WORKING PAPER

2016/03

Towards an Agent-Driven Software Architecture Aligned with User Stories

Samedi Heng,
Louvain School of Management

Yves Wautelet,
KU Leuven, Belgium

Manuel Kolp,
Louvain School of Management

Christelle Scharff,
Pace University, United States of America
Towards an Agent-Driven Software Architecture Aligned with User Stories

Samedi Heng, Louvain School of Management
Yves Wautelet, KU Leuven, Belgium
Manuel Kolp, Louvain School of Management
Christelle Scharff, Pace University, United States of America

Summary

Agile principles have taken an increasing importance in the last decades. Software Architecture (SA) definition is perceived as a non-agile practice as it is executed in a top-down manner, reminding waterfall development, and sometimes imposes heavy documentation. This paper proposes to systematically build an agent-oriented SA from a set of User Stories (US), the core artifact to document requirements in agile methodologies. Previous research has allowed to define a unified US meta-model for the generation of templates relating WHO, WHAT and WHY elements. This meta-model’s elements define a syntax issued from practitioners templates associated with semantics from Goal Oriented Requirements Engineering frameworks, more precisely i*. With a set of US following the templates of this previous model, the link between the US and SA concepts is systematically studied and a transformation process is proposed. The SA can decline agent behaviors aligned with requirements and organizational behaviors. Moreover, requirements (thus US) are subject to evolution through agile iterations; the SA can evolve with these changes in a semi-automatic manner. We thus argue that the Agent-SA produced with our transformation process contributes to the overall project agility.

Keywords: Agent Architecture, Agile Development, User Story, Agile Architecture, Multi-Agent System.

This paper was accepted and presented at 8th International Conference on Agents and Artificial Intelligence (ICAART 2016), 24-26 February, 2016, Rome, Italy.
Towards an Agent-Driven Software Architecture
Aligned with User Stories

Samedi Heng¹, Yves Wautelet², Manuel Kolp¹, and Christelle Scharff³

¹ Université catholique de Louvain, Belgium
{samedi.heng, manuel.kolp}@uclouvain.be,

² KU Leuven, Belgium
yves.wautelet@kuleuven.be

³ Pace University, United States of America
cscharff@pace.edu

Abstract. Agile principles have taken an increasing importance in the last decades. Software Architecture (SA) definition is perceived as a non-agile practice as it is executed in a top-down manner, reminding waterfall development, and sometimes imposes heavy documentation. This paper proposes to systematically build an agent-oriented SA from a set of User Stories (US), the core artifact to document requirements in agile methodologies. Previous research has allowed to define a unified US meta-model for the generation of templates relating WHO, WHAT and WHY elements. This meta-model's elements define a syntax issued from practitioners templates associated with semantics from Goal Oriented Requirements Engineering frameworks, more precisely i*. With a set of US following the templates of this previous model, the link between the US and SA concepts is systematically studied and a transformation process is proposed. The SA can decline agent behaviors aligned with requirements and organizational behaviors. Moreover, requirements (thus US) are subject to evolution through agile iterations; the SA can evolve with these changes in a semi-automatic manner. We thus argue that the Agent-SA produced with our transformation process contributes to the overall project agility.

Keywords: Agent Architecture, Agile Development, User Story, Agile Architecture, Multi-Agent System.

1 INTRODUCTION

[1] highlights the core opposition between Software Architecture (SA) definition within a software project – seen as a top-down practice implying heavy documentation – and agile development. The Agile Manifesto emphasizes four values in agile development: (i) Individuals and interactions over processes and tools; (ii) Working software over comprehensive documentation; (iii) Customer collaboration over contract negotiation; and (iv) Responding to change over following a plan. Agile principles seem thus opposed to SA design unless the SA can evolve together and parallel to the requirements. Such an evolution would be eased if the SA would follow agile-style requirements definition. In agile methodologies, requirements are expressed using User Stories (US).
US are written in natural language by the customer and they describe user functionalities at an abstract level. In this work, we consequently suggest to systematically derive part of the SA from the set of US of a project in the form of a multi-agent architecture better aligned with agile principles. At the end, we want to dispose of a SA definition that is not heavy but can be partly automated and changed with the evolution of the US models, only if they are properly structured following the model defined in [17]. This model’s elements define a syntax issued from practitioners templates associated with semantics from Goal Oriented Requirements Engineering (GORE) frameworks, more precisely i* [19]. The implementation of the agent-architecture is outside the scope of this paper, but an implementation model in line with the SA can be found in [10].

2 STRUCTURING USER STORY CONCEPTS

This section exposes the “building blocks” used within this research.

2.1 Macro-level: Ways to Organize User Stories

We first distinguish a US macro-level dealing with the grouping of US into depending (i.e. relating) sets. We thus consider here the US as potential building blocks serving for SA composition/decomposition. Figure 1 presents our meta-model of the US concepts.

![Fig. 1. US as Macro-Level Structures: Meta-Model](image_url)

The UserStory class represents the US as a whole. US are written by the customer or product owner at the earliest stages of the project and put in the product backlog with a (implementation) priority and an amountOfPoints which refers to the number of User Story Points (USP)\(^4\) [6, 11]. As a result, these elements have been added as attributes to the UserStory class. Additional attributes required for process management are also included within the UserStory class.

\(^4\) The amount of USP represents the estimated effort required to implement the US.
The concept of US is by essence granular in the sense that some US need to be refined into other ones since they are too abstract to be estimated, implemented and tested in their initial forms. These US are called Epics [6]. Epics are US with a high level of abstraction meaning that they must be refined/decomposed into smaller US to fully describe the requirement they refer to. These US are represented by the class Epic that inherits from the US class. As we will see later on, Epic US contain coarse-grained process elements that cannot be transformed at once in the SA (see Section 4.2).

Other concepts evoked in the meta-model of Figure 1 are non-relevant for the present research and will not be discussed.

2.2 Micro-level: Decomposing a User Story in Descriptive Concepts

Within Figure 2, the meta-model of the previous section is enriched with the constituting elements of the US. We refer to this US view as the micro-level.

Rather than using the US as a whole within the requirements analysis process, we suggest, in our research design, to decompose the US on the basis of their WHO, WHAT and, when available, WHY dimensions. To ensure uniformity, these elements are all characterized as Descriptive Concepts (DC). When decomposed into a set of DC, the dependency between DC is intended to be further studied (see Section 2.3). Each element of a US template relating to one of the 3 dimensions is then an instance of the DC class. For the template of a US: As a <role>, I need a <task> so that <goal>, we have 3 instances of the DC class: one for role, one for task and one for goal. The dimension attribute thus compulsorily takes one of the values WHO (for role), WHAT (for task) or WHY (for goal) and the syntax attribute takes the syntax of the concept name (role, task, goal, ...). The semantic attribute relates to the definition of the DC. The list of all the possible DC is given in the form of a meta-model allowing to define US templates in Section 2.3.

Finally, since different DC can be linked together, we introduce the Link class that represents the possible types of links between two DC. Typically such links can be (at least partially) found in a logical way through agents’ behavior at the level of the SA (see Section 6).
2.3 Unified-Model of User Stories’ Descriptive Concepts

[17] proposes to build a unified model for designing US templates. The interested reader will refer to the latter reference for research and process related details. Figure 3 represents the meta-model of US templates. A US template can be designed from taking an element of the WHO, WHAT and possibly WHY dimensions. The link between the classes conceptually represents the link from one dimension to the other. Specifically, the unidirectional association from the Role to one of the Capability, Task or Goal classes implies that the target class instantiates an element of the WHAT dimension (always tagged as wants/wants to/needs/can/would like in the model). Then, the unidirectional association from one of these classes instantiating the WHAT dimension to one of the classes instantiating the WHY dimension (always tagged as so that into the model) implies that the target class eventually (since 0 is the minimal cardinality) instantiates an element of the WHY dimension. A US template supported by our model is for instance: As a <role>, I would like <task> so that <hard-goal>.

Each concept is associated with a particular syntax (identical to the name of the class in Figure 3) and a semantic. The syntax and semantics of the model are summarized here. As a result of the research conducted in [17], all these couples syntax/semantic are issued of [19]. Specifically:

- A Role is an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor;
- A Task specifies a particular way of attaining a goal;
- A Capability represents the ability of an actor to define, choose, and execute a plan for the fulfillment of a goal, given certain world conditions and in the presence of a specific event;
- A Hard-goal is a condition or state of affairs in the world that the stakeholders would like to achieve;
- A Soft-goal is a condition or state of affairs in the world that the actor would like to achieve. But unlike a hard-goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated soft-goal.

More explanations for distinguishing Hard-goal, Task and Capability elements will be discussed in Section 4.
3 A Structure for Defining the Software Architecture

This section exposes the main building blocks of the abstract agent-architecture that will be used to map US concepts in order to automatically align part of the SA with the collected requirements. Examples of each of the elements are provided in the illustrative example of Section 6.

3.1 Agent Structure

Figure 4 proposes a generic template for the Agent concept. The definition of an Agent is composed of five parts: Attributes, Events, Plans, Beliefs and Methods.

We typically refer in this work to BDI agent implementation platforms such as JADE, JACK Intelligent Agents or the Jadex BDI Agent System [14, 13, 12].

The agent class allows to specify:

– the declaration of agent attributes;
– the events (both internal and external) that the agent (i) handles, (ii) can post internally (to be handled by other plans), or (iii) can send externally to other agents;
– the plans that the agent can execute;
– the beliefs the agent can use and refer to. The beliefs of an agent can be of type private, protected, or public. A private access is restricted to the agent to which the belief belongs. Protected access is shared with other agents of the same class, while public access is unrestricted;
– the declaration of agent methods (e.g., the constructor of an agent).

<table>
<thead>
<tr>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
</tr>
<tr>
<td>Event</td>
</tr>
<tr>
<td>post</td>
</tr>
<tr>
<td>send</td>
</tr>
<tr>
<td>Plan</td>
</tr>
<tr>
<td>&lt;plan&gt;</td>
</tr>
<tr>
<td>Belief</td>
</tr>
<tr>
<td>private belief</td>
</tr>
<tr>
<td>protected belief</td>
</tr>
<tr>
<td>public belief</td>
</tr>
<tr>
<td>Method</td>
</tr>
</tbody>
</table>

Fig. 4. Agent template

3.2 Event Structure

Events describe stimuli, emitted by agents or automatically generated, in response to which other or the same agents must take action.
Events are the origin of all activity within an agent-oriented system. In the absence of events an agent stays idle. Whenever an event occurs, an agent initiates a task to handle it. This task can be thought of as a thread of activity within the agent. The task causes the agent to choose between the plans it has available, executing a plan or a set of plans until it succeeds or fails.

There are different event types, each with different uses. Events can be described along three dimensions:

- **External or internal event**: external events are sent to other agents while internal events are posted by an agent to itself;
- **Normal or BDI event**: an agent has a number of alternative plans to respond to a BDI event and only one plan in response to a normal event. Whenever an event occurs, the agent initiates a plan to handle it. If the plan execution fails and if the event is a normal event, then the event is said to have failed. If the event is a BDI event, a set of plans can be selected for execution and these are attempted in turn. If all selected plans fail, the event is also said to have failed;
- **Automatic or nonautomatic event**: an automatic event is automatically created when certain belief states arise.

### 3.3 Belief Structure

A **Belief** describes a piece of knowledge that an agent has about itself and its environment.

Figure 5 shows a template of a belief. Every belief that an agent currently has is represented as tuples. It has key and value fields. The **key** `FieldType FieldName` declaration describes the key attributes, while the **value** `FieldType FieldName` declaration describes the data attributes of each belief. A belief may have zero or more key or value field declarations.

### 3.4 Plan Structure

A **plan** describes a sequence of actions that an agent can take when an event occurs.

Plans are structured in three parts as shown in Figure 6: the **Event** part, the **Belief** part, and the **Method** part. The **Event** part declares events that the plan handles (i.e., events that trigger the execution of the plan) and events that the plan produces. The latter can be either posted (i.e., sent by an agent only to itself) or sent (i.e., sent to other agents). The **Belief** part declares beliefs that the plan reads and those it modifies. The **Method** part describes the plan itself, that is, the actions performed when the plan is executed. In the following, we study in more detail each part of the plan template.
**Events in Plans** handle <event>

Whenever an agent detects that an event arises, it tries to find a plan to handle the event.

The event the agent can handle is identified by its plans’ handle <event> declarations. When an instance of a given event arises, the agent may execute one of the plans that can handle this event.

The handle <event> declaration is mandatory. Whenever an instance of this event occurs, the agent will consider this plan as a candidate response. Without a handle <event> declaration, a plan would never be executed by an agent.

post <event>

The post <event> statement declares that the plan is able to post this event when executed. This may be in one of the methods declared in the Method part. Only events that the agent posts internally are declared after the post statement.

send <event>

The send <event> declaration is similar to the post <event> declaration, except that it identifies events that the plan can send to other agents. A plan is able to send an event through one of the methods declared in the Method part.

**Used Beliefs in Plans** read <belief>

The read <belief> declaration indicates that the <belief> is to be read (and only read) within the plan.

modify <belief>

The modify <belief> statement indicates that the <belief> may be read and modified within the plan.

<table>
<thead>
<tr>
<th>Plan</th>
<th>Event</th>
<th>Used Belief</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>handle</td>
<td></td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td>post</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>send</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;event&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;event&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 6.** Plan template
Methods in Plans All the plan’s methods are declared in the method part. Two of these methods are worth to be pointed out: context() and main().

An agent may further discriminate plans that handle an event by determining whether a plan is relevant. The context() method allows the agent to determine which plan to execute when a given event occurs. To be relevant, the plan must declare that it is capable of handling the event that has arisen (via the handle <event> declaration) and that it is relevant to the event instance (via the context() method).

context() is a boolean method. If this method returns true, the plan is relevant to the event. If not, the plan is not relevant to the event.

The main() method is executed whenever a plan is executed. The main() method is just like the main() method in Java – it represents the starting point in a plan’s execution.

4 Bridging a set of US to a Multi-Agent Software Architecture.

The aim of this Section is, starting from the Descriptive Concepts defined into the model presented in Section 2.3, to study the transformation possibilities to the element constituting the agent SA.

4.1 Role

A Role within a US is transformed in an intentional Agent in the SA. This way, an organizational role (which is most often a software user) is literally mapped to an intelligent acting entity at run-time into the system.

4.2 Hard-goal

The Hard-goal is the most abstract element since it is a condition or state of affairs that must be attained but there is no defined way to attain it or more precisely several ways could be followed in practice contrarily to the task that represents an operational way to attain a Hard-goal. An example of a Hard-goal could, for example, be to Graduate for a diploma in Business Information Management; it can be the Hard-goal of a student but there are several ways to attain this Hard-goal.

Being highly abstract, Hard-goals are not right away transformed into a functional aspect in the software architecture. US containing Hard-goals can be (but not necessarily) Epic US. Their satisfaction is rather supported by one or several Task(s) and/or Capabilities that will themselves be transformed into the agent architecture (see Section 4.3).

4.3 Task & Capability

The Task and the Capability represent more concrete and operational elements but these two need to be distinguished. The Capability could in fact be modeled as a Task but the Capability has more properties than the former since it is expressed as a direct intention from a role. In order to avoid ambiguities in interpretation, we use of the Capability
Towards an Agent-Driven Software Architecture Aligned with User Stories

4.4 Soft-goals

Soft-goals are not transformed at operational level but other Hard-goals and Tasks contribute positively or negatively to their satisfaction. These latter Hard-goals and Tasks are themselves transformed into the agent architecture as evoked above.

5 SOFTWARE TRANSFORMATION PROCESS

Figure 7 graphically illustrates the transformation process defined in the previous section. With respect to the SA:

- A Role element is transformed into an Agent;
- A Task element is transformed into a Plan;
- A Capability element is transformed into an Event.

Let us also note that Domain Entities (DE) manipulated by the Agents to fulfill the elements evoked above will form the Beliefs in the SA. DE are not expressed as such in US but rather physical or logical resources manipulated by Roles in the realization of US. This means they are derived from a careful reading of the US.

On the bottom right side of Figure 7, we can also note three types of diagrams – the Structural Diagram (SD), the Communication Diagram (CD) and the Dynamic Diagram (DD) – representing complementary views of the agents properties and behavior at system level. Due to a lack of space and with the willingness to focus on the transformation process only, we will, in the context of this paper, focus on the SD only.
6 ILLUSTRATIVE EXAMPLE

This section exposes the illustrative example on which we apply the SA transformation process. Note that this illustrative example is taken from a bigger case but that, due to a lack of space, the set of US chosen and their transformation into the SA is limited.

6.1 User Stories

Our proposal will be illustrated using a running example about carpooling. Carpooling deals with sharing car journeys so that more than one person travels within a car. In this context, it becomes increasingly important to save gas, reduce traffic, save driving time and control pollution. ClubCar is a multi-channel application available as an Android application, SMS service and IVR system. Users of ClubCar are riders and/or drivers that can register by SMS, voice or through an Android app. Roughly speaking, the software allows drivers to propose rides and submit their details with dates, times, sources and destinations while riders can search for available rides [15].

As shown in Table 6.1, we have taken a sample of the US of the ClubCar application to illustrate the research developed in this paper. The first column depicts the Dimension of US $D_C$, the second column describes the element itself and the last column gives the type of the $D_C$.

<table>
<thead>
<tr>
<th>#</th>
<th>Dimension</th>
<th>Element</th>
<th>$D_C$ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WHO</td>
<td>As a DRIVER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to propose rides to Riders</td>
<td>Hard-goal</td>
</tr>
<tr>
<td>2</td>
<td>WHO</td>
<td>As a DRIVER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to propose a ride from A to B with</td>
<td>Task</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the price location and time of departure,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and number of seats available</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>WHO</td>
<td>As a DRIVER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to log in to the platform</td>
<td>Capability</td>
</tr>
<tr>
<td></td>
<td>WHY</td>
<td>so that I can register to the service</td>
<td>Task</td>
</tr>
<tr>
<td>4</td>
<td>WHO</td>
<td>As a DRIVER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to select the ride characteristics</td>
<td>Capability</td>
</tr>
<tr>
<td>5</td>
<td>WHO</td>
<td>As a DRIVER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to confirm the proposal</td>
<td>Capability</td>
</tr>
<tr>
<td>6</td>
<td>WHO</td>
<td>As a RIDER</td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>WHAT</td>
<td>I want to order a ride</td>
<td>Capability</td>
</tr>
</tbody>
</table>

Table 1. US sample issued of the ClubCar development

Note that, when there were several possibilities, an ad-hoc choice has been made since envisaging the multiple modeling solutions at the level of the unified template model is not the focus of the present research.
6.2 Agent Architecture

US1 contains a *Hard-goal* element in the WHAT dimension and constitutes an Epic US so that it is not forwarded as such to the SA but is satisfied through the realization of the Task element in US2. Nevertheless we can “manually identify the presence of a DE which is the Ride that will be transformed into a Belief (see hereunder).

As illustrated in Figure 8, the ProposeRideFromAtoB Plan is forwarded from the Task element depicted in US2. It is used by the Driver Agent to propose a ride to the Rider Agent. The Plan is executed when the CharacteristicsSelected Event (containing the information about the characteristics of the Ride offered by the Driver) occurs. The latter Event is forwarded from US4. The ProposeRideFromAtoB Plan also creates a Ride Belief, i.e. the Driver belief storing the proposed Ride that can then be proposed to a Rider Agent. Later, the Rider Agent can subscribe for the proposed Ride.

DriverProposalConfirmation is an Event transformed from US5 that is posted automatically whenever the value of the status field (of the Ride Belief) is set to submitted. It will then invoke a plan to inform the Rider that the Ride is available.

The RiderOrder Event (not illustrated in Figure 8) belonging to the Rider Agent is sent when the Rider Agent subscribed to a Ride offered by the Driver Agent. This Event is transformed from the Capability element in US6.

As evoked in the previous Section, other agent behavior is outside the scope of the this paper that focuses on systematic forward engineering of US to a SA, therefore the full SD as well as CD and DD are not represented here.

7 RELATED WORK

Agile development and SA have been identified as the fourth position among the top ten research questions that should be addressed by the community [8]. The actual researches on this area are focused on quantity of SA rather than quality [2]. SA is seen as a human activity based on designers experience meaning that most architectures are ad-hoc and only some follow a defined pattern [3]. Many practitioners have argued that model-driven engineering as well as user-center design are beneficial for constructing a SA in the agile environment [2]. We point to the development of a SA not following patterns but aligned on the structure of requirements.

To the best of our knowledge, there is no work on mapping the US to agent SA. In most existing research, US are generated and decomposed from the SA elements. [11] proposes the scaled agile framework that emphasizes on the importance of building the SA to support development. They however do not furnish any mechanism for building and and tracing the SA. [?] propose a similar framework that bridges US to SA elements; US are nevertheless considered as fine-grained elements only.

We can also focus on the use of US in agent methods. In Agile PASSI [5], US are used as requirement artifact for communication but its their only usage. US, specifically in MaSE [7], is one source of requirements for capturing the goal of the agent [18] but without formal transformation. [9] use fuzzy theory to provide a systematic

---

6 Due to a lack of space the Rider Agent has not been represented in Figure 8.
and methodical approach to explore various dependencies such as goal, task, resource or soft-goal from US. Again the technique is limited at analysis stage with no transformation. In contrast, [16] adopt agent-oriented modeling in agile development. They provide a method for decomposing a goal into sub-goals and link them to US elements. [4] use US as testing mechanism for agent-oriented software development. Only one US template is used and aligned to JBehave (http://jbehave.org).

8 CONCLUSION

The quest for agility has, in the recent years, been conducted in several domains and, behind the buzz word, companies are looking for a high-level of alignment between user requirements and software as well as rapid respond to change. This paper has proposed to map US – THE requirements models of agile software development methods like SCRUM and XP – with an agent-based SA in order to implement a software system made of Agents mapping organizational behavior at run-time. The transformation
process can be partially automated for rapid response to change and to avoid a heavy documentation process.

Future work includes a full validation on the ideas on a broader case study. A plug-in supporting the automatic transformation of US elements into the proposed SA is under development within the DesCARTES Architect CASE Tool.

References

