an independent phase after building the model (like PBGT has) contributes to have a wider perspective of the testing goals, enabling to think more thoroughly about the checks to run in each particular position of the model.

With PBGT, the test input data is also defined during the configuration phase. After constructing a PARADIGM model, it is simple to continuously add different input data and checks, while the model remains the same. By adding more configurations, it is possible to increase the number of tests cases and increase the percentage of code coverage. With Selenium, in order to add a different set of inputs and checks, the tester has to create different test scripts or, otherwise, convert scripts to a programming language (like Java for instance). Then, the tester can modify the code to build parameterized scripts that can be executed with different input data. These parameterized scripts are difficult to maintain when the original scripts require modifications.

Another significant difference between PBGT and Selenium is the moment upon which testers can start their testing activities. Although the study was based on existing applications, the PBGT approach can also be useful during the development of software applications. The construction of the PARADIGM models can start when eliciting requirements takes place. PARADIGM models can be built during the requirements elicitation phase, when there is no code developed yet. During the development, the PARADIGM models can be updated, if required, and later configured. The GUI itself (containing code) is only required during the mapping phase. With Selenium, the tester can begin his activities, by building scripts only when the application is implemented and running.

Regarding the comparison of different strategies for test case generation, and more specifically the comparison of *Invalids* with *Default* strategies, it was possible to verify that the *Invalids* strategy killed the majority of the mutants that were killed by *Default* strategy. However, it required additional test cases to realize it. Nonetheless, the *Invalids* could kill some mutants that *Default* could not kill. These were the mutants that affected the authentication functionality, that allowed invalid users to access private data, and causing incorrect data input verification. Comparing the *Default* algorithm with the *Random* one, it was possible to verify that the *Random* strategy was able to kill more mutants than the *Default* strategy

(it killed some that were only killed by the *Invalids* strategy), but it would take more time and test cases to kill them. In addition, since the algorithm is in fact entirely random, some test cases were repeated and thus more time was spent to kill different mutants.

In conclusion, the *Default* strategy can kill a significant number of mutants in a short time, while the *Random* strategy kills more mutants, but it consumes additional time. Further, since the *Default* strategy only runs one test case per test path, it always picks the first set of configurations that the user defined. Most of the time, the first set of configurations defined represent a valid set, to ensure the system works as expected. Additional configurations usually test the application with more unexpected data, such as invalid characters, different data types (e.g. input a string where a number was expected) and invalid credentials (such as username and password). This type of data can be considered to be more error-prone. With the *Random* strategy, the algorithm can choose these types of configurations as well as the valid ones, increasing the probability of killing a mutant. The *Invalids* strategy should be used when the tester wants to check authentication or validation issues in the web application.

The failures found by PBGT can be classified in two main groups: (1) the behavior is different than the expected and (2) the test driver tries to interact with GUI objects (widgets) that do not exist or are disabled. In the first group, there are the failures caused by widgets, such as buttons, text boxes and links, that stop working or function in a different way than expected; failures preventing the entry of correct data; authentication failures that allow access to restricted data; and data inputs ignored. In the second group, we can find failures caused by dynamic content which ceased to exist; and the blocking of the application caused by, for example, loss of connectivity to the database.

Finally, PBGT was not compared with any model-based test generation approach, such as the one used by GUITAR. A fundamental reason for skipping this comparison is that the approaches are incomparable. PBGT helps us to generate test input (sequences of events) as well as the test oracle (the mechanism that determines whether a test case passed or failed). GUITAR, on the other hand, generates only the test input; for the test oracle, it relies on software crashes, which in this case study was irrelevant, or reference testing, in which it compares the full current GUI state with that of a reference “golden” version available during regression testing, again irrelevant for this study. The long-term goal is to incorporate the PBGT approach with the algorithms used by GUITAR, so as to enhance both approaches.

Case Study 3

In terms of modeling efforts, from an universe of 6 subject systems, PARADIGM took less time than VAN4GUIM and even less than Spec#, in terms of crafting and configuring models. It is important to note that with Spec# and VAN4GUIM the focus is directed towards modeling user actions, which are demanding. However, with PARADIGM the modeling is focused on testing goals, which saves considerable time.

Models crafted in Spec# describe a state transition system, where states are modeled as *state variables* and methods are denoted as *actions*. *Actions* represent transitions within the system. Spec# has two different kinds of actions: *probe actions*, which do not update the internal state of a modeled system but only read its state, are useful to perform checks during the test case execution phase; and *controllable actions* update the state of the system and are used to model possible user actions. Further, the Spec# needs to be created manually, which is in fact very demanding. VAN4GUIM hides the formalism details of Spec# by providing a visual modeling front-end and a translation mechanism of such models to Spec#. Due to this reason the modeling efforts were lower when compared to writing models in Spec#, since instead of having to manually create the model according to the formalism details of Spec#, testers were able to do this graphically. Also, since VAN4GUIM has a built-in mechanism that translates VAN4GUIM models to Spec# it also saved considerable time. Yet, VAN4GUIM requires that probe actions need to be manually added to the translated Spec# models. Although this behavior needs to be performed manually, the modeling effort needed by VAN4GUIM notation is smaller when comparing with the development of Spec# models from scratch.

Concerning PARADIGM acceptability, graphical effectiveness and communicative aptitude, results indicate that PARADIGM registered a better rating than VAN4GUIM. PARADIGM registered a rating of 2.7, while VAN4GUIM 2.6. This means that the higher the proximity towards value 3, the higher and better the tester perception is. Given the obtained findings, it was possible to realize that PARADIGM language elements are expressive, easy to identify, effective and graphically rich, which facilitates PARADIGM's understanding and acquaintance by testers.

8.5 Summary

This chapter presented three case studies that were designed and executed to validate the PBGT approach and its supporting processes. The goal of the first case study was to measure the effectiveness of the PBGT approach including the effort required to start using it. Testers were capable to find failures in two fielded applications. Testers were also able to obtain immediate results in short time, after learning the PBGT approach. The second case study aimed at comparing PBGT with capture/replay approaches in terms of failure detection ability and time spent in testing. PBGT required more efforts in modeling and configuration than capture/replay. However, PBGT was able to find more failures than capture/replay. Case study three was designed to measure the efforts required to build PARADIGM models when compared with other modeling approaches like VAN4GUIM and Spec#. In addition, it also intended to assess PARADIGM language acceptance and graphical richness among testers. PARADIGM required less modeling efforts when compared with other approaches. Testers found that PARADIGM language elements are expressive, easy to identify and are graphically rich. The next chapter reasons about the dissertation and points future work directions.

9

Conclusions

The PBGT approach is an innovative model-based GUI testing technique that aims to promote reuse of GUI testing strategies. It introduces a new concept – UI Test Patterns – which define generic test strategies that are able to test different implementations of UI patterns. UI Test Patterns represent behavioral elements within PARADIGM – a DSL created specifically for PBGT – that allows to craft models describing testing goals. The DSL was engineered according to guidelines and best practices, and followed a systematic approach for its development. PBGT also features collection of UI Test Patterns that are subjected to be used in PBGT. PBGT is supported by an integrated modeling and testing environment (PBGT tool) that is comprised by a set of components that make the approach more forceful, easy to use and to adopt.

The PBGT approach fills a void between design and quality assurance. It leverages UI patterns that are commonly used for GUI design by providing a systematic approach that allows to test multiple implementations of UI patterns, aiming to ensure their functional correctness. This approach is able to improve current GUI modeling and testing techniques, by reducing the effort required to craft models and by directing the testing focus towards testing goals. It is evolutionary as it is able to evolve by supporting the addition and improvement of new testing strategies (and therefore new UI Test Patterns) in order to cope

with future UI tendencies. The PBGT process is effective, feasible, with a low learning curve. PBGT also requires diminished efforts, when compared with other MBT approaches, in modeling and configuring SUTs. These preceding remarks are substantiated by the empirical evaluation that was performed for PBGT via three case studies.

This chapter describes the main contributions of this dissertation and indicates future research directions.

9.1 Summary of Contributions

The contributions of this dissertation are:

- **A new model-based GUI testing paradigm (PBGT) for GUI modeling and testing**, described in Chapter 5 “Pattern-Based GUI Testing” (p. 45), that aims to promote reuse of GUI testing strategies. With PBGT the focus is directed towards modeling testing goals and therefore, the state space explosion problem, known among other MBGT testing techniques, is less prone to occur, since the state space is greatly reduced. The modeling efforts are lower when compared with other GUI modeling approaches.
- **The design of an innovative testing process**, described in Chapter 5 “Pattern-Based GUI Testing” (p. 53), with the goal to define the activities (including the PBGT components to support the process) and to guide testers during their testing activities.
- **A bridge to address the gap between the design and quality assurance of GUIs**, by leveraging UI Patterns featured in GUIs, towards the notion of UI Test Patterns. To the best of the author’s knowledge this is the first time that patterns are applied to GUI testing.
- **A collection of UI Test Patterns** that are described according to guidelines from the field of Pattern Languages (described in Chapter 6, “Pattern Library”, p. 63). Hence, it provides guidance and shares knowledge for testers and software engineers that want to use the PBGT approach and, therefore, starting benefiting from it in short time. Furthermore, these patterns act as guidance on how to effectively use UI Test Patterns for GUI Testing, which is a new emerging concept, and thus allow to achieve a common understanding on their usage.

- **A domain-specific language (DSL)** to be used in the context of PBGT, including a detailed description, according to best practices and guidelines, of the engineering process that was followed to develop a new DSL from scratch (described in Chapter 7, “DSL Engineering for GUI Modeling”, p. 85). In addition, the language provides the capability to extend the initial set of UI Test Patterns, to adjust current test strategies (or create new ones) to fulfill new UI trends.
- **A novel approach using Alloy in DSL engineering**, describing a methodology, called MAIDEN (detailed in Chapter 7, “DSL Engineering for GUI Modeling”, Section 7.4.2, p. 92), to find and tune DSL language constraints using Alloy.

9.2 Future Research

Although PBGT has already been used at industry level, it would be interesting to see further adoption. The goal would be to promote the PBGT approach in the testing community, along with sessions/talks at software companies so test teams could see the value of PBGT, and the benefits of its adoption.

More UI Test Patterns can be researched and later added to the PARADIGM language, in order to keep up the latest progress in the field of GUI trends. For instance, specific mobile events like swipe and long press. The reverse-engineering component (PARADIGM-RE) can also be improved by adding further heuristics and rules that would allow to explore and identify a larger set of UI patterns on different platforms.

Regarding the PBGT tool, at this moment the tool is not able to test iOS applications. The aim is to develop and integrate drivers for this purpose in the near future. Another point of improvement is related with usability, as it could facilitate even more the configuration of UI Test Patterns. The generated reports (failures found) will also be subject of enhancement. Another functionality on the product roadmap is the ability to share UI Test Patterns among PBGT’s testers, allowing to promote collaboration among them. Internationalization of the tool will also take place in the future, which will facilitate the promotion and adoption of the tool, to a wider market.

Test case generation in PBGT is an area that can be improved. The goal would be to investigate and analyze further test case generation algorithms, including additional test strategies and filtering options and identify where each of them

can be more effective.

Bibliography

- [ABPS12] Stephan Arlt, Cristiano Bertolini, Simon Pahl, and Martin Schäf. Trends in Model-based GUI Testing. *Advances in Computers*, 86:183–222, 2012. Cited on pages 33 and 40.
- [Ado14] Adobe. Flash Player | Adobe Flash Player | Overview. Available from <http://www.adobe.com/products/flashplayer.html>, 2014. Cited on page 134.
- [AFT⁺14] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 2014. Cited on page 35.
- [Agu12] Ademar Manuel Teixeira de Aguiar. *Framework documentation: A minimalist approach*. PhD thesis, 2012. Cited on page 55.
- [AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oxford, 1977. Cited on pages 63 and 64.
- [Ama14] Inc. Amazon.com. Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more. Available from <http://www.amazon.com/>, 2014. Cited on pages xi and 25.
- [App14] Apple. Cocoa Bindings Programming Topics: Creating a Master-Detail Interface. Available from <https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/Tasks/masterdetail.html>, 2014. Cited on page 46.
- [Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. Cited on page 89.
- [BHS07] Frank Buschmann, Kevlin Henney, and Douglas C Schmidt. Past, present, and future trends in software patterns. *Software, IEEE*, 24(4):31–37, 2007. Cited on page 63.
- [Big98] Ted J Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5(1):169–226, 1998. Cited on page 104.
- [Bin00] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000. Cited on page 17.

- [BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008. Cited on page 89.
- [BM07] Penelope A Brooks and Atif M Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342. ACM, 2007. Cited on page 30.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, and Michael Stal. Pattern-oriented software architecture, volume 1: A system of patterns, 1996. Cited on page 20.
- [BRB14] Gigon Bae, Gregg Rothermel, and Doo-Hwan Bae. Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study. *Journal of Systems and Software*, 97:15–46, 2014. Cited on page 40.
- [BZ14a] Chris Bank and Waleed Zuberi. *Mobile UI Design Patterns*. UXPin, 2014. Cited on pages 1 and 64.
- [BZ14b] Chris Bank and Waleed Zuberi. *Web UI Design Patterns*. UXPin, 2014. Cited on pages 1 and 64.
- [CG12] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): a definitive guide. In *Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering*, SFM’12, pages 58–90, Berlin, Heidelberg, 2012. Springer-Verlag. Cited on page 103.
- [Chi96] Ram Chillarege. *Orthogonal Defect Classification*. Handbook of Software Reliability Engineering, ed. Michael R. Lyu (Los Alamitos, CA): IEEE Computer Science Press, 1996. Cited on page 17.
- [Chi10] Ram Chillarege. Fault vs Failure. Available from <http://www.chillarege.com/content/fault-versus-failure.html>, 2010. Cited on page 17.
- [CJ02] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, 2002. Cited on page 54.
- [CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, first edition, 2007. Cited on page 88.
- [Cor14] Potix Corporation. ZK Calendar. Available from <http://www.zkoss.org/product/zkcalendar>, 2014. Cited on pages 80 and 129.
- [CRS⁺13] Senthil K Chandrasegaran, Karthik Ramani, Ram D Sriram, Imré Horváth, Alain Bernard, Ramy F Harik, and Wei Gao. The evolution, challenges, and future of knowledge representation in product design systems. *Computer-Aided Design*, 45(2):204–228, 2013. Cited on page 14.
- [Dub13] Julien Dubois. Tudu lists. Available from <http://www.julien-dubois.com/tudu-lists.html>, 2013. Cited on pages 70, 80, 82, and 121.

- [DWSG14] Jack Dorsey, Evan Williams, Biz Stone, and Noah Glass. Login on Twitter. Available from <https://twitter.com/login?lang=en>, 2014. Cited on pages [xi](#) and [23](#).
- [EFW01] Ibrahim K El-Far and James A Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2001. Cited on page [65](#).
- [EMNM10] E. Elsaka, W.E. Moustafa, Bao Nguyen, and A. Memon. Using Methods and Measures from Network Analysis for GUI Testing. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 240–246, April 2010. Cited on page [37](#).
- [EvdSV⁺13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*, pages 197–217. Springer, 2013. Cited on page [89](#).
- [Fac14] Facebook. Welcome to Facebook – Log In, Sign Up or Learn More. Available from <http://www.facebook.com>, 2014. Cited on pages [xii](#), [49](#), and [70](#).
- [Fad09] Dimitry Fadeyev. User interface design in modern web applications. *The Smashing Book*, 2009. Cited on page [15](#).
- [Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002. Cited on page [20](#).
- [Fow06] Martin Fowler. GUI Architectures. Available from <http://www.martinfowler.com/eaDev/uiArchs.html>, 2006. Cited on page [19](#).
- [Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. Cited on page [86](#).
- [FRBS04] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head first design patterns*. " O'Reilly Media, Inc.", 2004. Cited on page [19](#).
- [Gal07] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007. Cited on pages [13](#) and [16](#).
- [GB14] Richard C. Gronback and Nick Boldt. Graphical Modeling Framework. Available from <http://www.eclipse.org/modeling/gmp>, 2014. Cited on page [88](#).
- [GC96] Christian Gram and Gilbert Cockton. *Design principles for interactive software*. Springer, 1996. Cited on page [15](#).
- [Ger97] Paul Gerrard. Testing GUI applications. *EuroSTAR'97*, pages 24–28, 1997. Cited on page [17](#).
- [GF99] Dorothy Graham and Mark Fewster. Software Test Automation: Effective Use of Test Execution Tools. *Eua: Addison-wesley Professional*, 1999. Cited on pages [31](#) and [32](#).
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. Cited on pages [20](#) and [64](#).
- [Goo14] Google. Gmail. Available from www.gmail.com, 2014. Cited on page [49](#).

- [Gos05] John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps. Available from <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>, 2005. Cited on page 22.
- [GPU14] GPUUpdate. Formula 1, MotoGP, GP2, GP3, F3 and IndyCar news, photos and much more at GPUUpdate.net. Available from <http://www.gpupdate.net/en/search/>, 2014. Cited on pages xii and 75.
- [Gra14] Grandite. Open Modelsphere – Free Modeling Software Open Source GPL. Available from <http://www.modelsphere.org/>, 2014. Cited on page 89.
- [Gro14a] Inc. Filament Group. Filament group lab example from page from date range picker using jQuery ui 1.6 and jQuery UI CCS framework. Available from http://www.filamentgroup.com/examples/daterangepicker_v2/, 2014. Cited on pages xi and 23.
- [Gro14b] The Hillside Group. Design patterns catalog. Available from <http://hillside.net/patterns/patterns-catalog>, 2014. Cited on pages xi and 24.
- [Ham04] Bill Hamilton. *NUnit pocket reference*. " O'Reilly Media, Inc.", 2004. Cited on page 34.
- [He06] Yujing He. Comparison of the Modeling Languages Alloy and UML. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 671–677. CSREA Press, 2006. Cited on page 92.
- [HH13] David Howe and Lou Huang. The Address Book | SourceForge.net. Available from <http://sourceforge.net/projects/theaddressbook/>, 2013. Cited on pages 80 and 129.
- [Hun14a] Steffen Hung. Australian charts portal. Available from <http://australian-charts.com>, 2014. Cited on pages 67, 70, 73, 75, 80, and 111.
- [Hun14b] Steffen Hung. italiancharts.com – Italian charts portal. Available from <http://italiancharts.com/>, 2014. Cited on pages 68, 70, and 129.
- [IBM12] IBM. IBM – Rational Robot. Available from <http://www-03.ibm.com/software/products/en/robot/>, 2012. Cited on page 33.
- [IDT08] University of Applied Sciences Potsdam Interface Design Team. Pattern Browser – Interface Design Patterns. Available from <http://patternbrowser.org/code/pattern/pattern.php>, 2008. Cited on pages 1, 23, 64, and 78.
- [Int14] Last.fm Ltd. CBS Interactive. Last.fm – listen to free music and watch videos with the largest music catalogue online. Available from <http://www.last.fm/>, 2014. Cited on pages xi and 25.
- [Jac11] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press; 2nd Revised edition, 2011. Cited on pages 91 and 92.
- [KA04] Mikko Karinsalo and Pekka Abrahamsson. Software Reuse and the Test Development Process: A Combined Approach. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques, and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 59–68. Springer Berlin Heidelberg, 2004. Cited on page 55.

- [KMPK06a] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-based testing through a GUI. In *Formal Approaches to Software Testing*, pages 16–31. Springer, 2006. Cited on page 35.
- [KMPK06b] Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara. Model-based testing through a GUI. In *In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*, pages 16–31. Springer, 2006. Cited on pages 35 and 40.
- [KP⁺88] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988. Cited on page 19.
- [Kru92] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992. Cited on page 55.
- [KV10] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010. Cited on page 89.
- [LHRM07] Ping Li, Toan Huynh, Marek Reformat, and James Miller. A practical approach to testing GUI systems. *Empirical Software Engineering*, 12(4):331–357, 2007. Cited on page 30.
- [LIRC12] Bongshin Lee, Petra Isenberg, Nathalie Henry Riche, and Sheelagh Carpendale. Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2689–2698, 2012. Cited on page 14.
- [LW06] Kanglin Li and Mengqi Wu. *Effective GUI Testing Automation: Developing an Automated GUI Testing Tool*. Wiley, Hoboken, NJ, 2006. Cited on page 33.
- [M⁺04] Vincent Massol et al. *JUnit in Action*. Dreamtech Press, 2004. Cited on page 34.
- [MA07] Sushmita Mitra and Tinku Acharya. Gesture recognition: A survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(3):311–324, 2007. Cited on page 15.
- [Mar03] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003. Cited on page 64.
- [Mat08] Aditya P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008. Cited on page 54.
- [MD96] Gerard Meszaros and Jim Doble. A Pattern Language for Pattern Writing. In *The 3rd Pattern Languages of Programming conference*, pages 1–33, 1996. Cited on page 64.
- [Mem01] Atif M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware). Cited on pages 13, 14, and 40.

- [Mem02] Atif M Memon. GUI testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002. Cited on pages 18 and 30.
- [Men14] Jose Menendez. A Professional Calendar/Agenda View That You Will Use. Available from <http://www.codeproject.com/Articles/38699/A-Professional-Calendar-Agenda-View-That-You-Will>, 2014. Cited on pages 68, 80, and 129.
- [MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and How to Develop Domain-Specific Languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005. Cited on pages 85, 86, 87, and 104.
- [Mic14a] Microsoft. Kinect for Windows – Learn about how Kinect for Windows supports movement, voice, and gesture recognition technology. Find out how to get the Kinect sensor and download the sdk. Available from <http://www.microsoft.com/en-us/kinectforwindows/>, 2014. Cited on page 15.
- [Mic14b] Microsoft. Microsoft Silverlight. Available from <http://www.microsoft.com/silverlight/>, 2014. Cited on page 134.
- [Mic14c] Microsoft. Outlook – Sign In. Available from <https://login.live.com/>, 2014. Cited on page 49.
- [Mic14d] Microsoft. Spec# – Microsoft Research. Available from <http://research.microsoft.com/en-us/projects/specsharp/>, 2014. Cited on page 36.
- [Mic14e] Microsoft. Unit Testing. Available from [http://msdn.microsoft.com/en-us/library/aa292197\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx), 2014. Cited on page 34.
- [Mic14f] Microsoft. What is XAML? Available from <http://msdn.microsoft.com/en-us/library/cc295302.aspx>, 2014. Cited on page 22.
- [mob14] mobile.de. mobile.de – Germany’s Biggest Vehicle Marketplace Online. Search, Buy and Sell Used and New Vehicles. Available from <http://www.mobile.de/?lang=en>, 2014. Cited on pages xii, 67, 70, 73, 75, 78, 80, 81, 83, and 111.
- [MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005. Cited on page 124.
- [Moo07] Daniel Moody. What Makes a Good Diagram? Improving the Cognitive Effectiveness of Diagrams in IS Development. In Wita Wojtkowski, W.Gregory Wojtkowski, Joze Zupancic, Gabor Magyar, and Gabor Knapp, editors, *Advances in Information Systems Development*, pages 481–492. Springer US, 2007. Cited on page 130.
- [MP08] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. Visual Abstract Notation for GUI Modelling and Testing – VAN4GUIM. In *ICSOFT (SE/MUSE/GSDCA)*, pages 104–111. INSTICC Press, 2008. Cited on page 36.
- [MP13] Tiago Monteiro and Ana C. R. Paiva. Pattern Based GUI Testing Modeling Environment. In *TestBeds*, 2013. Cited on page 51.
- [MP14a] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. A GUI Modeling DSL for Pattern-Based GUI Testing - PARADIGM. In Leszek A. Maciaszek and Joaquim Filipe, editors, *ENASE*. SciTePress, 2014. Cited on pages 50 and 94.

- [MP14b] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. Towards a Pattern Language for Model-Based GUI Testing. In *Proceedings of the 19th European conference on Pattern Languages of Programs (EuroPLOP)*, 2014. Cited on page 64.
- [MP14c] Rodrigo M.L.M. Moreira and Ana C.R. Paiva. PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-Based GUI Testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 863–866. ACM, 2014. Cited on pages 51 and 103.
- [MP15] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. A Novel Approach using Alloy in Domain-Specific Language Engineering. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*, ESEO, Angers, Loire Valley, France, 2015. Cited on page 92.
- [MPM13] Rodrigo M. L. M. Moreira, Ana C. R. Paiva, and Atif Memon. A Pattern-Based Approach for GUI Modeling and Testing. In *Proceedings of the 24th International Symposium on Software Reliability Engineering*, ISSRE'13, Pasadena, CA, USA, 2013. IEEE Computer Society. Cited on pages 29 and 59.
- [MSP01] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing. In *In Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267. ACM Press, 2001. Cited on pages 30 and 35.
- [MY10] Yuan Miao and Xuebing Yang. An FSM based GUI test automation model. In *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*, pages 120–126, Dec 2010. Cited on page 35.
- [Nei14] Theresa Neil. *Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps*. "O'Reilly Media, Inc.", 2014. Cited on pages 1, 23, and 64.
- [Nin14a] Ninefold. Australian Dedicated Cloud Hosting & Virtual Servers – Ninefold. Available from <https://ninefold.com/>, 2014. Cited on pages xi and 24.
- [Nin14b] Nintendo. Wii U from Nintendo - Official Site - HD Video Game Console. Available from <http://www.nintendo.com/wiiu>, 2014. Cited on page 15.
- [NP14] Miguel Nabuco and Ana C. R. Paiva. Model-Based Test Case Generation for Web Applications. In *Computational Science and Its Applications–ICCSA 2014*, pages 248–262. Springer, 2014. Cited on pages 51, 59, and 60.
- [NPF14] Miguel Nabuco, Ana C. R. Paiva, and João Pascoal Faria. Inferring User Interface Patterns from Execution Traces of Web Applications. In *Software Quality workshop of the 14th International Conference on Computational Science and Applications (ICCSA)*, 2014. Cited on page 51.
- [NR68] Peter Naur and Brian Randell. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 oct. 1968, Brussels, Scientific Affairs Division, NATO. 1968. Cited on page 55.
- [NRBM14] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif M. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Autom. Softw. Eng.*, 21(1):65–105, 2014. Cited on pages 35 and 40.

- [NT11] Sagar Naik and Piyu Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, 2011. Cited on page 17.
- [Nym00] Noel Nyman. Using Monkey Test Tools. *Software Testing and Quality Engineering Magazine*, 2000-01, 2000. Cited on page 34.
- [Nym02] Noel Nyman. In Defense of Monkey Testing, 2002. Cited on page 34.
- [Ozi14] Stan Ozier. TaskFreak! web based task manager and todo list, project management made easy. Available from <http://www.taskfreak.com/original>, 2014. Cited on pages 70, 78, 80, and 122.
- [Pai07] A. C. R. Paiva. *Automated Specification-Based Testing of Graphical User Interfaces*. PhD thesis, 2007. Cited on pages 17, 31, 34, and 35.
- [Pat14] Patternry. Patternry Open – A Free Front-End Resource | Patternry. Available from <http://patternry.com/patterns/>, 2014. Cited on pages 1, 23, 64, 67, 72, 75, and 78.
- [PBG14] PBGT. PBGT - Pattern Based GUI Testing Wiki – Case Studies. Available from <http://paginas.fe.up.pt/~apaiva/pbgtwiki/doku.php?id=publications>, 2014. Cited on pages 113, 119, and 125.
- [PFTV05] Ana C. R. Paiva, J. C. P. Faria, Nikolai Tillmann, and Raul F. A. M. Vidal. A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *LNCS*, pages 450–464. Springer, 2005. Cited on pages 35 and 36.
- [PFV03] Ana Paiva, Joao C. P. Faria, and Raul F. A. M. Vidal. Specification-Based Testing of User Interfaces. In *Interactive Systems. Design, Specification, and Verification, 10th International Workshop*, volume 2844 of *LNCS*, pages 139–153. Springer, 2003. Cited on page 40.
- [PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction, INTERACT '97*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd. Cited on page 96.
- [Pot96] Mike Potel. MVP: Model-View-Presenter the Taligent Programming Model for C++ and Java. *Taligent Inc*, 1996. Cited on page 20.
- [Ran14] GmbH Ranorex. Automated Testing Software | Ranorex – Test Automation. Available from <http://www.ranorex.com/>, 2014. Cited on page 33.
- [Ras00] Jef Raskin. *The humane interface: new directions for designing interactive systems*. Addison-Wesley Professional, 2000. Cited on page 11.
- [Ras14] Robin Raszka. Pptrns – Mobile User Interface Patterns. Available from <http://pptrns.com/>, 2014. Cited on pages 23, 70, 72, and 75.
- [REG07] H. Reza, S. Endapally, and E. Grant. A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. In *Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 366–370, April 2007. Cited on pages 36, 37, and 40.

- [ROM08] Hassan Reza, Kirk Ogaard, and Amarnath Malge. A Model Based Testing Technique to Test Web Applications Using Statecharts. In *Proceedings of the Fifth International Conference on Information Technology: New Generations*, ITNG '08, pages 183–188, Washington, DC, USA, 2008. IEEE Computer Society. Cited on pages 36, 37, and 40.
- [RWC11] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 1st edition, 2011. Cited on page 88.
- [SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2009. Cited on page 88.
- [Sch05] Joost Schalken. Research Methods for the Empirical Assessment of Software Processes. In *Proceedings of the 12th Doctoral Consortium at Conference on Advanced Information Systems Engineering (DC-CAiSE 2005), Porto, PT, Vol.59, no.9*, pages 148–157, 2005. Cited on page 3.
- [Sel14] Selenium. Selenium – web browser automation. Available from <http://www.seleniumhq.org/>, 2014. Cited on pages 33 and 54.
- [SKK⁺14] Christian Schmitt, Sebastian Kuckuk, Harald Köstler, Frank Hannig, and Jürgen Teich. An evaluation of domain-specific language technologies for code generation. In *Proc. Int. Conf. on Computational Science and its Applications (ICCSA)(Jun–Jul 2014), to appear*, 2014. Cited on page 89.
- [SL14] Department of Computer Science Sikuli Lab. Sikuli Script - Home. Available from <http://www.sikuli.org/>, 2014. Cited on page 54.
- [Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010. Cited on pages 34, 40, and 57.
- [Son11] Sony. PlayStation®Move Motion Controller – PlayStation®3 Move Info, Games & Updates. Available from <http://us.playstation.com/ps3/playstation-move>, 2011. Cited on page 15.
- [SP14] Clara Sacramento and Ana C. R. Paiva. Web Application Model Generation through Reverse Engineering and UI Pattern Inferring. In *The 9th International Conference on the Quality of Information and Communications Technology, QUATIC*. IEEE Computer Society, 2014. Cited on page 51.
- [SPS91] Andrew Sears, Catherine Plaisant, and Ben Shneiderman. A new era for touchscreen applications: High precision, dragging icons, and refined feedback. *Advances in Human-Computer Interaction*, 3, 1991. Cited on page 15.
- [SS97] Richard K Shehady and Daniel P Siewiorek. A method to automate user interface testing using variable finite state machines. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 80–88. IEEE, 1997. Cited on page 35.
- [Sta13] StarUML. StarUML – The Open Source UML/MDA Platform. Available from <http://staruml.sourceforge.net/en/>, 2013. Cited on page 88.

- [Sta14] Stattrek. T distribution calculator. Available from <http://stattrek.com/online-calculator/t-distribution.aspx>, 2014. Cited on pages [xii](#) and [68](#).
- [SZ09] Mark Strembeck and Uwe Zdun. An approach for the systematic development of domain-specific languages. *Softw. Pract. Exper.*, 39(15):1253–1292, October 2009. Cited on pages [xii](#), [86](#), and [90](#).
- [TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008. Cited on page [34](#).
- [Tel14] Telerik. Telerik Mobile App Development Platform, .NET UI Controls, Web, Mobile, Desktop Development Tools. Available from www.telerik.com, 2014. Cited on pages [xii](#) and [78](#).
- [Tid11] Jenifer Tidwell. *Designing Interfaces*. O’Reilly, Sebastopol, CA, 2011. Cited on pages [1](#), [23](#), [64](#), [67](#), [72](#), [73](#), [75](#), [78](#), and [82](#).
- [TKH11] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based GUI testing of an Android application. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 377–386. IEEE, 2011. Cited on page [35](#).
- [Tox14] Anders Toxboe. Design patterns. Available from <http://ui-patterns.com/patterns/>, 2014. Cited on pages [1](#), [23](#), [49](#), [64](#), [67](#), [70](#), [72](#), [73](#), [75](#), [78](#), and [82](#).
- [UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. Cited on page [35](#).
- [VBD⁺13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. Cited on pages [86](#), [87](#), [89](#), and [90](#).
- [VCG⁺08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, Lecture Notes in Computer Science, pages 39–76. Springer, 2008. Cited on page [36](#).
- [Vic12] Ryan Vice. *MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF*. Packt Publishing Ltd, 2012. Cited on page [21](#).
- [VP14] Liliana Vilela and Ana C. R. Paiva. PARADIGM-COV - A Multidimensional Test Coverage Analysis Tool. In *In 9ª Conferencia Ibérica de Sistemas y Tecnologías de Información (CISTI)*, 2014. Cited on page [51](#).
- [vW08] Martijn van Welie. Interaction Design Pattern Library. Available from <http://www.welie.com/patterns>, 2008. Cited on pages [1](#), [23](#), [64](#), [73](#), [75](#), and [78](#).
- [Wac14] Clemens Wacha. home – PHP iAddressBook. Available from <http://iaddressbook.org/wiki/>, 2014. Cited on pages [67](#), [73](#), [75](#), [80](#), [82](#), and [122](#).

- [Weg83] Peter Wegner. Varieties of reusability. In *Workshop on Reusability in Programming*, pages 30–44, 1983. Cited on page 55.
- [Whi96] Lee J White. Regression testing of GUI event interactions. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 350–358. IEEE, 1996. Cited on page 30.
- [WW11] Daniel Wigdor and Dennis Wixon. *Brave NUI world: designing natural user interfaces for touch and gesture*. Elsevier, 2011. Cited on page 14.
- [Xie06] Qing Xie. Developing cost-effective model-based techniques for GUI testing. In *Proceedings of the 28th international conference on Software engineering*, pages 997–1000. ACM, 2006. Cited on page 35.
- [XM06] Qing Xie and Atif M Memon. Model-based testing of community-driven open-source GUI applications. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 145–154. IEEE, 2006. Cited on page 35.
- [Yah12] Yahoo! Yahoo! Design Pattern Library. Available from <http://developer.yahoo.com/ypatterns>, 2012. Cited on pages 1, 23, 64, 72, and 73.
- [Yah14] Yahoo! Yahoo! Mail – Sign in to Yahoo. Available from <http://mail.yahoo.com>, 2014. Cited on page 49.
- [Yin09] Robert K Yin. *Case study research: Design and methods*, volume 5. Sage, 2009. Cited on page 109.
- [ZW98] Marvin V Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998. Cited on page 3.