Improving the Gherkin Specification Language using Aspect-Oriented Techniques

John Aquilina Alamango[1], Christian Colombo[1], and Mark Micallef[1]

University of Malta

In the highly dynamic markets in which software customers operate, it is crucial that the software development process is able to incorporate the customers in the feedback loop, supporting the evolution of specifications and ensuring that software is according to the customers' requirements.

A specification language frequently used for this purpose is Gherkin [1] — a very simple language with three main keywords: *Given*, *When*, *Then*. The semantics of these keywords are loosely defined as *given some precondition, when a particular event occurs, then some postcondition is expected to hold*. Other than these three keywords, the specification writer can use natural language and it is then up to the developer to translate the specification into executable tests.

While Gherkin has become a mainstream tool in contemporary software development, one cannot help but notice a significant problem: the simplicity and flexibility which have made Gherkin so popular amongst non-technical industrial customers, are the same aspects which makes the resulting scripts highly repetitive. Each given-when-then statement — known as a *scenario* — will typically be written over and over again albeit with just a different parameter.

```
Feature: Testing Country Charges

Scenario: Test Incoming Phone Call Fee
Given I am receiving a phone call from Australia
When I answer the call
Then I should charged at 0.00c per minute

Scenario: Test Incoming Phone Call Fee
Given I am receiving a phone call from United Arab Emirates
When I answer the call
Then I should charged at 0.00c per minute
```

After having considered a number of industrial case studies, we noted two main repeating patterns: (i) As in the above example, a number of scenarios where identical except for one or more parameters (ii) In other cases, apart from the parameters, a scenario might have one or more extra postconditions which are not present in the other scenarios.

In view of the identified recurring patterns in Gherkin scripts, our aim is to constrain the flexibility of Gherkin just enough so that the user can be afforded just enough automatic support in the writeup of the scripts. In the rest of the paper we describe our approach in tackling the two identified issues.

Set theory to the rescue Our first part of the solution is to introduce the notion of collections of objects — sets — enabling the script writer to collate related objects into a set and then allowing one to refer to all the individual set elements at once. Furthermore, we observed that using set operators, one could define sets as follows:

```
EUCountry = {Malta, UK}
NonEUCountry = {Australia, United Arab Emirates}
AnyCountry = EUCountry + NonEUCountry
```

Resulting in reduced repetition in the definition of the scenarios:

```
Scenario: Test Incoming Phone Call Fee
Given I am receiving a phone call from <NonEUCountry>
When I answer the call
Then I should charged at 0.00c per minute
```

Aspect-oriented programming [2] While set theory concepts greatly helped in structuring Gherkin specifications, a number of examples couldn't benefit because the scenarios of the set elements would gave a single line which would be different. To this end, we propose an aspect-oriented programming extension to Gherkin, allowing the injection of the extra line to be done automatically. For instance, if a entry should be added in the log for every call (irrespective of whether it originates from a *NonEUCountry* or an *EUCountry*), then using the code below, we would be checking the log for every scenario without adding extra lines.

```
Matching: {"When I answer the call"}
[ Then call should be registered in call log ]
```

When we applied the proposed set-theory-extended-Gherkin to our industrial case studies, in three out of the four cases[1] we observed a significant reduction in the size of the script without any major impact on their readability. Perhaps more contentious is the aspect-oriented programming extension where more than one script file would have to be consulted in order to read a single scenario. This however could be alleviated with the right tool support where any matching aspects could be highlighted whilst browsing the corresponding scenario in another window. In the future we aim to provide such an editor which would ultimately enable the user to go from the Gherkin flavour presented here to the original Gherkin, providing backward compatibility while making the task of script writing easier.

References

1. Gherkin wiki. Accessed: 2013-12-8.
2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327âĂŞ–354, 2001.

[1] Two of the industrial case studies were performed at Ascent Software and Betclic respectively, while the other two were performed at companies who do not wish to disclose their identity.