Connecting User Stories and Code for Test Development

Mathias Landhäußer, Adrian Genaid
Karlsruhe Institute of Technology
Karlsruhe, Germany
landhaeusser@kit.edu, adrian.genaid@ira.uka.de

***Abstract*—User Stories are short feature descriptions from the user's point of view. Functional tests ensure that the feature described by a User Story is fully implemented.**

We present a tool that builds an ontology for code and links completed User Stories in natural language with the related code artifacts. The ontology also contains links to API components that were used to implement the functional tests. Preliminary results show that these links can be used to recommend reusable test steps for new User Stories.

***Keywords*-code mining; functional testing; reasoning; traceability; ontology**

I. Introduction

Agile development processes have been on the rise since the early 2000s. Informal documents and sketches dominate the requirement elicitation process. One early artifact of agile development processes is the User Story (see Fig. 1). It depicts a feature of the software from a stakeholder's point of view, while it is not as detailed and extensive as a Use Case [1].

Developers implement the needed functionality and write unit tests (and integration tests). Independently, testers write structured natural language test scripts and describe the usage of the new functionality step-by-step (in so called test steps). Also, test steps can define the desired outcome and/or the desired behavior of the software. Test steps that are identical for a number of stories should be reused instead of reimplemented. However, the manual process of finding reusable test steps is tedious and time consuming.

We propose to build an ontology of the entire code base together with the completed User Stories and their respective functional tests. This ontology can then be used to retrieve similar test steps to ease test code reuse. Furthermore the ontology can provide guidance for finding the parts of the API that are likely to be needed when implementing a new test script.

The remainder of this paper is structured as follows: Section II explores related work. Section III illustrates our approach and Section IV shows preliminary results. Section V concludes this paper and subsumes future enhancements as well as future opportunities.

II. Related Work

In this section, we focus on Behavior Driven Development (BDD) and behavior driven testing. Also we review some work on building and using software ontologies.

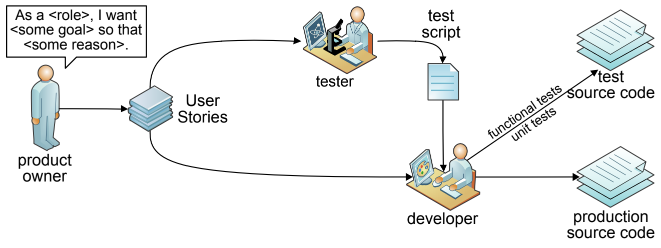


Figure 1. A Development and Test Process using User Stories.

A. Behavior Driven Development

BDD is an agile development technique that concentrates on specifying and testing a system in a new way: It focuses on the desired behavior of the new software as defined by stakeholders in natural language [2], [3]. User Stories lay the foundation of the process: They are a short description of a new feature and should be written according to the template *As a <role>, I want <some goal> so that <some reason>* (see Listing 1). User Stories are also used for testing: Test scripts illustrate the story's new feature and show the expected behavior of the software. They ensure that the functionality and behavior of the software meets the stakeholder's criteria. As you can see in Listing 2, a test script comprises several test steps, which explicitly define either preconditions (*Given*), actions (*When*), or expected results (*Then*).

Scripts for functional tests can be written in Gherkin, "a Business Readable, Domain Specific Language that lets you describe software's behaviour without detailing how that behaviour is implemented"[1] and there are grammars for many spoken languages. There are tools that translate Gherkin scripts into code stubs; e.g. SpecFlow for the .NET platform [4]. For every test step, there is one stub method generated that either asks for program state setup (*Given*), actions (*When*), or for result verification (*Then*). Provided with these stubs, the developer has to fill the gaps with the needed API calls to exercise the steps. The resulting test code can be run with a test environment that resembles the well-known xUnit test environments. Succeeding functional tests mark the completion of the implementation of the User

[1]https://github.com/cucumber/cucumber/wiki/Gherkin, accessed: 03/23/2012.

```
As a string manipulation library user, I want to
    have a fancycase method in order to gain
    fancy cased strings.
* The fancy case method should print the
    characters of a string alternating in upper
    and lower case.
* Whitespace should be ignored.
```

Listing 1. A Simple User Story for String Manipulation.

```
Scenario: Fancy strings
  Given I have the string "BDD is fun"
  When I put it in fancy case
  Then I see the string "BdD iS fUn"
```

Listing 2. A Simple Test Script for String Manipulation.

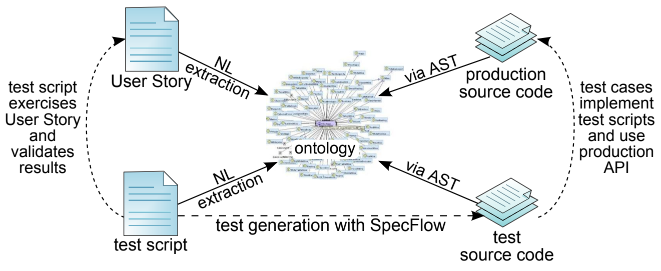
Story. While BDD tools help a great deal when it comes to translating the specification into test methods, there is a lack of tool support for finding reusable test steps.

B. Software Ontology Building and Population

Mining software requires that the software itself (i.e. source code) and related artifacts such as documentation and requirements documents are stored in a machine readable format. Both, formal and informal sources should be analyzed and prepared for the storage in a formally defined structure. Ideally, all artifacts are stored in a single database and can be queried using a single interface. Ontologies offer exactly that: They allow to specify concepts and relations and thus are an "explicit specification of a conceptualization" [5]. They are not limited to the specification of the concepts, but also allow for storing (or rather defining) instances of these concepts.

Queries to ontologies can not only ask for explicitly recorded, but also for inferred information. An example for information that can be inferred are transitive relations: If a relation `rel` holds between the elements `a` and `b`, and `b` and `c` respectively and `rel` is transitive, then we can infer that `rel` holds between `a` and `c` also. Such inference tasks are carried out by a reasoner. We use OWL2 [6] for the definition of the ontology. We chose Pellet [7] as reasoner because it proved to be reliable and fast on our ontology ([8] lists reasoners for OWL2). Ontologies can be queried using a SPARQLDL query engine [9], which uses the reasoner for answering questions.

Zhang et al. link ontologies of design documentation and source code [10]. Their system SOUND supports reverse engineering tasks. It allows easy querying for source code and documentation elements such as instances of a specific design pattern. Khamis et al. extract source code comments, process them using a natural language pipeline and populate an ontology [11]. During the processing, they calculate metrics to assess source code comment quality [12]. To the best of our knowledge, there is no previous work that aims directly at easing test implementation or API retrieval for natural language requirements.



User Story
NL extraction
via AST
production source code
test script exercises User Story and validates results
ontology
test cases implement test scripts and use production API
NL extraction
via AST
test script
test generation with SpecFlow
test source code

Figure 2. An Overview of Sources for Ontology Population.

Overlapping and/or complementary ontologies can be linked (ontology matching or alignment) [13]. A source code ontology such as ours could be aligned with one from a bug tracking database or a version control system. Especially the latter could provide important information about test step implementation. As developers are hardly supported in writing commit messages, we cannot rely on commit messages for linking source code modifications with User Stories or requirements respectively. Furthermore, we are less interested in new or modified production code, but more in reusable test code.

III. Exploiting a Combined Knowledge Base for Code and User Stories

Given that the User Stories and test scripts are written by non-technical staff (likely a member of the operating department), the natural language representation of the test steps can be inconsistent. Developers must know the software under test by heart to efficiently implement the tests. When different developers implement similar, overlapping, or equal test steps, one cannot take that knowledge for granted; developers then have to search the other test scripts for similar steps and/or the code base for the needed API interfaces.

To address the challenge of retrieving test steps that could be reused, we use an ontology that provides a structure for elements of source code (similar to Zhang's source code ontology [10]). Fig. 2 illustrates the population sources and their connections. We built F-TRec (Functional Test Recommender), a prototype for C# and SpecFlow tests which we evaluate in Sect. IV. In the following, we describe how we build and use the ontology.

A. Ontology Structure and Population

Source Code: We decided not to use classic vector space models since we plan to extend the retrieval algorithms to use API information (e.g. call graphs) and build an integrated knowledge base for the use in software projects. To allow a precise representation, the ontology provides distinct concepts for source code elements and relations. To populate the ontology, we first parse the project source code (production code as well as test code). We traverse

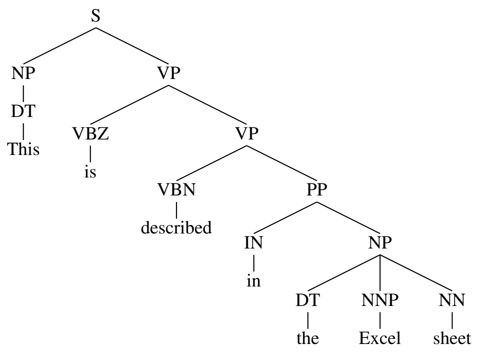


Figure 3. A Parse Tree containing a Compound Noun.

the abstract syntax tree and record all structural elements (e.g. classes, methods and so on). Then, we fill in the links between the extracted elements according to the source code structure. Furthermore, we analyze implementations of methods and similar elements to identify further links. Thus, the resulting ontology parallels the source code in structure. Here, we consider method calls, object creations, and inheritance relations.

Test Scripts: Secondly, we include test scripts into the ontology. Since for every user story there is a test script and a corresponding test implementation, test scripts are an ideal bridge between source code and User Stories (c.f. III-B). A test script consists of a feature title and test scenarios. A test scenario is described by a title and test steps (c.f. Listing 2); a test step is a sentence in natural language beginning with *Given*, *When* or *Then*. All test script elements are added to the ontology; the order of the steps is encoded in relations.

Requirements: The final ingredient for our ontology are natural language elements (NLEs) that we extract from User Stories and test steps. At the moment, we only extract (compound) nouns and verbs. We identify these elements by parsing the text sentence-wise with the Stanford factored parser [14].

The output of the parser is a parse tree representing the sentence structure; you can see the tree for the phrase *This is described in the Excel sheet* in Fig. 3. The root node `S` represents the entire phrase (sentence) and the leaves represent the words. The inner nodes describe the structure of the phrase; an inner node directly before a leaf stands for the part-of-speech tag (POS-tag) of the adjacent word leaf. Verbs are tagged with a POS-tag that starts with `VB`; POS-tags for nouns start with `NN`. Compound nouns are groups of nouns that are siblings in a noun phrase (`NP`) subtree, e.g. *Excel sheet* in Fig. 3. We extract non-compound nouns and verbs directly. Since we want to store normalized NLEs in our ontology, we lemmatize the extracted words; e.g., the passive verb *is described* is reduced to its lemma *describe*.

Then we employ the Stanford Named Entity Recognizer [15] to identify and filter person names that occur in the stories. Furthermore, the extracted NLEs have to be filtered,

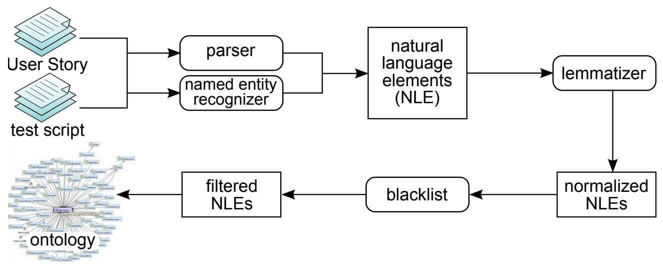


Figure 4. F-TRec's Natural Language Element Extraction Process.

as some of them do not contain valuable information (e.g. verbs like *to be*, pronominals, and nouns like *February*). Since the list of such NLEs is project specific, we administer it manually. Fig 4 illustrates the processing of natural language sources through a NLP pipeline to extract, filter, and add NLEs to the ontology.

B. Linking Stories and Code

For every User Story there is a test script that can be compiled into code stubs with SpecFlow. So every NLE that is extracted from a User Story is linked with the corresponding test script. Furthermore, a NLE that occurs in a test step is linked directly with that step. So a NLE in our ontology is linked either with a test script or a test step or both. NLEs linked with a test script are linked with all test steps of that script transitively.

Test steps exist in our ontology twice: One time as part of a test script and a second time as a C# test method. We can link these artifacts, because SpecFlow annotates the C# methods with the step text. SpecFlow eases the reuse of test steps in that it allows parameterized test step implementations. Then the annotation does not contain the natural language test step but a parameterized variant. The annotation for a reusable step implementation of line two in Listing 2 would be `Given("I have the string (.*)")`. We use the annotation to identify the matching implementation for a test step and link the step with its implementation.

C. Using the Knowledge

Given a new User Story, we extract all NLEs just as we did during ontology population; we refer to them as *query NLEs*. Then we search the knowledge base for all test steps that are linked with query NLEs. Listing 3 shows a query for steps associated with the query NLE *button*.

The result is very much like the one, a Boolean retrieval would give. The number of results can be large, thus we must sort the results by relevance. We assume that a test step, that is linked to more than one query NLE, is more likely to be helpful than a single-linked step. Simply counting the number of links is not sufficient, since some NLEs are used in many stories and thus bear less importance than others.

```
PREFIX o: <http://ontoserv/project1#>
SELECT ?step ?steptype ?steptext WHERE {
    Type(?nle, o:StoryConcept),
    PropertyValue(?nle, o:hasName, "button"),
    PropertyValue(?nle, o:hasStepReference, ?step),
    PropertyValue(?step, o:hasTestImplementation, ?impl),
    PropertyValue(?impl, o:hasAttribute, ?attr),
    PropertyValue(?attr, o:hasAttributeValue, ?steptext),
    PropertyValue(?attr, o:hasName, ?steptype),
}
```

Listing 3. A SPARQLDL-Query for Test Steps that are Connected to the Natural Language Entity *button*.

In F-TRec, we use the term frequency $tf_{i,j}$ of a NLE i in a test step j and the inverse document frequency idf_i to weight the NLEs: The idf_i is calculated as $idf_i = log\frac{N}{n_i}$, where N is the number of all resulting test steps and n_i is the number of test steps that contain NLE i. Together with the term frequency the weight is $w_{i,j} = tf_{i,j} * idf_i$ (c.f. [16]). Since $tf_{i,j}$ is (almost) always one[2], we refine the weight by multiplying it with the NLE's frequency rank regarding the new User Story. We use the frequency rank instead of the plain frequency to reduce the impact of frequent NLEs further. Finally, the weights of all query NLEs, that are linked with a given test step, are summed up to represent its relevance score. Afterwards, we sort the result set according to the relevance scores.

IV. Preliminary Results

We conducted a small case study to examine the fitness of our approach in which we analyzed an industry project with approx. 300 KLOC of code, 10 KLOC of which are functional test code. In total, we examined 39 completed User Stories with associated test scripts. To test whether F-TRec can provide useful recommendations, we built the ontology for the project and the User Stories. The last seven User Stories belong to the most recent sprint and were taken as test examples. For every User Story, we created the ontology without the story's information; then we searched for relevant test steps. As the seven test examples are already completed and tested, we can determine if the retrieved test steps were actually used.

Table I shows the results of our case study. For every test step in the result set, we checked whether it is in the gold standard. Returned steps that were expected were counted as true positives (TP), otherwise as false positives (FP). Expected steps not contained in the result set need extra consideration. Steps, that are contained in the ontology, but not in the result set, were counted as false negatives (FN). Steps that were implemented especially for testing the given User Story are not contained in the ontology and therefore are missing in the result set. We counted these steps as *new*, as we cannot expect F-TRec to deliver them; also the new steps were not considered when determining recall.

[2]Test steps are short and it is unlikely that a NLE is used in a step more than once.

For the seven test stories, many of the reusable steps were found using the NLEs from the story. Additionally, F-TRec recommended test steps with high relevance scores that could have been considered for implementation reuse, but where implemented anew in the gold standard. This also supports the usefulness of F-TRec.

The number of retrieved relevant steps depends on the nature of a User Story. A story that describes an extension of existing functionality is more likely to be testable with many existing steps (e.g. program state setup steps). New functionality that barely builds upon existing code will not use many existing test steps. In this case, F-TRec cannot find many true positives. This is especially the case for the stories one, two, six, and seven. We expect this effect to decrease as the project matures and additional stories enhance or complement existing features without the need of implementing isolated or base functionality.

Weighting the test steps as described in Sect. III-C is too simplistic and needs to be improved in the future. Due to that, we do not cut off the result list by now and hence the precision is rather low.

Manually providing links between synonymous NLEs, we were able to improve the results. We added two synonym relations to the ontology which affected stories one and four. Marking these synonyms increased precision and recall by 138 percent and 65 percent respectively (in average for these two stories). It seemed that the identified synonyms were not used to make the specification more "readable", but that the cause of synonym usage lies in the different vocabularies of the stakeholders – especially those with different roles in the development process. For example, a developer might talk about a *plot canvas*, while a user refers to a *plot window*. This finding calls for the usage of a glossary which not only defines a common vocabulary but also can be leveraged during step retrieval. As synonyms of technical and domain-specific terms are hard to detect, domain experts could help in maintaining the glossary during development.

Also, the handling of compound nouns can be improved. For example, the compound noun *data selection view* would only be linked with stories if they use the same compound noun. If the context of data selection is known to a human reader, the concept could simply be referred to by *view* without prepending *data selection* explicitly. Then F-TRec cannot establish a connection between the query NLE *view* and the actual concept *data selection view*.

V. Conclusion and Future Work

We proposed a novel approach to recommend reusable test steps to testers and developers. F-TRec is a prototypical implementation and populates an ontology with source code and natural language elements of User Stories and test scripts. Using this ontology, F-TRec retrieves relevant reusable test steps for new User Stories; besides the actually

Table I
RESULTS OF THE CASE STUDY.

Story	Test Steps				Recall	Precision
	TP	FP	FN	new		
1	7	55	3	9	70 %	11.3 %
2	0	0	0	4	-	-
3	3	33	0	0	100 %	8.3 %
4	3	63	4	3	42.9 %	4.6 %
5	16	71	2	0	88.9 %	18.4 %
6	1	41	0	9	100 %	2.4 %
7	1	63	0	2	100 %	1.6 %
Totals	**31**	**326**	**9**	**31**	**77.5 %**	**8.68 %**

used test steps, we were able to identify steps that could have been reused but were reimplemented.

Even though first results are encouraging, much work remains: The precision of our retrieval is rather low – we plan to use context information of User Stories to further improve the ranking of retrieved steps (e.g. whether a story deals with the user interface or with calculation functions). Steps that stem from stories in the same context could be more relevant than others. We saw, that there are "common steps" that are used in many test scripts (especially preconditions); our current approach penalizes these steps – even if they are relevant for a query. Also, we will take the steps' type into consideration; further studies will show, whether precision can be improved if one concentrates on a specific step type.

We plan to extend the NLE list to be a fully fledged glossary. Using the glossary during the writing of new stories and test scripts, F-TRec could urge product owners and testers to use a consistent vocabulary.

In terms of result presentation, we also can improve F-TRec. The presentation of the result list should be grouped by the step type to ease readability. Since the test steps are directly linked to the test step implementations, we could evaluate their API calls and source code comments. Comments could be provided to testers in order to assess whether the recommended test steps are reusable. Developers could additionally profit from links to API components, that are likely to be of interest when implementing test steps.

ACKNOWLEDGEMENTS

Agilent Technologies kindly provided us with code and requirements of the analyzed project. Adrian Genaid is supported by Agilent Technologies.

REFERENCES

[1] M. Cohn, *User Stories Applied: For Agile Software Development*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[2] D. North, "Introducing BDD," Published in Better Software magazine, Mar. 2006, accessed: 03/23/2012. [Online]. Available: http://dannorth.net/introducing-bdd/

[3] D. Chelimsky, *The RSpec book : behaviour-driven development with RSpec, Cucumber, and Friends*, 1st ed. Raleigh, NC: The Pragmatic Bookshelf, Dec. 2010.

[4] "Specflow – pragmatic BDD for .NET," accessed: 03/26/2012. [Online]. Available: http://www.specflow.org

[5] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, Apr. 1993.

[6] W3C, "OWL 2 Web Ontology Language," Oct. 2009, accessed: 03/23/2012. [Online]. Available: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

[7] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, and Y. Katz, "Pellet: A practical OWL-DL reasoner." *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[8] W3C owl working group, "List of OWL reasoners," Nov. 2011, accessed: 03/23/2012. [Online]. Available: http://www.w3.org/2007/OWL/wiki/Implementations&oldid=26281

[9] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL query for OWL-DL," in *In 3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007.

[10] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "An ontology-based approach for traceability recovery," in *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006)*, 2006, pp. 36–43.

[11] N. Khamis, J. Rilling, and R. Witte, "Generating an NLP Corpus from Java Source Code: The SSL Javadoc Doclet," in *New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 22 2010, pp. 41–45.

[12] N. Khamis, R. Witte, and J. Rilling, "Automatic Quality Assessment of Source Code Comments: The JavadocMiner," in *Proc. of NLDB 2010*, 2010, pp. 68–79.

[13] J. Euzenat and P. Shvaiko, *Ontology matching*, 1st ed. Springer-Verlag, 2007.

[14] D. Klein and C. D. Manning, "Fast exact inference with a factored model for natural language parsing," in *In Advances in Neural Information Processing Systems 15*. MIT Press, 2003, pp. 3–10.

[15] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *In ACL*, 2005, pp. 363–370.

[16] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, 2nd ed. Prentice Hall, 2009.