Cost Study of Test Automation Over Documentation for Microservices

Shan Shan, Marcela Reyna, Jiting Xu, Chris Skirvin, Nanditha Patury and Russell Scott

ABSTRACT

There are many development test frameworks targeted specifically for developers; e.g. Pytest [1], TestNG [2]. Additionally, there is a culture around developers where code is often preferred over documentation, e.g, Cucumber [3], Robot [4]. we define the category of test automations which provides readability as Test Over Documentation, and identify the common characteristics of it. Readability in the tests has many benefits, like helping improve code quality, however, adding readability to the test code affects the original structure of the test code, adding more cost to test development if the test code is poorly developed. Test automation is expensive, test over documentation can be even more expensive. The cost problem of building and maintaining test automation is one of the biggest problems in the arena of test automation [5], thus it is the same for Test Over Documentation. Test Over Documentation has many subcategories, each add readability to the tests in a different way, thus the cost is different. In this paper, we study the cost problem of Test Over Documentation, evaluate the cost of each subcategory for testing Microservices [6], hoping to give the testers some reference for tooling selection. Additionally, we propose a new design pattern named Element Pattern to help write Test Over Documentation with minimum cost and best readability. We also investigate if automatic test generation using model based testing [7] can help reduce cost. The experiment results compare different Test Over Documentation approaches against our proposed benchmark metrics.

1Shan Shan, Marcela Reyna, Chris Skirvin, Nanditha Patury, Russell Scott, Cognitive Scale, 9500 Arboretum Blvd L-1, Austin, TX 78759, U.S.A.
2Jiting Xu, Ebay, 7700 West Parmer Lane, Austin, TX, 78729, U.S.A.
INTRODUCTION

Software Testing utilizes 40%-50% of total resources, 30% of total effort and 50%-60% of the total cost of software development approximately [8]. Test automation has been proposed as one solution to reduce these costs. Since it usually requires substantial upfront investments, thus, minimizing the cost of building and maintaining test automation is a major problem in the arena of test automation.

There is a category of test automation which provides readability in the tests. Readability in test code has many benefits. First, it improves software quality. Readability improves the communication and collaboration between product owners, test engineers, developers, and customers. Tests are written in a plain English, a business language that all stakeholders can understand. As a result, improved software quality is not strictly the concern of a Quality Analyst (QA) because the entire software development team is responsible for the quality of the software. Figure 1 is an example written using cucumber which belongs to Behavior Driven Tests [9]. Part 1 is high-level test cases written in plain English; Part 2 is low-level coded functions. Second, Readability helps managing and maintaining the test cases. Assume there are hundreds or thousands of test cases, readability helps testers follow test cases easily. Third, Readability makes it possible for non-technical people to write test automation with little or no training. For example, developers can create steps as in Figure 1. Part 2 in advance, then non-technical people can use those steps to create test cases later. Finally, readability makes auto generation of test automation simple and easy by hiding coding implementation details under textual description. For example, in [10], skyfire is developed to generate cucumber test cases from UML state machine diagram automatically. Test scenarios in natural language are generated, which is easier than the automatic generation of test cases in pure code because code implementation details (like variable definitions, method signatures, etc.) are ignored.

However, adding readability to test automation could increase the cost of test automation development and maintenance potentially. When the author works on inBloom [11] project, a $100 million student data collection project funded by the Gates Foundation, each developer spent approximately 40% of their time on developing and maintaining test automation code using cucumber as test automation framework according to experience. This phenomenon is seen in many companies. Some companies are forced to give up test automation and go back to manual testing because of the release deadline, high maintenance and development cost.

Thus, the objective of our study is the cost problem of test automations with readability. Automation cost includes fixed automation costs (hardware, test software license, tool training, etc.) and variable automation cost (test case design for automation, test case implementation, test results analysis, etc.) [5]. In this paper, only the cost of test case implementation and cost of test case maintenance are considered. We study the category of test automations which provides readability, and name it Test Over Documentation. We also evaluate each subcategory of Test Over Automation.
mainly from cost perspective using theoretical analysis and experiments, hoping to give testers some references for tooling selection.

Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity. REST [12] based micro-services examples are most popular because of their simplicity; services communicate directly and synchronously with each other over HTTP, without the need for any additional infrastructure. Thus, the testing problem of REST based micro-services is what most IT companies need to solve. The two fundamental parts of micro-services are RESTful Web Services and Web GUI. In this paper, we focus on studying the cost problem of Test Over Documentation for RESTful Web Services and Web GUI.

For the rest of this paper; In Section 2, the formal definition of Test Over Documentation will be given. There are too many tools supporting Test Over Documentation, it is impossible to compare one by one. In Section 3, we choose one from each subcategory after analyzing the current test automation toolkits. In Section 4 we propose Element Pattern to help create Test Over Documentation. In Section 5, we analyze the cost of each Test Over Documentation subcategory in theory; in Section 6, we investigate how to use model based testing to reduce the cost of Test Over Documentation; In Section 7, we propose first time benchmarks metrics to compare different approaches. Finally, in Section 8, the paper concludes.

TEST OVER DOCUMENTATION

Based on reviewing the test automation literature, analyzing current test automation toolkits in the market and learning from real project experience, we divide test automations into two categories based on readability: Test Over Documentation and Test without Documentation. Test without Documentation is the category of test Definition 1: Test Over Documentation is a category of test automations which

<table>
<thead>
<tr>
<th>TABLE I. SUBCATEGORIES OF TEST OVER DOCUMENTATION.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Over Documentation</td>
</tr>
<tr>
<td>Behavior Driven (BDD)</td>
</tr>
<tr>
<td>Specification BDD</td>
</tr>
</tbody>
</table>
maps test code into human-readable template or scenarios to provide readability in the tests.

Behavior Driven (BDD) is generally regarded as the evolution of Test Driven Development (TDD) [18] and Acceptance Test Driven Development (ATDD) [19]. BDD is focused on defining fine-grained specifications of the behavior of the targeting system, in a way that they can be automated. The main goal of BDD is to get executable specifications of a system. BDD relies on ATDD, but in BDD tests are clearly written and easily understandable, because BDD provides a specific ubiquitous language that helps stakeholders to specify their tests. Data-driven testing is the creation of test scripts to run together with their related data sets in a framework. The framework provides re-usable test logic to reduce maintenance and improve test coverage. Input and result (test criteria) data values can be stored in one or more central data sources or databases, the actual format and organization can be implementation specific. Keyword-driven testing, also known as table-driven testing or action word based testing, is a software testing methodology suitable for both manual and automated testing. A keyword defines an action against the system under test. A keyword is mapped to a method in the test code.

Test Over Documentation represents a category of test automations (Table I), including Specification [13] Behavior Driven (BDD) and User Story [14] BDD, Keyword Driven (KDD) [15, 16] and some Data Driven (DDD) [17]. DDD tests can be written with or without documentation. The DDD tests with readability belongs to Test Over Documentation. The DDD tests with readability can be implemented using pure code, BDD and KDD. Thus, in this paper, DDD is studied inside BDD and KDD.

We have also identified the common characteristics of Test Over Documentation:

• Readability in the test cases
  All Test Over Documentation approaches provide readability. Tests are written in a human-readable, domain-specific language.
• Mapping test code to textual description
  In Test Over Documentation, methods or code blocks always map to textual descriptions. Some approaches map each line of a test case to a method. Each line can be named a step, a keyword, an action. For the rest of the paper, to be simple, we call each line a step. Step definitions are keyed by their snippets of text from the test scenarios. Steps, when used loosely, has two (closely related but vitally distinct) meanings, depending on context. In some approaches, inside test files, steps are the textual descriptions which form the body of a test case; inside a coding file, steps refer to the matcher methods. However, some approaches embed and map the specifications in readable language instead of steps to the executable test code.
• Two parts
  Test Over Documentation has two parts, the outwards facing test case steps and the inward facing step definitions. Thus, most test automation over documentation project has two types of files: tests file contains only high-level test cases; coding file has the low-level definition of steps. There is an exception, specification BDD embeds the specifications directly in the test code in the same file.
From practical experience, we find the cost of each subcategory of Test Over Documentation are quite different. For example, the cost of mapping scenarios to test code (Figure 1) is very expensive using User Story BDD, some testers prefer writing tests using pure code to avoid the cost. However, some subcategories like Specification BDD do not have the mapping cost. In Section 3 we analyze the toolkits of different subcategories and select one for each to study the cost problem.

**STUDY THE TOOLKITS**

The research approach here is to study the current toolkits, classify them into a few groups based on their main characteristics, choose one from each group to implement the same set of test suites, then compare using our proposed benchmark metrics. Most toolkits are hybrid tools. For example, Robot supports BDD, KDD and DDD. However, its main characteristics is KDD, so it is considered as a KDD toolkit.

<table>
<thead>
<tr>
<th>Specifications BDD</th>
<th>Tool</th>
<th>Language</th>
<th>Tool</th>
<th>Language</th>
<th>Tool</th>
<th>Language</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>User Stories BDD</th>
<th>Tool</th>
<th>Language</th>
<th>Tool</th>
<th>Language</th>
<th>Tool</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JBehave</td>
<td>Java</td>
<td>Behave</td>
<td>Python</td>
<td>Lettuce</td>
<td>Ruby, Java</td>
</tr>
<tr>
<td></td>
<td>Radish</td>
<td>Ruby</td>
<td>SpecFlow</td>
<td>Python</td>
<td>Behat</td>
<td>Freshen</td>
</tr>
<tr>
<td></td>
<td>Ruby</td>
<td>C#</td>
<td>PHP</td>
<td>Python</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RBehave</td>
<td>Ruby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE II. THE BDD TOOLKITS ANALYSED IN OUR STUDY.
TABLE III. THE KDD TOOLKITS ANALYSED IN OUR STUDY.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Robot</th>
<th>FitNesse [45]</th>
<th>RedwoodHQ [46]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Python, Java</td>
<td>Java, C++, Python, Ruby, Delphi, C#, etc.</td>
<td>Java, Python, Groovy, C#</td>
</tr>
</tbody>
</table>

Behaviour Driven Test Automation Frameworks.

There are various toolkits supporting BDD. One subcategory of BDD tools use User Stories (Story BDD) as input language as in Table II. Usually those tools use Gherkins [20] as the language to define test cases. Gherkins is designed to be non-technical and human readable, and collectively describes use cases relating to a software system. Cucumber written in Ruby is one of the most popular BDD tools which supports Gherkins. There are many other similar tools use Gherkins, like Radish [21], Behave [22], Lettuce [23], PyTest-BDD [24] and Freshen [25] are python implementations; JBehave [26] is a pure Java framework; RBehave [27] is the ruby version of JBehave; SpecFlow [28] and NBehave [29] are C# implementation; Behat [30] for PHP. Since python is our testing language, we choose Cucumber and Lettuce for stories-based BDD because Lettuce is the python implementation of Cucumber.

Another subcategory of BDD uses functional Specifications for units (Specification BDD) that are being tested. These specifications often have a more technical nature than user stories and are usually less convenient for communication with business personnel than are user stories. Such a specification may exactly specify the behavior of the component being tested, but is less meaningful to a business user. As a result, specification-based testing is seen in BDD practice as a complement to story-based testing and operates at a lower level. Specification testing is often seen as a replacement for free-format unit testing. We choose the most popular JavaScript tool Mocha to study the cost problem of specification based BDD tools.

Keyword Driven Test Automation Frameworks.

Robot is a Keyword Driven Test automation framework for acceptance testing. Using Robot, real language is used for keyword description, so it’s easy to follow test case—even for non-technical person, which, together with its simple usage and easy library extension, make it great tool for test case automation. [47] describes their experience of using Robot framework for automation of existing functional regression test cases within short time and with great success and thus saving costs and enhancing the quality of the software project. RedwoodHQ is an action driven Development framework. Action is a keyword that describes what event it is about to do. FitNesse is a keyword Driven framework too, each test in FitNesse is a wiki page. FitNesse has lots of common aspects with Robot framework: Keywords are written in Java and included as libraries by creating JAR-files for example. Several
keywords can be combined to create an individual test case. Parameters can be given to those keywords to test different scenarios. Tests can be executed by a script and report is generated based on the execution. Table III has all the KDD toolkits we have looked at. After careful analysis, Robot framework was chosen because python is our backend testing language and Robot framework is a python based test automation framework. Besides, Robot framework is one of the most popular KDD test automation frameworks so it is easy to get tutorials and community supports.

After selecting a representative from each category, we use the selected toolkit to implement the same set of test cases and then evaluate the cost of different Test Over Documentation approaches.

### ELEMENT PATTERN

Various design patterns are proposed to design and implement test automation. Some of the patterns are inherited from developer’s design patterns directly, for example, all the patterns in Gang of Four [48] can be applied on test automation. There are some patterns targeted specifically for test automation, e.g. Recorded Tests Pattern, Modular tests Pattern, Hybrid testing pattern, Page Object Pattern and Single Page Pattern [49, 50, 51, 52, 53], etc. However, all the existing patterns provide us no guidance on how to split test logic into steps for Test Over Documentation.

It is recommended that in BDD the test cases are phrased declaratively rather than imperatively—in the business language, with no reference to elements of the UI through which the interactions take practice [54, 55]. However, we have seen an increased development cost if defining the steps are solely based on business logic. New steps are added whenever the business logic is missing, even if it refers to the same web GUI element. For example, in Figure 2, the step definition of "When I navigate to the Dashboard home page" is too specific. Similar steps like "When I navigate to the Realm page" need to be implemented if the test case needs to navigate to the Realm page. In fact, a generic step "When I navigate to the "(*)" could be defined to cover the similar business logics to reduce cost, "Dashboard home page" and "Realm page" are passed in as arguments.

However, if step definitions are too abstract, the test automation code ends up with poor readability. Figure 3 is a robot test cases with poor readability. Although good code reusability and maintenance cost are obtained, the test cases become very difficult for non-technical people to understand.

It is very difficult to find a standard and optimum way to define steps when writing test cases in business language because business logic varies from project to project. Test Over Documentation contains many other subcategories, so the goal here is to find a pattern to help define test over documentation steps with less consideration of business language. This pattern will be useful for Test Over Documentation. It will be used to estimate the cost of business oriented BDD as well in Section 7. This pattern should help create a set of steps with minimum cost. Each
step can be reused, shared and still produce good readable information. After learning all the lessons from failures, we propose Element Pattern, the concept of which is to map each operation for a single web element (page, browser, edit box, button, image, dropdown box, etc.) or API endpoint to a test code method or step (for example, click a button). In Definition 2, we present Element Pattern in a formal way. We prove Element Pattern is the optimum way of splitting test logic theoretically in Theorem 1. Experiments are done to validate the Element Pattern in Section 7 too.

Definition 2: Element Pattern is a test software design pattern which emerges when the tester splits the test code logic of a RESTful web application so that each operation for a single web element or API endpoint maps directly to a test code method or step.

Theorem 1: Element Pattern is the best way to divide test logic for RESTful Web applications

Proof: Given splitting test logic into a set of steps denoted as $S = \{s_1, s_2, \ldots s_n\}$, $n \geq 1$ using Element Pattern. Assume there is another better way which splits tests into another set of steps denoted as $U = \{u_1, u_2, \ldots u_m\}$, $m \geq 1$.

Claim 1: Any step $s_i$ in $S$ is the smallest unit we can have. To prove that, assume there is a step $u_j$ in $U$ which is smaller than the relevant step $s_k$ in $S$ (by relevant steps, we mean $u_j$ and $s_k$ overlap), then readability is lost as shown in Figure 3 for RESTful testing; for Web GUI, each step is a single operation of a single web element, it is impossible to have smaller step definition. Thus, claim 1 is correct.

Claim 2: The size of set $U$ is no less than the size of $S$. Assume the size of $U$ is smaller than the size of $S$, based on Claim 1, it is derived that some API endpoint or some operation of a single web element is not covered. Thus, claim 2 is correct.

In conclusion, element pattern is the optimum way to split test logic for RESTful web applications.

It is not required for BDD to identify the whole business flow. KDD which needs the keywords defined, which will later be used to create test scripts, require the application functionality clearly defined in terms of different flows it contains. With Element Pattern, a step could be added/deleted/updated at any time whenever a new web element or API endpoint is added/deleted/updated, which is very convenient.

---

**Scenario:** Valid user login

Given the s1 securityEvent collection is empty
# hitting static URL
When I access "/static/html/test.html"
Then I can see "Static HTML page"
When I navigate to the Dashboard home page
Then I should be redirected to the Realm page
When I select "Illinois Daybreak School District 4529" and click go
And I was redirected to the "Simple" IDP Login page
When I submit the credentials "linda.kim" "linda.kim1234" for the "Simple" login page
Then I should be redirected to the Dashboard landing page

Figure 2. An Example of Expensive Test Cases Using Business Language.
COST ANALYSIS OF TEST OVER DOCUMENTATION

Before using experiments to compare the cost of different Test Over Documentation approaches, we make some theoretical analysis based on analysing the selected toolkits and practical project experience. In this section, we analyze the cost of specification BDD, Story BDD and KDD.

• Value passing or state sharing between steps:

  In KDD, keywords can return value because they are methods. However, in Story BDD, steps can’t have return value because they are usually written in English-like language, they must either use global hooks or class variables to pass values between steps. As a result, it is difficult to define and use steps from different files or libraries.

  Using Specification BDD, variables could be used between steps easily because code and text description are in the same file. However, same as Story BDD, it is difficult for Specification BDD to use steps from different files or libraries. In KDD, there is no such problem. Steps from any libraries can be used together freely. In this way, KDD facilitate code sharing even across companies and organizations to reduce cost.

• Extra mapping cost:

  In KDD, methods are mapped to keywords directly, no extra mapping cost. For Specification BDD, there is no extra mapping cost neither. However, most of the popular Story BDD frameworks have extra mapping cost.

• Maintenance and development cost:

  In KDD, a keyword is a method, more flexible, easy to refactor and maintain. By facilitating modularization and abstraction, the KDD framework dramatically improves the reusability and maintainability of the test code as well as the readability of both test code and test cases. Because of inherited from object-oriented approaches modular structure of keywords, keywords could be reused often. In this way, compared with BDD, KDD cut development and maintenance cost.
From the above theoretical analysis, Story BDD has the highest cost, KDD is the cheapest, Specification BDD has no extra mapping cost from textual step description to code implementation, however, it is difficult to share and reuse steps. In Section 7, experiment results are used to estimate and compare the cost of each approach.

MODEL-BASED TESTING TO AUTOMATE TEST GENERATION

We also investigate if Model-Based Testing [56-62] help reduce the cost of test over documentation through automatic test generation. In [10], a Model-Based Testing (MBT) tool, skyfire is presented and can easily be used to generate effective UAT test scenarios from UML state machine diagrams. In [63], another Model-based Testing tool, MISTA is implemented to generate Integration and System Test Automation using robot framework, other languages like java and c, etc. are supported too. MISTA supports automated generation of executable robot test code. For our project, we create a robot keyword generator which reads swagger API docs [64] and generate a robot keyword for each API endpoint. As shown in Figure 4, MISTA transfers robot keywords and graph model into executable robot tests. In Section 7, we compare MBT Keyword Driven approach with others using experiments.

![Figure 4. Model Based Keyword Driven Architecture Implemented on MISTA and Robot.](image)

COMPARISON AND EXPERIMENT RESULTS

To compare different approaches for Test Over Documentation, we propose a set of benchmark metrics as shown in Table IV. Benchmark metrics allows the QA Engineer to quantitatively compare the cost and readability of different test automation approaches. “Total Test Case Lines Written” calculates the total number of lines written for a single test case. For our study, a single test case consists of two factors which must be summed together once identified. The first factor is a description of the test steps required to validate a feature written in a natural language or “textual” style. For example, “Open browser and navigate to desired
TABLE IV. PROPOSED BENCHMARK METRICS TO COMPARE TEST OVER DOCUMENTATION APPROACHES.

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Test Case Lines Written</td>
<td>#lines</td>
<td>The total number of lines written for a single test case and step definitions.</td>
</tr>
<tr>
<td>Lines for Test Cases</td>
<td>#Test Cases</td>
<td>The total number of lines written for an automated test case.</td>
</tr>
<tr>
<td>Code Reused</td>
<td>Code Reused (%)</td>
<td>The percentage of code re-used from existing publicly shared libraries.</td>
</tr>
<tr>
<td>Sharable Code</td>
<td>#Code Sharable</td>
<td>The total number of lines written by the QA Automation Engineer which can be shared with the organization.</td>
</tr>
<tr>
<td>Sharable Functions</td>
<td>#functions sharable</td>
<td>The total number of programmatic functions or methods which could be shared with the organization.</td>
</tr>
<tr>
<td>Readability</td>
<td>Readability</td>
<td>An indication of the level of difficulty required to understand a test case.</td>
</tr>
</tbody>
</table>

TABLE V. DESCRIPTION OF READABILITY VALUE SET.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>The test case contains poor readability such as a test case are without any textual descriptions. Or if a textual description is provided, it is difficult to understand. The test case is in a keyword format or the test case automation code and test case textual code are in the same source file. This level also represents a test case which has good readability so that a non-technical person understands the test case with little to no training.</td>
</tr>
<tr>
<td>Level 1</td>
<td>The test case is written in human-readable language which provides good readability. Additionally, a non-technical people can easily understand the purpose of the test case.</td>
</tr>
<tr>
<td>Level 2</td>
<td>The test case is written in human-readable business language which provides optimal readability; all stakeholders can understand the test case with no explanation.</td>
</tr>
</tbody>
</table>

website. Validate when button x is clicked, window y is opened.” For this example, each sentence has a weight of 1. For the second factor in determining the “Total Test Case Lines Written”, we consider the number of lines of code implemented for a test case where each line of code also has a weight of 1. Once these variables are known, they can be summed together to determine the Total Test Case Lines Written. This represents a measure of the total amount of work needed to implement a documented test case. “Lines of Test Cases” represents the amount of work done for documented test cases. As with determining the Total Test Case Lines Written, each sentence of the documented test case has a weight of 1 which should be summed per test case. “Code Reused” is the percentage of code reused from existing publicly shared libraries. The greater the value of “Code Reused” the further test automation cost is reduced. This would then go on to indicate that the QA team could potentially reuse more existing code as opposed to developing an in house solution. “Sharable Code” is the number of lines of code which could be shared in future test automation projects and is additionally developed by the in house test automation team. “Sharable Functions” is the number of programmatic functions or methods which can easily be shared across projects and organizations. As with “Code Reused”, the greater the value of “Sharable Functions” and
“Shareable Code”, the further test automation maintenance is reduced. “Readability Metrics” is an indication of how easy or difficult it is to understand a test case from the perspective of the non-technical team member. For our study, “Readability Metrics” consists of 4 levels (0 – 3), the meaning of each level is described in Table V.

In this section, Keyword Driven approach is denoted as KDD. Keyword Driven approach using shared libraries is denoted as KDD*. Additionally, BDD represents for Behavior Driven and MBT KDD is the Model-based Keyword Driven approach.

Element Pattern

To evaluate the proposed Element Pattern, we use the dashboard component of in Bloom project as a sample testing target. The Dashboard is a proof-of-concept application that leverages a rich RESTful API to demonstrate educator access to a broad range of student information. The Dashboard is implemented as a Java web application. We test the student feature and the dash feature of dashboard with BDD written in business language and BDD using Element Pattern. The experiment results are in Figure 5. For student feature testing, without Element Pattern, the tests are written in business language, as a result a set of 31 steps are created; with Element Pattern, only 12 steps needed. The experiment results show that although it is recommended to write BDD using business language, the cost of writing BDD tests written in business language is more than 2 times of that of using Element Pattern.

Restful API Testing

We use different approaches to test RESTful APIs and compare the cost. We use element pattern to split the test logic, one step is one API endpoint, then implement the same set of test cases using pure python, Behaviour Driven (lettuce), KDD (robot), KDD* (robot framework and Requests Library [65]) and MBT KDD separately. Pure code implementation is the baseline for comparison, so we make it use Element Pattern too. The experiment results as in Table VI show that using element pattern, BDD’s cost is almost equal to that of pure code and KDD’s cost is less than that of pure code which means there is no extra cost to add readability in tests using KDD. The experiment results in section 7.1 shows that the cost of BDD in business language is more than 2 times of BDD using Element Pattern, so the cost of BDD in business language can be more than 2 times of pure code. Although BDD written in business language has best readability (Level 3), its cost is the highest - more than 2 times of pure code implementation. Using KDD and robot frameworks extensive libraries (KDD*), the cost is lower than pure code, BDD and KDD. Experiment results show that MBT KDD is the cheapest among all the approaches. In our project, to test the same set of API endpoints, it takes 4 testers about one month to automate tests using pure code, however, it takes one tester only a week if
using MBT KDD. KDD* has the best code reuse rate, 83% of the test code are from existing libraries, however it is very difficult to understand the test cases because the steps defined in robot frameworks’ Requests Library are too generic and abstract. Pure Code, BDD and KDD approaches have more functions which could be shared and reused in the future because those approaches write more new methods from than others.

![Figure 5. Comparison the splitting approach with and without Element Pattern.](image)

**TABLE VI. RESTFUL API TESTS.**

<table>
<thead>
<tr>
<th>Approach</th>
<th>#lines</th>
<th>code reused</th>
<th>#code sharable</th>
<th>#test cases</th>
<th># functions sharable</th>
<th>Readability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Code</td>
<td>1419</td>
<td>0%</td>
<td>637</td>
<td>782</td>
<td>77</td>
<td>Level 0</td>
</tr>
<tr>
<td>Spec BDD</td>
<td>1496</td>
<td>0%</td>
<td>714</td>
<td>631</td>
<td>77</td>
<td>Level 2</td>
</tr>
<tr>
<td>KDD</td>
<td>1419</td>
<td>0%</td>
<td>637</td>
<td>631</td>
<td>77</td>
<td>Level 1</td>
</tr>
<tr>
<td>KDD*</td>
<td>1351</td>
<td>83%</td>
<td>179</td>
<td>296</td>
<td>15</td>
<td>Level 0</td>
</tr>
<tr>
<td>MBT KDD</td>
<td>148</td>
<td>75%</td>
<td>327</td>
<td>296</td>
<td>16</td>
<td>Level 1</td>
</tr>
</tbody>
</table>

**TABLE VII. WEB GUI TESTS.**

<table>
<thead>
<tr>
<th>Approach</th>
<th>#lines</th>
<th>#test cases</th>
<th>code reused</th>
<th>#code sharable</th>
<th>#functions sharable</th>
<th>Readability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Code</td>
<td>2605</td>
<td>1391</td>
<td>0</td>
<td>1283</td>
<td>69</td>
<td>Level 0</td>
</tr>
<tr>
<td>Story BDD</td>
<td>2674</td>
<td>1391</td>
<td>0</td>
<td>1283</td>
<td>69</td>
<td>Level 2</td>
</tr>
<tr>
<td>Spec BDD</td>
<td>2531</td>
<td>1391</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Level 0</td>
</tr>
<tr>
<td>KDD</td>
<td>2605</td>
<td>1391</td>
<td>0</td>
<td>1283</td>
<td>69</td>
<td>Level 1</td>
</tr>
<tr>
<td>KDD*</td>
<td>1391</td>
<td>1391</td>
<td>100%</td>
<td>0</td>
<td>0</td>
<td>Level 1</td>
</tr>
</tbody>
</table>

**TABLE VIII. ELECTRON GUI TESTS.**

<table>
<thead>
<tr>
<th>Approach</th>
<th>#lines</th>
<th>code reused</th>
<th>#code sharable</th>
<th>Readability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectron</td>
<td>1026</td>
<td>0%</td>
<td>0</td>
<td>Level 1</td>
</tr>
<tr>
<td>Robot*</td>
<td>431</td>
<td>100%</td>
<td>0</td>
<td>Level 1</td>
</tr>
</tbody>
</table>
GUI Testing

The Web GUI of all our projects share a same set of web elements, like table, calendar, button, etc. We use Element Pattern to split the testing logic into a set of steps, then use pure code, Story BDD (cucumber), Specification BDD (Mocha), KDD (Robot framework with Javascript) and KDD* (Robot framework and Selenium2Library [66]) approaches to implement the same set of test cases separately. Those steps are tested with Chrome, Firefox, Internet Explorer and Safari on Windows, Mac and Unix platform. The test results are listed in Table VII. The experiment results show that using element pattern, Story BDD’s cost is almost equal to that of pure code. Although the total cost of Specification BDD is lower than Story BDD, Story BDD has better readability than Specification BDD. Besides, Specification BDD makes steps sharing more difficult than Story BDD. There is no extra cost using KDD compared with pure code implementation. KDD* is the cheapest among the three approaches because no new code need to be written.

We also test Electron GUI [67] on Windows and MAC platform. Because Electron has built-in support from Chromium [68], so we could use Robot framework’s selenium2Library to test Electron application as well. Experiment results in Table VIII show that the cost of testing electron GUI using robot framework’s selenium2Library is less than half of the cost of testing using Spectron [68]. In our electron project, using Spectron it takes 2 testers more than a month to finish automating a set of test cases, using KDD* (Robot) only 1 tester 1 month.

CONCLUSION

Cost is a major concern of test automation in industry. In this paper, we study the group of test automations which provides readability in the tests and define the group as Test Over Documentation formally. We identify the common characteristics of Test Over Documentation and evaluate the cost of Test Over Documentation subcategories theoretically and experimentally. BDD in business language has the best readability, but has the highest cost - more than 2 times extra cost in our case. KDD with existing library not only has no extra cost, but also reduce the cost. MBT KDD is the cheapest approach which reduces the cost significantly. We also propose a new design pattern named Element Pattern to help create Test Over Documentation and investigate using Model-based testing to reduce test automation cost.

REFERENCES


