Towards a Pattern Language for Model-Based GUI Testing

Rodrigo M. L. M. Moreira
INESC TEC & Dept. of Informatics Engineering
Faculty of Engineering of the University of Porto
Porto, Portugal
pro08007@fe.up.pt

Ana C. R. Paiva
INESC TEC & Dept. of Informatics Engineering
Faculty of Engineering of the University of Porto
Porto, Portugal
apaiva@fe.up.pt

ABSTRACT

Graphical user interfaces (GUIs) have become popular as they appear in everydays' software. GUIs have become an ideal way of interacting with computer programs, making the software friendlier to its users. GUIs have grown, and so has the usage of UI Patterns featured in GUIs. UI Patterns are recurring solutions to solve common GUI design problems. We developed the notion of UI Test Patterns that, are able to test different implementations of UI Patterns. Therefore, we created a new methodology called Pattern-Based GUI Testing (PBGT) that aims at systematizing and automating the GUI testing process. PBGT samples the input space using UI Test Patterns, which provide a reusable and configurable test strategy, in order to test a GUI that was implemented using a set of UI Patterns. In this paper we present three UI Test Patterns: Login, Master/Detail and Sort.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability/Verification*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

Keywords

Model-Based GUI Testing; Pattern-Based GUI Testing; GUI Modeling; GUI Testing

1. INTRODUCTION

Over the past decades, GUIs have become extremely important and have evolved, as they appear in the majority of todays' software systems. They are responsible to facilitate the interaction between users and the underlying software system. Therefore, GUIs play a crucial role, since the users' perception on the GUI will dictate the success and acceptance of the software system.

EuroPLoP '14, July 09 - 13, 2014, Irsee, Germany.

ACM 978-1-4503-3416-7/14/07 ...$15.00.
http://dx.doi.org/10.1145/2721956.2721972.

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 [14]. Nowadays, due to the increasing functionality on GUIs, these numbers are even higher. Hence, as GUIs logic grow in complexity, it becomes more vital to test them.

Software testing represents a demanding task in software development. It allows increasing confidence in the software quality and cannot be seen as a separate activity that is commonly performed at the end of the process. As GUIs tend to grow, GUI testing demands further research endeavors. GUI testing is difficult, time-consuming, and costly, with few tools and techniques to assist the testing process. Manual testing is a lengthy activity, error prone and exasperating. As GUIs implementation grow in complexity, so do the efforts required to perform manual GUI testing. Moreover, current GUI testing methods still require manual activity. In order to reduce the cost of software development and maintenance, a lot of research has been conducted over the past decades.

Automating software testing can substantially reduce the effort required for adequate testing. Furthermore, it helps to greatly reduce the time and the cost of software testing throughout the entire development life cycle. Model-Based Testing (MBT) is a software testing technique upon which test cases are derived from a model that describes some (usually functional) aspects of the system under test (SUT). It allows checking the conformity between the implementation and the model of the SUT, introducing more systematization and automation into the testing process. This technique uses a model program to generate test cases and/or to act as an oracle. An oracle is the authority which provides the correct result to make a judgment about the outcome of a test case – whether the test passed or failed. MBT has been successfully implemented for GUI testing purposes [22, 2, 4, 18, 11, 26, 25, 13].

Despite all of the advances in automated testing tools and frameworks over the last decade, manual testing still represents the majority of testing effort. Furthermore, the experience and skills of the tester, yet play a key role in finding failures in the software.

Until recently, testing GUIs for their functional correctness has been disregarding UI Patterns. UI Patterns describe an ideal solution for a specific functionality in order to solve a common GUI design problem. Certain function-

alities can have different layouts. However, the aspect that is particular relevant for our work is the common behavior featured in such GUIs. This behavior is recurrent and has the propensity of being repeated over time. Using UI Test Patterns it becomes possible to test a GUI that was implemented with a specific UI Pattern or a set of UI Patterns across their several implementations. Our new GUI testing approach, aims to provide a new model-based GUI testing paradigm that promotes reuse of GUI testing strategies.

The patterns described in this paper are a sub-set of patterns from a total of 7 (until now) of the Pattern Language to be used for the PBGT domain. Furthermore, these UI Test Patterns have been used to model and test real web applications in [22, 31, 19] and more recently mobile applications (Android) [3], where we successfully were able to find faults in the software.

1.1 Audience

The patterns presented in this paper aim 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.

Furthermore, these patterns will act as guidance on how to effectively use UI Test Patterns for GUI Testing, which is a new emerging concept, and will allow to achieve a common understanding on their usage. The idea is to also raise awareness on the topic of Model-Based GUI Testing, by demonstrating the capability of using UI Test Patterns to model and test GUIs.

1.2 Structure

The remainder of this paper is structured as follows. Section 1 provides an introduction on the topics of GUI testing, automated testing and identifies the audience of the paper. Section 2 presents the topic of Pattern-Based GUI Testing and provides an illustrative example about the approach. Section 3 describes the Pattern Language to be used in the context of PBGT. Finally, Section 4 draws conclusions and indicates future work directions.

2. OVERVIEW ON PATTERN-BASED GUI TESTING

GUIs are implemented using UI Patterns [22]. In our terms, and in the context of our work, we describe UI Patterns as recurring solutions of a given functionality, to solve common GUI design problems. Since a UI Pattern can be implemented in different ways, with different layouts and particularities, what is important to notice is their common behavior. Recently, we have developed an approach that is able to test UI Patterns for their functional correctness [22]. We have developed the notion of UI Test Patterns. A UI Test Pattern provides a generic way to test the different implementations of UI Patterns. We realized that GUIs that have similar design, based on a given UI Pattern, should share a common testing strategy. We created a new methodology called Pattern-Based GUI Testing method (PBGT) that aims at systematizing and automating the GUI testing process. UI Test Patterns are one of the key points in PBGT. They provide a reusable and configurable test strategy, in order to test a GUI that was implemented using a set of UI Patterns. By promoting reusability with PBGT, we are able to reduce the modeling and testing efforts allowing to save costs and time in GUI testing.

With PBGT, the tester is able to create models of GUIs to be tested, according to a set of testing goals. Models are crafted with PARADIGM [19, 21], which is a graphical Domain-Specific Language (DSL) to be used in the context of PBGT. These models contain UI Test Patterns that are configured to fulfill the testing goals. PBGT is supported by a tool (PBGT Tool) that provides a fully integrated modeling and testing environment (further details about the tool and its usage can be found in [20]). The tool automatically generates test cases (from the models) and executes them. The tester then verifies the output, to check which tests passed or failed. Moreover, the tool is able to test web-based applications and mobile applications (Android). Future releases will target iOS and desktop-based applications.

UI Test Patterns are divided into two categories: (1) Base UI Test Patterns that represent the initial set of the patterns and; (2) High-Order UI Test Patterns that aim to promote, even further, reuse when constructing GUI models for testing. The latter category allows to define new UI Test Patterns, by combining Base UI Test Patterns. These categories are associated with two specific user-roles. The developer implements the Base UI Test Patterns, and is able to extend the language by adding test patterns. The tester is responsible for using existing UI Test Patterns, and can also to combine existing ones.

2.1 Example on using UI Test Patterns

To better understand the usage of UI Test Patterns and their benefits, we now provide a practical example featuring real web applications. One recurring problem in web applications is authentication, ensuring or restricting access to certain functions or data. In order to access these protected functionalities, 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 [34]. This UI Pattern can be implemented in different ways, in terms of layout and outcomes (Figure 1). 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 1 displays the different implementations of the authentication mechanism, of four web applications: Gmail [9], Yahoo! Mail [38], Outlook [16] and Facebook [7]. 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. Further information about the PBGT approach can be found at the *wiki* of the project [29].

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. In the next section we will present a more detailed view on the UI Test Patterns defined for PBGT.

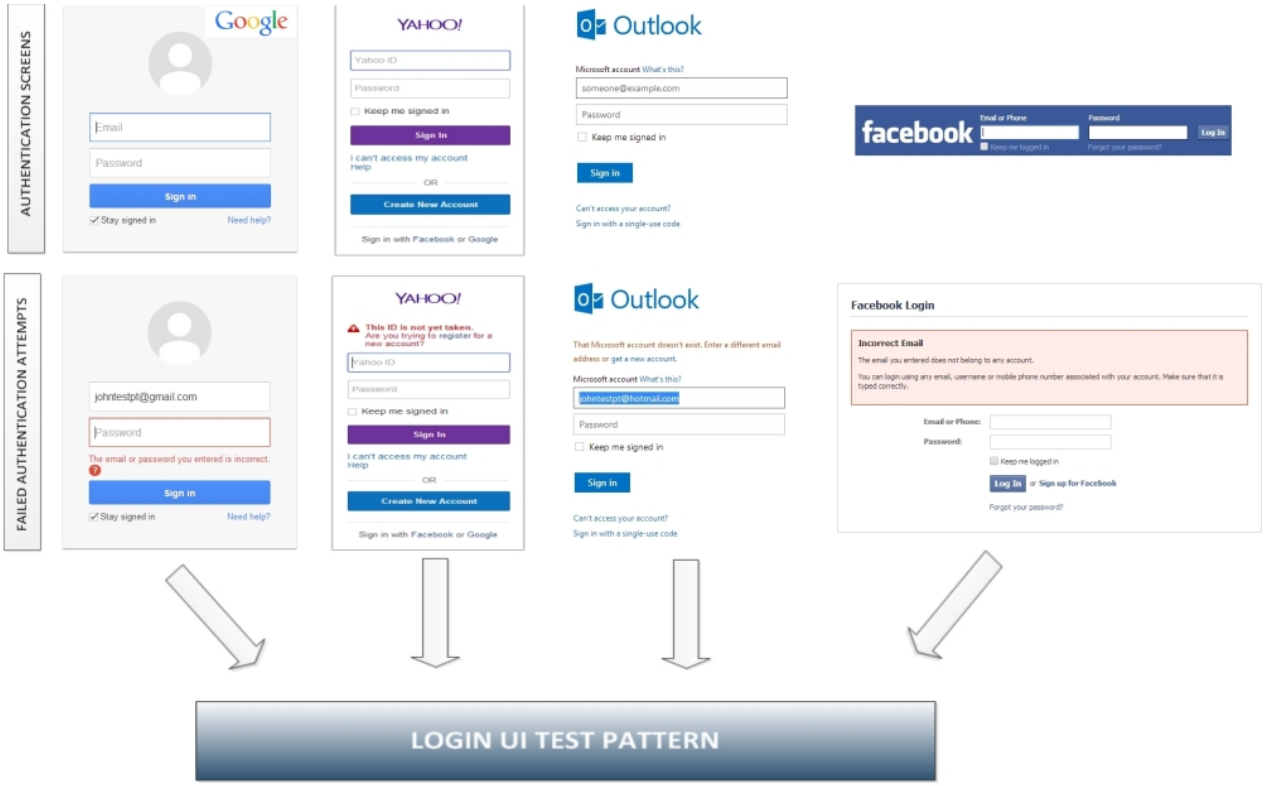

Figure 1: Different implementations and outcomes of the Login UI Pattern.

3. THE PATTERN LANGUAGE

GUIs regularly feature UI Patterns [22]. UI Patterns are graphical and interaction standards of a given functionality. UI Patterns can be realized using several different implementations, yet they share common behavior. UI Test Patterns provide a generic test strategy to test these different implementations.

In order to better understand some of the contents that will be presented in this section, prior reading on GUI testing and PBGT methodology is advised, which can be found in [22, 20].

3.1 UI Test Pattern Formal Definition

A UI Test Pattern describes a generic test strategy, formally defined by a set of test goals, for later configuration, denoted as $< Goal, V, A, C, P >$ 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.

After researching generic testing strategies for the most common UI Patterns, from various sources [33, 39, 28, 34, 36, 35], we have designed a set of UI Test Patterns.

3.2 Pattern Format

Patterns can be described in several styles [1, 8, 12, 15]. 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, we will present the patterns, having in mind simplicity and comprehension, according to the following structure, with guidance from [15] and an adaptation (*,[1]):

[1]According to feedback gathered from experts on the field of patterns at EuroPLoP 2014.

- **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.

3.3 Further Considerations

The pattern language described in this paper aims to be platform independent. It is very flexible, since it can be customized/configured to model and test any UI. Moreover, this pattern language has been used to model and test several fielded applications in [22, 3, 19, 23, 31]. We were able to find faults in the software, proving the benefit and applicability of the pattern language.

The patterns that are going to be described next can be connected in order to achieve a specific flow to fulfill the test goals of the SUT. The PBGT approach comprises a DSL called PARADIGM. It defines language elements and connectors. These connectors are based on ConcurTaskTrees (CTT) [27], which are a popular graphical notation for task representation. 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.

As a remark, the patterns that will be detailed below, are instances of the formal definition (defined in SubSection 3.1). Therefore, 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 to 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 it 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 [6].

3.4 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 [34] (referred as *Login* in [30]).

Known Uses

- TaskFreak [24].
- Mobile.de [17].
- Tudu Lists [5].

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 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.

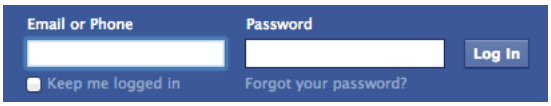

Figure 2: UI part featuring the authentication functionality from Facebook [7].

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.

3.5 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. Set of checks **C**: {"detail has value(s) X", "detail does not have value(s) X", "detail is empty", "detail has N elements"};
4. Sequence of actions **A**: [select *master*].

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 [33].
- Breadcrumbs [30, 34, 33, 28].
- Tab Bars [30] (referred as *Tabs* in [39]; referred as *Navigation Tabs* in [34]).
- Sequence Map [33].
- Accordion [33, 39, 34].
- Collapsible Panels [36].
- Details on Demand [36].

Known Uses

- Australian-charts.com [10].
- Mobile.de [17].
- iAddressBook [37].

Example

A tester aims to verify the functional correctness of the GUI, that features a Master/Detail area (Figure 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.

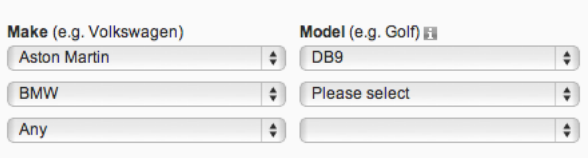

Figure 3: UI part of Mobile.de [17] that features the Master/Detail area.

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.

3.6 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).

Testers target 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),...,(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),... , 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"}.

During configuration, the user has to define the value for N, provide input data for the pair variables and criteria (v_1, c_1),...,(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 [34] (referred as Sortable Columns in [35]).
- Table Sorter [28, 36] (referred as Sortable Table in [33]).

Known Uses

- Mobile.de [17].
- TaskFreak [24].

Example

A tester aims to verify the functional correctness of the GUI, that features a sort functionality (Figure 4). The tester will use the Sort UI Test Pattern to verify the correctness of the GUI. Thus, the testing goals are: "Ascending" and "Descending". 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.

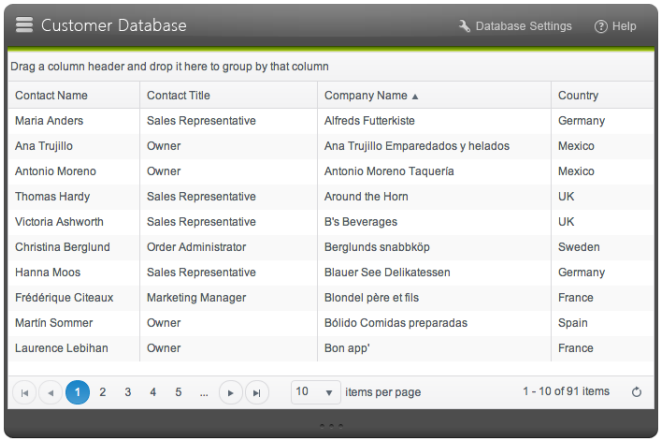

Figure 4: UI part from Telerik [32] that features a sort functionality.

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.

4. CONCLUSIONS

We have introduced a set of three UI Test Patterns (as a part of the Pattern Language) to be used in the context of PBGT. PBGT is a new model-based GUI testing paradigm that aims at promoting reuse of GUI testing strategies. We have used the UI Test Patterns to model and test real web applications in [22], where we successfully were able to find faults in the software. Currently, UI Patterns have to be identified by the tester, representing a manual task. In the future we aim to automate this process, so we could automatically identify a set of elements that represent UI Patterns and then be able to map them to the associated UI Test Pattern. Moreover, we intend to increase the number of UI Test Patterns, in order to cope with latest design capabilities in GUIs.

5. ACKNOWLEDGMENTS

The authors express their deepest appreciation to their shepherd – Ernst Oberortner – for the guidance, support, and availability for preparing this paper.

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.

6. REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oxford, 1977.

[2] S. Arlt, C. Bertolini, S. Pahl, and M. Schäf. Trends in Model-based GUI Testing. *Advances in Computers*, 86:183–222, 2012.

[3] P. Costa, M. Nabuco, and A. C. R. Paiva. Model-based testing for Mobile Applications. In *The 9th International Conference on the Quality of Information and Communications Technology, QUATIC*. IEEE Computer Society, 2014.

[4] M. Cunha, A. C. R. Paiva, H. Sereno Ferreira, and R. Abreu. PETTool: A Pattern-Based GUI Testing Tool. In *2nd International Conference on Software Technology and Engineering (ICSTE'10)*, SFM'12, pages 202–206, 2010.

[5] J. Dubois. Tudu lists. http://www.julien-dubois.com/tudu-lists.html. Accessed June, 2014.

[6] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2001.

[7] Facebook. Welcome to Facebook – Log In, Sign Up or Learn More. http://www.facebook.com. Accessed January, 2014.

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[9] Google. Gmail. www.gmail.com. Accessed February, 2014.

[10] S. Hung. Australian charts portal. http://australian-charts.com. Accessed February, 2012.

[11] A. Kervinen, M. Maunumaa, T. Päkkönen, and M. 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.

[12] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[13] A. 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).

[14] A. M. Memon. GUI testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.

[15] G. Meszaros and J. Doble. A Pattern Language for Pattern Writing. In *The 3rd Pattern Languages of Programming conference*, pages 1–33, 1996.

[16] Microsoft. Outlook – Sign In. https://login.live.com/. Accessed January, 2014.

[17] mobile.de. mobile.de – Germany's Biggest Vehicle Marketplace Online. Search, Buy and Sell Used and

New Vehicles. http://www.mobile.de/?lang=en. Accessed November, 2013.

[18] R. M. L. M. Moreira and A. C. R. Paiva. Visual Abstract Notation for GUI Modelling and Testing – VAN4GUIM. In *ICSOFT (SE/MUSE/GSDCA)*, pages 104–111. INSTICC Press, 2008.

[19] R. M. L. M. Moreira and A. C. R. Paiva. A GUI Modeling DSL for Pattern-Based GUI Testing - PARADIGM. In L. A. Maciaszek and J. Filipe, editors, *ENASE*. SciTePress, 2014.

[20] R. M. L. M. Moreira and A. C. R. Paiva. PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-Based GUI Testing. In *ASE '14: Proceedings of the 29th IEEE international conference on Automated Software Engineering*. ACM, 2014.

[21] 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.

[22] R. M. L. M. Moreira, A. C. R. Paiva, and A. 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.

[23] M. Nabuco, A. C. R. Paiva, and J. P. 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.

[24] S. Ozier. TaskFreak! web based task manager and todo list, project management made easy. http://www.taskfreak.com/original. Accessed June, 2014.

[25] A. Paiva, J. C. P. Faria, and R. 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.

[26] A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal. A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *LNCS*, pages 450–464. Springer, 2005.

[27] F. Paternò, C. Mancini, and S. 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.

[28] Patternry. Patternry Open – A Free Front-End Resource | Patternry. http://patternry.com/patterns/. Accessed January, 2014.

[29] PBGT. Pattern-Based GUI Testing Wiki. http://paginas.fe.up.pt/~apaiva/pbgtwiki/doku.php?id=publications, 2014. Accessed August, 2014.

[30] R. Raszka. Pttrns – Mobile User Interface Patterns. http://pttrns.com/. Accessed January, 2014.

[31] C. Sacramento and A. 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.

[32] Telerik. Telerik Mobile App Development Platform, .NET UI Controls, Web, Mobile, Desktop Development Tools. www.telerik.com. Accessed January, 2014.

[33] J. Tidwell. *Designing Interfaces*. O'Reilly, Sebastopol, CA, 2011.

[34] A. Toxboe. Design patterns. http://ui-patterns.com/patterns/. Accessed January, 2014.

[35] I. D. T. UASP. Pattern Browser. http://patternbrowser.org/code/pattern/pattern.php. Accessed January, 2014.

[36] M. van Welie. Interaction Design Pattern Library. http://www.welie.com/patterns, 2008. Accessed January, 2014.

[37] C. Wacha. home – PHP iAddressBook. http://iaddressbook.org/wiki/. Accessed June, 2014.

[38] Yahoo! Yahoo! Mail – Sign in to Yahoo. http://mail.yahoo.com. Accessed January, 2014.

[39] Yahoo! Yahoo! Design Pattern Library. http://developer.yahoo.com/ypatterns, 2012. Accessed December, 2013.