Automatically Generating Tests from Natural Language Descriptions of Software Behavior

Sunil Kamalakar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

September 23rd, 2013

Blacksburg, Virginia

Keywords: Behavior-Driven Development, Test-Driven Development, Natural Language Processing, Probabilistic Analysis, Automated Code Generation

Automatically Generating Tests from Natural Language Descriptions of Software Behavior

Sunil Kamalakar

(ABSTRACT)

Behavior-Driven Development (BDD) is an emerging agile development approach where all stakeholders (including developers and customers) work together to write user stories in structured natural language to capture a software application's functionality in terms of required "behaviors". Developers then manually write "glue" code so that these scenarios can be executed as software tests. This glue code represents individual steps within unit and acceptance test cases, and tools exist that automate the mapping from scenario descriptions to manually written code steps (typically using regular expressions). Instead of requiring programmers to write manual glue code, this thesis investigates a practical approach to convert natural language scenario descriptions into executable software tests fully automatically. To show feasibility, we developed a tool called Kirby that uses natural language processing techniques, code information extraction and probabilistic matching to automatically generate executable software tests from structured English scenario descriptions. Kirby relieves the developer from the laborious work of writing code for the individual steps described in scenarios, so that both developers and customers can both focus on the scenarios as pure behavior descriptions (understandable to all, not just programmers). Results from assessing the performance and accuracy of this technique are presented.

Acknowledgments

My deepest gratitude is to my advisor, Stephen H. Edwards. I have been amazingly fortunate to have an advisor who gave me the freedom to explore ideas on my own, and the guidance when needed. His patience, commitment and support helped me immensely throughout my work.

I would like to thank David A. Mittelman, my committee member, for his innovative ideas and numerous discussions on technical topics. I am fortunate to have worked in his Genomics Lab at Virginia Bioinformatics Institute where I was able to explore completely new avenues.

I would also like to thank Eli Tilevich, my committee member, for his support and prompt feedback. His courses were the most enjoyable ones at Virginia Tech.

I also thank Tung M. Dao for his ideas and encouragement.

Next, I would like to thank my wife, Yashaswi for her continuous encouragement and support, without her none of this would have been possible. Lastly, I would like to express my gratitude to my parents and my sister, Shilpa, without them as my source of inspiration and guidance, I would not be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Behavior-driven Development (BDD) is a relatively new agile development technique that builds on the established practice of test-driven development. Test-Driven Development (TDD) [2, 18], is an approach for developing software by writing test cases incrementally in conjunction with the code being developed: “write a little test, write a little code.” TDD provides a number of benefits [26], including earlier detection of errors, more refined and usable class designs, and greater confidence when refactoring. TDD facilitates software design by encouraging one to express software behaviors in terms of executable test cases.

Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis. It was originally conceived by Dan North as a response to limitations observed with TDD. With TDD programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails [27].

In BDD we specify each “behavior” of the system in a clearly written and easily understandable scenario description using natural language. These natural language scenarios help all stakeholders, not just programmers, to understand, refine, and specify required behaviors. Through the clever use of “glue code” provided by programmers once the scenarios are written, it is possible to execute these natural language scenarios as operational software tests. BDD is focused on defining fine-grained specifications of the behavior of the target system. The main goal of BDD is to produce executable specifications of the target system [31], while keeping the focus on human-readable scenario descriptions that can be easily understood by customers, as well as developers.

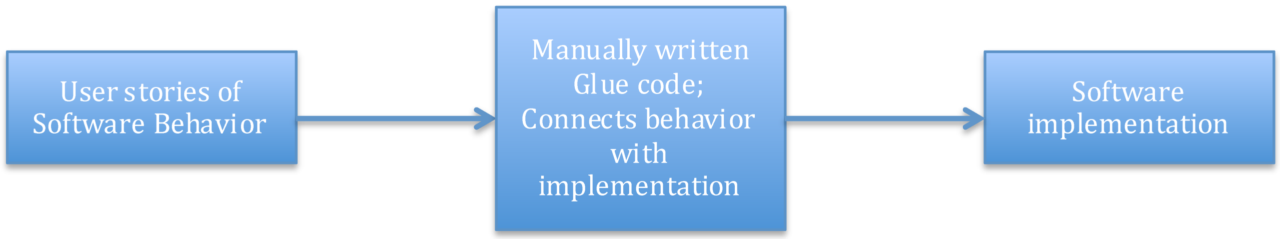

User stories of Software Behavior
Manually written Glue code; Connects behavior with implementation
Software implementation

Figure 1.1: Behavior-driven development work flow.

1.1 Problem Statement

One weak point of BDD is the "glue code"—programmers are still required to produce program "steps" that correspond to the basic actions described in the natural language scenarios. A number of tools have been developed to make the process of writing this glue code easier and streamlined, and to automatically map phrases in the scenarios onto such steps (typically using regular expressions). However, a manually-written bridge between the scenarios and the programmatic actions that correspond to them is still necessary.

Figure 1.1 shows the traditional work flow in BDD where user stories are compiled as software behaviors initially. Then, developers use BDD software tools to connect user stories onto software implementation. The natural language user stories contain inherent information about the behavior of the software. The question we put forth is: **Can we reduce the burden on programmers by utilizing information in the natural language, so that they don't have to hand-write the glue code?**

1.2 Motivation

The main theme and motivation of the thesis is:

> *In Behavior-Driven Development, instead of requiring programmers to write manual glue code, it is practical to convert natural language scenario descriptions into executable software tests fully automatically.*

A software tool which is capable of fully automating "glue code" creation will have the following benefits:

- **Reduction in effort, time and cost:** Developers who would have otherwise spent time and effort on manually writing the glue code, can now spend more time and effort on the other important aspects of SDLC. Time, effort and cost are all inter-related terms in a software project, and auto-generation techniques will end up showing more productivity and faster time to market (provided we have a reliable auto-generation engine).

- **Increase in adoption of BDD:** Adoption of BDD, requires an overhead where user stories or software behavior are mapped onto software implementation. Though this provides executable specification, programmers with a background in TDD, may see this as an initial hinderance. Research and industrial outlook on BDD suggests that it promises to be a long term beneficial method, and auto-generating step definitions could go a long way in helping its adoption.
- **Ubiquitous communication:** BDD in itself helps in improving communication among all stakeholder of the project, because everyone works together to create user stories. With an auto-generation approach, domain specific terminologies that the customer understands are forced to be carried over to the codebase. This reinforces the idea of having ubiquitous communication and understanding among customers and developers, not just at the specification level, but also at the implementation level.

1.3 Proposed Solution

Our solution is to process the user stories written for BDD using state-of-the-art Natural Language Processing (NLP) techniques. Simultaneously, we scan the implementation/skeleton code of the project and capture every required property of the code, using reflection techniques. Once we have these two pieces, we employ a custom probabilistic matcher to map the BDD instructions onto code segments. Finally, based on the results of the probabilistic matcher, the code generator spits out code representing these mappings.

This thesis will primarily focus on my work: *Kirby*, a software tool written in Java, for auto-generating glue code from BDD scenarios. Kirby is capable of auto-generating code for projects implemented in Java. It outputs a specific code format in JUnit, which is a widely used unit testing framework for Java. The reason we choose Java is because, it is one of the most popular programming languages in use today, along with being a strictly object-oriented and typed-language.

Figure 1.2 shows the proposed approach where the mapping/bindings are generated by Kirby. This provides a layer of abstraction, where it appears that the user stories can be directly executed on the software implementation without manually writing these bindings.

1.4 Thesis Organization

The Introduction presented the problem and the motivation to solve it. Further, Chapter 2 elaborates on certain topics which provide a background on the concepts and techniques

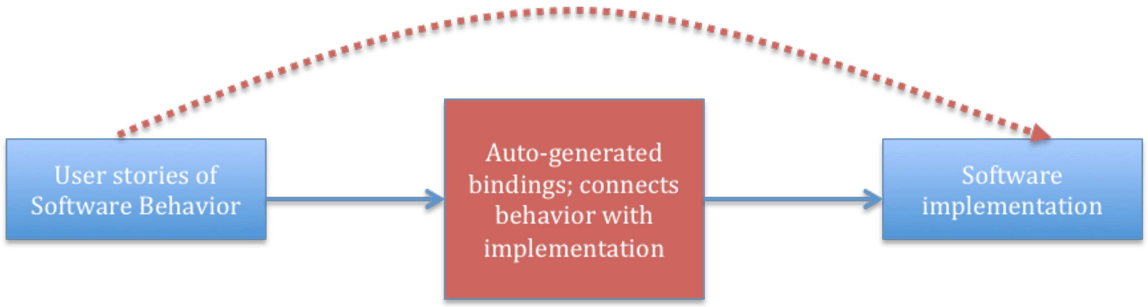

User stories of Software Behavior
Auto-generated bindings; connects behavior with implementation
Software implementation

Figure 1.2: Proposed behavior-driven development flow.

used in Kirby, and also discusses the ideas and previous work that has influenced the design and development of Kirby. Chapter 3 provides a system overview from the perspective of an end user using Kirby, along with showcasing the user interface. Chapter 4 provides a detailed design and implementation of the whole system, along with examples to help understand the implementation challenges. Chapter 5 provides an evaluation of Kirby from a usage, correctness and performance perspective. Finally, Chapter 6 discusses the key contributions of Kirby and some future directions that we envision.

Chapter 2

Background

Complexity is an essential part of building software. This idea was enforced by Frederick Brooks, who wrote his famous essay, “No Silver Bullet: Essence and Accidents of Software Engineering” [4], in 1986:

> *The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed. I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems. If this is true, building software will always be hard. There is inherently no silver bullet.*

According to Brooks, software complexity can be divided into two categories: accidental complexity and essential complexity. Essential complexity refers to a situation where all reasonable solutions to a problem must be complicated because the simple solutions would not adequately solve the problem. It stands in contrast to accidental complexity, which arises purely from mismatches in the particular choice of tools and methods applied in the solution. The complexity of software cannot be reduced without eliminating requirements; hence essential complexity is inherent and inevitable. But, accidental complexity arises from the approach chosen to solve the problem, and can be eliminated/reduced by using appropriate tools and development processes [19].

This essay brings to light that there is no straightforward way i.e. silver bullet to automate software engineering by having a computer to implement a solution for any given problem. Even after a quarter century of this writing, no silver bullet exists.

2.1 Agile Development Methods

In the last half a century, there have been various software development models and processes that have been adopted as a means to develop complex software. Making the essential complexity easier to deal with, and reducing the accidental complexity have been an important concern for these models.

Traditional software engineering processes like Waterfall model, Spiral model have been implemented in large-scale industrial software development and have had good success in the past. But, with the increasing complexity of software and a need for frequent changes to the requirements, Incremental and Iterative software development models have leaped ahead in the last decade. Agile development methods fall into this category.

The term Agile Software Development, often referred to as just Agile, was coined in the year 2001 with the formulation of the Agile Manifesto [3], which describes it as:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> - *Individuals and interactions over processes and tools*
> - *Working software over comprehensive documentation*
> - *Customer collaboration over contract negotiation*
> - *Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*

The Agile Manifesto in itself is an umbrella covering various actual development methodologies such as Scrum, eXtreme Programming (XP) and Rational Unified Process (RUP). After years of process and document heavy development methods, agile methods have been seen as a breath of fresh air [28]. This has created considerable interest in both industry and academia.

The agile process of development is very flexible and enablers of the technology have tried to incorporate various other techniques such as test-driven development, behavior-driven development, pair programming, continuous verification and integration into these approaches.

2.2 Test-Driven Development

Test-Driven Development (TDD) is a basic technique of agile development [2], and its existence is motivated by "Responding to change over following a plan", which is the fourth point of the Agile Manifesto [3].

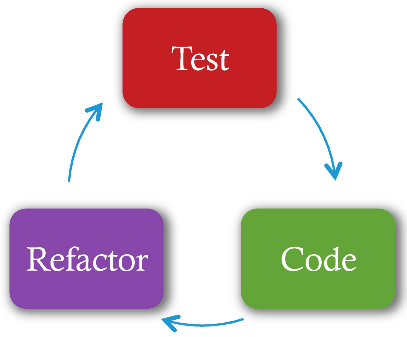


Figure 2.1: Test-driven development workflow.

Process

TDD is an incremental and iterative process, where there are three independent, well-defined phases.

1. Write tests
2. Write implementation code to pass the test
3. Refactor code

The main principle of TDD, also known as Test-First Development, is that test code is written even before implementation code. These validation tests provide a degree of confidence when dealing with code changes or refactoring. We verify this by running the test, which should fail when the implementation code does not exist. Then the implementation code written in the second phase has just enough content to pass the test. Once we have a passing test, we have the retrospective phase where we look back and refactor the code by structuring, optimizing and cleaning up any code smells.

This whole cycle, shown in Figure 2.1, is also known as the Red-Green-Refactor cycle. The initial step of writing tests will fail on execution, because of the lack of implementation code, indicating the Red phase. As we progress with the implementation, once we have a passing test, it is indicated by the Green phase. Software frameworks, which help in practicing TDD, follow this intuitive color convention in order to indicate the state of progress for the execution of tests.

Benefits

Benefits of TDD include [29]:

- *Testability:* Testability of the system is improved through the act of writing tests first, requiring developers to think about how the code can be tested and designed to that requirement, no longer leaving testing as an afterthought. This, in turn, leads to a common reduction in effort required to test components developed using TDD.
- *Simplicity:* During the implementation of a feature, the test-driven developer performs the simplest activity to result in turning all tests green. This approach to development has the effect of simplifying code design and achieving the results that come with that including improved readability and understandability.
- *Quality:* Quality improves through the development of a regression test base rooted in the requirements. Building regression tests from the beginning allows for the continual verification of impact due to changes, giving developers and product owners a greater confidence that changes don't introduce negative side effects [25, 11].
- *Design:* Applications developed using TDD have testability built into them. This requires testability to be incorporated into the software architecture, which influences the design of the system from the grounds up. Advocates of TDD consider this a very important consequence of test-first approach.

While these are some of the high-level benefits of TDD, various other benefits exist in terms of cost, reliability, manageability and documentation. While these advantages are substantial, the greatest benefit of Test-Driven Development is producing "clean code that works" [2]. Test-Driven Development is primarily meant to yield good, clean code. It's not about the quality of the software; it's about the quality of the code.

Adoption of TDD requires some effort on the developer, who needs to develop a new mindset in the way they think about writing software. Writing good tests is an acquired skill and developers should follow a theme [23], also known as *F.I.R.S.T.* To be useful in real world scenarios, tests need to be: *Fast, Independent, Repeatable, Self-validating, Timely.*

Test Frameworks

Various unit and acceptance test frameworks exist which help in writing automated tests. xUnit is an unit test framework architecture that supports automated tests, and it is a de facto standard for unit test tools and is ported to several programming languages. Multiple xUnit implementations share the same basic framework architecture. The most important concepts of the architecture are assertion, test case, test fixture and test suite. An assertion is a true-false statement that is used in the actual test.

Below is an example of one scenario of a tic-tac-toe game, which is written using TDD. It is written in the Java language using the JUnit framework.

1. *Write test:* A single test is written initially. In this case we come up with a test where we need to decide if a user has won the tic-tac-toe game, if he has selected all cells in a column. The test code is simple that it creates a Game class, and calls the method play on it, representing the user action. The play method needs to return a true value, only after a user has won the game, in this case - if he has entirely filled a column.

```java
@Test
public void userWithThreeColumnsShouldWin() throws Exception {
  int user1Id = 0;
  Game game = new Game();
  assertFalse(game.play(0, 0, user1Id));
  assertFalse(game.play(1, 0, user1Id));
  assertTrue(game.play(2, 0, user1Id));
}
```

2. *Write implementation code to pass the test:* Now, we need to write the implementation class, which is called Game. It needs a data structure to hold the tic-tac-toe board, which can be represented as a two dimensional matrix. It also needs to contain the play method, which returns a boolean value indicating a win. Since the test is written keeping the 0th column in mind, the implementation code can contain only a validity check on that column, in order to pass the test. Though in reality, it would need to do more than that.

```java
public class Game {
  private int[][] grid = new boolean[3][3];

  public boolean play(int i, int j, int player) {
    grid[i][j] = player;
    return grid[0][0] == player &&
        grid[1][0] == player &&
        grid[2][0] == player;
  }
}
```

3. *Refactor Code*: It is very evident in this case that the code is going to break for other columns, which needs to be fixed. We fix this by performing a minor refactoring, and we can also add tests for those columns. As a developer, you need to decided based on the FIRST principle, which tests are required and which are not.

```
public boolean play(int i, int j, int player) {
 grid[i][j] = player;
 boolean retval = false;

 for(int i=0; i<=2; i++) {
  retval = (grid[0][i] == player) &&
      (grid[1][i] == player) &&
      (grid[2][i] == player);
  if(retval == true) break;
 }

 return retval;
}
```

This code is by no means complete, it's purpose is to introduce the reader to TDD. Once we have a green light that the refactor works as expected, we move onto the next test, and continue with the Red-Green-Refactor cycle.

Acceptance Test-Driven Development

Acceptance Test-Driven Development (ATDD) [14] is a software requirements specification and verification process. It is also known as specification by example, where there is an emphasis on automation of acceptance tests along with the specification of customer readable requirements. The main motivation for ATDD is that it keeps all stakeholders of the project involved and focused on the main goals of the project, along with improving communication through readable specification.

The main difference between TDD and ATDD is that ATDD looks at the specification and tests from a customer point of view, providing a high-level view of the system and it's requirements. TDD on the other hand, works more at the modular or unit level which is closer to the code.

ATDD process involves three roles [18]: *Customer, Tester and Developer.*

The process for ATDD is similar to TDD and involves three phases:

1. *Write Acceptance tests:* Based on the requirements brought forth by the customer, the tester will work with him to understand the specification better and come up with tests with an acceptance criteria. Concrete examples are used, and the customer needs to understand the tests, which can be in natural language or Domain Specific Language (DSL).

2. *Automate Acceptance tests:* The tester works with the developer to map the concepts in the concrete examples onto their implementation. This allows for automated execution of these acceptance tests.

3. *Implementation:* The developer architects and implements the system, to meet the acceptance criteria.

Though ATDD has seen momentum and has been adopted in academia and industry, no extensive evaluation using it has been reported [18].

2.3 Behavior-Driven Development

Despite the numerous advantages that TDD brings to the table, there are some inherent problems with the approach. The main problem with TDD is in the language it uses [1]. The word *"test"* is used too many times, which is interpreted as verification. TDD influences design and has various other benefits. TDD would benefit from a shift in its vocabulary. Instead of unit and test, we should use behavior and specification.

With TDD, programmers wanted to know [27]:

- Where to start
- What to test and what not to test
- How much to test in one go
- What to call their tests
- How to understand why a test fails

As a response to the limitations observed in TDD, Behavior Driven Development (BDD) was originally conceived by Dan North in the November 2006 issue of Better Software [27].

The formal definition:

> *BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software.*

A simpler definition:

> *Behavior-driven development is about implementing an application by describing its behavior from the perspective of its stakeholders.*

BDD is not a revolutionary idea, it derives from experiences in eXtreme Programming, Acceptance Test-Driven Planning, Lean principles and Domain-Driven design[27].

The main theme of BDD is that it allows software behavior to be specified in natural language, which can act as automated tests (with the help of software toolkits). It is still TDD with an extra layer of abstraction. And our belief is that*: BDD is TDD done right.*

Principles

The main goal of using BDD is delivering value to customers, and over time it has grown to become a more complete method [6].

This development is based on three basic principles [6]:

- *Enough is enough:* we should do just enough up-front analysis and design to get us started, any more is wasted effort.
- *Deliver stakeholder value:* the word stakeholder does not refer only to the customer who pays for the software, but to everyone who should benefit from using it. We should only do what delivers value or increases our ability to deliver value to stakeholders.
- *it is all about behavior:* at any level of granularity, we use the same linguistic constructs to describe the behavior.

The requirements of the system are expressed as user stories. These are further extended with acceptance criteria in the form of scenarios. These scenarios are then automated to form automated acceptance tests. The design and implementation of the functionality required by the user stories is done in the same way that TDD does.

Language

In this section we will discuss the language that BDD uses to specify requirements. It includes features, user stories and scenarios.

Features

The high-level requirement of the system are expressed as features. A feature provides a brief natural language description of the requirement. It uses ubiquitous domain language.

A template for writing a feature files is shown below.

```
Feature: name
As a - [user role]
I want to - [behavior]
In order to - [provide business value]
```

The template is self-explanatory, with the emphasis on business value that is going to be provided for a particular user role, through the behavior.

An example of a feature description is shown below for the tic-tac-toe example.

```
Feature: Winning Tic-tac-toe
As a - player
I want to - check if I have won the game
In order to - assess the current status of the game
```

These user roles represent different types of users of the system. Here, player of tic-tac-toe is the user, who wants to assess the status of the game to know if he has already won the game.

Stories

Features are sometimes too large to estimate accurately and completely in a single iteration. So these features are broken down into stories [6]. Each of these stories represents the who, what and why of the requirement, and takes a subset of the feature to represent. It follows the same template as the feature shown above, but are more detailed in nature.

Scenarios

Stories contain various scenarios, which represent acceptance criteria for the story. These scenarios are fine-grained and provide a concrete example of the behavior that the story intends to capture. The specification of stories is an iterative process, and scenarios help in the refinement of the story.

```
Scenario: name
Given - context
When - action
Then - outcome
```

"Given-When-Then" describes a story. *"Given"* describes an initial context, *"When"* specifies the action or an event that occurs. *"Then"* specifies the outcome which is the acceptance criteria.

A more complex story can be described using the template shown below, which uses conjunctions to combine clauses together.

```
Scenario: name
Given [Initial context] And [some more context]
When [Event] And [some other event]
Then [Outcome] And [some other outcome]
```

An example of a scenario, for the same tic-tac-toe game based on the same feature/story shown above would be:

```
Scenario: player with three columns should win
Given a tic-tac-toe game
When I play row 0, column 0
And I play row 1, column 0
And I play row 2, column 0
Then I should have won the game
```

This scenario describes how a user should have won the game if he marks all the columns in a row.

These scenarios are written in Domain Specific Languages (DSL), similar to natural language. It is easy for all stakeholders to interpret this information.

BDD toolkits

Many tools for BDD have been created for use in different contexts. These frameworks primarily help with the implementation phase, where they can provide bindings from the natural language scenarios onto code implementation. The sample set of different BDD toolkits for some programming languages are listed below.

- Ruby: *RSpec (http://rspec.info), Cucumber (http://cukes.info)*
- Java: *JBehave (http://jbehave.org), JDave (http://jdave.org)*
- C#: *NBehave (http://nbehave.org)*
- PHP: *PHPSpec (http://www.phpspec.net)*

We will discuss briefly about some of the widely used BDD frameworks for Ruby language.

RSpec

RSpec [6] is a software framework for writing low-level code specification in the language of BDD for Ruby programming language. It tries to use the convention of behavior rather than test [1]. Based on the name, it provides a domain-specific language for describing, structuring and defining specifications. The main focus of RSpec is the usage of the right conventions in describing software behavior.

An example of Ruby code in RSpec for a simple Calculator class performing a basic subtraction operation is shown below.

```
describe Calculator do

  before(:each) do
    @calculator = Calculator.new
  end

  subject {@calculator}

  context "handling subtraction with" do
    it "two numbers" do
      subject.subtract(5,2).should == 3
    end
  end
end
```

RSpec's conventions makes this example more like natural language rather than programming code. This is because of the use of words like *describe, context, it, should* as suggested by Astels [1].

Cucumber

For supporting high-level constructs of BDD, Cucumber is a widely used software framework for the Ruby language. It is to be noted that Cucumber also provides libraries for the Java language called Cucumber-JVM.

Make acceptance tests executable is the main goal of Cucumber. To do this, user stories are written in a DSL specified in *Gherkin* [6]. Gherkin has the same form as the *"Given-When-Then"* clause. Once the user stories are ready, the developer associates each step of the scenario onto a block of programmatic code which are called *step definitions*. Lastly, cucumber has a command line interface through which we can execute each scenario as an acceptance test.

For the Calculator example, the scenarios and step definitions in Cucumber are shown below.

```
#Calculator_feature.txt

Feature: Basic arithmetic calculator
As a - calculator user
I want to - perform basic arithmetic operations
In order to - speedup manual arithmetic calculations

Scenario: Arithmetic subtraction
Given a Calculator
When I subtract "2" from "5"
Then result should be "3"
```

We use the bindings provided by Cucumber to write the step definitions to connect the user stories with the implementation using regular expressions.

```
#Calculator_steps.rb

Given /a Calculator/ do
  @calc = Calculator.new
end

When /I subtract (\d+) from (\d+)/ do |num1, num2|
  @result = @calc.subtract(num1, num2)
end

Then /result should be (.*)/ do |result|
  @result.should == result.to_f
end
```

Once we have these step definitions ready, we can write out the implementation to pass the test.

```
#Calculator.rb

class Calculator
  def subtract(subtract_value, source_value)
    result = source_value - subtract_value;
  end
end
```

Behavior-Driven development research

There are few published studies on BDD, most of which take a relatively narrow view, treating it as a specific technique of software development. There is no one well-accepted definition of BDD, and a widely accepted understanding is far from clear and unanimous. The supporting tools for BDD are mainly focused on the implementation phase of the development process, which is a mismatch to BDD's broader coverage of the software development lifecycle [31]. But, BDD is the best of several worlds with the combination of ubiquitous language, TDD and automated acceptance testing. It tries to connect all of them together in a seamless fashion, optimizing the interactions and connections onto a single approach.

Keogh [16] embraces a broader view of BDD and argues its significance to the whole lifecycle of software development, especially to the business side and the interaction between business and software development. In addition, this author argues that BDD permits to deliver value by defining behavior, and it is focused on learning by encouraging questions, conversations, creative exploration, and feedback. BDD also aids to decouple the learning associated with TDD from the word "test", using the more natural vocabulary of examples and behavior to elicit requirements and create a shared understanding of the domain. Even though the study does not provide a comprehensive list of the BDD characteristics, it demonstrates convincingly that BDD has broader implication to software development processes than being just an extension of TDD [31].

Lazar et al. [20] highlight the value of BDD in the business domain, claiming that BDD enables developers and domain experts to speak the same language, and that BDD encourages collaboration between all project participants.

2.4 Natural Language Processing

Natural Language Processing (NLP) is the branch of computer science focused on developing systems that allow computers to communicate with people using everyday language. It's aka Computational Linguistics, because we need computational methods to aid the understanding of human languages. In this section we will briefly discuss some of the basic NLP techniques that form the basis for understanding the underlying concepts used in the thesis.

POS Tagging

Part-Of-Speech (POS) tagging is the process of marking each word in a corpus/text with it's corresponding part-of-speech. This tagging needs to be based on, both the definition of the word and the context in which it is used. The context is determined by adjacent words, related words, and it's usage in a phrase or sentence. This process, involves word-sense disambiguation based on the context, because ambiguity is ubiquitous in natural language.

The Penn Treebank project [22] provides Table 2.1, where all the available POS tags are listed for English text and their tag/abbreviation is specified.

An example of POS tagging would be:

Sentence: *"John saw the saw and decided to take it to the table."*

Tagged words: John/NNP saw/VBD the/DT saw/NN and/CC decided/VBD to/TO take/VB it/PRP to/TO the/DT table/NN ./.

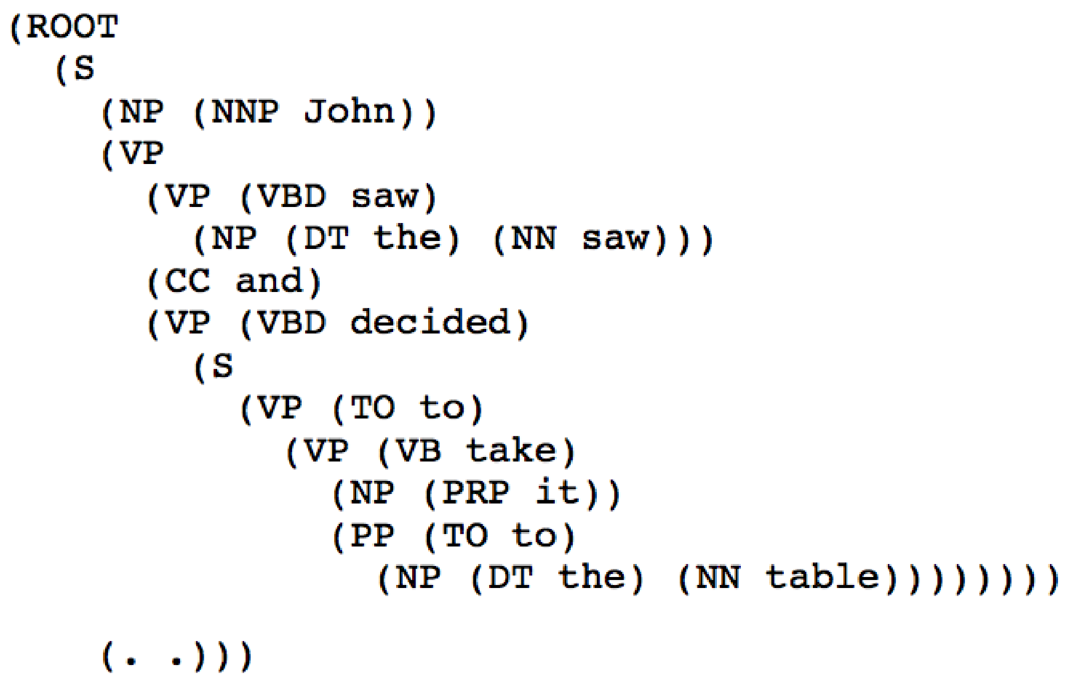
Similar to POS tagging, there is also value in understanding the collections of related words in a sentence, which may represent a phrase, where the words of the phrase are generally co-located. Table 2.2 provides the list of phrase tags and their meaning as provided by the Penn Treebank project for English [22].

Tag	**Description**
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
RB	Adverb
SYM	Symbol
TO	to
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present

Table 2.1: Penn treebank POS tag abbreviations with description.

Tag	**Description**
NP	Noun Phrase
VP	Verb Phrase
PP	Pronoun Phrase

Table 2.2: Penn treebank phrase tag abbreviations with description.



```
(ROOT
  (S
    (NP (NNP John))
    (VP
      (VP (VBD saw)
        (NP (DT the) (NN saw)))
      (CC and)
      (VP (VBD decided)
        (S
          (VP (TO to)
            (VP (VB take)
              (NP (PRP it))
              (PP (TO to)
                (NP (DT the) (NN table))))))))

    (. .)))
```

Figure 2.2: Sample textual phrase structure tree representation.

Treebank

A treebank is a parsed corpus in which every sentence that is part of the corpus is parsed and annotated with syntactic information. The most common way of representing this syntactic information is in the form of a tree, hence the name treebank.

These treebanks can be created manually, by annotating each sentence within the corpus, but various parsers exist which can follow particular grammar rules to automate the creation of these parse tree structures. These parsers can be distinguished based on the type of annotation that they perform, as either Phrase Structure or Dependency Structure.

Phrase Structure Treebank are parsers that annotate the phrase structure of a sentence. The well-known phrase structure treebank algorithms are Penn Treebank [22] and International Corpus of English – Great Britain (ICE-GB) [9] for the English language.

Example of Phrase structure annotation:

Sentence: *"John saw the saw and decided to take it to the table."*

Tree structure output from Penn Treebank parser shows the hierarchical grouping of elements of the sentence, with a combination of the phrase and POS tags, which is shown in Figure 2.2.

Dependency Structure Treebank are types of parsers that annotate the dependencies that

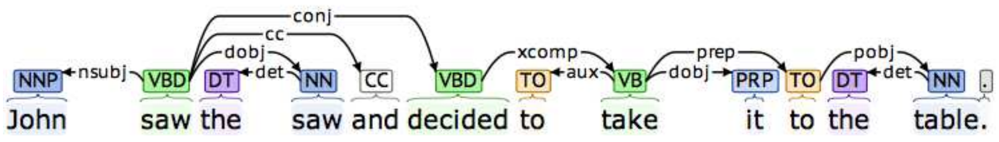

conj
cc
dobj
det
nsubj
xcomp
aux
prep
dobj
pobj
det
NNP VBD DT NN CC VBD TO VB PRP TO DT NN .
John saw the saw and decided to take it to the table.

Figure 2.3: Sample dependency structure tree representation.

exist between words of a corpus. The dependency structure views the verb as the structural center of the clause structure and other words to be directly or indirectly dependent on them.

There is an evolving research in the field of combining the properties of dependency and phrasal annotations and also in the conversion from dependency form to phrase annotation [35].

Example of Dependency annotation:

Sentence: *"John saw the saw and decided to take it to the table."*

Similar to POS tags, there exists various dependency tags, which indicates the type of dependency that exists between the words of a corpus. In Figure 2.3, you see a number of typed dependencies that exist between the words of the sentence. The second word *"saw"* is associated with the first word *"John"* in a nominal subject (nsubj) dependency. Marie et. al [7] documents a list of all such dependencies in various StanfordNLP [7] parsers.

WordNet

WordNet [24] is a large lexical database of English that is developed by Princeton University. It groups nouns, verbs, adjectives and adverbs into sets of cognitive synonyms, known as synsets. Based on context and the sense of the word, different meanings of the word are classified. It contains a total of 90,000 different word senses and more than 166,000 pairs that connect different word senses with a semantic meaning [30].

WordNet defines various relationships like hypernym, hyponym, troponym that exists between semantic words and characterizes them as is-a-kind-of and is-a-part-of hierarchies. Each word in a synset is also assigned a textual description that provides the precise meaning of the word in that sense. It also provides a frequency count, which helps in ascertaining the usage of the word in common practice.

Measuring Similarity

Similarity is a complex concept that has been widely discussed in the linguistic, philosophical, and information theory communities [12]. Given two input text segments, we want to automatically determine a score that indicates their similarity at a semantic level, thus going beyond the simple lexical matching methods traditionally used for this task. An effective method to compute the similarity between short texts or sentences has many applications in natural language processing and related areas [13].

Approaches to computing similarity on short text segments can roughly be classified into four major categories: word co-occurrence/vector-based document model methods, corpus-based methods, hybrid methods, and descriptive feature-based methods [13].

Lemmatization

Lemma or Morphology is the canonical/dictionary form of a set of words. And lemmatization is the algorithmic process of determining the lemma of a given word, which can be inferred by grouping together the different inflected forms of a word so that they can be analyzed as a single term. This process of identification of the lemma is no simple task; we need to understand the context, part-of-speech of the word, and have a dictionary setup which can provide the lemma for a grouped set of words.

Lemmatization and Stemming are morphological analysis techniques. Both of them are similar in goals, but, the main difference being that stemming is a completely algorithmic process and does not use a standard dictionary to group multiple inflected forms onto a common word. It tries to get the stem of the word, which is a part of the word.

Example:

Lemmatization reduces various inflected forms of car. *("Car", "Cars", "Car's") to "car"*

2.5 Automatic Code Generation

Automatic code generation will have a profound impact on software projects for developing a complex and evolving management information system. As a result, the development costs will be reduced, the development cycle will be shortened, and the software quality will be promoted [21].

Generative programming is a software engineering methodology that automates the generation of system implementations from higher level abstractions represented as textual or graphical models. Meta-modeling techniques provide great advantages for modeling domain-specific software systems at higher abstraction level [15]. Code generation and model transformation are the general approaches adopted to implement automatic synthesis facilities for systems. Various other software tools are capable of generating source code based on user input in the form of UML or XML which describes the software system.

Modern IDE's such as Eclipse, NetBeans, Microsoft Visual Studio allow for automatic source code generation based upon programmer interactivity. Programmers can interactively customize snippets of code. A good example would be in user interface development, where the programmer/designer can drag-and-drop UI components, and the required markup or source-code is automatically generated.

Other techniques for automatically generating code based on natural language processing have been described [36]. They use a parse tree to identify the functional keyword that corresponds to a program instruction. Further, the program instruction along with a regular expression pattern matcher is used to generate a concept map from which source code can be generated using a code output rule. But this patent is very generic in its approach, and we have not see an implementation of a system using this technique.

2.6 Automatic Code Generation for BDD

In a closely related work, Soeken et al [30] propose an assisted flow for BDD, where the user enters into a dialog with the computer. The computer suggests code fragments extracted from the sentences in a scenario, and the user confirms or corrects each step. This allows for a semi-automatic transformation from BDD scenarios as acceptance tests into source code stubs. Thus, providing the first step towards automating BDD scenarios.

They show some excellent usage of NLP techniques to interpret BDD scenarios, and their tool is built over Cucumber. Though Kirby borrows certain ideas from their technique, the scale at which we try to automate, and overall goals of the two projects are very different.

2.7 Summary

Based on existing research, BDD promises to have a widespread impact. But, BDD techniques need to evolve and concentrate on all phases of Software Development Life Cycle (SDLC). Automatic code generation has been limited to specific problem types, where these generators require a well-defined input format. Unless one follows a known grammar with unambiguous, clearly defined instructions, it is difficult to auto-generate program code from natural language on the fly. Though there is potential research carried on for semi-automatic code generation for BDD, going from there to full automation without programmer intervention is a whole new gamut. As far as we know, converting BDD scenarios onto programmatic code in a completely automated fashion, is a "*first of its kind*" work.

Chapter 3

Kirby's User Interface

Based on the previous work surveyed in Chapter 2, it is clear that automatically generating tests from natural language descriptions of software behavior has not been attempted in a fully automated fashion. This chapter provides an introduction to our application, *Kirby*, which attempts to address this problem. Section 3.1 discusses code generation strategies from the perspective of the end user who is going to use Kirby as a software tool with BDD. Section 3.2 provides an overview of the User Interface that we developed to assist developers in using Kirby.

3.1 Assisted BDD Workflow

In this section we will walk through a *"Banking Application"* example that is going to be developed using Behavior-Driven Development principles with Kirby.

Actors

There are two actors as part of the banking application that needs to be developed.

Mr. Gates, the customer: is interested in developing a "Banking application" to serve his Artista Corporation. He first needs to showcase a proof-of-concept (POC) of the application. This POC needs to have the bare minimum account functionality with deposit and withdraw operations supported.

John, the developer: is an experienced consulting developer who is new to Behavior-Driven Development, but is eager to learn about it.

Requirements and Specification

Mr. Gates has a clear idea of what needs to be developed for the proof-of-concept banking application. He and John sit together and sketch out the requirements. Once they come to a common understanding, they write out BDD features as a formal acceptance criteria for the application. The scenarios described below are only a subset of the real-world scenarios that can be sketched out for the application.

```
//BankAccount.story

Narrative:
As an account holder
I want to deposit and withdraw money from my bank account
In order to perform bank transactions

Scenario: deposit money to empty account
Given a bank account with initial balance of 0
When we deposit an amount of 100 into the account
Then the balance of the account should be 100

Scenario: withdraw money from a bank account
Given a bank account with initial balance of 1000
When we withdraw "100" dollars from the account
Then the balance of the account should be 900

Scenario: deposit and withdraw money from a bank account
Given an account with balance = 100
When an amount of 20 is deposited into the bank account
And we remove "40" from the bank account
Then the balance of the account should be 80
```

Interpretation

Gherkin as a language is very expressive and intuitive for describing scenarios, and any natural language phrasing can be used in each clause of a scenario. The general strategy used by Kirby is to translate each scenario into a single test. The "Given" clause(s) specify the object creation actions or other setup actions needed at the beginning of the test. The "When" clause(s) represent method calls on objects involved in the test, while "Then" clause(s) represent assertions that validate expected outcomes. Clauses can be interleaved as needed. Understanding this basic interpretation of clauses may help stakeholders write effective scenarios that can be translated successfully by Kirby.

The wordings and the organization of the clauses in the three scenarios for the banking application shows flexibility in expressing the same information. This flexibility is a important and inherent feature of describing specification in natural language.

Development cycle

John has work to do, once these user stories are ready. He needs to follow the Red-Green-Refactor cycle of TDD. So, he will first use Kirby to generate the glue code, which will be created in Java language as JUnit tests.

So, John creates a Java project with a folder called "BDD Scenarios". This folder contains the feature/story files for the project. When we run Kirby on this input set, it still generates the step definitions in a file called BankAccountSteps.java which is going to be located in the src directory. This file contains a class called BankAccountSteps which has three methods - one for each of the scenarios in the story file. The method names are a camel-cased textual representation of the scenario name.

You have to remember that no implementation is still written. The generated step definitions/methods do provide feedback to John, that the implementation may not yet exist. The current output of Kirby is shown below. When he runs the generated file as a JUnit test, it will fail with the error mentioning that the implementation class may not exist.

```
//BankAccountSteps.java

package org.vt.cs.bdd.steps;
import org.junit.Test;

/**
* Auto-Generated test class created by Kirby
*/
public class BankAccountSteps {

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositMoneyToEmptyAccount() throws Exception {

    org.junit.Assert.fail("No matching Class found for clause - [Given a bank
        account with initial balance of 0]. This could be because the
        implementation class does not exist");
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testWithdrawMoneyFromABankAccount() throws Exception {

    org.junit.Assert.fail("No matching Class found for clause - [Given a bank
        account with initial balance of 1000]. This could be because the
        implementation class does not exist");
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositAndWithdrawMoneyFromABankAccount() throws Exception {

    org.junit.Assert.fail("No matching Class found for clause - [Given an
        account with balance = 100]. This could be because the implementation
        class does not exist");
  }
}
```

Now that we have a failing behavior, we can go ahead with the implementation of the class called BankAccount. Initially, we may only create the base class with the required constructor, in the Java project that is created.

```
package org.cs.vt.bdd.examples;

public class BankAccount{

  private double balance;

  public BankAccount(double balance) {
    this.balance = balance;
  }
}
```

Now, if we run Kirby on the story file again, it should recognize that a class called BankAccount has been created. Once it recognizes the class it should create code corresponding to the "Given" clause. The intricate details about how Kirby makes the choice about these classes, is explained in Section 4.2.

```java
//BankAccountSteps.java

package org.vt.cs.bdd.steps;
import org.junit.Test;

/**
* Auto-Generated test class created by Kirby
*/
public class BankAccountSteps {

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositMoneyToEmptyAccount() throws Exception {

    BankAccount bankAccount = new BankAccount(0.0D);
    org.junit.Assert.fail("No matching method found for clause [When we deposit
         an amount of 100 into the account]. This could be because the matching
         method is not present.");
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testWithdrawMoneyFromABankAccount() throws Exception {

    BankAccount bankAccount = new BankAccount(1000.0D);
    org.junit.Assert.fail("No matching method found for clause [When we
        withdraw 100 dollars from the account]. This could be because the
        matching method is not present.");
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositAndWithdrawMoneyFromABankAccount() throws Exception {
    BankAccount bankAccount = new BankAccount(100.0D);
    org.junit.Assert.fail("No matching method found for clause [When an amount
        of 20 is deposited into the bank account ]. This could be because the
        matching method is not present.");
    org.junit.Assert.fail("No matching method found for clause [And we remove
        40 from the bank account]. This could be because the matching method is
         not present.");
  }
}
```

In the code that is generated by Kirby, you see that the BankAccount object is created with the correct constructor being called. The naming convention for the objects that are created use the camel-case name of the class. Now, if you look at the messages that have been generated, for the "when" and "then" clauses, it is clear that the methods that are required for deposit and withdraw are not present in the BankAccount class.

In TDD style, we will add each method, then run the behavior, and further implement the actual code. Once all those, steps have been completed, the implementation for the BankAccount would look like the code shown below. Any sort of error checking is not done on the code shown here, because there are no behaviors that capture that information. If we need error checking, we will have to add the behavior and follow the TDD steps for its implementation.

```java
package org.cs.vt.bdd.examples;

public class BankAccount{

  private double balance;

  public BankAccount(double balance) {
    this.balance = balance;
  }

  public double getBalance() {
    return balance;
  }
  public void setBalance(double balance) {
    this.balance = balance;
  }

  public void deposit(double amount) {
    balance = balance + amount;
  }
  public void withdraw(double amount) {
    balance = balance - amount;
  }
}
```

With this implementation, when we use Kirby to auto-generate the glue-code, we will end up with the correct representation of the behavior.

```
//BankAccountSteps.java

package org.vt.cs.bdd.steps;

import org.junit.Assert;
import org.junit.Test;

/**
* Auto-Generated test class created by Kirby
*/
public class BankAccountSteps {

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositMoneyToEmptyAccount() throws Exception {

    BankAccount bankAccount = new BankAccount(0.0D);
    bankAccount.deposit(100.0D);
    Assert.assertEquals(bankAccount.getBalance(), 100.0D, 0.0D);
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testWithdrawMoneyFromABankAccount() throws Exception {

    BankAccount bankAccount = new BankAccount(1000.0D);
    bankAccount.withdraw(100.0D);
    Assert.assertEquals(bankAccount.getBalance(), 900.0D, 0.0D);
  }

  /**
  * Test method generated by Kirby
  */
  @Test
  public void testDepositAndWithdrawMoneyFromABankAccount() throws Exception {
    BankAccount bankAccount = new BankAccount(100.0D);
    bankAccount.deposit(20.0D);
    bankAccount.withdraw(40.0D);
    Assert.assertEquals(bankAccount.getBalance(), 80.0D, 0.0D);
  }
}
```

These step-definitions which are auto-generated by Kirby represent the required behavior. With these auto-generated step definitions, John can actually run them as JUnit tests on the implementation and verify their behavior.

It is also interesting to note that we do not need concrete implementations for Kirby, but a code template or skeleton is sufficient to auto-generate the required step definitions.

3.2 User Interface

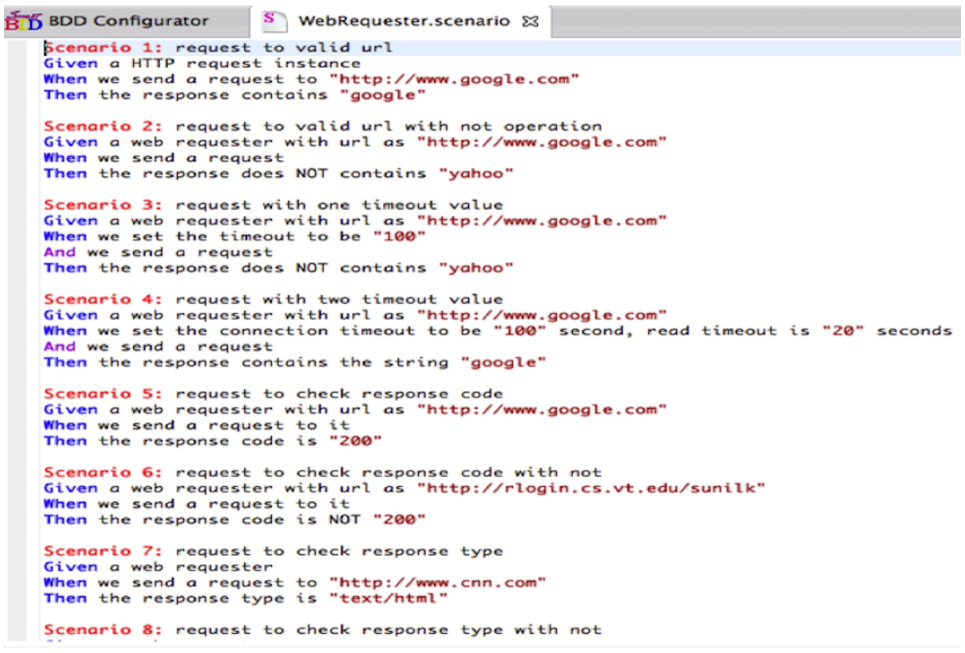
In order to assist developers using Kirby, we created a simple User Interface (UI). The UI was developed as a plug-in to Eclipse, which is a popular IDE. It helps in:

- Composing stories with the help of syntax highlighting for gherkin based scenario descriptions.
- Auto-generating step definitions, which can be viewed and validated by the developer.
- Running auto-generated step definitions.
- Viewing the results of the run, in a tabular form which indicates the success or failure of each scenario.

User stories can be written in any general text editor, and it should follow the convention of the Gherkin language. But, in order to assist developers and customers to work together with a unified editor, we created a editor within the Kirby UI which is capable of syntax highlighting various components of gherkin language. You can see in Figure 3.1, that a particular set of scenarios are input into the editor, and keywords like Scenario, Given, When, Then, And are highlighted making it easier for the user to interpret this information. You will also observe that the quoted text is again highlighted, which generally indicates variable information in the scenario, which act as parameters.

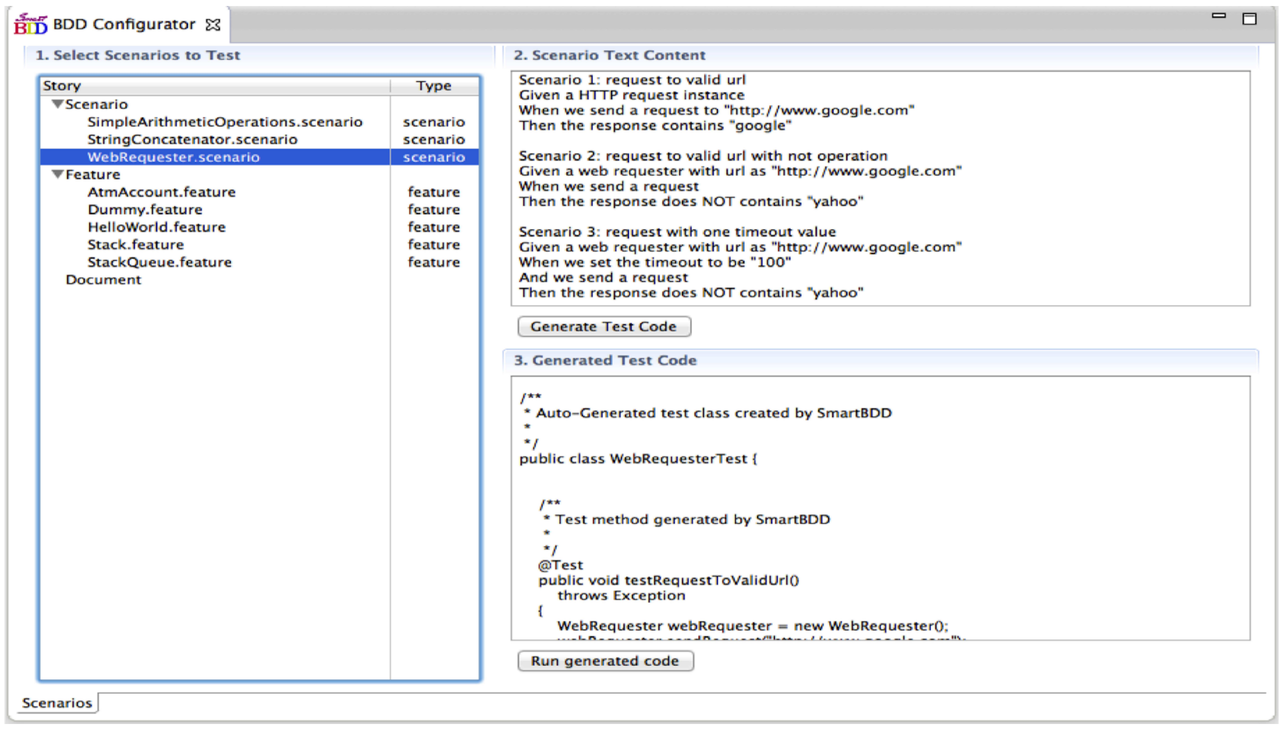
Now that the stories have been created, the developer would want to be able to auto-generate the programmatic code, representing the scenario. In order to do this, he would use the main view of Kirby as shown in Figure 3.2. This view provides a convenient way of viewing the text in a story that the user has selected from the project. For a selected user story, the developer can click on a button, to auto-generate the corresponding step-definitions. He can view the auto-generated step definitions in the editor which is to the bottom right of the main view, and validate it for correctness.

Once the validation is done by the developer, he can click on a button to run the behavior as a JUnit test. Once the run is complete, the user is provided with a results page which indicates the status of the run as shown in Figure 3.3. A tabular form of showing the results is followed, and for the selected behavior, a green color result indicates a success and a red color indicates a failure.



```
Scenario 1: request to valid url
Given a HTTP request instance
When we send a request to "http://www.google.com"
Then the response contains "google"

Scenario 2: request to valid url with not operation
Given a web requester with url as "http://www.google.com"
When we send a request
Then the response does NOT contains "yahoo"

Scenario 3: request with one timeout value
Given a web requester with url as "http://www.google.com"
When we set the timeout to be "100"
And we send a request
Then the response does NOT contains "yahoo"

Scenario 4: request with two timeout value
Given a web requester with url as "http://www.google.com"
When we set the connection timeout to be "100" second, read timeout is "20" seconds
And we send a request
Then the response contains the string "google"

Scenario 5: request to check response code
Given a web requester with url as "http://www.google.com"
When we send a request to it
Then the response code is "200"

Scenario 6: request to check response code with not
Given a web requester with url as "http://rlogin.cs.vt.edu/sunilk"
When we send a request to it
Then the response code is NOT "200"

Scenario 7: request to check response type
Given a web requester
When we send a request to "http://www.cnn.com"
Then the response type is "text/html"

Scenario 8: request to check response type with not
```

Figure 3.1: Kirby UI - scenario text editor.

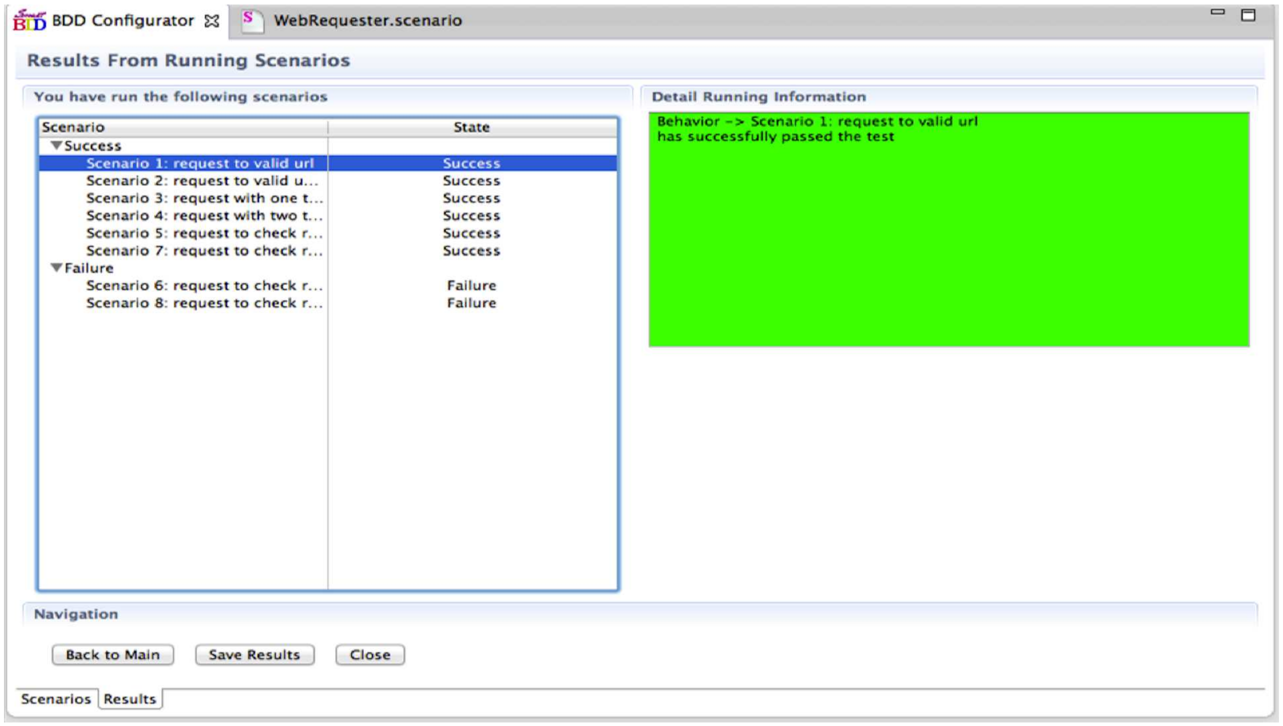


Story	Type
▼Scenario	
SimpleArithmeticOperations.scenario	scenario
StringConcatenator.scenario	scenario
WebRequester.scenario	scenario
▼Feature	
AtmAccount.feature	feature
Dummy.feature	feature
HelloWorld.feature	feature
Stack.feature	feature
StackQueue.feature	feature
Document	

```
Scenario 1: request to valid url
Given a HTTP request instance
When we send a request to "http://www.google.com"
Then the response contains "google"

Scenario 2: request to valid url with not operation
Given a web requester with url as "http://www.google.com"
When we send a request
Then the response does NOT contains "yahoo"

Scenario 3: request with one timeout value
Given a web requester with url as "http://www.google.com"
When we set the timeout to be "100"
And we send a request
Then the response does NOT contains "yahoo"
```

```
/**
 * Auto-Generated test class created by SmartBDD
 *
 */
public class WebRequesterTest {


    /**
     * Test method generated by SmartBDD
     *
     */
    @Test
    public void testRequestToValidUrl()
        throws Exception
    {
        WebRequester webRequester = new WebRequester();
```

Figure 3.2: Kirby UI - main view.


BDD Configurator
WebRequester.scenario
Results From Running Scenarios
You have run the following scenarios
Detail Running Information
Navigation
Back to Main
Save Results
Close
Scenarios
Results

Scenario	State
▼Success	
Scenario 1: request to valid url	Success
Scenario 2: request to valid u...	Success
Scenario 3: request with one t...	Success
Scenario 4: request with two t...	Success
Scenario 5: request to check r...	Success
Scenario 7: request to check r...	Success
▼Failure	
Scenario 6: request to check r...	Failure
Scenario 8: request to check r...	Failure

Behavior -> Scenario 1: request to valid url
has successfully passed the test

Figure 3.3: Kirby UI - results view.

Chapter 4

Kirby's Design and Implementation

With the design of Kirby, we exactly knew what the system requirements were. But, there were various questions and concerns that we had to address, as we went about designing and building the system.

4.1 High-level Design

This section provides a high-level overview and design of Kirby. We will briefly discuss the workflow with Kirby and it's component architecture.

4.1.1 Workflow

Kirby tries to completely automate the generation of the individual steps directly from the natural language scenario descriptions. Kirby is named after *Kirby cucumbers*, a variety of cucumbers that are both short and bumpy in appearance. Kirby shortens the process of executing BDD scenarios by eliminating the manual task of writing step definitions.

The workflow we envision for BDD with Kirby is illustrated in Figure 4.1. Developers alternate between creating or revising scenarios and writing (or creating stubs) implementation code. Since the implementation code is written with the scenario in mind, we believe the language that is used in the code will naturally reflect the language of the scenario (with subtle variations). At any time, the scenarios can be executed directly on the implementation code by using Kirby.

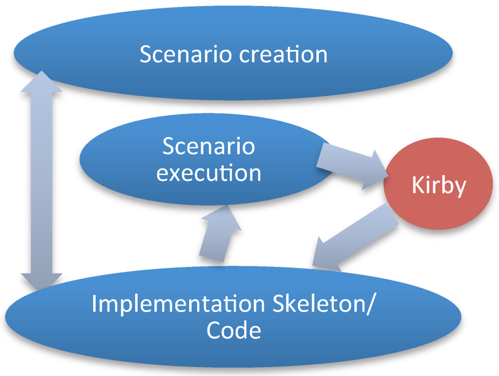


Figure 4.1: System workflow with Kirby.

4.1.2 Architecture

Since the goal of Kirby is to relieve the developer from writing step definitions manually, we need to develop a mechanism to map the natural language scenario descriptions onto code implementation/skeleton that is being written alongside the scenarios.

Kirby uses both the scenario descriptions and the co-developed software (complete code or stubs) as input when generating executable tests. It uses a *Natural Language Augmentation Engine* to process and augment the information in each clause of the scenario to understand its structure and semantics. At the same time, Kirby also uses a reflection-based *Class Information Extractor* to obtain details about the classes and methods that have been written in the implementation. The *Probabilistic Matcher* uses a variety of algorithms to determine the best matches between noun phrases and verb phrases in the behavioral description, and objects and methods available in the implementation. Once suitable matches have been found, the *Code Generator* synthesizes this information to produce JUnit-style tests as shown in Figure 4.2.

4.2 Implementation

This section provides details about the implementation of Kirby, the technologies used and low-level information about each of the components in the architecture.

We choose the language Java, because it is one of the most popular programming languages in use today. It is also strictly object-oriented and typed-language. Because of this nature, the challenges we face in Java are a lot harder than what we might have faced in a scripting language like Ruby or Python. Once we address these challenges, it would be relatively easy to port Kirby to other languages.

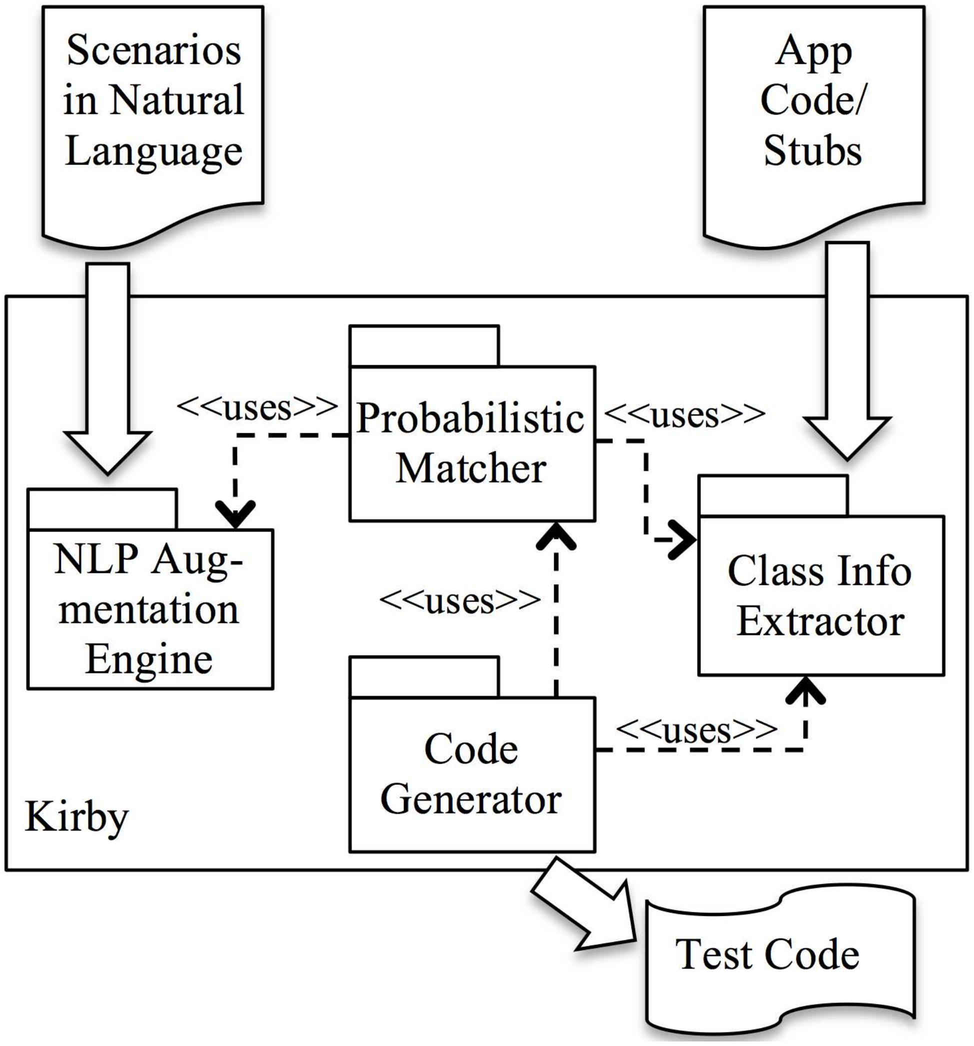


Figure 4.2: High-level architecture of Kirby.

4.2.1 Code Information Extractor

The Code Information Extractor is responsible for extracting information from the implementation code. This component is an important part of the architecture, since it makes available the entire information about the required classes for automating step definitions. The code information extractor does not need concrete implementations of classes, but a template or skeleton with the correct signature is sufficient.

Since we use Java, which follows a strict object oriented structure, we keep track of the public methods and public members that are part of each class. The exact information that we keep track-of for individual classes are:

- *Class name*
- *Public constructors*
- *Public member variables*
- *Public static variables*
- *Public methods*
- *Public static methods*
- *Parameter names in methods and constructors*

Kirby uses a streamlined Java reflection API [10] to keep track of the code information. The interface that we use for exposing each class details is provided below.

```java
/**
* This interface represents the the details that we
* want to pull out of a given Class implementation.
*/
public interface IClassDetails {

 public String getClassName();

 public List<Constructor<?>> getPublicConstructors();

 public List<Method> getPublicMethods();

 public List<Field> getPublicMembers();

 public void setOperatingClass(String baseDirectory, String pathOfClass)
     throws ClassNotFoundException;

 public Class<?> getOperatingClass();
}
```

Search space

When users talk about creating classes in the "Given" clause of the natural language description of the behavior, it is possible that they want to create objects that are not directly created in the project.

Example:

```
Given a URL with value "http://google.com", called google
And a web requester with url equal to google
```

In the example provided, we need to provide a behavior for a Web Requester which sends a HTTP request to a given URL. There are various libraries in Java, that provide a URL class, one that the Java library provides is *java.net.URL*. And the WebRequester class needs to be designed to make use of the capabilities of the URL class. But, if we only keep information about the classes in the project, we will not be able to create a URL class which is in a different package.

To avoid this scenario, Kirby uses a hierarchical tree to keep track of classes. This hierarchy is determined by not only looking at the classes in a project, but also the classes that are accessible from it. We determine the classes that are accessible, based on parameters of public constructors and methods in the class. The premise is that, if we need to create a class, and its method or constructor has a complex type, then there is a likelihood that those classes that represent the complex type might need to be created. In the above example, if the WebRequester class has a constructor which accepts a type java.net.URL, then the URL class is also part of our search space.

Parameter Names

Native Java reflection mechanisms do not provide the name of the parameters in constructors or methods. But, this information is important to Kirby, because parameter names can be used to understand which word(s) represent parameter values. So, in-order to obtain this information, we use a library called Paranamer (http://paranamer.codehaus.org/) that allows the parameter names of non-private methods and constructors to be accessed at runtime. But, this library requires the Java byte-code for the application to include debug information, which is typical in development environments.

Using the Code Information Extractor API

There is a Facade design pattern which is used to expose all the class details in a project, while hiding the inner implementation and object creation structure. As a developer, using the Code Information extractor to obtain class information is simple.

```java
String projectPath = "/data/bdd-project";

//Create the facade class for getting Java class details
JavaClassDetailsFacade facade = new JavaClassDetailsFacade();

//Call the generateClassInfo method to get a map of Classname VS ClassInfo
//which contains all the required information about the class
fileVsClassInfo = facade.generateClassInfo(projectPath);
```

4.2.2 NLP Augmentation Engine

The Natural Language Processing (NLP) Augmentation engine parses the natural language scenario descriptions, processes and analyses them, and augments it with various information such as lemma, hierarchy and dependency. This augmented information is required for interpreting the meaning and content of each of the clauses in the scenario descriptions. Various sub-components of the NLP Augmentation engine are described below.

Parser and Transformer

As the name suggests, the parser is responsible for parsing the information provided in the scenario description. It uses the power of *Regular Expressions*, which is provided by the Java language in the *java.util.regex* package. The input to the parser is a story/feature/scenario file. The input file is parsed, to understand what is the narrative or meaning of the story. Then, it is capable of understanding all the scenarios that are described in the story/feature. For each scenario, it understands what are the clauses that it contains and the type of clause that it represents in the Gherkin language.

The transformer uses all the information provided by the parser and transforms each data set into an *Object-Oriented (OO)* structure that can be further used to represent the natural language description. We have Story, Scenario, Clause, Narrative as OO classes, which is the basic output of the Gherkin transformer.

Augmenter

As described, the Augmenter is responsible for adding supplemental information which helps in extending the interpretation of scenario descriptions. For doing this we need a NLP library which can process English language. We choose the widely used StanfordNLP library (http://nlp.stanford.edu/) [7] which is provided by the Natural Language Processing Group at Stanford University. This library provides every information that we need to process the natural language. A discussion of the type of augmentation for a scenario is shown below.

Given a scenario for a trader trading stocks:

```
Scenario: check stock threshold
Given a stock of symbol GOOGLE and a threshold of 15.0
When the stock is traded at price 5.0
Then the alert status is OFF
```

The Augmenter will augment each clause by creating a parse tree. This parse tree provides information such as Part-Of-Speech (POS) tag, Phrase Tag and the Phrase Structure treebank as described in section 2.4.

For the "Given" clause in the scenario description of the stock, the parse tree would look like:

```
(ROOT
  (PP (VBN Given)
    (NP
      (NP
        (NP (DT a) (NN stock))
        (PP (IN of)
          (NP (NN symbol) (NN GOOGLE))))
      (CC and)
      (NP
        (NP (DT a) (NN threshold))
            (PP (IN of)
              (NP (CD 15.0)))))))
```

Here you see that each word of the sentence is the leaf of the tree and is tagged with a part-of-speech. There is a grouping of words to form a phrase tag.

Other than the parse tree information, we also augment it with the dependency treebank as described in section 2.4. We use the PennTreebankLanguagePack class from the StanfordNLP library for typed dependencies. Internally we factor in collapsed dependencies, which can take the list of dependencies and collapse them together to provide a smaller subset of important dependencies. Below is a listing of the collapsed dependencies for the same "Given" clause of the stock scenario.

```
root(ROOT-0, Given-1)
det(stock-3, a-2)
dep(Given-1, stock-3)
nn(GOOGLE-6, symbol-5)
prep_of(stock-3, GOOGLE-6)
det(threshold-9, a-8)
dep(Given-1, threshold-9)
conj_and(stock-3, threshold-9)
prep_of(threshold-9, 15.0-11)
```

Other than providing these dependencies, the augmenter can also mark *stop-words* in the clause. These are words in a sentence, that do not add any value to the sentence for the purpose of comparisons. They may include words like *"a, also, am, an, and"* etc. The augmenter marks these words, so that when we perform comparison of sentences, these words can be ignored to get better results.

NLP Augmenter also provides *lemmatization* capability by keeping track of the lemma of each word in the clause. The lemma of each word is used for comparison rather than the actual word itself, so that we do not use different inflected forms of the word.

Parameter Extractor

The Parameter Extractor is responsible for picking out the parameter(s) that are specified in a clause. There are two ways in which parameters can be specified in Kirby. One simple way is to provide the parameter information in quotes. The limitation with this approach is that the ordering of the parameters in the clause should be relative to the ordering of parameters in the method/constructor. Another way is to specify the parameter values in the clause to be associated with the parameter name in the method/constructor. We use typed dependency and parameter type to determine the parameter values, and this does not have the limitation of ordering which is present in the quoted parameter approach.

Considering that there is a constructor for the stock class:

```
public class Stock {
   public Stock(String symbol, double thresholdValue);
}
```

Quoting Parameters:

Given a stock "GOOGLE" with lower limit "15.0"; Parameters here are "GOOGLE" and "15.0" in that order.

Dynamic Parameter Extraction: Based on the parameter names in the constructor - "symbol" and "thresholdValue", parameter values can be extracted.

Given a stock of symbol GOOGLE and a threshold of 15.0; Parameters here are "GOOGLE" and "15.0" for the constructor.

Given a stock of threshold value 15.0 and symbol GOOGLE; Parameters here are "GOOGLE" and "15.0" for the constructor.

Also, a hybrid approach where certain parameters are specified in quotes, while others are specified as a relation to a word in the constructor parameter can be used. The same strategy also works for extracting parameters in methods.

NLP Wrapper

From an implementation perspective, we provided a well-rounded wrapper around the StanfordNLP library and added more capabilities that we need on top of it. The API for the

NLP wrapper is shown below.

```
/**
* This interface acts as the wrapper around the Stanford NLP library.
*/
public Interface NLPWrapper {

  public Tree obtainParseTree(String text);

  public void printParseTree(Tree parseTree);

  public List<TypedDependency> obtainTypedDependencies(String text);

  public Map<String, POSTag> getTaggedWords(Tree parseTree);

  public Map<String, POSTag> getTaggedWordsForPOSType(Tree parseTree, POSTag
      tag);

  public Map<String, POSTag> getTaggedWordsForPhraseType(Tree parseTree,
      PhraseTag phraseTag);

  public Map<String, POSTag> getStopWords(Tree parseTree);

  public String lemmatize(String sentenceStr);

  public List<String> getRelatedWords(Tree parseTree, String word);
}
```

4.2.3 Probabilistic Matcher

The Probabilistic Matcher is an interesting aspect of the architecture, since it is responsible for computing probability values of match between the clause in the behavioral description and the code implementation. Using natural language in scenarios provides a great deal of flexibility in the way we specify software behavior. But, matching this natural language with program features is challenging.

In order to understand what the code segment is intending to perform, we have to rely on good naming practices followed by developers. For the Java programming language, *camel-casing* class names and methods is a widely used and accepted software engineering practice. We use this convention to understand what the class or the method name represents and intends to do.

```
public class UnitConverter {
  public float convertToKilometers(float num);
  public float convertToMiles(float num);
}
```

Here you can see that the class is called UnitConverter, which is a combination of unit and converter. The method names are also conveying information in the way they are written - convertToKilometers() represents a string of "convert to kilometers". By interpreting the information in the camel-cased naming, we have come up with a strategy to understand the meaning of code segments.

The intention of the Probabilistic matcher is to compute probabilistic values for a "Given" clause onto each class identified in the search space. Similarly for the "When" and "Then" clauses, we need to compute probabilistic values for each method of the identified class.

Probabilistic Matcher Algorithms

There is no one-stop solution or algorithm that works perfectly in computing the probabilistic matching values. For this purpose, we had to improvise by using a combination of different types of algorithms.

- *Levenshtein:* The most basic algorithm that we use is the edit-distance algorithm like levenshtein [32]. This algorithm works favorably in a situation where the user specifies a partial or exact wording used in the code, but it will fail miserably when a synonym is used in natural language. When developers use a test-first approach to development, there is a strong likelihood that they will be influenced by the language used in the behavior, which would help the case for an edit-distance algorithm.

- *Cosine Similarity:* is a measure of similarity between two vectors that measures the cosine of the angle between them. In Information Retrieval, each term is notionally assigned a different dimension and a document is characterized by a vector where the value of each dimension corresponds to the number of times that term appears in the document. Cosine similarity then gives a useful measure of how similar two documents are likely to be in terms of their subject matter [5]. For multi-word or sentence matching, a vector space model like cosine similarity gives better values [33]. When we have relatively longer sentences, with various words in the sentence which do not have a direct mapping onto words in the code segments, cosine performs better than levenshtein.

- *Cosine WordNet Similarity:* Since natural language is very flexible in the way we describe scenarios, customers/developers may not use the exact same word notations across code and behavior. So, we need to be capable of interpreting synonyms and alternate word forms for matching. Both levenshtein and cosine similarity perform dismally in the presence of synonyms. To handle this, we extended the cosine similarity algorithm to include information from WordNet [24]. It uses the path measures between words in the sentence and WordNet library to compute vector based similarity measure.

- *DISCO:* DIstributionally related words using CO-occurrences [17] is a library that helps in retrieving the semantic similarity between arbitrary words. We use this library to compute sentence similarity measures not just for words, but also for short sentences. Second-order similarity measures between two words or sentences based on actual usage in large datasets like Wikipedia is captured by this model.

The Probabilistic Matcher uses weighted averaging to adapt its matching model based on the individual values obtained from the competing algorithms described above. If one algorithm, such as DISCO or edit-distance, does not provide any results in a given situation, the matcher modifies the weights of the probabilities (confidence levels) produced by the other algorithms.

Various matchers are used by Kirby, for matching "Given" clause with classes, "When" and "Then" clauses with methods. Further we also use the constructor and parameter matchers for all clauses. These matchers are discussed in more detail.

Class Matcher

The Class Matcher is responsible for picking out the correct class for a "Given" clause. The search-space for all the set of classes that we need to compare against is described in Section 4.2.1.

Based on convention followed in naming classes, it is a sound software engineering principle to have nouns or noun phrases as class names. So, the class matcher extracts all the nouns that are part of the noun phrases in the "Given" clause. Further, all stop-words are removed and each remaining word is represented by its lemma. Once we have this information from the NLP augmentation engine, we compare it against the camel-cased name of each of the class in the search space. The class with the highest probability is our most likely match.

For the "Given" clause: *Given a stock of symbol GOOGLE and a threshold of 15.0*

The different noun phrases are "a stock", "symbol GOOGLE", "a threshold"; We pick out the different noun forms from these phrases and run them against each of the class names. A sample output of running the class matcher in our case is shown in Table 4.1 which contains a subset of the classes in the project on which the class matcher was run.

From the result in Table 4.1, we see that there is a 96% probability that we are talking about the stock class, which is our class of interest.

For a "When" and "Then" clause, we would need to know, which is the class that these clauses are talking about. For understanding this information, we look for words that are used which represent a particular class that has already been created in the given clause. If one is found, we know that we need to perform operations upon that class. Upon a no match scenario, we try to look at the method that the clause is talking about. The class which contains the method with the highest probability is most likely our class of interest. Also note that the when and then clauses can only perform operations upon classes that are already created using the given statement.

Class-Name	**Probability(%)**
Cell	5.50
Trader	24.61
Event	4.13
SwingRenderer	17.70
ActionMap	11.73
Game	4.99
Color	6.04
GameObserver	15.49
Stock	**96.79**
String	11.82
SimpleCalculator	17.62

Table 4.1: Example class matcher probability statistics.

Constructor Matcher

The constructor matcher is responsible for picking out the correct constructor once we have a class of interest. It uses the constructor information provided by the code information extractor, and the parameter values from the Parameter Extractor to assign the highest probability to a particular constructor. The number of arguments to the constructor and the type of the arguments determine which constructor is chosen.

In the previous stock example, if the stock class had 3 constructors:

```
public class Stock {
   public Stock();
   public Stock(String symbol);
   public Stock(String symbol, double thresholdValue);
}
```

Then, we would pick the constructor which has the highest match as shown in the Table 4.2. This highest match is computed for the maximum number of parameter matches.

Constructor	**Probability(%)**
Stock()	0
Stock(String)	0
Stock(String, double)	**100**

Table 4.2: Example constructor matcher probability statistics.

Method Matcher

The Method Matcher is responsible for computing the probabilities of the "When" and "Then" clause against method names of a class. Sound software engineering principles talk about method names representing actions containing verbs. Similar to the Class Matcher, the Method Matcher extracts the verb forms that are part of the verb phrase. For each of the verb phrases, we compute the probability of a likely match against a clause.

For the "When" clause: *When the stock is traded at price 5.0*

The verb phrases are: "is traded";

Table 4.3 provides us with the probability match values for the methods in the Stock class. You see that the tradeAt() method has perfect probability of 100%.

Method-name	**Probability(%)**
setThreshold	0.0
getSymbol	18.17
tradeAt	**100.0**
getPrices	35.39
resetAlert	15.32
getStatus	18.36

Table 4.3: Example method matcher probability statistics for when clause.

There are times, when we deal with the "Then" clause, relying on just verb phrase matches may not be sufficient because of the "should be", "is" language that is used in describing the acceptance criteria. In these cases, we would need to expand our match to also include noun phrases.

For the "Then" clause: *Then the alert status is OFF*

The verb phrases are: "is OFF"; Noun phrases are: "the alert status"

Table 4.4 shows the probability values for the "Then" clause, you do see that the getStatus() method has the highest probability for the clause. If you look at resetAlert() method, it also has a high probability based on the mapping of the words in the sentence, but a slightly lower value.

Method-name	**Probability(%)**
setThreshold	0.0
getSymbol	15.65
tradeAt	0.0
getPrices	11.01
resetAlert	53.85
getStatus	**66.85**

Table 4.4: Example method matcher probability statistics for then clause.

4.2.4 Code Generator

Once phrases in the scenario have been matched with classes, methods and parameters, the code generator interacts with the other components to produce executable tests. Information from the probabilistic matcher is combined with code features retrieved by the code information extractor to generate the test code in Java using JUnit as our base unit-testing framework. Each of the clauses expressed in a scenario is treated differently. "Given" clauses map to one or more constructor calls. "When" clauses refer to a method call. "Then" clauses refer to one or more assertions from a code generation perspective.

The code is generated using a sophisticated on-the-fly approach based on the CodeModel library (http://codemodel.java.net/). With the help of this library we are able to generate java source files using a complete object-oriented approach. This approach is complicated, but flexible than a string based source-code generation technique.

The final outcome for the stock scenario described in section 4.2.1, will have java source code generated representing the equivalent programmatic code for the behavior.

```java
package org.vt.cs.bdd.steps;

import org.junit.Test;
import org.junit.Assert;

import org.cs.vt.bdd.examples.Stock;

/**
* Auto-Generated test class created by Kirby
*/
public class TraderSteps {
    /**
   * Test method generated by Kirby
   */
  @Test
  public void testCheckStockThreshold() throws Exception {
    Stock stock = new Stock("GOOGLE", 15.0D);
    stock.tradeAt(5.0D);
    Assert.assertEquals(stock.getStatus(), "OFF");
  }
}
```

The code generator needs to handle fine-grained low-level syntax constructs of the language. You would observe that the name of the class will be the name of the feature/story file appended with "steps", since it is representing the step definitions. Name of methods would be the name of the scenario in a camel-cased convention. The name of each of the objects created in the "Given" clause is its camel-cased name, unless the behavior provides a name to the object in the clause. The "When" and "Then" clause can operate upon any object already created in the "Given" clause. Each of the parameters that are identified here are converted to their respective types in the code representation - "GOOGLE" becomes a string, "15.0" becomes 15.0D as a double.

The "Then" clause represents an assertion of some sort. Based on the return-type of the method that is described in the clause, we decide the correct assert method to call. Boolean return values would perform a *assertTrue()* operation, while String values may perform a *assertEquals()* operation. A negative tone in the "Then" clause would cause an operation such as *assertFalse() or assertNotEquals().*

4.2.5 Error Handling

There are various instances during the process of converting behavior onto step-definitions where things can go wrong. To help developers who are going to use Kirby, we try to provide useful messages, so that they can know about the failure and rectify them. The code generator will generate a fail() method call in the test specifying the reason for the error/ambiguity. Some of the common errors are described in this section.

- *No matching Class found for clause - [clause text]. This could be because the implementation class does not exist:* If the implementation class does not exist, or if the class matcher is not able to find an appropriate class for the "Given" clause, then this error may occur. The developer can then go and write the implementation class, or change the clause to reflect upon the name of the class he intends to talk about.

- *No matching constructor found for clause - [clause text]. This could be because the implementation class does not have a matching constructor:* If we found a class of interest, but it does not have the correct constructor, then this message would appear. Once the class is verified, the developer can add a constructor with the right parameters.

- *A matching constructor has been found for clause - [clause text]. But the number of arguments are incorrect:* This message would appear if the clause tries to pass in incorrect number of arguments than that are required by the constructor. Developers can then fix the clause or the constructor as desired.

- *No matching method found for clause - [clause text]. This could be because the matching method is not present*: Similar to the class not found, if a class is identified but no matching method is found for the "Then" or "When" clause, we would see this error. The method can be added or the clause can be fixed to resolve this error.

- *A matching method has been found for clause - [clause text]. But the number of arguments are incorrect:* If we find a matching method, but the number of arguments specified in the clause are incorrect, then we would see this error. Either the clause or the method can be fixed to reflect the expected behavior.

- *An argument specified in the clause - [clause text], was not created earlier. Please check your scenario again:* Kirby allows complex type parameters to be passed to constructors/methods. An instance of an object needs to be passed as an argument, and if the developer forgot to create that instance in the behavior description, we would see this error.

- *A matching variable was not found for clause [clause text]. Please check your scenario again, and specify the variable:* Kirby allows users to specify the name of a variable that is created. If the behavior followed this convention to name a variable, then if the probabilistic matcher cannot determine what variable is being described, this error message would appear.

- *An exception prevented Kirby from generating code. Please contact the author with debug information:* If anything causes Kirby itself to create an exception which is not handled, this error would appear indicating a bug in Kirby.

Chapter 5

Evaluation

Although BDD is an emerging technique with a growing user community, it is difficult to find large numbers of publicly available scenarios written for tools like Cucumber and JBehave. However, at the same time, it is important to evaluate new techniques against real-world situations.

5.1 Dataset

We compiled a collection of 80 BDD scenario descriptions written in Gherkin, primarily from books on Behavior Driven Development [6][8][34], online tutorials published for use by developers learning to use other BDD tools, and scenarios that we created to showcase the advanced capabilities of Kirby.

As shown in Figure 5.1, more than 50% of the scenarios are directly picked up from the books as is, and 20% from online tutorials. Another 29% are hand-written, which is a combination of scenarios that are modified versions of the book scenarios and other complex scenarios which act as unit-tests for Kirby.

The only alteration that we performed on the natural language behavior description was to provide quotation for parameters in a subset of scenarios. This was required to handle tabular data format, and also to understand the variability characteristics of scenarios for parameter extraction.

The modification of scenarios is done, so as to get them to work accurately as we desire, which is explained below. Some of the scenarios were described at a very high-level, without much emphasis on how the behavior representation should be. These type of scenarios do not fit well into our model. Though they are not uncommon, we hope to give enough feedback in the cycle, so that developers can modify them to be converted accurately. An example of a scenario which is very less descriptive is shown below.

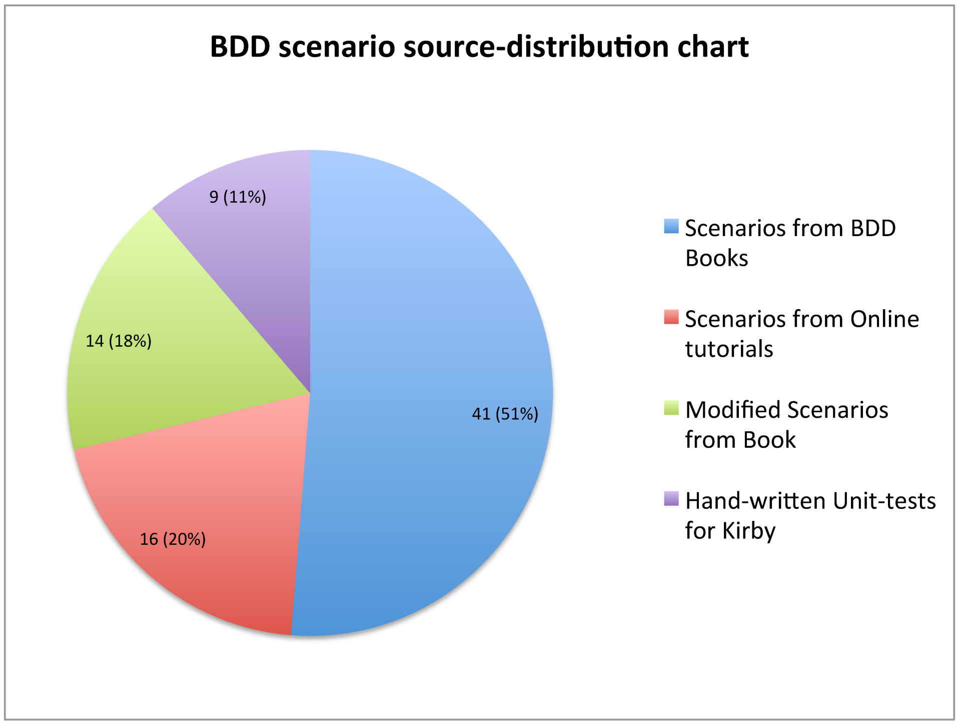


Figure 5.1: Distribution of BDD scenarios by source.

```
Scenario: Greet users who are logged in
Given I am logged in as "matt"
When I visit the homepage
Then I should see "Hello matt"
```

These type of scenarios were modified by describing the scenarios in more detail.

```
Given a web page "http://www.gmail.com"
When I login as user "Matt" with password "pass"
And I visit the homepage
Then I should be greeted with "Hello matt" in the title
```

There were other type of scenarios which did not follow the convention of having a “Given” clause to establish a context. Kirby does not handle such scenarios.

```
Scenario: Miles to kilometers
When I convert "26.2" miles to kilometers
Then the result should be "42.1648" kilometers
```

These type of scenarios were modified to fit into the Kirby natural language scenario convention.

```
Scenario: Miles to kilometers
Given a unit converter
When I convert "26.2" miles to kilometers
Then the result should be "42.1648" kilometers
```

The modifications made on the scenarios were very simple in nature to add subtle information, making the scenarios more descriptive.

Also, based on the scenario, we had to write skeleton/implementation code as a developer inorder to have a comparison point for mapping clauses onto code segments. Implementation code was written in such a way that a developer using BDD would, to write minimal code for getting the tests to pass. Also, to simulate a real-world project, we had the cumulative set of scenarios within the same project. This resulted in a project with about 40 different classes, which we believe is a reasonably sized search space for classes.

We then ran this collection through Kirby to assess its accuracy and performance, with the belief that these examples are representative of real-world practice.

5.2 Correctness

In order to measure the correctness of auto-generating step definitions, we built an evaluation engine into Kirby. This engine provides a textual interpretation for every clause encountered. This interpretation is compared with a hand-written expectation of the intended output for the clause. By performing these comparisons, we are able to determine the correctness and output characteristics for each clause in the scenario, and also for the whole scenario.

We can define certain terms for measuring correctness on the whole scenario:

- *Generated Successfully:* If every clause in the scenario has the expected step definition, then the scenario conversion is completely correct.
- *Generated partially without error(s):* If even one clause in the scenario is not converted into equivalent step definition, because Kirby could not determine it's class, method etc., then the scenario conversion has a partially generated state without any errors.
- *Generated with error(s):* If even one clause in the scenario has an incorrect step definition, then the scenario conversion has errors in them.

Based on these measures, we ran Kirby with a predefined configuration (which will be discussed later), and results are shown in Figure 5.2. For whole scenario conversions, we were able to accurately convert about 73% of the 80 behavior descriptions into step definitions correctly.

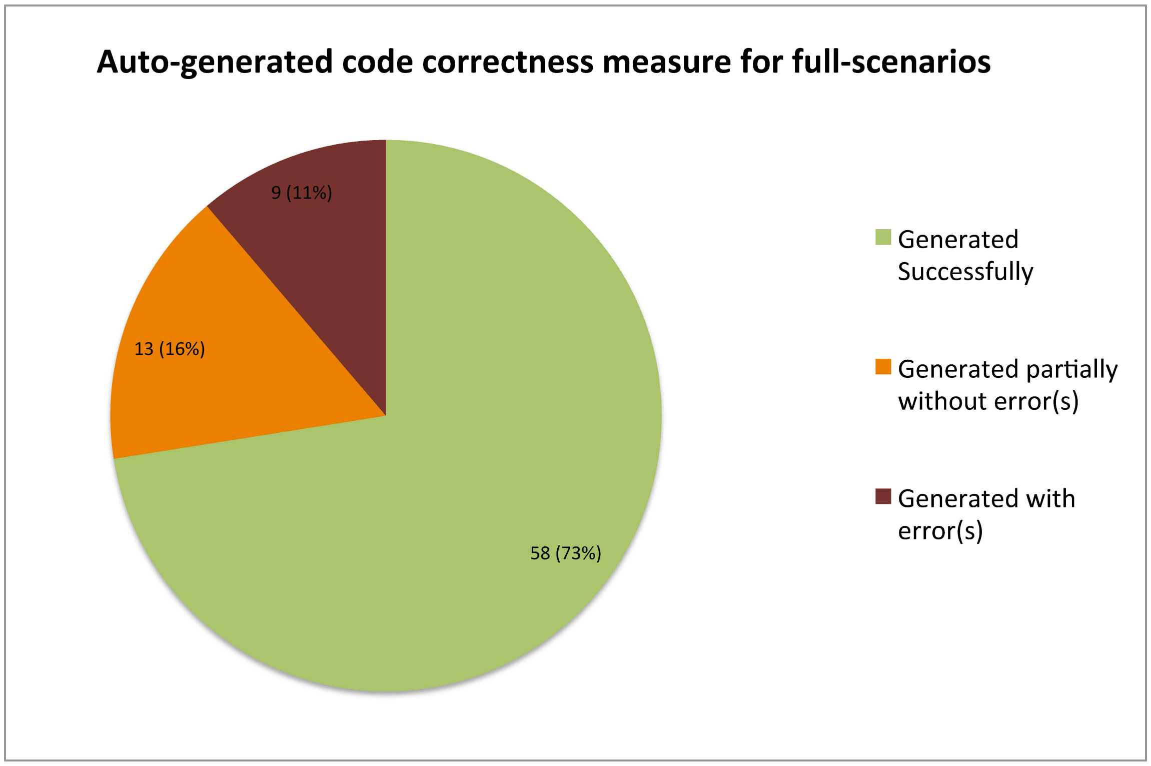


Figure 5.2: Kirby whole scenario conversion correctness pie-chart.

For about 16% of the scenarios, we were not able to determine any step definitions either because the probability values were very close to each other, or no match was found, or the confidence measure was too low. For these types of scenarios, the developer should be able to iteratively refine the implementation or the behavior description to achieve better correctness, based on the feedback provided by Kirby.

Because we took a large class set with about 40 classes, which had many classes seemingly related to each other like [Greeter, WebLoginGreeter, WebPage, WebRequester], [BankAccount, SavingsAccount, CheckingAccount], causing ambiguity in certain scenario descriptions that were not completely descriptive. Because of this, we see false positives about 11%. In real-world large projects, we expect this number to be slightly lower, and by being more descriptive in the scenario description, the modified scenarios were able to be converted completely into the "Generated Successfully" state.

Measurement of correctness for the scenario, lacks fine grained results about each one of the clauses that are going to be individually converted onto step definitions. So, looking at the results for "Given", "When" and "Then" clauses provides more information about the overall conversion capabilities of Kirby.

In Figure 5.3, you see that we are able to achieve about 90% correctness for the "Given" clauses, which is significantly higher than the overall scenario accuracy. We observed that

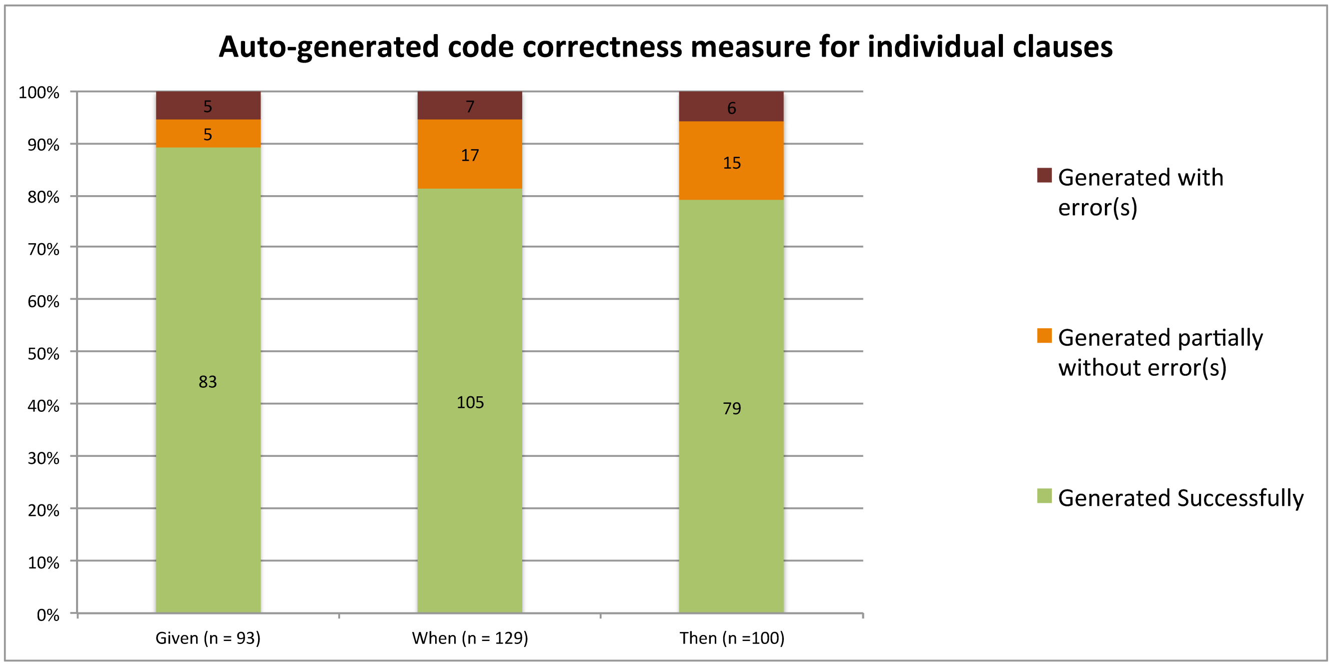


Figure 5.3: Kirby clause conversion chart for Given-When-Then clauses.

whenever a given clause is encountered, it generally provides us with enough context to pick out the correct class of interest. The accuracy of the "When" and "Then" clauses were around the 80% mark. The reason for the slightly lower scores, was because of the 10% not generated successfully state of the given clause. The accuracy of the "When" and "Then" clauses are heavily dependent on the accuracy of the "Given" clause, because if we do not pick the correct class, then there is a very high likelihood that we will not pick any correct methods.

The accuracy measure that we described, are heavily dependent on the capability and the probabilistic value provided by the Probabilistic matcher. As described in Section 4.3.2, there are multiple algorithms that we use in-order to decide the overall probabilistic value of match. Since a weighted average scheme is used to measure the overall confidence scores, let us have a look at how each individual algorithm performs w.r.t correctness if there was no combination of algorithms.

You would notice in Figure 5.4 that the accuracy of both levenshtein and cosine algorithms are on the lower end of the spectrum. But, Cosine-WordNet and Disco which perform semantic analysis by also considering synonyms into account, perform better in terms of conversion accuracy. Disco algorithm performs very well, but has a higher error state. So, we use a adaptive weighted average algorithm to estimate the probabilistic value, which performs the best among competing algorithms. The weighted averages for each one of the algorithms have been chosen after careful experimentation.

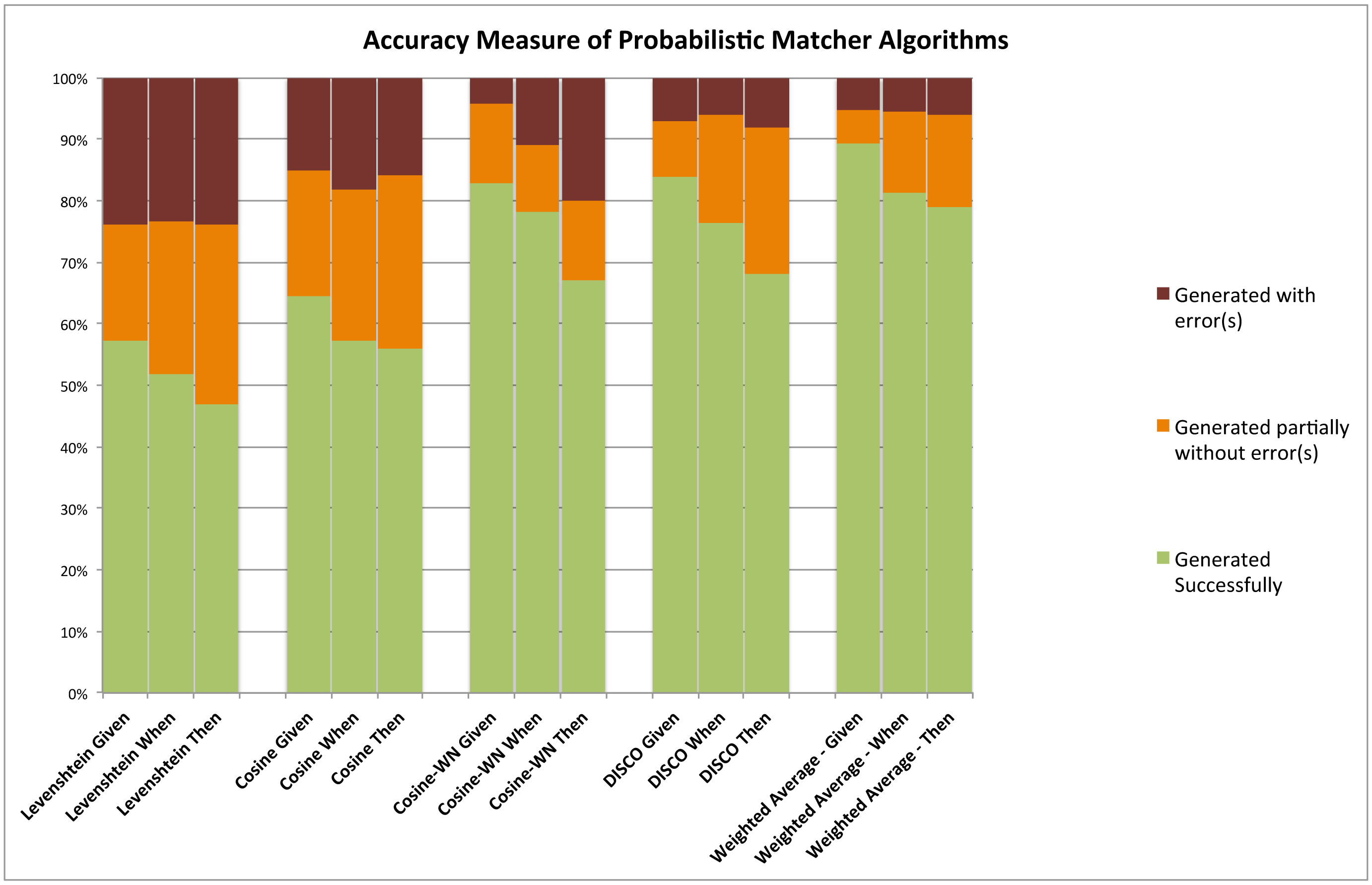


Figure 5.4: Kirby clause conversion chart for probabilistic matcher algorithms.

Another interesting measure to look at, is the confidence measure of each algorithm when it decided to pick a particular class or method. This can be extrapolated to look at the confidence scores on different metrics of conversion, on the large set of clauses that we have. This is shown in Table 5.1.

In the "Generated Successfully" scenario, we see that the weighted average value is about 76% confident that we have chosen the correct value. Cosine-WordNet and Disco give very good results in this case. If we look at the error cases, the weighted average score is about 41%, which seems high for scenarios which are reported incorrectly. You would also observe that Cosine, Cosine-WordNet and DISCO throw in large confidence values for false positives. As expected, the average confidence in the partially generated state is very low.

These values can be used to investigate about the effectiveness of the probabilistic matching algorithms in Kirby and fine tuning the weighted average scores. It can also be used to set the values of minimum threshold that we expect the confidence score to be, so that we can avoid false positives.

State	**Levenshtein**			**Cosine**			**Cosine - WN**			**DISCO**			**Weighted Average**		
Generated successfully	Given	0.64	0.6	Given	0.68	0.66	Given	0.92	0.9	Given	0.9	0.84	Given	0.8	**0.76**
	When	0.59		When	0.67		When	0.91		When	0.8		When	0.74	
	Then	0.57		Then	0.62		Then	0.89		Then	0.8		Then	0.73	
Generated with error(s)	Given	0.27	0.26	Given	0.38	0.38	Given	0.6	0.57	Given	0.32	0.34	Given	0.39	**0.41**
	When	0.25		When	0.38		When	0.53		When	0.39		When	0.39	
	Then	0.25		Then	0.39		Then	0.56		Then	0.31		Then	0.44	
Generated partially without error(s)	Given	0.09	0.09	Given	0.03	0.05	Given	0.13	0.16	Given	0.04	0.04	Given	0.06	**0.06**
	When	0.08		When	0.06		When	0.17		When	0.06		When	0.06	
	Then	0.09		Then	0.07		Then	0.18		Then	0.03		Then	0.07	

Table 5.1: Average confidence measure for probabilistic matcher algorithms.

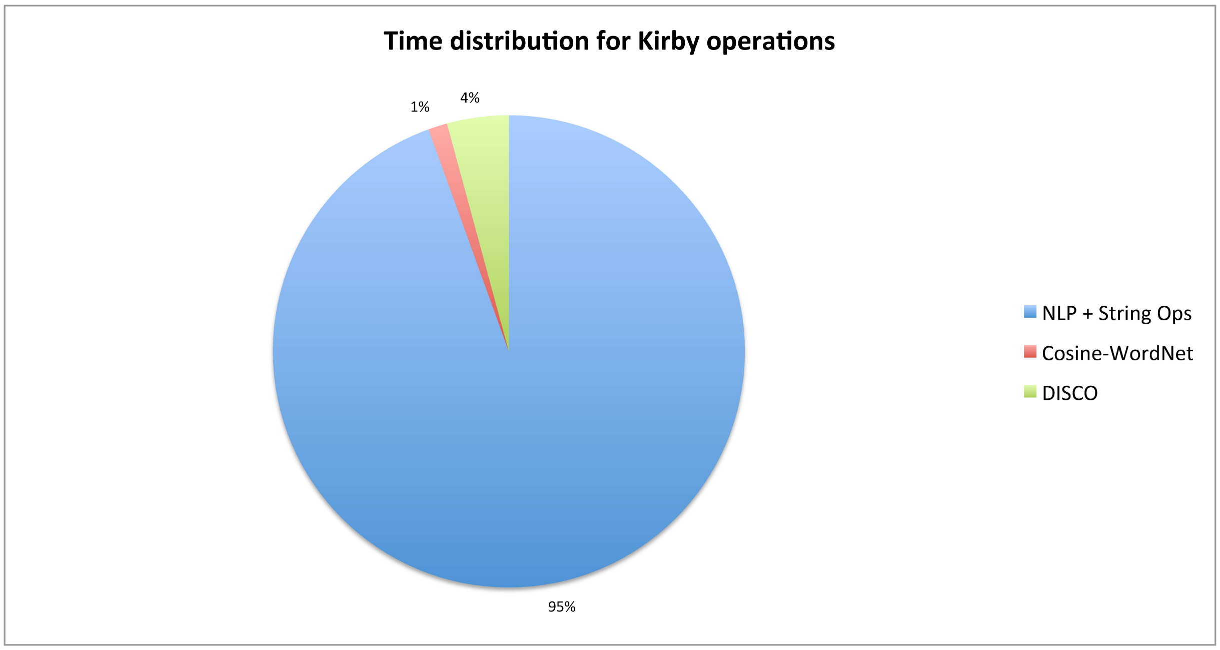


Figure 5.5: Time analysis based on different operations in Kirby.

5.3 Performance

In addition to accuracy, performance is also a concern. As a result, we collected timing data on the processing of individual clauses and whole scenarios. Timing measurements were performed on a simple development laptop, which was configured with a Intel i5 processor clocked at 3.1 GHz, 3MB L3 cache, 8GB of memory and a SSD drive.

Kirby's development quickly showed that most of the time is consumed in Natural Language Processing and string manipulation operations, rather than on Probabilistic matcher algorithms. Operations such as lemmatization along with NLP techniques used for parse tree construction and dependency analysis especially consumed most of the time.

Figure 5.5 shows that almost all the time is spent on NLP operations which takes about 95% of the processing time in Kirby. Cosine and Levenshtein hardly consume any time because they do not involve any dictionary lookup. DISCO and Cosine-WordNet consume about 5% of the time put together. It is possible to use smaller dictionaries for DISCO and Cosine-WordNet which will further reduce the time taken by these operations.

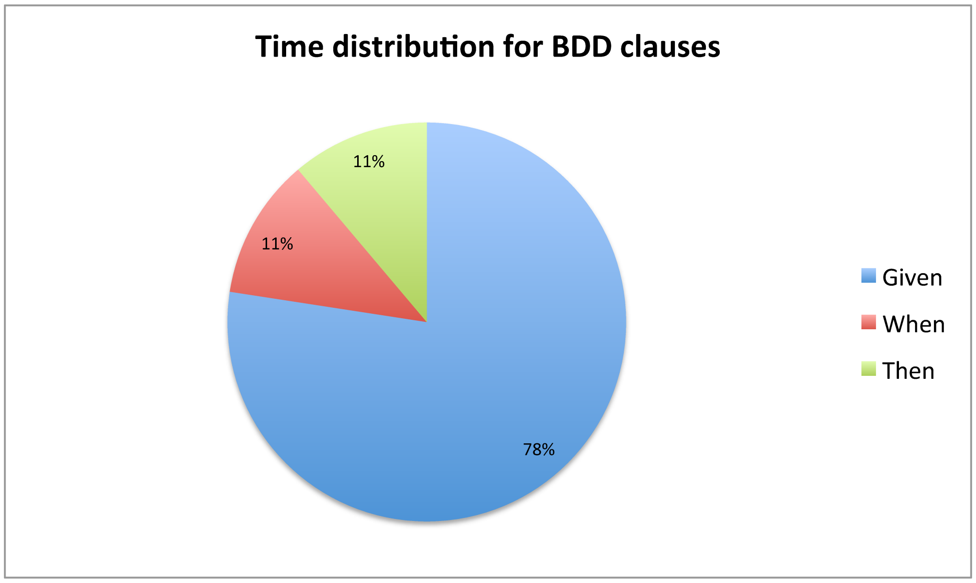


Figure 5.6: Average time distribution analysis for different clauses in Kirby.

Clause	**NLP**(Avg.)	**Probabilistic Matcher**(Avg.)	**Sum**(Avg.)
Given	2.96s	0.17s	3.13s
When	0.43s	0.08s	0.51s
Then	0.42s	0.03s	0.45s
Whole Scenario	4.67s	0.39s	4.96s

Table 5.2: Average time measure for different clauses and scenarios in Kirby.

Table 5.2 provides the time measures for different clauses and scenarios. For each scenario, it took less than 5 seconds on an average to generate its step-definition, showing that Kirby is a practical solution for real-world projects. It is interesting to observe that the "Given" Clause takes the most amount of time in our dataset. This is mostly because of the large number of classes in the search space for the given clause, which is about 40 classes in number. The When and Then clauses take a considerably lesser amount of time because most of the classes that we dealt with did not have large number of methods within them.

Figure 5.6 shows a pie-chart of the time spent on the different clauses in a BDD scenario, and you would see that more than 3/4th of the time is spent on finding the correct class for a Given clause. Then and When clause consume relatively a lower amount of time. But you have to note that this graph is for our dataset, and if you have classes in mature projects, with layers of inheritance, then the time taken by each of clauses could be very different.

Figure 5.7 shows the time distribution for the different clauses along with the time taken for each of the time-consuming probabilistic matcher algorithms. It also shows the time

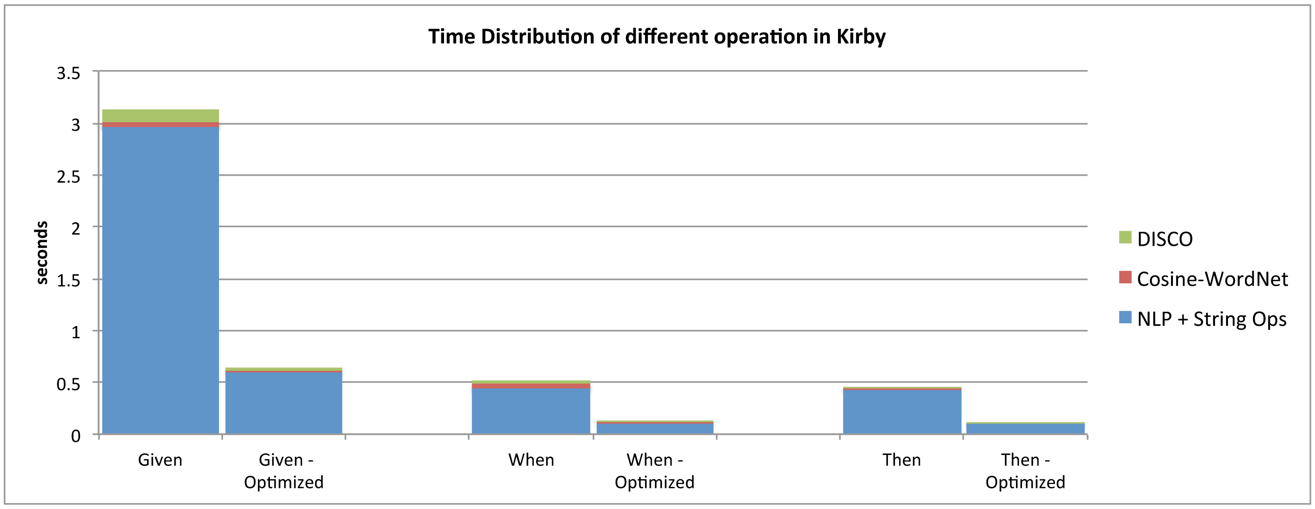


Figure 5.7: Time distribution analysis based on different clauses and algorithms in Kirby.

that was taken for a performance optimized variation of Kirby. In this optimization, we considered only clauses for matching if there was at least one word overlap between the behavior description and the class/method name. Though the time it took for this performance optimization was 5x lower, averaging 1 second for scenario conversions, the “Generated successfully” conversion correctness measure was brought down by about 8%.

Chapter 6

Conclusions

By designing and building Kirby, a prototype tool for auto-generating step definitions from structured natural language scenario descriptions, we have worked to explore the design space of code generation in Behavior-Driven Development. To show effectiveness of Kirby, we conducted an evaluation using numerous set of developer/customer written scenarios. This evaluation suggests ways in which Kirby succeeded, and ways in which it did not, and potential future improvements.

6.1 Contributions

We started out with the problem: *In Behavior-Driven Development, can we reduce the burden on programmers by utilizing information in the natural language, so that they don't have to hand-write the glue code?* With the design and evaluation of Kirby, we have shown that it is practical to convert natural language scenario descriptions into executable software tests fully automatically. Thus, providing a resounding answer to the problem that we set out to solve.

The results from the evaluation shows that we were able to achieve about 73% accuracy in auto-generating step definitions on existing whole BDD scenarios. But, if we look at individual clauses within the scenarios, we were able to achieve about 86% accuracy in generating individual code segments. Auto-generation of each clause took about 1.2 seconds on average, and whole scenarios took about 5 seconds on an average proving that this approach is suitable for real-world projects.

But, these results miss the main take-away—for every scenario where we were either not able to generate code segments or inaccurately generated code segments, we were able to refine the behavior description using the feedback from Kirby. These refined behavior descriptions were able to achieve 100% correctness in step definition generation. This proves the point—if developers and customers are aware of the existence of Kirby and it's approach to generation, they can iteratively define scenarios with almost certainty that the step definitions are going to be completely accurate.

The main contribution of this thesis was in the design of a prototype tool for achieving the goal of auto-generating step definitions from natural language scenario descriptions, and presenting results from applying the prototype to a reasonable set of real-world examples.

6.2 Future Work

With Kirby, we believe that we have laid the foundation for the future of development software designed to be used with Behavior-Driven Development projects for auto-generating step definitions. However, while Kirby shows that this approach is feasible, the road to a production-quality solution still contains a few bumps.

With NLP techniques, there is a possibility to include other techniques such as Morphological segmentation, Named-Entity-Recognition(NER), better Relationship Extraction strategies, etc. These techniques will help in inferring more information about the scenario descriptions, which will help us in making informed choices about the meaning of each clause. Also, all techniques that we use currently make static analysis of the given scenario, but if we can introduce machine learning approaches using reinforced learning that could help in fine-tuning our approaches and achieving better results.

With Code information extraction, we rely solely on Java reflection technique to extract data from Java Byte-code. We miss information such as comments, local variable name, content of methods, etc. With this additional information. we can better understand what the code is performing, which would help us in achieving higher accuracy when we perform probabilistic matching.

The probabilistic matcher is an interesting part of the architecture where we combine multiple competing algorithms in a weighted average manner. But, we need to perform extensive evaluation of the different algorithms and measure their efficiency. Using vector-based document model methods, corpus-based methods, hybrid methods, and descriptive feature-based methods for comparing short-text segments needs to be experimented with and evaluated.

Though the natural language description of scenarios is completely flexible, it would help if we can structure it to some extent by introducing our own grammar rules. These may be required for cases where we need to define variables, pass them around and have multiple objects interacting with each other. Though these cases explained here are supported by Kirby, access to these features needs to be streamlined and easy to understand for developers and customers alike.

The Eclipse-based plugin for Kirby is an useful add-on. But, we have not considered an evaluation of it's user-experience. We envision that we can provide a feature which can have an auto-complete mechanism where the user inputs the scenario and the corresponding code for it is shown in real-time. This would help in refining scenarios and may greatly help in the adoption of our technique.

Finally, we need to release our project to the open-source community and get developers interested in using our approach. Their feedback and usage pattern in large-scale projects can highly influence the future direction of Kirby.

Bibliography

[1] Dave Astels. A new look at test-driven development, 2005.

[2] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 1 edition, November 2002.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[4] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[5] Yihua Chen, Eric K. Garcia, Maya R. Gupta, Ali Rahimi, and Luca Cazzanti. Similarity-based classification: Concepts and algorithms. *J. Mach. Learn. Res.*, 10:747–776, June 2009.

[6] David Chelimsky and Dave Astels and Bryan Helmkamp and Dan North and Zach Dennis and Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends.* Pragmatic Bookshelf, 2010.

[7] Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, CrossParser '08, pages 1–8, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[8] Ian Dees, Matt Wynne, and Aslak Hellesoy. *Cucumber Recipes: Automate Anything with BDD Tools and Techniques.* Pragmatic Bookshelf, 2013.

[9] Steve Disney. The world english model revised: Definite article use in ice?gb and ice?hk. *University College Plymouth Marjon*, 2010.

[10] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. Running students' software tests against each others' code: new life for an old "gimmick". In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 221–226, New York, NY, USA, 2012. ACM.

[11] Dick Hamlet. Connecting test coverage to software dependability. In *Proc. 5th Intl. Symp. on Soft. Rel*, pages 158–165, 1994.

[12] Vasileios Hatzivassiloglou, Judith L. Klavans, and Eleazar Eskin. Detecting text similarity over short passages: exploring linguistic feature combinations via machine learning. In *In Proceedings of the 1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 203–212, 1999.

[13] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2):10:1–10:25, July 2008.

[14] Ron Jeffries and Grigori Melnik. Guest editors introduction: Tdd–the art of fearless programming. *IEEE Software*, 24(3):24–30, 2007.

[15] Xu Ke and Krzysztof Sierszecki. Generative programming for a component-based framework of distributed embedded systems. In *Proc. of the 6 th OOPSLA Workshop on Domain Specific Modeling*, pages 545–70, 2006.

[16] E Keogh. BDD: A Lean Toolkit. March 2010.

[17] Peter Kolb. DISCO: A Multilingual Database of Distributionally Similar Words. In Angelika Storrer, Alexander Geyken, Alexander Siebert, and Kay-Michael Würzner, editors, *KONVENS 2008 – Ergänzungsband: Textressourcen und lexikalisches Wissen*, pages 37–44, 2008.

[18] Lasse Koskela. *Test Driven: TDD and Acceptance TDD for Java Developers.* Manning Publications, October 2007.

[19] Nikolai Koudelia. Acceptance test-driven development. Master's thesis, University of Jyväskylä, 2010.

[20] Ioan Lazăr, Simona Motogna, and Bazil Pârv. Behaviour-Driven Development of Foundational UML Components. *Electronic Notes in Theoretical Computer Science*, 264(1):91–105, August 2010.

[21] Haode Liao, Jun Jiang, and Yuxin Zhang. A study of automatic code generation. In *Proceedings of the 2010 International Conference on Computational and Information Sciences*, ICCIS '10, pages 689–691, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *COMPUTATIONAL LINGUISTICS*, 19(2):313–330, 1993.

[23] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[24] George A. Miller. Wordnet: A lexical database for english. *COMMUNICATIONS OF THE ACM*, 38:39–41, 1995.

[25] G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 2004.

[26] Nachiappan Nagappan, E. Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, June 2008.

[27] Dan North. Introducing BDD. March 2006.

[28] Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '03, pages 308–, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Bartlett A. Shappee. Test first model-driven development. Master's thesis, Worcester Polytechnic Institute, 2012.

[30] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 269–287, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Carlos Solís and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *EUROMICRO-SEAA*, pages 383–387. IEEE, 2011.

[32] R. William Soukoreff and I. Scott MacKenzie. Measuring errors in text entry tasks: an application of the levenshtein string distance statistic. In *CHI '01 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '01, pages 319–320, New York, NY, USA, 2001. ACM.

[33] Sandeep Tata and Jignesh M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *SIGMOD Rec.*, 36(4):75–80, December 2007.

[34] Matt Wynne and Aslak Hellesoy. *The cucumber book : behaviour-driven development for testers and developers*. The pragmatic programmers. Dallas, Tex. Pragmatic Bookshelf, 2012.

[35] Fei Xia and Martha Palmer. Converting dependency structures to phrase structures. In *Proceedings of the first international conference on Human language technology research*, HLT '01, pages 1–5, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics.

[36] Jay J. Yu and Adam M. Fleming. Automatic code generation via natural language processing. United States Patent 7765097, July 2010.