Automatically Generating Tests from Natural Language Descriptions of Software Behavior

Sunil Kamalakar, Stephen H. Edwards and Tung M. Dao
Department of Computer Science, Virginia Tech, 2202 Kraft Drive (0902), Blacksburg, Virginia, U.S.A.

Keywords: Behavior-Driven Development, Test-Driven Development, Agile Methods, Software Testing, Feature Description, Natural Language Processing, Probabilistic Analysis, Automated Testing, Automated Code Generation.

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 can then manually write "glue" code so that these scenarios can be translated into executable 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). This paper takes the position that, instead of requiring programmers to write manual glue code, it is practical to convert natural language scenario descriptions into executable software tests *fully automatically*. To show feasibility, this paper presents preliminary results from a tool called Kirby that uses natural language processing techniques 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). Preliminary results from assessing the performance and accuracy of this technique are presented.

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), (Beck, 2002; Koskela, 2007) 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 (Nagappan et al., 2008), 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 (2013) as a response to limitations observed with TDD. In BDD we specify each "behavior" of the system in a clearly written and easily understandable scenario description written using natural language. These natural language scenarios help all stakeholders—not just programmers—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 (Solis and Wang, 2011), while keeping the focus on human-readable scenario descriptions that can be easily understood by customers as easily as by developers.

However, 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 developed to make the process of writing this glue code easier and more

Kamalakar S., H. Edwards S. and M. Dao T..
Automatically Generating Tests from Natural Language Descriptions of Software Behavior.
DOI: 10.5220/0004566002380245
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 238-245
ISBN: 978-989-8565-62-4

streamlined, and to automatically map phrases in the scenarios into 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.

This paper takes the following position:

> Instead of requiring programmers to write manual glue code, it is practical to convert natural language scenario descriptions into executable software tests *fully automatically*.

To defend this position, the remainder of this paper describes work in progress on a tool called *Kirby*: a BDD support tool that automatically translates natural language scenario descriptions into executable Java software tests. This paper describes the approach used and presents preliminary results that demonstrate feasibility.

The paper is organized as follows: Section 2 briefly reviews the related work including typical BDD practices. Section 3 describes Kirby and its architecture and implementation strategy. Section 4 illustrates how the approach works with examples, and Section 5 summarizes our preliminary evaluation. The paper concludes with a discussion of future work in Section 6.

2 RELATED WORK

There are few published studies on BDD, most of which take a relatively narrow view, treating it as a specific technique of software development (Solis and Wang, 2011). Keogh (2010) embraces a broader view, arguing its significance to the whole lifecycle of software development, especially on the business side and the interaction between business and software development. Lazar et al. (2010) 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.

Many tools for BDD have been created for use in different contexts, the best known of which is Cucumber (Cucumber, 2013, Wynne and Hellesøy, 2012). Cucumber is a BDD tool written for the programming language Ruby. Developers and customers write semi-structured natural language scenario descriptions, and developers write the corresponding test "steps" in Ruby, using regular expressions to match natural language phrases used in the scenarios. Similar tools exist for other languages (JBehave, 2013, JDave, 2013, NBehave, 2013, PHPSpec, 2013). Traditionally, TDD has been used in writing unit tests and BDD has evolved to specify acceptance tests. Nevertheless, software developers should be able to leverage the capabilities of BDD to specify unit tests in a intuitive manner as well.

Cucumber uses the Gherkin language (Gherkin, 2013) for writing semi-structured scenario descriptions. It is a "business readable, domain specific language" (Cucumber, 2013) that lets you describe software behaviors without detailing how those behaviors are implemented. Gherkin simultaneously serves two purposes: documentation and automated test description. Gherkin structures descriptions this way:

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

In Gherkin, each behavior is called a scenario, and scenarios can further be grouped into stories or features. For example, here is a short scenario describing a stock trading behavior in Gherkin:

```
Scenario: check stock threshold
  Given a stock with symbol GOOGLE and
    a threshold of 15.0
  When the stock is traded at a price
    of 5.0
  Then the alert status is OFF
  When the stock is sold at a price of
    16.0
  Then the alert status is ON
```

The scenario name is a shorthand description of what the scenario is supposed to do. Scenarios use a declarative syntax containing "Given", "When" and "Then" clauses. "Given" clauses describe an initial context for some behavior, "When" clauses describe the occurrence of one or more events or actions, and "Then" clauses describe the expected outcome(s). The sentences (or phrases) after each keyword are free-form, but each must match a specific step (or glue method) written by the developer(s).

While existing BDD tools such as Cucumber and JBehave rely on regular expressions to recognize key phrases in scenarios in order to map them to steps, we propose a strategy based on a more comprehensive natural language processing (NLP) approach. Other techniques for automatically generating code based on NLP have been described (Yu and Fleming, 2010). Budinsky (1996) describes code generation techniques that generate templates for user-specified design patterns. Various software tools for UML, XML processing, etc., are capable of generating source code based on user input. These systems require a well-defined input format, and unless one follows a known grammar with unambig-

uous, clearly defined instructions, it is difficult to auto-generate program code from natural language on the fly.

In another closely related work, Soeken et al. (2012) 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 and thus provides a first step towards automating BDD. Our approach borrows certain ideas of NLP, but aims for complete automation.

3 A NEW APPROACH

BDD is a highly iterative and incremental process where one switches back and forth between working on scenarios and developing application code that meets the scenario's requirements. We propose a new tool called *Kirby* that completely automates the generation of the individual steps in executable tests 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 test steps. However, while Kirby shows that this approach is feasible, the road to a production-quality solution still contains a few bumps.

The workflow we envision for BDD with Kirby is illustrated in Figure 1. Developers alternate between creating or revising scenarios and writing (or creating stubs for) implementations of features. Since the implementation code is written with the scenario in mind, the language that is used in the code we believe naturally will 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, which will generate the needed step definitions for us automatically from the language used in the scenarios themselves.

3.1 The Design of Kirby

Gherkin as a language is very expressive and intuitive for describing scenarios—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.

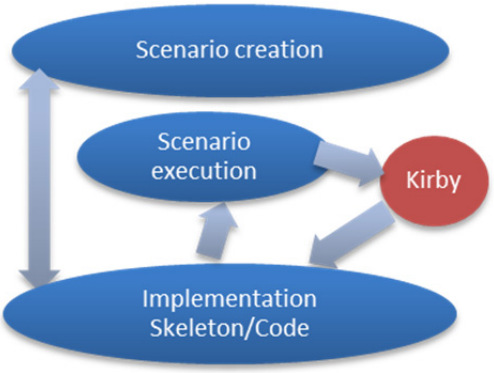

Figure 1: BDD Workflow with Kirby.

The "When" clause(s) represent method calls on objects involved in the test, while "Then" clauses represent assertions that check 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.

The high level architecture for Kirby is shown in Figure 2. Since the goal is to relieve the developer from writing step definitions manually, we need to develop a mechanism to map the natural language scenarios onto the code implementation/skeleton that is being written alongside the scenarios.

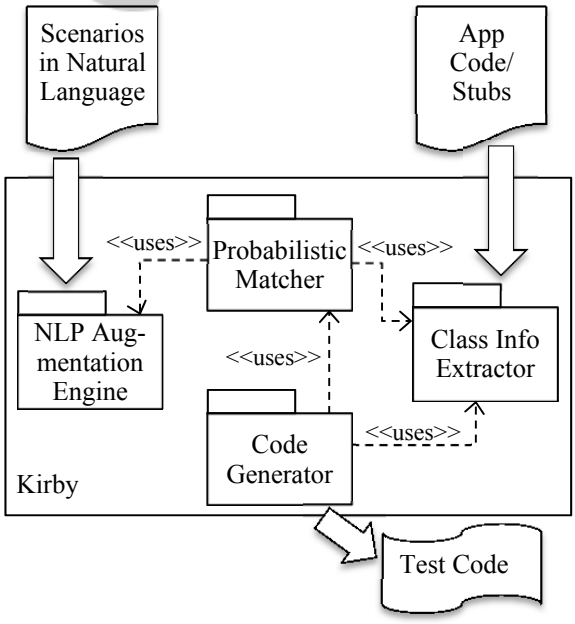

Figure 2: Kirby's high-level architecture.

Kirby uses *both* the scenario descriptions and the co-developed software (complete code or stubs) as input when generating executable tests. It uses a

NLP 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 application. Once suitable matches have been found, the *Code Generator* synthesizes this information to produce JUnit-style tests.

3.2 NLP Augmentation Engine

The natural language processing performed in Kirby occurs in its NLP Augmentation Engine, which preprocesses a scenario to extract structural information about the organization of its clauses. The Stanford NLP library (Stanford NLP Group, 2013) is used to create a Phrase Structure Tree (PST) for each clause, allowing the noun phrases and verb phrases to be extracted. We also use its capability to augment information about the types of dependencies that exist between the words of each clause.

Preprocessing includes removal of stop-words that do not add any value to the meaning of the sentence, while making sure that the PST representation for the clause still remains intact. The augmentation also ensures that we keep the lemmatized version of each word encountered in the clause to reduce ambiguity by consolidating different inflected forms of the word into a single form for matching.

3.3 Code Information Extractor

The Code Information Extractor is responsible for extracting information from the implementation code. Kirby uses a streamlined Java reflection API (Edwards et al., 2012) to keep track of all the classes in a particular project, plus the classes that are accessible and utilized by those classes. These classes represent our search space for the objects that need to be created based on the natural language information available in the scenario clauses. Since we use Java, which follows a strict object oriented structure, we keep track of the public methods and also public members that are part of each class. If the Java bytecode for the application includes debug information (which is typical during development), the Code Information Extractor also keeps track of the names of each method's parameters.

3.4 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 behavior 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 quite challenging. There is no one-stop solution or algorithm that works perfectly in this situation. An edit-distance algorithm like Levenshtein (Soukoreff and MacKenzie, 2001) may work favorably in a situation where the user specifies the partial or exact wording used in the code, but it will fail miserably when a synonym is used in natural language. For multi-word or sentence matching a vector space model like cosine similarity gives better values (Tata and Patel, 2007).

Kirby takes a hybrid approach that combines multiple algorithms that include confidence measures, weighing them against each other to choose the most likely match. Kirby's cosine similarity measure has been extended to include WordNet (Fellbaum, 1998) so that word synonyms can be handled. For computing the semantic similarity between words based on how they are used in written language, Kirby uses a tool called DISCO (Kolb, 2008) that provides a second order similarity measure between two words or sentences based on actual usage in large datasets like Wikipedia. The challenges faced in these computations vary depending upon what specific kind of matching is needed in a particular clause: class matching, parameter matching, constructor matching, or method matching.

The Probabilistic Matcher uses weighted averaging to adapt its matching model based on the individual values obtained from the competing algorithms. 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. The weights used were obtained through experimentation as discussed in Section 5.

3.5 Code Generator

Once phrases in the scenario have been matched with classes, methods, and values, 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 (or sequence of calls). “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 (CodeModel, 2013). We also handle ambiguity and error handling at this layer. If the probability values are very close to each other, or if no match is found, or if the confidence measure is too low, the Code Generator will generate a `fail()` method call in the test specifying the reason for the ambiguity.

4 EXAMPLES IN ACTION

To see how this strategy works in practice, consider the example scenario shown in Section 2, which comes from the JBehave web site (JBehave, 2013):

```
Scenario: check stock threshold
  Given  a stock with symbol GOOGLE and
    a threshold of 15.0
  When the stock is traded at a price
    of 5.0
  Then the alert status is OFF
  When the stock is sold at a price of
    16.0
  Then the alert status is ON
```

The NLP Augmentation Engine parses the clauses as shown in Figure 3.

To see how the proposed approach operates in practice, we will examine the actual results produced by the Kirby prototype on this scenario. In this example, the parse of the “Given” clause identifies three nouns, “stock”, “symbol”, and “threshold”.

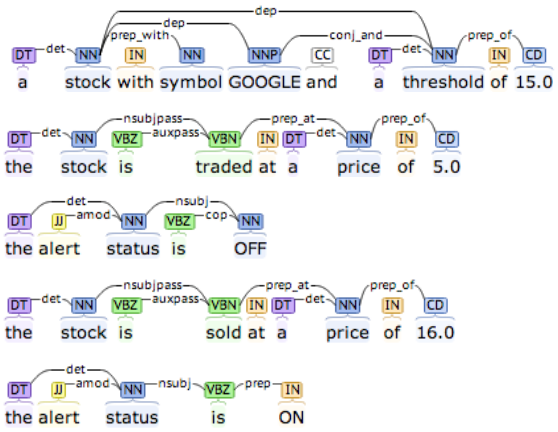

Figure 3: Parse results for the “stock” scenario.

The Code Information Extractor found the class `Stock` as part of the application under development. This class does include a constructor that happens to have parameters named “symbol” (a String) and “threshold” (a double). As a result, the Code Generator translates this “Given” clause into:

```
Stock stock =
    new Stock("GOOGLE", 15.0);
```

Similarly, the first “When” clause includes two nouns and a verb phrase. By looking at the labelling of the parse, it is possible to determine that “stock” is the subject of the verb—in this case, the receiver of a method call that represents an action. The verb phrase determines the action, while the second noun is an object that represents a parameter value. Because the Code Information Extractor reports a `tradeAt()` method with one parameter named “price”, the Probabilistic Matcher produces a high-confidence match.

The “Then” clause is handled similarly, where the noun “status” is matched to an existing method named `getStatus()` that returns a string, and its presence in a “Then” clause triggers the use of an assertion to compare this method’s return value with an expected result. The complete test produced is:

```
@Test
public void testCheckStockThreshold()
{
  Stock stock =
    new Stock("GOOGLE", 15.0);
  stock.tradeAt(5.0);
  assertEquals(
    "OFF", stock.getStatus());
  stock.tradeAt(16.0);
  assertEquals(
    "ON", stock.getStatus());
}
```

Now consider another BDD scenario, this time for a program implementation of Conway’s “Game of Life” (also inspired by scenarios posted on the JBehave web site):

```
Scenario: multiple toggle outcome
  Given a game called gameOfLife, with
    width of 5 and height of 6
  When I toggle the cell at column = 2
    and row = 4
  And I switch the cell at "4", "2"
  And I alternate the cell at row 4 and
    column 2
  Then the string representation of the
    game should look like
    "_ _ _ _ X
     _ _ _ _ _
     _ _ X _"
```

The same parsing strategy results in the following JUnit test:

```
@Test
public void testMultipleToggleOutcome()
{
  Game gameOfLife = new Game(5, 6);
  gameOfLife.toggleCellAt(2, 4);
  gameOfLife.toggleCellAt(4, 2);
  gameOfLife.toggleCellAt(2, 4);
  assertEquals(
    "_ _ _ _ X\n_ _ _ _ _\n_ _ _ X _",
    gameOfLife.
      getStringRepresentation());
}
```

The example above shows how Kirby provides flexibility in the way that different word choices, such as "toggle", "alternate", or "switch", can all be mapped to the method `toggleCellAt()` by using for flexible similarity measures. This example also shows different free-form choices for expressing parameter values.

Finally, here is a third scenario for a class that is used to make web requests and examine the resulting responses:

```
Scenario: request contains string
  Given a URL with value
    "http://google.com", called google
  And a web requester with url equal to
    google
  When we set the timeout to be 100
  And we send a request
  Then the response contains "google"
```

The corresponding JUnit test generated by Kirby is:

```
@Test
public void testRequestContainsString()
{
  URL google =
    new URL("http://google.com");
  WebRequester webRequester =
    new WebRequester(google);
  webRequester.setTimeout(100);
  webRequester.sendRequest();
  assertTrue(webRequester.
    getResponse().contains("google"));
}
```

Note that in all of the generated test methods above, the code corresponding to the "Given" clauses have been embedded directly in the test methods. If a person were writing tests by hand, he or she would most likely take advantage of a `setUp()` (or `@Before`) method so that the starting conditions for the test could be reused across multiple tests. In this case, however, the "source code" of the tests is the natural language scenarios from which the Java test code is automatically generated. Since scenarios may or may not contain overlapping "Given" clauses, and programmer updates and modifications to tests are expected to be made in the scenario descriptions themselves, the Code Generator does not generate separate `setUp()`-style methods.

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. At the same time, however, it is important to evaluate new techniques against real-world situations. To this end, we compiled a small collection of 12 BDD scenario descriptions written in Gherkin—primarily from tutorials published for use by developers learning to use other BDD tools. We then ran this collection through Kirby to assess its accuracy and performance, with the belief that these examples are representative of at least some portion of real-world practice.

Table 1 shows the accuracy of each of the four individual matching algorithms employed in Kirby to match parsed nouns to classes or objects. In addition, the "weighted average" shows the accuracy of the final result produced by the Probabilistic Matcher when it chooses results from the four competing algorithms based on confidence weights. Table 2 shows the same information for matching verbs and verb phrases to methods (note that "Given" clauses typically use constructors rather than method calls, and so are not included in Table 2).

Table 1: Matching algorithm accuracy for nouns and noun phrases (classes and objects).

	Clause Type		
Algorithm	**Given**	**When**	**Then**
Edit distance	40%	55%	50%
Cosine	69%	78%	80%
Cosine (WordNet)	56%	56%	49%
DISCO	84%	96%	88%
Weighted average	100%	100%	100%

Table 2: Matching algorithm accuracy for verbs and verb phrases (methods).

	Clause Type		
Algorithm	**Given**	**When**	**Then**
Edit distance	-	23%	30%
Cosine	-	58%	61%
Cosine (WordNet)	-	32%	28%
DISCO	-	76%	80%
Weighted average	-	100%	100%

From these tables, it is clear that using a single algorithm will not produce acceptable accuracy. However, by running all algorithms and considering the confidence scores associated with each, it is possible to pick results from the algorithm that produces the most likely match in any given clause, increasing overall accuracy significantly. At the same time, however, this simply shows the feasibility of improving accuracy by combining algorithms using confidence measures. The tiny set of scenarios used cannot be taken as truly representative of real-world practices.

In addition to accuracy, speed is also a concern. Kirby's development quickly showed that some algorithms depend critically on a dictionary of known words, and the smaller the dictionary, the less capable the algorithm—but larger dictionaries significantly increase processing time. DISCO in particular, as well as the WordNet extension to the cosine measure, is susceptible. As a result, we collected timing data on the processing of individual clauses and whole scenarios, both using the most comprehensive dictionaries available, and also using a smaller dictionary intended to reduce processing time. Unfortunately, the smaller dictionary also reduced accuracy—resulting in a 25% loss in accuracy for word and phrase matching. Table 3 summarizes the run time performance.

Table 3: Average running time for phrase analysis/matching.

Clause type	Comprehensive Dictionary	Smaller Dictionary
Given	4.57 s (s.d. 3.98)	3.47 s (s.d. 3.61)
When	0.59 s (s.d. 0.27)	0.45 s (s.d. 0.28)
Then	0.84 s (s.d. 0.45)	0.73 s (s.d. 0.46)
Complete scenario	8.34 s (s.d. 3.78)	6.47 s (s.d. 3.73)

From Table 3, it is clear that NLP is time-consuming in relation to simpler approaches like regular expression matching. It is interesting to note that the bulk of the time is associated with class and constructor matching in given clauses, while method-based matching in later clauses is much faster. It is also interesting that using a smaller dictionary sacrificed a noticeable amount of accuracy, but did not drastically improve speed.

6 CONCLUSIONS

This paper takes the position that fully automated translation of natural language behavioural descriptions directly into executable test code is practical. By describing the design of a prototype tool for achieving this goal, and presenting results from applying the prototype to a small collection of real-world examples, this paper also shows the feasibility of one technique for accomplishing this task.

At the same time, however, this prototype represents work in progress and has not undergone a significant evaluation in the context of authentic BDD usage by real developers. As future work, it is necessary to collect a much larger library of existing BDD scenario descriptions—preferably from open-source projects, since the corresponding applications would also be needed—to serve as a baseline for truly evaluating effectiveness. Further, additional improvements in performance (and potentially accuracy) are also needed.

REFERENCES

Beck, K. 2002. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S. 1996. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, May 1996.

CodeModel. 2013. http://codemodel.java.net [Accessed May 15, 2013].

Cucumber. 2013. http://cukes.info/ [Accessed May 15, 2013].

Edwards, S. H., Shams, Z., Cogswell, M., and Senkbeil, R. C. 2012. 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*, pp. 221–226, ACM, New York, NY, USA.

Fellbaum, C. (ed.). 1998. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, USA.

Gherkin. 2013. https://github.com/cucumber/cucumber/wiki/Gherkin [Accessed May 15, 2013].

JBehave. 2013. http://jbehave.org/, Accessed May 15, 2013.

JDave. 2013. http://jdave.org/ [Accessed May 15, 2013].

Keogh, E. 2010. BDD: A lean toolkit. In *Proceedings of Lean Software Systems Conference*, 2010.

Kolb, P. 2008. DISCO: A multilingual database of distributionally similar words. In *Proceedings of KONVENS-2008*, Berlin.

Koskela, L. 2007. *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning Publications Co., Greenwich, CT, USA.

Lazr, I., Motogna, S., and Pírv, B. 2010. Behaviour-driven development of foundational UML components. *Electronic Notes in Theoretical Computer Science*, 264(1): 91–105, Aug. 2010.

Nagappan, N., Maximilien, E. M., Bhat, T., and Williams, L. 2008. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3): 289-302, June 2008.

NBehave. 2013. http://nbehave.org/ [Accessed May 15, 2013].

North, D. 2013. Introduction to BDD. http://dannorth.net/introducing-bdd/ [Accessed May 15, 2013].

PHPSpec. 2013. http://www.phpspec.net/ [Accessed May 15, 2013].

Soeken, M., Wille R., and Drechsler, R. 2012. Assisted behavior driven development using natural language processing. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns* (TOOLS'12), Springer-Verlag, Berlin, Heidelberg, pp. 269-287.

Solis, C. and Wang, X. 2011. A study of the characteristics of behavior driven development. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 383–387.

Soukoreff, R. W., and MacKenzie, I. S. 2001. 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*. ACM, New York, NY, USA, pp. 319-320.

Stanford NLP Group. 2013. http://nlp.stanford.edu/ [Accessed May 15, 2013].

Tata, S., and Patel, J.M. 2007. Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM SIGMOD Record*, 36(2): 7-12, June 2007.

Yu, J.J.-B., and Fleming, A.M. 2010. Automatic code generation via natural language processing, U.S. Patent 7765097, July 27, 2010.

Wynne, M., and Hellesøy, A. 2012. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers, LLC.