Ontologies + Mock Objects = Runnable Knowledge

Iaakov Exman, Anton Litovka and Reuven Yagel

Software Engineering Department, The Jerusalem College of Engineering - Azrieli
POB 3566, Jerusalem, 91035, Israel

Abstract. Ontologies constitute the highest abstraction level in software systems. But in order to obtain runnable and testable software knowledge we can supplement ontologies with mock objects. This work formulates a basic Generation Algorithm to actually obtain testable high level code. The Generation Algorithm has been implemented in a system by means of existing tools. The approach has been tested with several case studies. We then examine fundamental issues, say whether the supplementary mock objects are essential in all cases, or can be substituted by *perfect* ontologies.

Keywords. Ontologies, Mock objects, Runnable knowledge, Testing, Software knowledge, Generation algorithm.

1 Introduction

Our work is based upon a few assumptions. First, any software system can be described by a set of hierarchical abstractions, in which the level of abstraction increases in a bottom-up direction. Second, the highest level of abstraction is a set of ontologies, from which one can derive the next level, viz. a UML design model. This assumption is the basis of what we call KDE (Knowledge Driven Engineering) [29].

We have been developing a KDE infrastructure [12], [29], in which we have gradually solved theoretical and practical problems of design and implementation.

In this paper, we specifically deal with the issue of supplementing classes and objects derived from the ontologies, by means of mock objects, to allow actually running and locally testing a high-level code/design for a software system.

There are two possible ways to arrive at the starting set of ontologies for a given software system. Either one manually proposes the set as we have done in [29], or one tries to derive the system ontologies from domain ontologies, as recently investigated. In both ways, one often gets an incomplete set of ontologies that must be supplemented by mock objects to be run and tested.

In the remaining of the paper we formulate the basic Generation Algorithm of runnable and testable code starting from ontologies and scenarios (section 2), overview the system which implements the Generation Algorithm (section 3), illustrate the approach with case studies (section 4) and conclude with a discussion (section 5).

Exman I., Litovka A. and Yagel R..
Ontologies + Mock Objects = Runnable Knowledge.
DOI: 10.5220/0005178500710078
In *Proceedings of the 5th International Workshop on Software Knowledge* (SKY-2014), pages 71-78
ISBN: 978-989-758-051-2

1.1 Related Work

Ontologies in the software context are reviewed by Calero et al. [7]. Several papers about ontology-driven software development are collected in Pan et al. [20]. Model Driven Engineering with ontology technologies is discussed in Parreiras [21].

In recent years, testing methods were put into wide practice through the works of Beck [4] end others. New terms were coined, such as TDD (Test Driven Development). In these methods, scripts demonstrate system behaviors, besides specifying the interface. Scripts' execution can be automated, thus called automated functional testing. The Agile software movement has emphasized early testing methods, e.g. Freeman and Pryce [13]. Its main goal is faster understanding of the software being developed by means of short feedback loops.

Scenarios appear naturally in various methods that implement TDD practices and their extensions. FitNesse by Martin (cf. [1]) is a wiki-based web tool for non-developers to write formatted acceptance tests, viz. tabular example/test data.

BDD (Behavior Driven Development) (North [18]), is an extension aiming at understanding requirements by stakeholders. The Cucumber (Wynne and Hellesoy [27]) and SpecFlow [26] tools accept stories written in a somewhat constrained natural language, directly supporting BDD. They are easily integrated with unit testing and user/web automation tools. Yagel [28] reviews these practices and tools.

ATDD (Acceptance Test Driven Development), is another extension of TDD, also known as Agile Acceptance Testing (Adzic [2]). Further extensions are Story Testing, Specification with examples (Adzic [3]) or Living/Executable Documentation (Brown [6], Smart [25]).

2 The Generation Algorithm

The Generation Algorithm which generates runnable knowledge test scripts [29] is composed of two intermingled techniques: ontologies as sources of classes, objects and their properties; scenarios which are run in a step-wise fashion.

Overall Generation Algorithm

The overall structure of the Generation Algorithm in our system is a set of loops, starting with ontologies and scenarios provided by the user.

The first important step is to look for term matches between the ontologies and the scenarios. If there is no ontology match for scenario objects, mock objects are generated for them. Mock objects have the same interfaces as real objects, but lack real implementation: their outputs are synthetically obtained, just to enable testing.

Then, for each step in the scenario's order one looks for the following parts:

- *Given* – an initial context;
- *When* – an event that occurs;
- *Then* – specific outcomes that must occur.

Finally, the algorithm fills the test steps with generated code.

The overall Generation Algorithm is displayed as pseudo-code in Fig. 1.

```
Run the Program with given Ontology and Scenarios.
    Run  Tool for running tests on scenarios file and save output
    Search for matches between terms from the Ontology and the scenarios.
    for each found match in a concrete step search for properties of those terms.
    for each step gotten from test tool
        Find objects which are not included in ontology.
        Generate mock objects for them.
    for each step returned by test tool in scenario's order
        if part is "Given"
                for each found objects
                        Generate global instance of objects.
                        Update attributes values, call to function using parameters
                                                        and generated mock objects.
        if part is "When"
                for each object that is not common in Given part of Scenario
                        Generate new instance.
                        Update attributes values, call to function using parameters
                                                        and generated mock objects.
                for each object that is common in Given part of Scenario
                        Update attribute values, call to function using parameters,
                          generated mock objects and objects created in this step.
        if part is "Then"
                for each object that is not common in Given part of Scenario
                        Generate new instance.
                        Update attributes values, call to function using parameters
                                                        and generated mock objects.
                for each global objects found
                        Check object attribute values and methods' returned values
                         using given step parameters and created in this step objects.
    Fill test steps with generated code.
    Create test files.
End.
```

Fig. 1. *Generation Algorithm* – displayed as pseudo-code.

We now focus on the ontologies and mock objects of the Generation Algorithm, as seen in Fig. 2. One observes that the main interactions between ontologies and scenarios in the Generation Algorithm are to check their consistency, expressed by the shared terms. Any lack of consistency is solved by adding mock-objects.

```
Run the Program with given Ontology, Scenarios.
    Run  Tool for running tests on scenarios file and save output
    Search for matches between terms from the Ontology and the scenarios.
    for each found match in a concrete step search for properties of those terms.
    for each step gotten from test tool
        Find objects which are not included in ontology.
        Generate mock objects for them.
```

Fig. 2. *Generation Algorithm* focused on ontologies and scenarios. In bold blue fonts the positive actions shared among ontologies and scenarios. In bold red fonts the non-shared terms, causing generation of mock objects.

3 Runnable Software Knowledge Generation System

Here we concisely describe the Generation System that outputs the runnable Software Knowledge. First we refer to the system design, then to its implementation.

The Generation System is composed of especially developed classes with existing external tools. The desing is shown as a UML class diagram in Fig. 3. The main class, 'Steps' alludes to the main loop steps of the Generation Algorithm.

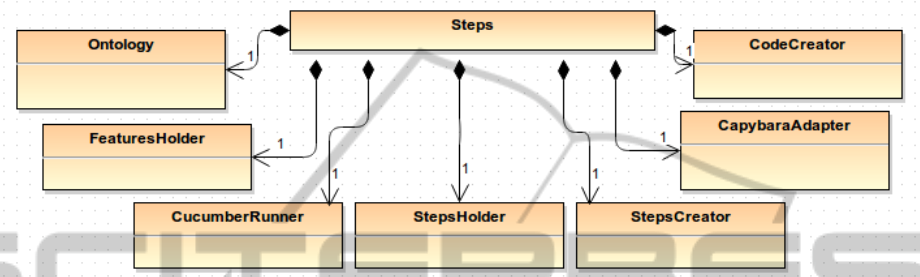

Fig. 3. *Generation System* class diagram – 'Steps' is the main class. The external tools are Cucumber and Capybara. Special classes deal with: ontologies, steps, features and code creator.

The generic classes in the Generation System are as follows: a- *Ontology* – it inputs and reads ontology files, finds matching terms in ontologies and in scenarios; b- *FeaturesHolder* – manages test files and connects between their lines and steps; c- *CodeCreator* – generates the tests' code for individual matches; d- *StepsHolder* – manages the generated tests' code; e- *StepsCreator* – distributes the generated code lines within steps.

The classes related to the external tools are: f- *Cucumber runner* – executes Cucumber and analyses its output; g- *Capybara adapter* – finds matches with methods that control UI.

The Generation System was implemented with the Ruby language. Ruby is a dynamic, object oriented, cross-platform and interpreted language with open source code [10]. Ruby was chosen due to several important reasons such as high development rate, rich meta-programming possibilities, and being supported by a large set of public available solutions. A most important factor is the initial orientation of the Ruby community toward development through behavior (BDD), since BDD is the basis of this work.

Different (gem) libraries supplied by the Ruby community were used. One of them is the Nokogiri library for xml files [17], [22], here used for reading ontologies.

The Generation System itself is implemented as an additional (gem) library designed for the MVC-Framework Ruby on Rails [14]. The tests are generated using APIs from the RSpec [18], and Capybara [8] libraries.

4 Case Studies

We have tested our system by various case studies. Two of them are presented here:

1st the “email account login” case study, fully compatible with the input ontology. 2nd the “post-to-blog” case study, demanding an extra mock object.

4.1 The Email Account Login Ontology

The account login system ontology is seen in Fig. 4. It is a print-screen from the ontology editor tool Protégé [30].

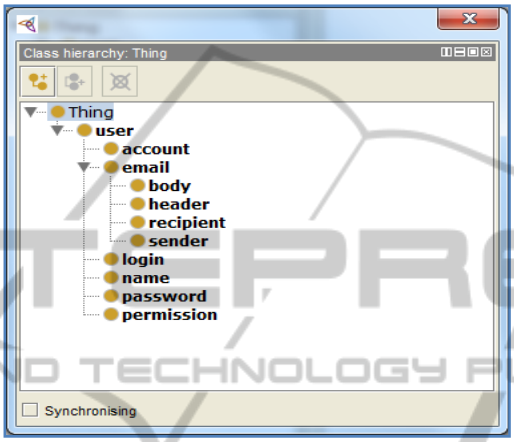

Fig. 4. Email Account Login Ontology – It displays a class hierarchy, in which ‘thing’ is the top concept (class). The next sub-class is ‘user’, which in turn has account, email, login, name, password and permission sub-classes. Email has 4 sub-classes.

```
Scenario: Successful login
    Given the following user records
      | name  | email              | password | password_confirmation|
      | alice | alice@gmail.com    | secret   | secret               |
    Given I am on the signin page
    Given I fill in "session_email" with "alice@gmail.com"
    Given I fill in "session_password" with "secret"
    When I press "Log in"
    Then I should see "Welcome alice"
```

```
Generated Test: Given /^the following user records$/ do |table|
                  table.hashes.each do |hash|
                    User.create(hash).save
                  end
                Given /^I on the signin page$/ do
                    visit '/signin'
                  end
                Given /^I fill in "(.*?)" with "(.*?)"$/ do |arg1|
                    element = page.find_by_id(arg1)
                  end
                When /^I press "(.*?)"$/ do |arg1|
                    click_on arg1
                  end
                Then /^I should see "(.*?)"$/ do |arg1|
                    page.should have_content(arg1)
                  end
```

Fig. 5. Successful Account login sample – Above, the scenario written in a Gherkin specification. Below, the corresponding generated test.

A successful login case is shown in Fig. 5. Here, all the scenario terms, viz. name, email, password, are found in the ontology. So, there is no need for any mock object.

4.2 Post-to-Blog: Added Mock Objects

Here we consider a scenario for post-to-blog, as shown in Fig. 6. In this second study case, the term "Text" found in the scenario does not match any term in the ontology. Therefore, a mock object is needed to supplement the ontology terms.

```
Scenario: Post-to-Blog
Given I have written Text
And Text name is "Hello"
And Text contains is "Hello World"
And I on the article page
When I fill in "article_title" with Text name
And I fill in "article_body" with Text contains
When I press "Publish"
Then I should see "Article was successfully created"
```

Fig. 6. Successful Post-to-Blog sample – also written in a Gherkin specification.

The corresponding generated test for the above scenario is shown in Fig. 7.

```
Generated Test: Given /^I have written Text$/ do
                  @text = mock('Text')
                end
              And /^Text name is "(.*?)"$/ do |arg1|
                  allow(@text).to receive(:name).and_return(arg1)
                end
              And /^Text contains is "(.*?)"$/ do |arg1|
                  allow(@text).to receive(:contains).and_return(arg1)
                end
              And /^I on the article page$/ do
                  visit '/article'
                end
              When /^I fill in "(.*?)" with Text name$/ do |arg1|
                  element = page.find_by_id(arg1)
                  element.set(@text.name)
                end
              And /^I fill in "(.*?)" with Text contains$/ do |arg1|
                  element = page.find_by_id(arg1)
                  element.set(@text.contains)
                end
              Then /^I should see "(.*?)"$/ do |arg1|
                  page.should have_content(arg1)
                end
```

Fig. 7. Successful Post-to-Blog generated test – corresponding to the scenario in Fig. 6. But note the mock object inserted in the 2nd line (marked by bold red font).

5 Discussion

This work has described a system to implement the Generation Algorithm to produce Runnable Software Knowledge that is testable. It starts from sets of ontologies and scenarios, and has recourse to mock objects if there are terms in the scenario that do not match any ontology term.

Some of the external tools, such as Capybara, may contain concepts in addition to those in the ontologies' input set. In principle, the concepts of the external tools are based on additional ontologies in the tool domain. These additional ontologies could be explicitly input to the Generation System, but it would be a superfluous repetition.

The source code for the described tool can be found here [15].

5.1 Future Work: Perfect Ontologies or Essential Mock Objects?

A deep issue to be investigated is whether one can in general formulate *perfect* ontologies, including all possible terms found in the scenarios. Or the most common situation is that all input ontologies are typically incomplete, and must be supplemented by mock objects.

It also could be the case that *perfect* ontologies would be so extensive, as to demand an excessive efficiency price.

Other issues of interest open to investigation are: a- extensive examples of a variety of software systems and their respective sets of ontologies and scenarios; b- systematic composition of larger systems from smaller components that have already been tested by the current approach.

5.2 Main Contribution

The main contribution of this work is an approach to Knowledge Driven Engineering of software systems from the highest abstraction levels. The approach uses only ontologies, scenarios and mock objects, which constitute a sort of complete input set to software development.

References

1. Adzic, G., Test Driven .NET Development with FitNesse, Neuri, London, UK, 2008.
2. Adzic, G., Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri, London, UK, 2009.
3. Adzic, G., Specification by Example – How Successful Teams Deliver the Right Software, Manning, New York, USA, 2011.
4. Beck, K., Test Driven Development: By Example, Addison-Wesley, Boston, MA, USA, 2002.
5. Boehm, B.W.: "Software Engineering Economics", IEEE Trans. Software Eng., 1984.
6. Brown, K., Taking executable specs to the next level: Executable Documentation, Blog post, (see: http://keithps.wordpress.com/2011/06/26/taking-executable-specs-to-the-next-level-executable-documentation/), 2011.
7. Calero, C., Ruiz, F. and Piattini, M. (eds.): Ontologies in Software Engineering and Software Technology, Springer, Heidelberg, Germany, 2006.
8. Capybara Library - https://github.com/jnicklas/capybara.
9. Chelimsky, D., Astels, D., Dennis, Z., Hellesoy, A., Helmkamp, B., and North, D,: The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends, Pragmatic Programmer, New York, USA, 2010.

10. Cooper, P.: Beginning Ruby: From Novice to Professional,II ed., Apress, Berkeley, USA, 2009.
11. Exman, I, Llorens, J. and Fraga, A.: "Software Knowledge", pp. 9-12, in Exman, I., Llorens, J. and Fraga, A. (eds.), Proc. SKY'2010 Int. Workshop on Software Engineering, 2010.
12. I. Exman and R. Yagel, "An Approach to Self-consistency Verification of a Runnable Ontology Model", in Fred, A., Dietz, J.L.G., Liu, K. and Filipe, J. (eds.) Knowledge Discovery, Knowledge Engineering and Knowledge Management, Revised Selected Papers of the 4th Int. Joint Conference, IC3K'2012, Barcelona, Spain, October 2012, pp. 271-283, Vol. 415 of Communications in Computer and Information Science, Springer Verlag, Berlin, 2013. DOI = 10.1007/978-3-642-54105-6_18.
13. Freeman, S., and Pryce N.: Growing Object-Oriented Software, Guided by Tests, Addison-Wesley, Boston, MA, USA, 2009.
14. Hartz, M.: Ruby on Rails Tutorial: Learn Web Development with Rails, online tutorial at http://www.railstutorial.org/book, 2013.
15. Test Generation Tool, 2014: https://github.com/AntonLitovka/tgt
16. Moq – the simplest mocking library for .NET and Silverlight: (see http://code.google.com/p/moq/), 2012.
17. Nokogiri –an HTML, XML, SAX, and Reader parser at http://nokogiri.org/.
18. North, D.: "Introducing Behaviour Driven Development", Better Software Magazine, (see http://dannorth.net/introducing-bdd/), 2006.
19. NUnit: (see http://www.nunit.org), 2012.
20. Pan, J.Z., Staab, S., Assmann, U., Ebert, J. and Zhao, Y. (eds.): Ontology-Driven Software Development, Springer Verlag, Heidelberg, Germany, 2013.
21. Parreiras, F.S.: Semantic Web and Model-Driven Engineering, John Wiley, Hoboken, NJ, and IEEE Press, USA, 2012.
22. Powers, H.: Instant Nokogiri, Packt Publishing, Birmingham, UK, 2013.
23. Robbins, M.: Application Testing with Capybara, Packt Publishing, Birmingham, UK, 2013.
24. RSpec mocks library: (see: https://github.com/rspec/rspec-mocks), 2013.
25. Smart J. F.: BDD in Action Behavior-Driven Development for the whole software lifecycle, Manning, 2014 (expected).
26. SpecFlow – Pragmatic BDD for .NET: (see http://specflow.org), 2010.
27. Wynne, M. and Hellesoy, A.: The Cucumber Book: Behaviour Driven Development for Testers and Developers, Pragmatic Programmer, New York, USA, 2012.
28. Yagel, R.: "Can Executable Specifications Close the Gap between Software Requirements and Implementation?", pp. 87-91, in Exman, I., Llorens, J. and Fraga, A. (eds.), Proc. SKY'2011 Int. Workshop on Software Engineering, SciTePress, Portugal, 2011.
29. Yagel, R., Litovka, A. and Exman I.: KoDEgen: A Knowledge Driven Engineering Code Generating Tool, The 4th International Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K) - SKY Workshop, Vilamoura, Portugal, 2013.
30. Protégé. A free, open-source ontology editor and framework for building intelligent systems, http://protege.stanford.edu/