Modelling Gherkin Scenarios Using UML

Javier J. Gutiérrez *javier@us.es*
University of Seville
Sevilla, Spain

Isabel Ramos *iramos@us.es*
University of Seville
Sevilla, Spain

Manuel Mejías *risoto@us.es*
University of Seville
Sevilla, Spain

Carlos Arévalo *carlosarevalo@us.es*
University of Seville
Sevilla, Spain

J.M. Sánchez-Begines *juan.sanchez@iwt2.org*
IWT2 Research Group
Sevilla, Spain

David Lizcano *david.lizcano@udima.es*
Universidad a Distancia de Madrid
Madrid, Spain

Abstract

Gherkin scenarios are examples of the behavior of the system under development. They may be part of the requirement specification, they may be part of the test suite and they are an excellent tool for gathering information among stakeholders, testers and developers. However, little work have been done formalizing Gherkin scenarios and modelling them as part of UML diagrams. This paper introduces an abstract syntax and concrete syntax for modeling Gherkin scenarios in UML Use Case diagrams. This paper also introduces a tool for running Gherkin scenarios from UML Use Case diagrams as test cases.

Keywords: Gherkin, scenario, use case, UML, Model-Driven Development.

1. Introduction

Several friends love to play online. One day they have a great idea that will change the world (or at least their world). They assembly together for building an online tournament system for competing among them. They love the idea, they are full of motivation and they start immediately. Figure 1 depicts some of their first requirements as user stories.

As a tournamet player
I want to submit match results
In order to update my clasification

As a tournamet player
I want to see my calendar
In order to discover my next rival

As a tournamet player
I want to see the clasification
In order to see my position

Fig 1. User stories for the tournament system,

They don't realized that they are not working in the same project (figure 2). What is a tournament system? How do their tournament works? Have they the same idea? Maybe some of them think in a tournament system like a soccer league system where all team play among them and points are obtained winning matches. Other friends could think in a tournament system like American basketball (NBA) or American football (NFL) leagues where groups of teams play among them to enter in the playoff for the championship. Other friends could think in a bracket system like a tennis tournament. And, finally, other friends are not thinking in these questions and they are focused on sign-up of teams or players management.

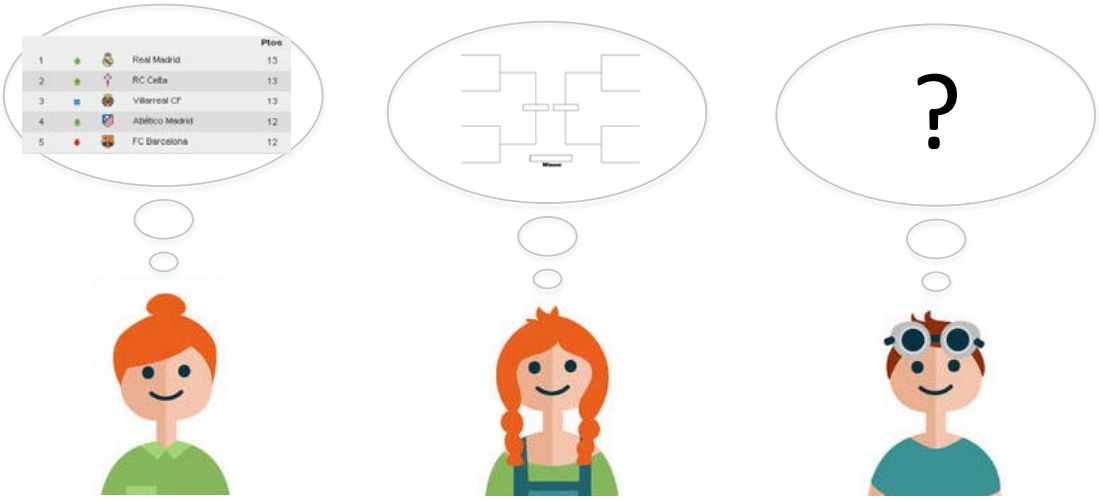

Fig 2. One team. Same project?

Requirement engineering is about communication and collaboration. Communication breakdowns are cited in [2] as one of the challenge for agile requirement engineering. Previous history exposes a common case of miscommunication. Every friend thinks she knows the system but each one could be developing a different system.

There are several approaches for bridging communication gaps: requirement workshops, prototyping, etc. This paper uses one of this approach: Gherkin scenarios. A Gherkin scenario is the answer for: "give me an example" using the Gherkin syntax [1]. This syntax is introduces in next section. Gherkin scenarios are useful for [12]:

1. Achieving a better understanding of the functional requirement to implement
2. Bridging the communication gap between stakeholders, users, testers and developers.
3. Being implemented as test cases using tools like Cucumber [15].

Let's imagine that we can explain this technique to the band of friends and they start to ask for examples, like: "give me an example about a player winning a game", or "give me an example of a draw among players". These questions must be answered with details of how the Tournament system should work. No chance for hiding details, so Gherkin scenarios help the team to communicate among them and also helps the team to think about which is the right behavior for the system under development. With these answers, team may share a common vision of the system.

This paper introduces an abstract syntax and a concrete syntax for modelling Gherkin scenarios together with UML Use Cases [14]. An abstract syntax, also called metamodel, describes the structure and rules of a language for defining models [5]. A concrete syntax indicates the elements for defining models using the concepts from the abstract syntax. the principle contributions of this article are as follows:

- Formalization of Gherkin syntax in a metamodel.
- A notation for including Gherkin scenarios in UML Use Case diagrams.
- A supporting tool for running Gherkin scenarios as test cases.

This paper is organized as follows. Section 2 describes related works. Section 3 introduces the Gherkin syntax for scenarios and some examples of Gherkin scenarios modeled using plain text. Section 4 introduces an abstract syntax for Gherkin scenarios and a concrete syntax for including scenarios in UML Use Case diagrams. Then, section 5 introduces a supporting tool for running the Gherkin scenarios as test cases from a UML Use Case diagram. Finally, section 6 exposes conclusions and future work.

2. Related works

Little attention has been paid to Gherkin scenarios despite the high interest in agile requirements and testing. For example, The Schön, Thomaschewski and Escalona survey about agile requirements [11] reports that a 56% of papers found are related to user stories but only a 22% are focus on scenarios. Little attention have also been paid to Gherkin scenarios from a MDE perspective. Next paragraphs cite works related with the contributions introduced in section 1.

Paper [3] describes a set of rules and patterns for translating UML Statecharts diagrams into Gherkin Scenarios. Scenarios obtained include a high number of steps and they mix several groups of Given-When-Then sections, so they are more focused on testing than in requirement elicitation.

Paper [8] describes an experiment using a formal notation (UML Sequence diagram), a semi-structured natural-language notation (Gherkin) and an extension to a fully-structured language for model management tasks (Epsilon). Paper reports that best results were achieved using Gherkin.

Paper [4] introduces a metamodel for user stories. This metamodel is focused in the semantic of the elements and it classified this elements into roles (similar to actors), tasks (actions performed by an actor), Hard-goals (a goal defined in the domain of the business), Capabilities (functionality of the system that is needed to commit a business goal) and soft-goals (a goal defined as the consequences of hard-goal). Relations among these elements and the elements from UML diagrams are also defined. This metamodel and its relations with UML are implemented in the Descartes architect CASE-tool which allows to generate a User Story view, a Use Case view and a Class, Sequence and Activity diagram views automatically.

Paper [12] introduces a semi-automated process for generating the implementation of the steps from scenarios using an analysis of the scenario itself. For example: nouns are transformed into classes, verbs into methods and the code of the step is generated using those classes and scenarios.

Paper [10] has the same goal than paper [12]. However this paper performs a lexical analysis of the scenarios for generating a code scaffolding of test

Next section introduces Gherkin syntax.

3. A brief introduction to Gherkin

Gherkin is a syntax for modelling examples of the realization of functional requirements [12]. These examples are called scenarios and they includes specific information needed for a real execution of functional requirements. Gherkin was created as one part of the Cucumber tool. The main goal of Gherkin is to define examples of the behavior of the system using a non-technical notation for stakeholders and business people. Figure 3 shows an example of Gherkin scenarios for the first user story in figure 1 (“submit match result” story).

```
4   Feature Submit match result
5       As player named Kol
6       I want to submit the result of matches
7       In order to update my classification
8
9   Background:
10      Given a calendar season where all players play among them
11      And three players called Kol, Tis and Aka
12
13  Scenario: Update results
14      Given First match of the season is between Kol and Tis
15      And A result for that match of 2-1 (best of three)
16      When "Kol" introduces "2" wins for him and "1" fot "Tis"
17      Then "Kol" is in the classification with "2" wins
18      And "Tis" is behind him with "1" victory
19
20  Scenario: Climbing the classification
21      Given "Kol" has 2 wins in the classification
22      And "Kol" plays a match against "Aka"
23      When "Kol" introduces "1" win for him and "2" wins for "Aka"
24      Then "Kol"'s classification is updated with "3" wins
25
26  Scenario: final classification
27      Given "Kol" results are "2-1" and "1-2" against "Tis" and "Aka"
28      And "Tis" result is "0-3" against "Aka"
29      When any of the players see their classfication
30      Then "Aka" is "first" with "5" wins and "1" lose
31      And "Kol" is "second" with "3" wins and "3" loses
32      And "Tis" is "third" with "1" wins and "5" loses
```

Fig 3. Gherkin scenarios.

Gherkin scenarios are concrete examples of the expected behaviour. Therefore, scenarios include concrete values, users and results. If you review the Gherkin scenarios in figure 3, you will discover details about how the tournament system work. In this example, the tournament system works as a soccer league. System stores victories and losers of every player (scenario update results). Classification is driven by the number of victories. Draws among players are resolved using the number of loses (scenario final classification). No information is provided about players with same number of victories and loses, which mean that we have an opportunity to write a new scenario (or several ones) to explain the behaviour of the system in this case.

Gherkin uses a basic grammar with a little number of reserved words. Reserved words are listed in table 1 and they are described in next paragraphs.

Table 1. Gherkin reserved words

Feature, Scenario, Given, When, Then, And, But, Background, Scenario Outline, Examples

A feature is a piece of behaviour of the system under test. A feature may be written using the classic pattern of "As..I want..In order.." (stories from figure 1 use this pattern), however, a feature is bigger than one iteration and it needs to be sliced into user stories [15]. A scenario

is an example of the realization of the behaviour described in a feature. A scenario is composed of a list of any number of steps.

Gherkin syntax defines three types of steps: Given, When and Then. Given steps defines the state of the system before the actor starts interacting with the system. When steps describe the key action that the user performs. Then steps observe outcomes. These observations should be some kind of output related to the business value/benefit of the feature.

Background is an optional section that defines a context to the scenarios using Given steps. The background is executed before the realization of each of the scenarios. An alternative syntax for a scenario may be used with reserved words: “scenario outlines” and “examples”. A "Scenario outline" scenario allows to define several set of concrete values in an “examples” section. In the example from figure 4, scenario outline “Climbing the classification” could be tested three times, one for each line with in the examples section.

```
38  Scenario Outline: Climbing the classification
39      Given "Kol" is "first" with "2-1"
40      And "Tis" is "second" with "1-2"
41      When "Kol" plays against "This" and result is <Result>
42      Then "Kol" is <Kol_Pos> with <Kol_Result>
43      And "Tis" is <Tis_Pos> with <Tis_Result>
44
45  Examples:
46      | Result | Kol_Pos | Kol_Result | Tis_Pos  | Tis_Result |
47      | "2-1"  | "first" | "4-2"      | "Second" | "2-4"      |
48      | "1-2"  | "first" | "3-3"      | "Second" | "3-3"      |
49      | "0-3"  | "second"| "2-1"      | "First"  | "4-2"      |
```

Fig 4. Scenario outline example.

Reserved words “And” and “But” (also from table 1) are syntactic sugar. They indicate the same type of step than the previous one.

Next section introduces one of the main original contributions of this paper: abstract and concrete syntaxes for the Gherkin syntax.

4. Abstract and concrete syntaxes for Gherkin Scenarios

As seen in section 1, the abstract syntax defines the elements of the Gherkin syntax, their semantic and their relations and a concrete syntax defines a notation for modeling Gherkin scenarios. Previous section introduced the semantic of the Gherkin elements and a concrete syntax for modelling scenarios as plain text (figure 3). This section formalizes the elements of the Gherkin syntax (table 1) in an abstract syntax compatible with the concrete syntax of plain text and, then, this sections introduces a second concrete syntax for graphic diagrams.

4.1. Abstract syntax for Gherkin Scenarios

Previous section has introduced the Gherkin elements. This section defines and abstract syntax for those elements. This abstract syntax formalizes the information and relations among the elements. As seen in section 1, abstract syntax sets the building blocks and the construction rules that any concrete syntax must follow. A concrete syntax indicates how to represent the elements and relations defined in the abstract syntax. Therefore, the abstract syntax is the base for defining concrete syntax. The abstract syntax for Gherkin scenarios is introduced in figure 5.

As seen in previous section, a feature is performed by one actor (association with roles performedBy-as from figure 5). Role "as" in the association indicates that an actor is the "as" part of the feature template (see example from figure 3). A feature may contains a background

with a set of Given steps for defining de context (associations Feature-Background with role context and Given-Background with role beforeScenarios from figure 5). A feature also contains a collection of scenarios with examples of the behavior of the feature. Features with scenarios are valid for the metamodel in figure 5 if development team is using features only for documenting the functional requirements of the system.

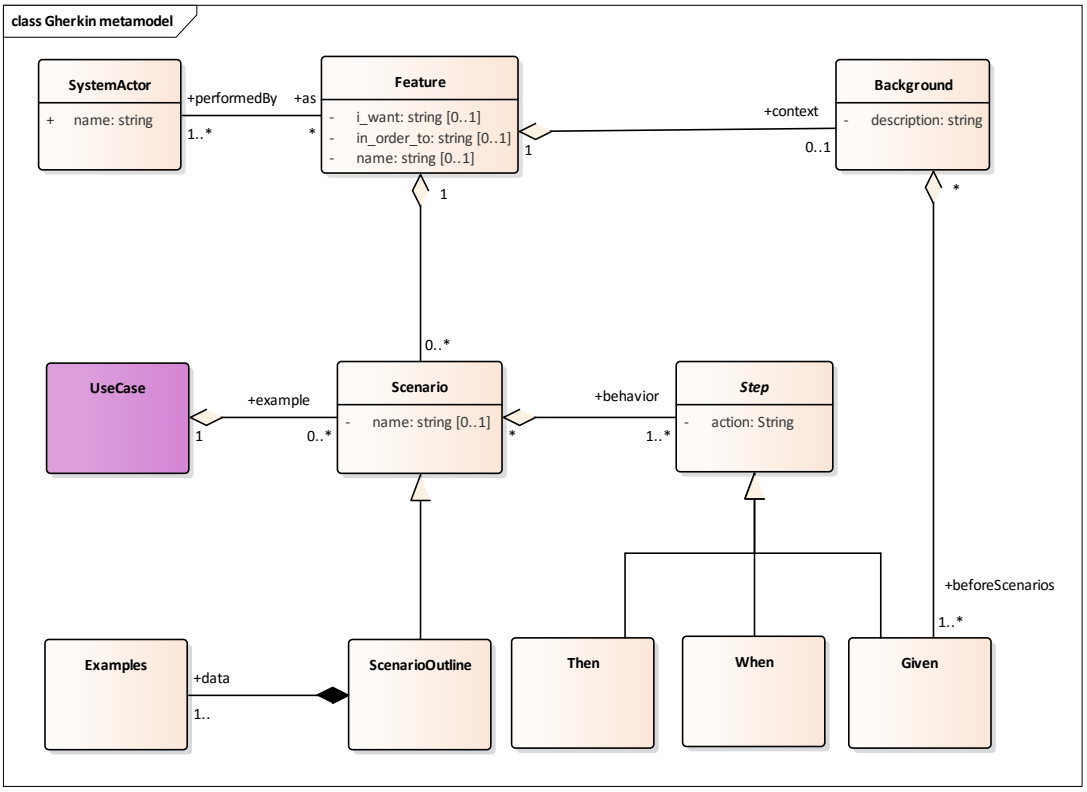

Fig 5. Metamodel for Gherkin scenarios.

Given, When and Then steps are the behavior of the scenarios. These three steps has been generalized with and abstract class called Step. This generalization is completed (a step must be a given or when or then step) and disjoint (a step cannot be more than one type and same time).

Scenario outline has been modelled as a specialization of scenario due it include and additional relation with the Examples element that contains the values for the steps (as seen in example from figure 3).

Another main contribution in this paper is how to include Gherkin Scenarios in UML Use Case diagrams. For this reason, we need a mechanism for associate use cases with Gherkin scenarios. This association is introduces in the abstract syntax (association Use Case-Scenario from figure 5) and it indicates that Gherkin scenarios are examples of use cases. Next section exposes how to draw these Gherkin scenarios together with Use Cases in UML Use Case diagrams.

4.2. Concrete syntax for Gherkin scenarios

As seen in section 1, a concrete syntax defines a notation for modeling the elements form the abstract syntax. One concrete syntax was already introduced for Gherkin scenarios. This syntax uses plain text for representing the Gherkin scenarios. With textual syntax, Gherkin scenarios may be executed by tools like Cucumber as test cases and they may be stored into a code repository.

However, UML Use Case diagrams are modelled used a graphical notation (figure 6). Therefore, for combining scenario with use cases effectively we need a notation for modelling

Gherkin scenarios as graphic elements that may be include in an UML Use Case diagram. Next paragraphs introduce a graphical notation for the elements defined in the previous abstract model. This graphical notation is another of the main contributions of this paper. Table 2 list all elements from Gherkin abstract syntax (also called metamodel) and indicates which UML element is used for modelling them in a UML Use Case diagram. Given, When and Then steps are represented by their supertype Step.

Table 2. Graphical elements and stereotypes for Gherkin elements.

Element from abstract syntax	UML element	Stereotype
SystemActor	No needed	--
Feature	No needed	--
Scenario	Use Case	Scenario
Background	Use Case	Background
Scenario outline	Use Case	Scenario
Step	Text note	--

Gherkin Actor and Feature elements from Gherkin metamodel do not need a graphical definition. The actor of a scenario modelled into a UML Use Case Diagram is the same actor that the main actor of the use case attached to the scenario. In a similar way, instead using features, Use Cases are used to attach scenarios.

UML Use Case definition [14] indicates that: "A UseCase is a kind of behaviour classifier that represents a declaration of an offered behaviour. [...] .Use cases define the offered behaviour of the subject without reference to its internal structure.". Previous definition matches with the goal of features. Therefore use cases may be used for attaching all elements of a feature (like scenarios and background). Scenarios may also be considered as a kind of use case and the UML Use Case symbol may be also used for modelling Gherkin scenario elements and Gherkin Background elements. For the sake of simplicity, all steps are defined in the same textual notation as the examples from figure 3. No need for special notation for Gherkin User Outline and Gherkin Example elements because they may be modelled using the same graphical elements that scenarios and steps.

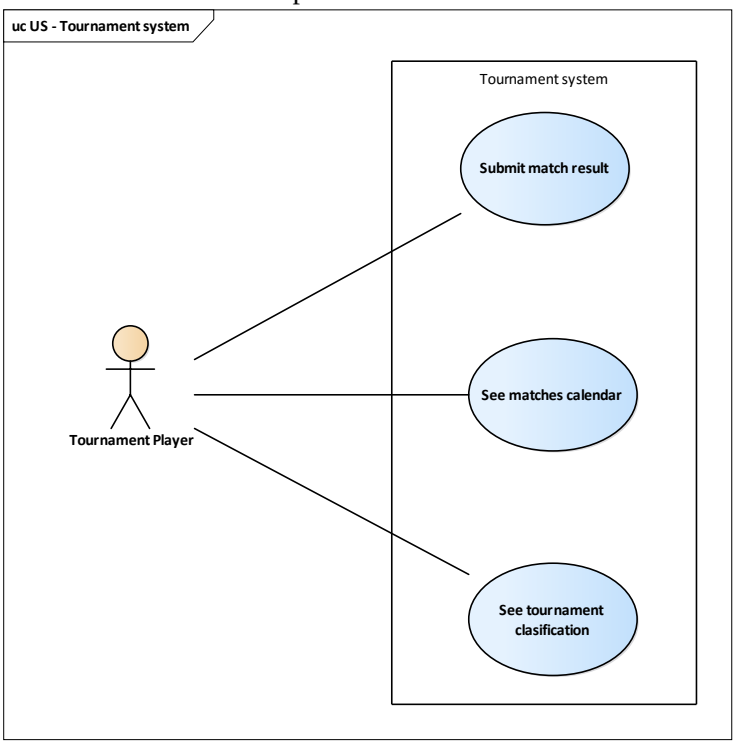

Fig 6. Example of use cases for Tournament System.

Two stereotypes has been also defined in table 2. Stereotype "Scenario" indicates a UML Use Case that models a piece of behaviour as a concrete example of the behaviour from other use case. Stereotype "Background" indicates a UML Use Case that models a piece of behaviour as a set or preconditions for other scenarios of the same Use Case.

Next paragraphs introduces the example of the tournament system (from sections 1 and 2) modelled used the concrete syntax defined in previous paragraphs.

Figure 6 shows a subset of the use cases for a system to manage a league of card games. Use cases in figure 6 describes the same behaviour than user stories from figure 1.

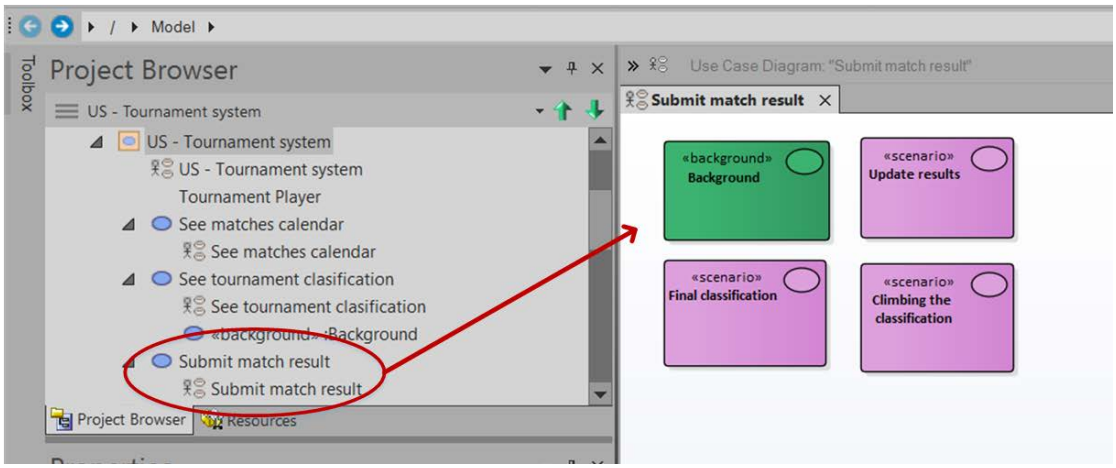

Fig 7. Concrete Syntax: diagrams and elements

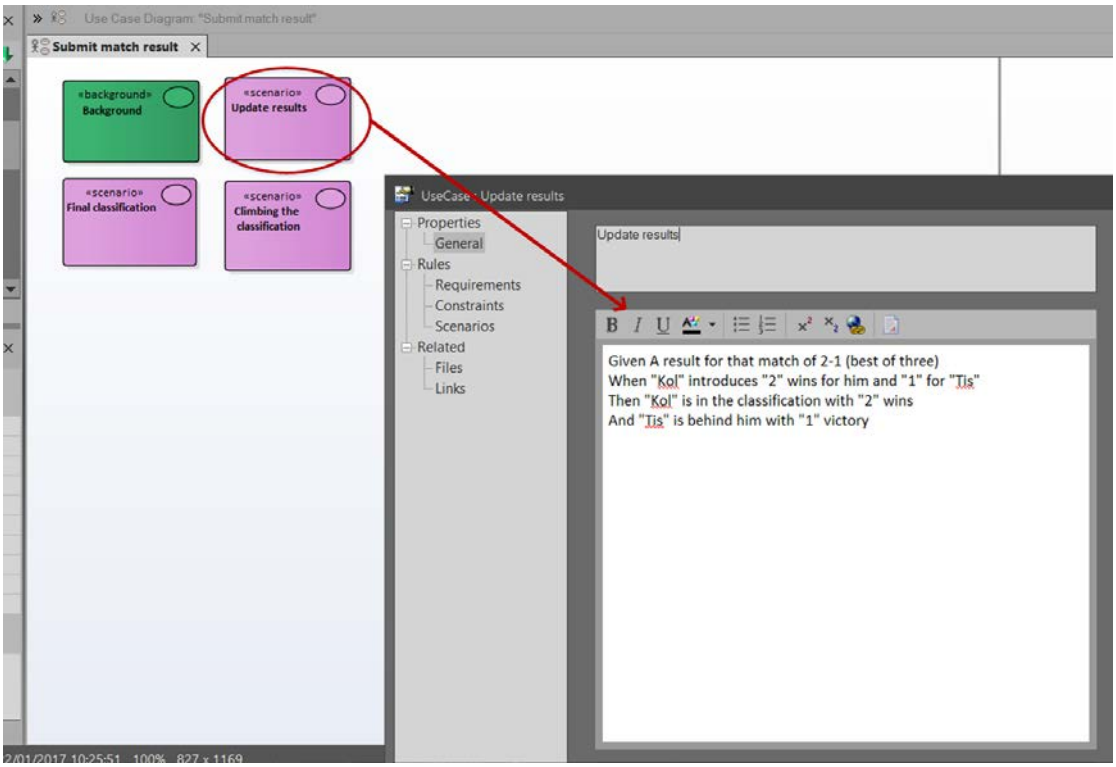

Fig 8. Concrete syntax: details

The background and scenarios depicted in section 2 (figure 3) are modelled as stereotyped use cases, using the stereotypes from table 2. These use cases have been defined in an inner use case diagram, as seen in figure 7.

Every scenario is defined in a UML Use Case diagram and this diagram is linked to a use case (see figure 7). All scenarios inside that diagram belongs to the same use case. The actor

of the scenario from figure 7 is Tournament Player (figure 6), the same actor of the use case the scenarios belong to.

Finally, the behaviour of the Gherkin Scenarios is defined as text as defined in table 2. One example is showed in figure 8. Please note that the behaviour and format of the scenarios (for example, Update Results from figure 8) is the same than the one from figure 3.

This section has defined how to draw Gherkin scenarios (modelled using the abstract syntax from previous section) as part of a UML Use Cases diagram. As seen in previous versions, Gherkin scenarios may be used as test cases by tools like Cucumber. Next section describes how to link the concrete graphical syntax introduced in this section with Cucumber and other related tools.

5. NeSP, a tool for supporting graphical Gherkin scenarios

As seen in a previous section, Gherkin scenarios may be executed as test case using the tool Cucumber. Cucumber, first, searches for all files that contains Gherkin scenarios defined as plain text (like the example in figure 3). Then, Cucumber tries to find the source code associated to each step in every scenario and executes it. When running on Java, Cucumber uses Java annotations or Java lambdas to associate methods to Gherkin steps. JUnit test library may be used for defining asserts and for compiling the information of the scenarios and generate a final report.

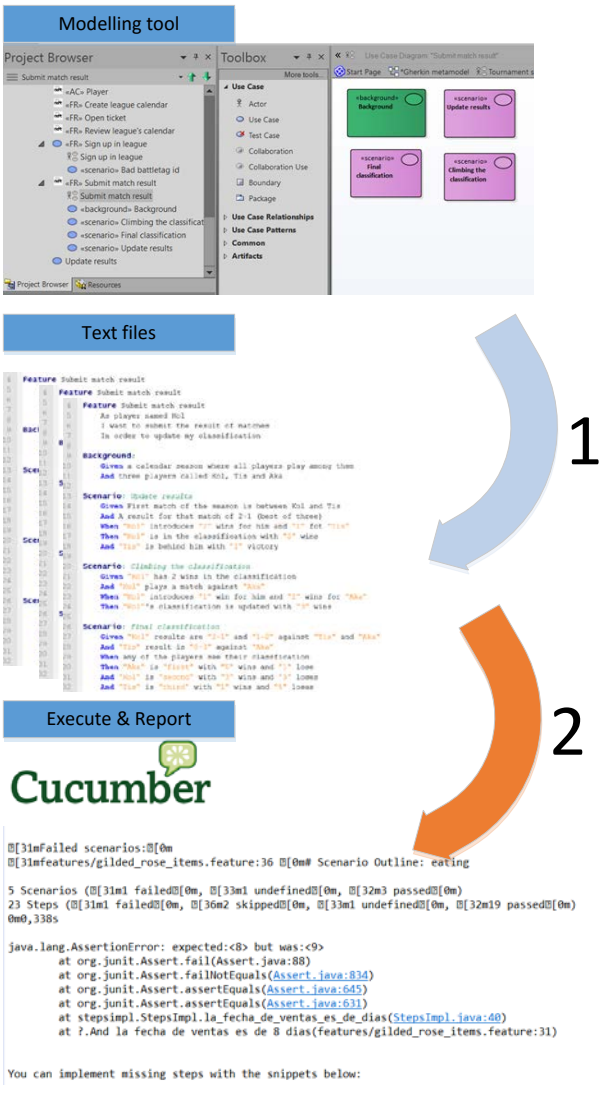

Fig 9.NeSP overview

Cucumber uses Gherkin scenarios defined in plain text like the examples from section 2. Cucumber cannot find by itself Gherkin scenarios defined in UML Use Case diagrams. Therefore, another contribution for this paper is a transformation from the UML Use case diagrams syntax (example in figure 6) to the plain text syntax (example in figure 3).

This transformation is pretty straightforward due both concrete syntaxes use the same metamodel (the abstract syntax from figure 5) and almost the same elements.

The only difference is that textual syntax uses the element “Feature” (from table 1) as container for scenarios and UML Use Case syntax uses use cases with the same purpose. Cucumber does not use the attributes of "Feature", it only uses the backgrounds and scenarios inside the feature. Therefore there is no need to create the attributes of the feature for the purpose of running scenarios as test cases.

The transformation has been implemented in a prototype tool called NeSP. Main functionality of NeSP is showed in figure 9 and it is described below.

1. NeSP scans the UML specification and it extracts information about scenarios from the UML Use Case diagrams.
2. Then, NeSP creates a structure of files and folders with those scenarios
3. NeSP calls Cucumber to execute the previous files as test cases.
4. Finally, NeSP shows the result of the execution of the test cases.

This tool works with UML Use Case diagram modeled with the tool Sparx Enterprise System [13]. NeSP source code is available in GitHub under an open source license [9].

Next section introduces conclusions and ongoing works.

6. Conclusions and Ongoing Works

As seen in the introduction section, requirement elicitation is about people communication and collaborating among them. Authors of this paper facilitated a Gherkin scenarios workshop few weeks before writing this paper. One of the exercises of that workshop was to write a little elevator pitch in a card describing the last project of the attenders. No surprise when several attenders that worked together in the same project wrote different pitches. For example, they identify different final users for their project.

Gherkin scenarios helps to avoid different visions in the same project. However, this communication tool must not be limited for the syntax or tools used for creating Gherkin scenarios. The abstract syntax introduced in this paper is a valid tool for opening the door to new representations for Gherkin scenarios.

This paper has introduced alternative syntax for working with Gherkin scenarios using UML Use Case models. The original contributions of this paper have been the formalization of the elements of Gherkin scenarios (with the abstract syntax from section 4.1), the extension of the notation for defining Gherkin scenarios (with the concrete syntax from section 4.2) and a supporting tool for running Gherkin scenarios in UML as test cases. Ongoing work is described below.

However, little practical experiences have been done yet. One of the main ongoing works is to promote practical usage of Gherkin scenarios in organization which use UML Use Cases. For example there still are an important usage of UML Use Cases in governmental projects in our country. Authors expect that, with the contributions in this paper, organizations which work in those projects will gain interest in the usage of Gherkin scenarios.

NeSP tool (section 5) stress the idea that Gherkin scenarios can be defined with different syntaxes. But NeSP tool has not been designed with usability in mind. It has a basic command line interface, it needs Java for running and it has many dependencies of external libraries. Authors of this paper have experience developing supporting tool for Model-Driven Engineering [5] and UML modeled with Sparx Enterprise Architect, like [7], [6]. Therefore, a second ongoing work is to evolve NeSP tool searching for an easiest usage.

Acknowledges

This research has been supported by Pololas project (TIN2016-76956-C3-2-R) and by the SoftPLM Network (TIN2015-71938-REDT) of the Spanish the Ministry of Economy and Competitiveness.

References

1. Adzic G.: Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications. USA. 2011
2. Cao, L., Ramesh, B.: Agile Requirements Engineering Practices: An Empirical Study. IEEE Software, 25 (1), 60–67 (2008)
3. De Carvalho, R. A., Silva, F. L. C., Soares, R.: Mapping Business Process Modeling Constructs to Behavior Driven Development Ubiquitous Language. eprint arXiv:1006.4892, (2010)
4. Wautelet, Y., Heng, S., Hintea D., Poelmans, S.: Bridging User Story Sets with the Use Case Model. ER 2016: Advances in Conceptual Modeling. Lecture Notes in Computer Science, 9975, 127-138 (2016)
5. Favre, J. M.: Towards a basic theory to model driven engineering. In: Proc. 3rd Workshop in Software Model Engineering (Satellite workshop at the 7th International Conference on the UML). (2004)
6. García Borgoñón, L., Barcelona, M. A., García García, J. A., Alba, M. , Escalona, M. J.: Software Process Modeling Languages: A Systematic Literature Review. Information and Software Technology 56, 103–116 (2014)
7. García García, J. A., Escalona, M. J., Domínguez Mayo, F. J., Salido, A.: NDT-Suite: A Methodological Tool Solution in the Model-Driven Engineering Paradigm. Journal of Software Engineering and Applications 7-4, 206-218 (2014)
8. Hoisl, B., Sobernig, S., Strembeck, M.: Comparing Three Notations for Defining Scenario-based Model Tests: A Controlled Experiment. 9th International Conference on the Quality of Information and Communications Technology, 95–104 (2014)
9. NeSP Source Code (2017), https://github.com/javierj/NeSP Accessed July 10, 2017
10. Schoeneman, L., Liu, J. B.: Integrating Behavior Driven Development and Programming by Contract. Master Thesis Bradley Universit (2013)
11. Schön, E., Thomaschewski, J., Escalona, M. J.: Agile Requirements Engineering: A Systematic Literature Review. Computer Standards & Interfaces. 49, 79–91 (2017)
12. Soeken, M., Wille, R., Drechsler, R.: Assisted Behavior Driven Development Using Natural Language Processing. TOOLS'12 Proceedings of the 50th international conference on Objects, Models, Components, Patterns. 269–287 (2012)
13. Sparx Enterprise Architect, https://www.sparxsystems.es. Accessed April 18, 2017
14. Unified Modeling Language 2.5 (2015), http://www.omg.org/spec/UML/2.5. Accessed July 10, 2017
15. Wynne, M., Hellesøy, A.: The Cucumber Book: Behaviour-Driven Devel-opment for Testers and Developers. The Pragmatic Bookshelf. USA (2012)