Lessons learnt from using DSLs for Automated Software Testing

Mark Micallef, Christian Colombo
{mark.micallef | christian.colombo}@um.edu.mt
PEST Research Lab
Faculty of Information & Communication Technology
University of Malta

***Abstract*—Domain Specific Languages (DSLs) provide a means of unambiguously expressing concepts in a particular domain. Although they may not refer to it as such, companies build and maintain DSLs for software testing on a day-to-day basis, especially when they define test suites using the Gherkin language. However, although the practice of specifying and automating test cases using the Gherkin language and related technologies such as Cucumber has become mainstream, the curation of such languages presents a number of challenges. In this paper we discuss lessons learnt from five case studies on industry systems, two involving the use of Gherkin-type syntax and another three case studies using more rigidly defined language grammars. Initial observations indicate that the likelihood of success of such efforts is increased if one manages to use an approach which separates the concerns of domain experts who curate the language, users who write scripts with the language, and engineers who wire the language into test automation technologies thus producing executable test code. We also provide some insights into desirable qualities of testing DSLs in different contexts.**

I. Introduction

In her widely cited paper about the future of software testing, Bertolino [1] claims that domain specific languages (DSLs) have emerged as an efficient solution towards allowing experts within a domain to express specifications in that domain. She goes on to claim that success of domain-specific approaches should be built upon and extended to the testing stage of software engineering. An intuitive place to start would be to explore DSLs in the context of automated software testing such that languages constructed by domain experts can be leveraged to specify not only requirements but also test cases which validate those requirements. This ties in nicely with the industry's ever growing appetite for automated testing. To a large extent, the use of DSLs for specifying automated tests is already widespread in the software development industry through the use of Gherkin [2] — a language designed to express test scenarios and is often referred to as the *Given-When-Then* notation:

Given `I am a user`
When `I log in using valid credentials`
Then `I should see the welcome page`

The language provides constructs for defining a number of scenarios which in turn consist of a number of *steps*, each being either a precondition to the scenario (*Given*) followed by a stimulus to the system (*When*) and finally the postcondition (*Then*). In the software development industry, hundreds of such scenarios are developed in a typical project and are used as a vehicle for communicating specifications amongst team members and eventually end up being turned into automated testing code via technologies such as Cucumber [2]. The language is very flexible in that it only requires steps to start with one of the given-when-then keywords whilst the rest of the step is specified as free text by the user of the language. Developers then write code which translates each step into interaction with the system, thus creating an automated test suite. In essence, the collection of phrases after a given-when-then keyword form a domain specific language as defined by whoever is maintaining scenarios written in Gherkin.

In this paper we present observations from five case studies involving the use of domain specific languages. The first two case studies consisted of working with two industry partners[1] and introducing Gherkin-driven test automation frameworks. Whilst the industry partners were interested in setting up automated testing frameworks, we were mainly interested in observing the challenges encountered when using Gherkin to develop a domain specific language. We subsequently went on to investigate the possibility of using more rigorously defined languages for specifying automated tests through three case studies: (1) Android applications, (2) eCommerce websites and (3) graphics-based games.

II. Background

This section provides a brief background of domain specific languages and automated testing — the two main subject areas of this paper.

A. Domain Specific Languages

Fowler and Parsons define *Domain Specific Languages (DSLs)* as "computer programming languages of limited expressiveness focused on a particular domain" [3]. They are championed as a mechanism to improve expressiveness within a domain and have been applied to a wide variety of applications in software engineering.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

[1] One partner was an online gaming company whilst the other was an international publishing house.

The literature contains numerous guidelines to guide DSL design [3][4][5][6][7] but if one consolidates the sources, a number of common characteristics of a good DSL emerge. Firstly, all sources strongly argue that a DSL should exhibit *simplicity* in that it is easily understood by humans yet parseable by machines. Related to this, a DSL should, where possible, exhibit *similarity* to another language with which users are already familiar (e.g. English). This contributes significantly to the initial acceptance and long-term adoption of the language. Thirdly, a DSL should be highly *domain specific and parsimonious* so that any notions not related to the domain in question are omitted; yet at the same time language designers should strive to make a language as *complete* as possible with respect to that domain. Finally, a domain specific language should ideally be *extensible* thus making it easy to add features and operations to the language; and *reusable* in that the same grammar can be used as the basis of a new language when required.

B. Automated Testing

Test automation technologies can be split into two broad categories: record-playback and scripted automation [8]. The former involves using recording software which observes a system being used and then simply replaying a series of actions back when required; whilst the latter involves someone writing code which when executed interacts with the system under test as part of an automated test suite. In this work, we are mainly interested in scripted testing.

There are a number of tools/APIs available which enable programmatic interaction with a system under test to facilitate test scripting. The Selenium/Webdriver [9] project is an open source endeavour which has become the de facto industry standard for web test automation. The project provides API bindings for a variety of programming languages and enables developers to launch and control web browsers for the purposes of automated testing. The technology interacts with a website under test by querying and interacting with the browser and the document-object model (DOM) of the website. Two related technologies which build on the Selenium/Webdriver project are Selendroid [10] and Appium [11]; two technologies which allow automated interaction and querying of mobile devices using a DOM-based querying mechanism. Finally, for systems which do not expose their internal representation of the user interface, Sikuli [12] is an automated testing API which utilises image recognition techniques. That is to say, instead of (for example) searching through a website's DOM looking for a button with the text "OK", Sikuli takes a screenshot of the app and looks for an image of such a button. The technology is slower and more brittle than the alternatives but in certain situations, it is the only option available. We discovered this when attempting to automate the testing of graphics-based games.

Whilst automating the execution of tests is becoming more and more popular [13], with some considering it a necessity [14], test automation efforts tend to be expensive and susceptible to losing steam as a project grows and evolves in ways that makes maintaining old tests difficult. In this paper we argue that hiding the low level intricacies of test automation technologies within a domain specific language goes a long way towards improving the usability of the technology, mainly through separating the role of the domain expert who curates the DSL and the engineer who wires phrases of the language into test automation tools. This is discussed further in the following section.

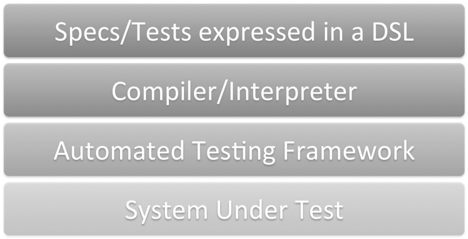

Fig. 1. The layered approach proposed as a guideline for developing a DSL for software testing

III. Gherkin Case Studies

Our initial steps on this journey consisted of two case studies with industry partners, one in the iGaming industry and the other was an international publishing house. We embedded a researcher with each of the industry partners for a period of three months with the researcher forming part of a product development team in a test engineer role. The arrangement was mutually beneficial in that the partners would benefit from the services of an embedded researcher with test automation experience who would champion and drive the creation of an automated testing framework, whilst the research group was able to observe challenges which arose throughout the process.

Following initial discussions, two main problems were identified. Firstly, whilst test automation was the primary outcome sought by our partners, they acknowledged that their testers *did not necessarily have the required skills to implement automated tests*. Whilst an effort would be made to recruit experienced technical testers, the balance of technical to non-technical testers was unlikely to ever be close to even. The second problem, stated bluntly, was that *delivery deadlines were a reality* and sometimes test DSL development and test automation would take a back seat whilst a product was pushed out of the door on time.

Based on these challenges, we proposed a layered approach (see Figure 1) to enable domain experts to curate a language of notions describing their particular domain. The language would subsequently be used to express specifications and tests for the domain with the hope of making the testing process quicker. These tests can be processed by a language compiler or interpreter and translated into executable code which forms part of an automated testing framewornabling testers lacking automation skill to easily automate testing. The framework was envisaged to be a collection of tools which automate (1) interaction with, and (2) querying of the state of, the system under test. Such a layered approach also enables a multi-role approach to test automation, in that different team members can participate in the process depending on the level. Therefore, a product owner or business analyst would be able to define test scenarios at the top level, non-technical test analysts can read these scenarios and carry out manual testing whilst software developers or test engineers can pick through a backlog of such scenarios and wire them in to the automated testing framework over time. All this is carried out with the language (and test scenarios written in the language) acting as the glue throughout. The multi-role features of the layered approach would address the challenge of not having a full-time supply of technical testers, since the few available could be shared across product teams and take on the role of picking through language definitions, gaining an understanding of expressed notions by talking to other members of the team and subsequently simply wiring language constructs into the test automation framework. In the meantime, non-technical testers could take on the role of manually executing unimplemented tests thus addressing the problem of test automation work causing delays and missed deadlines.

A. Lessons Learnt

The two case studies provided us with a number of interesting observations. The layered approach worked in that it *delivered an automated test suite in a few weeks* and its development did not seem to affect team productivity. The process of creating a DSL as part of the test suite also seemed to *improve communication* amongst team members as this common language developed and started being used. Although these positive outcomes dominated the first few weeks of the case studies, the cracks started to show as the number of test scenarios began to grow. Due to the fact that Cucumber provides a very loose grammar, the language grew organically with people adding their own versions of notions as they went along. This resulted in substantial duplication as people would express the same notions in different ways. For example, `Given I log in to the system`, `Given I log on to the system`, and `Given I log in correctly`, all express the same notion and resulting user actions. Similarly some team members condensed the sequences of actions into one whilst others used the longer atomic format. For example, `Given I log in and purchase a product` could also be expressed as:

```
Given I log in
And I search for a product
And I select the first item in the list
And I add the item to my shopping cart
```

Depending on who was writing a particular test scenario, the same notion could be expressed at different levels of abstraction. Besides making things harder on technical testers who were trying to maintain a clean coherent codebase, these issues created a sense that the language was getting out of hand and was less likely to be trusted as a common vehicle for communicating domain notions within the team. The problem was further compounded when specifications began to change as products evolved. In the case of one industry partner, the perceived cost of maintaining the language and automated tests became so high that the project was almost abandoned as technical testers were assigned manual testing jobs in order to meet deadlines. All this led to the following important lessons:

1) Whilst the layered approach works, *long term feasibility requires a dedicated language owner*. Her job would be that of ensuring consistency within the language, avoiding duplication, and so on.
2) Success also depends on having *a development process which caters and makes space for DSL development*. An ad-hoc approach is unlikely to work, especially when deadlines approach.
3) As the language grows, *tool support may be needed* to help users of the language look up available notions much in the same way that they would use a dictionary in a natural language.
4) Finally *management buy-in is essential* in that if the process of developing and maintaining a DSL is perceived as being in competition with software delivery then at some point, management will almost certainly abandon any such efforts.

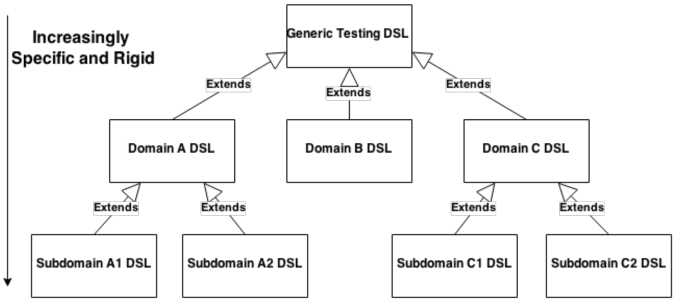

Fig. 2. Testing DSLs can be visualised as a hierarchy of increasing specificity and rigidity

B. Viewing Testing DSLs as a Hierarchy

Following our initial case studies, we propose that testing DSLs can be seen to exist in a form of hierarchy (see Figure 2) whereby the root DSL is very generic and only captures core concepts about testing. That is to say, it captures concepts such as test suites, tests, assertions, setup/teardown and so on. In fact, Gherkin can be said to exist at this level given it basically defines the notions of features (test suites), scenarios (tests), and the Given-When-Then notation (test setup, exercise, test teardown); whilst leaving all other notions to be defined as free text by users of the language. Languages further down the hierarchy however would be able to express more specific notions with regards to a particular domain or subdomain. For example, a DSL for testing graphics-based games would be able to express all the notions of the root DSL but also provide grammatical structures for defining concepts in the graphics-based games domain. However, it would not feasibly express notions about *all* possible graphics-based games. In fact, it would probably express common notions such as starting a game, the score of a game, winning a game, and so on. If one needed to model concepts tailored to a specific game then one would need to move one level further down in the hierarchy and create a DSL specifically for that purpose.

With this in mind, we regrouped and decided to try and further investigate the use of DSLs in testing by exploring case studies at various levels of this abstraction hierarchy. We hypothesised that a more rigid approach to language definition would lead to better results in terms of long term evolution and maintainability of the language. This is discussed in the following section.

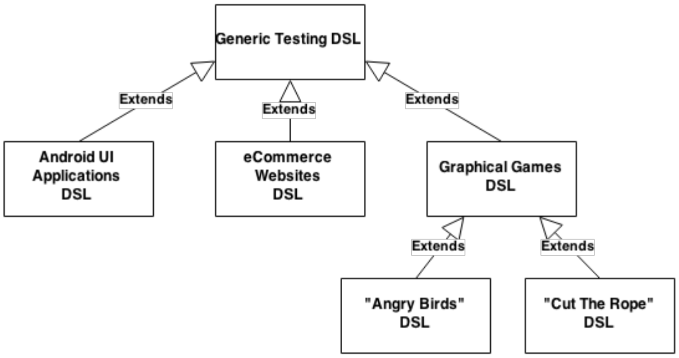

Fig. 3. Selected case studies visualised as a language hierarchy

IV. Structured Language Case Studies

Given the proposed hierarchical view of testing DSLs discussed in Section III-B, the motivation behind the next part of our investigation was that of exploring the design and use of DSLs at various levels of the hierarchy. Whilst DSLs lower down in the hierarchy will more likely need dedicated parsers and code generators than Gherkin-based languages, they should reduce the symptoms observed in our initial case studies whereby notions were expressed in a multitude of ways and the language quickly grew out of control. Also, more structured languages would provide benefits in terms of having scripts being automatically checked during compilation and as part of an integrated development environment would benefit from features such as code helpers[2]. The first challenge in this regard was that of case study selection.

A. Case study selection

Three case studies were selected involving domains with publicly available systems. The envisaged methodology was that of understanding the domains involved and designing a DSL for each one such that we would be able to specify tests and wire them in to test automation tools. The domains were selected based on their envisaged level in the hierarchy discussed in Section III-B and were as follows (see Figure 3)[3]:

1) Android Applications: The Android operating system is now one of the main players in the mobile device market and well over a million applications are available for download online. Whilst developers are essentially free to develop any kind of user interface they desire, it is generally advisable that applications follow Google's established user interface guidelines. These define both the vocabulary of the typical interface components (e.g. buttons, sliders, status bars, etc), user interactions (e.g. tap, long-tap, pinch, etc) and visual guidelines regarding how components are best placed in relation to each other on an interface. These guidelines are updated from time to time but usually in very small increments, thus making this domain a very stable one. Our aim in this case study was to design a language that would enable the creation of tests which expressed interaction with applications that adhere to the Android User Interface Guidelines; yet without expressing concepts related to specific applications.

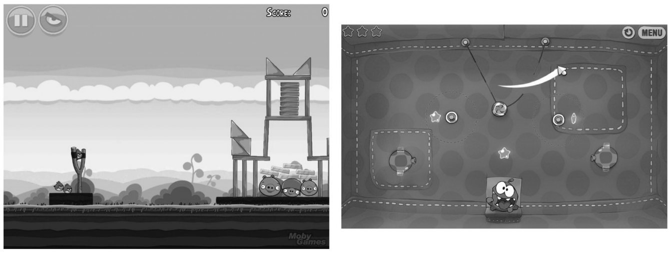

Fig. 4. *Angry Birds* (left) and *Cut the Rope* (right) were selected as candidates for the graphical games case study.

2) eCommerce Websites: Most users of the Internet would be familiar with eCommerce websites, sites which usually allow customers to search through a product database and purchase items online. A substantial amount of software developers are also likely to work on such systems at some point in their career. Yet whilst such websites embody a certain element of common functionality (e.g. searching for products, viewing products, adding items to a card, etc), companies would typically need to build automated test suites from the ground up. In this case study, we wanted to explore the possibility of defining a language which was able to express notions across eCommerce websites and also provide the engineering mechanism that would actually execute the same script regardless of the website. That is to say that a script fragment stating `Add the current item to the shopping cart` could be transparently executed against Amazon.com, BookDepository.com, or any other eCommerce website which supports the notion of adding an item to a shopping cart.

3) Graphical Games: Finally, we wanted to explore a case study which descended further down the language hierarchy. We selected graphics-based games for two main reasons. Firstly, in all likelihood, one graphics-based game will vary enormously from another due to the creative nature of development. This means that we would need to split the development of such a DSL across (at least) two levels in the hierarchy. That is to say that whilst we could design a DSL to express common notions such as starting a game, winning a game, etc, we would not be able to express notions about *all* games. Secondly, the engineering challenge in automating interactions with graphics-based games was intriguing. To this end we selected two popular games as part of our case study, *Angry Birds* and *Cut the Rope* (see Figure 4). The former is a popular game in which the user catapults various birds at structures in an attempt to kill the enemy pigs, whilst the latter is a game in which the user attempts to deliver candy to a hungry monster by cutting ropes in the right timing and sequence so as to successfully manipulate a swinging pendulum puzzle.

B. Language Design Discussion

In this section we compare and contrast language design issues and priorities across the three cases studies. An initial observation we made was that a DSL for software testing essentially expresses notions in two domains: the software testing domain, and the domain of the system being tested. In

[2] In software development, code helpers in an integrated development environment provide developers with suggestions as to possible ways to complete a line of code which they are working on.

[3] More details on the DSLs mentioned in this paper can be found at `www.um.edu.mt/ict/cs/pest/publications/testing_dsls`

the testing domain, we are interested in defining the notions of a test suite, a test, and various parts of the testing life cycle. Therefore, an early design decision across all three case studies was that and DSL would essentially consist of two parts, with one part being common to all testing DSLs and expresses notions about the testing life cycle whilst the second part would vary depending on the individual domain. In fact, all our languages had similar notions along the lines of:

```
define testsuite "login tests"
   define setup "setup"
      ...
   end

   define teardown "teardown"
      ...
   end

   define test "valid login"
      ...
   end
end
```

The interesting observations surfaced when we looked at the domain-specific component of each case study. In Section II-A, we outlined a number of quality attributes related to DSLs which are found in the literature. Whilst some of these attributes (e.g. simplicity and orthogonality) remained important across all our domains, other attributes become more important than others depending on the case study in question. These attributes were *domain specificity*, *extensibility*, and *reusability*. We discuss observations for each attribute below.

1) Domain Specificity: In the case of domain specificity, whilst one might think it an absolute necessity that a domain specific language be highly domain specific, the reality is that some domains are not so easily definable from the outset. The problem of fluctuating requirements in software development is well known and has in fact led to the conception of new development processes that deal with the phenomenon. In the context of our case studies, we found it relatively easy to create a highly domain specific language for our well-defined domain (Android applications), mainly due to the fact that we focused on extracting notions from the Android User Interface Guidelines and embodying them in a well-defined grammar. However, on the other end of the spectrum where we were trying to define a DSL for graphical games, the situation was very different. Whilst we were able to define concepts such as starting a game, winning a game, losing a game, the score of a game, and so on, we could not be any more specific. Whether we define the notion of catapulting a bird, driving a car or aligning coloured blocks would depend on the particular game we are testing.

In our eCommerce case study, we surveyed a number of websites and enumerated all their features. As one would expect, there was quite a bit of overlap with many sites offering similar features. However, there were features which were very site specific. For example, some sites offered product reviews while others did not. Also, amongst sites which offered reviews, the style differed (e.g. one supported star ratings whilst another supported thumbs-up/down ratings, and so on). So in terms of domain specificity, the eCommerce case study was more of a compromise candidate.

2) Extensibility: We observed that the importance of extensibility features in a DSL seemed to be inversely proportional to its domain specificity. That is to say that whilst it was important that our DSL about graphical games be extensible to accommodate new concepts, it was less so for eCommerce systems and almost completely unnecessary for Android applications. This was mainly due to the fact that our Android DSL modelled the Android User Interface Guidelines completely and the only plausible reason one would want to extend the language was when a new version of the Android Operating System was released; an occurrence which would arguably require an associated new release of our DSL in any case.

3) Reusability: All developed languages provided reusability support in the form of allowing users of the language to group sequences of language phrases into procedures which could then be called by name. For example:

```
define procedure "add out of stock book to cart"
   search for "Harry Potter" in "books"
   select first item from search results
   add current item to cart
end

define test "buyOutOfStockBook"
   add out of stock book to cart
   verify that the item is not added to the cart
end
```

We noticed that in the case of Android applications, this came in very handy due to the fact that test cases made up of highly atomic statements such as tapping, pinching, waiting, etc tended to make tests less readable. Therefore a complex sequence of interactions could be grouped together and labelled as `zoomInToLondonOnGoogleMaps` for a example. We found less need for reusability in the case of eCommerce websites because the notions modelled by the language were by their nature on a business logic level. The same was observed in the case of the DSL for graphical games. Having said that, we argue that reusability should still be provided within languages because it is a useful mechanism for improving readability and maintainability of scripts. For example, in the case of our games DSL, we found it very useful to define procedures such as `advanceToLevel5`, which would result in an automated process of playing the game through to level 5 before a test for a feature in level 5 can be carried out.

C. Engineering Challenges

The nature of these case studies presented us with a number of engineering challenges. Recall that the automation of tests specified using one of our DSLs was an important deliverable for industry partners. The fact that we departed from the current industry standard immediately presented us with the challenge of needing to *provide a compiler for our DSLs*. Many compilation frameworks are available but we decided to use the xText language framework [15], based mainly on its ability to automatically build a compiler and code editor (as an Eclipse plugin) from a BNF-like grammar. Given some very basic language specification training, we think that most domain experts would be able to curate their DSL in BNF notation.

The second engineering challenge involved the interaction or wiring in of our scripts such that they are able to interact with the systems under test. In the case of Android applications and eCommerce websites, this was relatively easy to do through the use of standard technologies such as Selendroid [10] and Webdriver [9]. In this case, language compilation resulted in the generation of JUnit code which internally utilised these APIs. The challenge was larger in the case of graphical games due to the fact that games do not utilise standard user interface components. Whilst in a case study with an industry partner we would have asked engineers to provide us with test hooks to interact with the game, in our case we did not have that luxury. We therefore decided to automate game interaction using Sikuli [12], a test automation API which utilises image recognition techniques. Hence, language curators would not only have to provide a grammar for the game-specific component of the language, but also a series of images which would be linked to notions (e.g. an image of a red bird in angry birds). Similarly, we utilised OCR techniques to read the score on the screen when it was required by a test. The approach worked but risks being brittle, especially in more fast paced games where the time it takes to take a screenshot and process it is too slow to keep up with the changing state of the game.

Finally, you may recall from Section IV-A that in the case of eCommerce systems, there was a desire to develop a site-agnostic technology stack in which scripts would execute seamlessly regardless of the website being tested. This was achieved through the development of dedicated classifiers. A classifier takes as input a user interface component, or a whole HTML page and delivers a verdict as to whether or not it is likely to be (for example) a *search button* or a *product information page*. Once a script in the eCommerce DSL was compiled, the test automation framework would execute it and when it comes across an instruction like `click on the search button`, it would extract all clickable components from the current page and feed them to the search classifier in order to find the one which is most likely the right button.

V. Conclusion and Future Work

In this paper we presented five case studies involving the design and utilisation of domain specific languages in the field of automated software testing. We proposed that such languages can be seen to exist in a hierarchy whereby the root DSL is highly generic and embodies generic testing concepts whilst DSLs lower down in the hierarchy become increasingly specific to particular domains. Our case studies seem to indicate that a layered approach to language design and test automation helps separate concerns between domain experts, users of the language and engineers who wire the language into test automation frameworks. We also observed that without management buy-in and seamless incorporation of DSL development into the software development process, such efforts are likely to fail. Finally, we observed that the depth of a DSL on the hierarchy has an impact on which attributes of the language should be given more importance.

With regards to future work, we are currently looking to gauge industry opinion on the development of structure DSLs for software testing; ideally followed up by a number of long-running case studies in which we can observe the development and long-term evolution of such DSLs. Finally, in a bid to reduce barriers-to-entry of DSLs, we are looking at improving the Gherkin language itself such that we can make it more structured whilst maintaining the familiarity which current practitioners have with the technology.

References

[1] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[2] M. Wynne and A. Hellesoy, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.

[3] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[4] C. Hoare, “Hints on programming language design,” DTIC Document, Tech. Rep., 1973.

[5] J. L. Bentley, “Little languages,” *Commun. ACM*, vol. 29, no. 8, pp. 711–721, 1986.

[6] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[7] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, “Design guidelines for domain specific languages,” *arXiv preprint arXiv:1409.2378*, 2014.

[8] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. Addison-Wesley Reading, 1999.

[9] U. Gundecha, *Selenium Testing Tools Cookbook*. Packt Publishing Ltd, 2012.

[10] P. Costa, A. C. Paiva, and M. Nabuco, “Pattern based gui testing for mobile applications,” in *2014 9th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2014, pp. 66–74.

[11] G. Shah, P. Shah, and R. Muchhala, “Software testing automation using appium,” 2014.

[12] T. Yeh, T.-H. Chang, and R. C. Miller, “Sikuli: using gui screenshots for search and automation,” in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 2009, pp. 183–192.

[13] M. Fewster and G. Consultants, “Common mistakes in test automation,” in *Proceedings of Fall Test Automation Conference*, 2001.

[14] D. Graham and M. Fewster, *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012.

[15] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.