Flexible *Working Architectures*: Agile Architecting Using PPCs

Jennifer Pérez, Jessica Díaz, Juan Garbajosa, and Pedro P. Alarcón

Technical University of Madrid (UPM), E.U. Informática, Madrid, Spain
jenifer.perez@eui.upm.es, yesica.diaz@upm.es,
{jgs,pedrop.alarcon}@eui.upm.es

Abstract. Software systems need software architectures to improve their scalability and maintenance. However, many agile practitioners claim that the upfront design of software architectures is an investment that does not pay off, since customers can rarely appreciate the value delivered by architectures. Furthermore, conventional architectural practices may be considered unacceptable from the Agile values and principles perspective. In this paper, the development of *working architectures* in agile iterations is presented as an attempt to solve the problem of designing software architectures in Agile. This contribution is based on the new concept of Plastic Partial Component (PPC). PPCs are highly malleable components that can be partially described, what increases the flexibility of architecture design. PPCs based architectures let reinforce some of the agile values and principles. Our experience of putting this contribution into practice is illustrated through the agile development of a Testing Framework for Biogas Plants.

1 Introduction

It is a well accepted fact in Software Engineering that architectures make software systems simpler and more understandable. Software architectures describe the structure of a software system by hiding the low-level details and abstracting the high level important features [1]. Software architectures also accommodate non-functional requirements. The design, specification, and analysis of the structure of software-intensive systems have become critical issues in software development [2]. As a result, software architectures emerged as a solution for the design and development of large and complex software systems.

The Agile Manifesto [3] is the basis of agile methodologies. It establishes the following two principles: *"Working software is the primary measure of progress"* and *"Delivering working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale"*. These two agile principles imply that, the limited time that the development of a *working product* takes the developers, should be mostly invested in coding to satisfy the delivery deadline. Therefore, agile practitioners often consider that the upfront design and definition of software architectures is an investment in time and effort that is not paid off.

Literature is full of references that advocates against architecture in Agile, as customers rarely can appreciate the value that architecture delivers. A common belief is that *"If you are sufficiently agile, you don't need an architecture - you can always refactor it on the fly"*. However, it has been argued that an inaccurate architectural design leads to the failure of large software systems and large refactoring might create significant defects [4]. As it is illustrated by Dybå and Dingsøyr in [5], several authors advocate that the lack of focus on architecture is bound to engender suboptimal design-decisions. This lack is in contradiction with an Agile principle that establishes that *"Continuous attention to technical excellence and good design enhances agility"*. In addition, according to Babar and Abrahamsson [6] software architectures may be also essential to improve and scale up *Agile Software Development*[1] in large software-intensive systems. Cockburn [7] claims that the issue with architecture in Agile is not either *architecture yes* or *architecture no*: he thinks that the issue is how much effort should be invested in architecture, assuming that (architecture) practices can be valuable for the customer. Kruchten concludes in [8] that in software architectures there are cost and value, also for agile. Then, the key question is: *"Are we able to avoid the obstacles that hamper agile practitioners to design software architectures without renouncing their values and principles?"*. There are some works [7,8,9,10,11] that intend to harmonize Agile and architecture by outlining high level approaches or organizational guidelines, but do not provide specific techniques or practices to design architectures that favor agile values and principles. Our understanding is that having flexibility at the time of defining software architectures is essential, so that, practices can be aligned with Agile values and principles.

In this paper, we deal with the problem of designing software architectures in Agile. From the wide-scope of tasks that software architectures comprise: (i) to analyze and describe the properties of systems at a high level of abstraction; (ii) to validate software requirements; (iii) to estimate the cost of the development and maintenance processes; (iv) to reuse software; and (v) to establish the bases and guidelines for the design of large complex software systems [1]. Our contribution is focused on the structural viewpoint of software architectures, i.e. the description of software architectures.

We present our experience using *Plastic Partial Components* (PPCs) [12] to specify software architectures in an Agile context. A Plastic Partial Component (PPC) is a new concept to support internal variation of architectural components by hooking *crosscutting* and *non-crosscutting* concerns (*aspects* and *features*) that are unaware of the linking context. Despite the fact that PPCs were originally defined for Software Product Lines (SPLs) [13], we have taken advantage of their extension mechanisms for designing software architectures in Agile. Using PPCs, architectural components can be iteratively and incrementally developed in each iteration and, by extension, the software architecture that they make up. This architecture is incrementally and iteratively designed in each iteration by adding/removing: (i) aspects and/or features to/from its PPCs, and (ii) components and connections to/from the architecture. From this proposal, a new concept in

[1] In this article we will use the term Agile representing Agile Software Development.

software architectures emerges, called *working architecture*. A *working architecture* is the architecture that is obtained along with each *working product* in each agile iteration. We illustrate our proposal of using PPCs in Agile through our experience of developing a framework, in cooperation with industrial partners, for monitoring, testing and operating biogas power production plants.

It is necessary to emphasize that our contribution is focused on the structural viewpoint of software architecture, i.e. the description of software architectures. Software architectures address: (i) the description of systems properties at a high level of abstraction; (ii) validation of software requirements; (iii) estimation of the cost of the development and maintenance processes; (iv) software reusability; and (v) establish the bases and guidelines for the design of large complex software systems [1].

The structure of the paper is as follows: Section 2 introduces the main notions of agile and PPCs. In addition, it explains the agile methodology SCRUM. Section 3 discusses related works about software architecture practices in Agile. Section 4 explains why and how PPCs fit for use with Agile. It also explains our proposal about how to specify *working architectures*. Section 5 presents a case study that is used to illustrate our contribution, and exemplifies the use of PPCs in Agile. Finally, conclusions and further work are presented in section 6.

2 Background

2.1 Agile Software Development

Agility is just an umbrella term for a variety of methods structured into values, principles and practices, with a common reference in the Agile Manifesto [3]. Shore et al. [14] define *values* as ideals, *principles* as the application of these ideals to the industry, and *practices* as principles applied to a specific type of project. The relevance of values and principles is increasing as long as large organizations are requiring their application [15]. Some of these agile principles are: *customer satisfaction through early and continuous delivery of valuable software*; *continuous attention to technical excellence*; or *welcome changing requirements, even late in development*. Some common Agile methods are eXtreme Programming (XP) [16], Lean Development [17], and Scrum [18], the one used within this work.

Scrum implements an iterative and incremental life cycle (see Figure 1). Three roles, the *Product Owner*, *Team*, and *ScrumMaster* make up all together the *Scrum Team* [18]. The Product Owner represents the key stakeholder interests. The Team is in charge of developing the product functionality and the customer is often a membership of the team. The Scrum process is responsibility of the ScrumMaster. Requirements are captured as User Stories (USs) by the customer together with the rest of the Scrum Team members during the *pre-game phase*, at the beginning of the project. The list of USs is stored in the *product backlog*. Later on in the process, USs are prioritized and divided into short time-framed iterations called *sprints*. A sprint is a 2-4 weeks period of development time. The scope and *goals* of each sprint are agreed at its beginning by the Product Owner

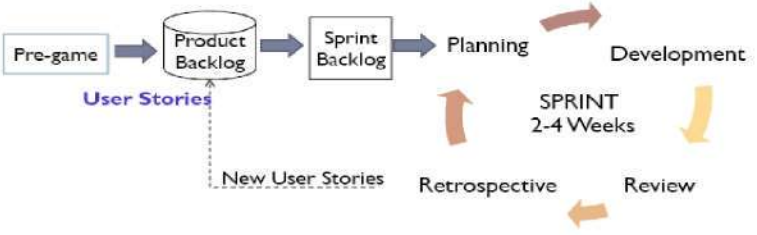

Fig. 1. Scrum Lifecycle

and the Team at the *sprint planning meeting*. The output from this meeting is stored in the *sprint backlog*. Each sprint should deliver a valuable increment of the final product functionality. During the execution of each sprint, the Team will *meet daily* in a 15-minute meeting to track work progress. At the end of each sprint, the *sprint review* and *retrospective* meetings will be held. In the sprint review meeting the Product Owner will communicate whether goals were met, and might introduce changes into the USs. In the retrospective meeting the Team and ScrumMaster discuss what went well, and what could be improved for the next sprint, and works as an estimation and tracking activity to put into practice continuous improvement.

2.2 An Overview of Plastic Partial Components (PPCs)

The notion of *Plastic Partial Component (PPC)*[2] was originally defined for Software Product Lines Engineering (SPLE) [13]. SPLE adoption requires explicitly to specify the commonalities and variabilities of SPLs at the architectural level. This implies not only to specify variants for modifying the configuration of software architectures, but also to define variations inside components. PPCs are a solution to support the internal variation of architectural components.

PPCs variability mechanisms are based on Invasive Software Composition Principles [19]. Invasive Software Composition defines components as fragment boxes that hook a set of reusable fragments of code. Specifically, these fragments of reusable code can be aspects making components easier to be maintained and by extension software architectures. The variability of a PPC is specified using *variability points*, which hook fragments of code to the PPC. These fragments of code are specific features of a software product, which can crosscut the software architecture or not. For this reason, we classified these features into: *crosscutting-features* and *non-crosscutting-features*. A *crosscutting-feature* is a common feature of the software architecture, which is encapsulated into a separate entity called *aspect*. Whereas, a *non-crosscutting-feature* is the specific functionality of a component, which is encapsulated into a separate entity called *feature*. Therefore, variability points can hook *aspects* or *features* to the PPC. A PPC is defined by specifying: (i) its variability points; (ii) the aspects and/or

[2] This section presents just an overview of PPCs. A broader description with additional references to literature sources can be found in [12].

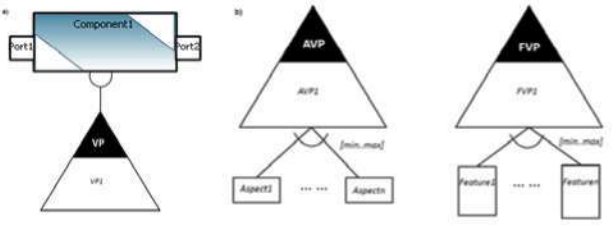

Fig. 2. a) Linking Plastic Partial Components and Variability Points. b) Variability Points and Variations (Aspects and Features).

features that are necessary to complete the definition of the component for any software product; and (iii) the hooks between the variability points and the aspects and/or features. As a result, the complete definition of a PPC for a specific product is done by means of the selection of aspects and/or features through the variability points.

A PPC is a specialization of a component and inherits all the properties and behavior of a component. A PPC is characterized by the definition of a set of variability points, i.e. the place where the different variants are hooked to the PPC (see Figure 2.a). A ***variability point*** of a PPC is characterized by three properties: (i) the kind of variation, (ii) the type of variability point depending on the variants that it offers to be selected (i.e. crosscutting or non-crosscutting features), and (iii) the weaving between variants and the component.

The *kind of variation* is based on the variability management of software architectures that Bachmann and Bass set out [20]. This property is provided to support variability, and it defines the number of variants of a product family that must be selected (*mandatory*) or can be selected (*optional*) for developing a specific product of the family. The kind of variation is specified as cardinality (0..1, 1..1, 0..n, 1..n, m..n).

There are two *types of variability points* (see Figure 2.b): (i) those that permit the selection of variants that are *crosscutting-features* (*aspects*); (ii) and those that permit the selection of variants that are *non-crosscutting features* (*features*). An *Aspect Variability Point (AVP)* can only offer aspects to be selected; and a *Feature Variability Point (FVP)* can only offer features to be selected. It is important to emphasize that both aspects and features can be linked to more than one variability point to facilitate reuse.

Variability points allow us to specify the *weaving* between the PPC and the variant. The weaving principles of Aspect-Oriented Programming (AOP)[21] provide the needed functionality to specify *where* and *when* extending the PPCs using variants. Therefore, AOP weaving primitives (*pointcuts*, *weavings* and *aspects*) are applied to weave a PPC with both, aspects and features. The *pointcut* definition consists of defining *where* to insert the code of the variant. An example of pointcut could be calling a service that the PPC provides. The definition of the *weaving operator* consists of establishing *when* to insert the code of the variant with regard to the pointcut: *before*, *after* or *instead of* (around). In our

proposal, it will be before, after or insteadOf the call of the service of the PPC. Thereby, the PPC, the pointcut, the weaving operator and the variant are the elements that define a *weaving*.

However, there are some differences between our definition of aspects and weavings and those AOP provides. In our proposal, the pointcut and the weaving operator are specified outside the aspects and features, and inside variability points. As a result, unlike AOP, our aspects and features are unaware of the linking context, and they are completely reusable.

The description of working architectures using PPCs is supported by a graphical modeling tool called Flexible-PLA. Flexible-PLA has been developed following the MDD approach [22,23] to take advantage of its metamodel definition and its corresponding graphical metaphor [12]. It has been automatically generated from the metamodel and the graphical metaphor. It was possible because they were specified using the Eclipse Modeling Framework (EMF) and its Graphical Modeling Framework(GMF) [24]. As a result, Flexible-PLA is an open-source tool that is available for the research community.

3 Related Work

To make come true agile software architectures, it is necessary: (i) to provide mechanisms to flexibly describe them and (ii) to define how they should be designed throughout the Agile software life cycle, i.e, how to perform agile architecting. In any case, Agile values and principles should be respected.

On the one hand, with regard to software architecture description, Scott Ambler in [25] proposes a model based on views and concerns for designing software architectures in Agile. We take a step forward and define a specific formalism to systematically introduce crosscutting and non-crosscutting features in the Agile Software Development of architectures. This is due to the fact that we realized that it is necessary to be flexible enough for supporting not only *external changes* (modification of the architectural configuration by adding or removing components or connections), but also *internal changes* (modification inside components). It is important to keep in mind that these internal changes must preserve abstraction and encapsulation (black boxes) of software architectures and Agile flexibility requirements. So, internal changes are treated as crosscutting and non-crosscutting features that are easily added and removed to/from agile software architectures.

On the other hand, with regard to agile architecting, one of the main Agile issues is to improve the scalability of their products by designing their software architectures. Several proposals have been outlined to fix this issue [9,10,11]. Most of them agree with the idea that Agile should incorporate architectural information when it is applied to develop large-scale software-intensive systems. However, the proposed approaches are rather "high-level". Cockburn [7] proposes to start with a simple architecture that handles all the *big rocks*. Then, it can be evolved or refactored as other requirements appear; but it should not be an objective to get the architecture at the end of the project. Boehm and Turner [9]

recognize that hybrid approaches to balance agile and plan-driven approaches are necessary; McMahon [10] recommends employing in agile architectures two levels. The first level develops a high-level agile architecture including the major system components, assumptions, and a brief description of each component. The second level focuses on the high-risk areas for each iteration (big rocks). M. Ali Babar et al. [11] analyze the role of the architecture in Agile through a case study by integrating software product lines and agile practices, and carry out a description of the organizational processes. All these works recognize and recommend the role of architectures in Agile but the practices they provide are rather general.

From our point of view, it is not necessary to create new mechanisms from scratch to design software architectures in Agile. We can adopt existing mechanisms that assists architects to flexibly develop software architectures. This flexibility can be obtained from mechanisms that allow us to specify variability. Agile methodologies can take advantage of variability mechanisms to flexibly adapt software architectures and to incrementally develop them together with the working product. In fact, our proposal uses PPCs [12] to incrementally design software architectures in Agile.

4 Flexible Working Architectures

Agile establishes an iterative and incremental software development, in which iterations are short time-framed and always deliver valuable software (*working product*). When Agile Methodologies are applied to develop large-software intensive systems, software architectures are required to scale their working and final products. Software architectures bridge the gap between requirements and implementation [1], and by extension between USs and the implementation. Therefore, mechanisms for designing flexible architectures along with the working product in each agile iteration are required to deal with the obstacles that hamper agile practitioner to design software architectures. These mechanisms must support for easily adding/removing components and connections (external variation) and adding/removing features and aspects inside components (internal variation), considering variation like incremental steps in software development.These assumptions and PPCs are the base of our proposal for developing software products in Agile by designing their software architectures and preserving the Agile values and principles. In this section, it is explained how PPCs can help us to flexibly build *working architectures* throughout the ASD life cycle.

4.1 Plastic Partial Components in Agile

PPCs variability mechanisms are successfully applied to SPLs to support internal variation of architectural components among the products of a SPL. However, in Agile, variability mechanisms are not used to define variations (aspects and features) among products. On the contrary, they are used to flexibly add, remove and modify aspects and/or features throughout the iterations of an Agile

lifecycle. Variability mechanisms behave as extensibility mechanisms to flexibly compose pieces (aspects, features, components) of software as if we were building a puzzle. As a result, PPCs get closer and closer to customer needs by means of specifying the aspects and features only when they are strictly required by a working product. From the PPC definition it is possible to conclude that PPCs facilitate to meet the agile principle and values:

- **Partial:** PPCs are Partial because they can be incompletely specified. They can be *working components* delivered and refined each iteration as part of the working product. Therefore, PPCs allow us to incrementally develop architectural components by only taking into account the required functionality for each iteration, and to construct them in time to the working product.
- **Plastic:** PPCs are Plastic because they are highly malleable. This is thanks to their extensibility mechanisms, which allow us to flexibly adapt software components by easily adding or removing fragments of code. As a result, they are ready to be extended or modified at any moment.

PPCs are always composed of *mandatory* aspects or features and every variability point of a PPC is either *mandatory and unique* or *mandatory multiple and multiple*. In agile, aspects and features of PPCs are mandatory unless they will be removed over iterations. So, the kinds of variation are constrained to the following cardinalities:

- 1..1: *mandatory and unique:* it is mandatory to select the unique aspect or feature of the variability point.

- m..n: *mandatory and multiple:* variability point is not used in Agile as a point of decision, therefore it is mandatory to select the multiple features or aspects of the variability point. For this reason, the number of selections m is equals to the number of variants n, and the cardinality is n..n.

Therefore, when PPCs are applied in Agile, the two types of variability points are characterized by the kind of addition, update or deletion, not by the kind of selection that performs. An *Aspect Variability Point (AVP)* can only add, modify or remove aspects; and a *Feature Variability Point (FVP)* can only add, modify or remove features.

4.2 Agile Architecting

In this section, we present our experience of applying PPCs in Agile. In our proposal, all the components of the architecture are PPCs, that are incrementally developed in time to the working product. They constitute the new concept of *working architecture*. A working architecture is the architecture that is iteratively and incrementally designed together with the working product. This idea was also proposed in [26,27,28] as continuous architecting. Continuous architecting allow us to tackle architecture degradation and keep the system in sync with changing conditions. In addition, successful agile architecting requires to define the role of the architect in an Agile team. The *architecture team* is part of the Agile team and interacts with the rest of members at the making-decision process by tracking architectural concerns and balancing them with business priorities.

Thereby, architecture can also support one of the agile values, communication [29]. Next, from our experience we explain how to develop agile *working architectures* using PPCs. We make a distinction between the first performed iteration and the others.

– **First Iteration**

Once the architecture team, the development team, and the customer have defined the USs of the software product, the customer selects the USs for being developed in the first iteration. This selection is performed taking into account the priorities of the customer and the advice and recommendations of the development and architecture teams. Next, the development and architecture teams can start the architecture design and implementation of the selected USs. Before implementing USs, it is important to analyze them to identify *candidate components* for a *working architecture* of the working product. Whereas traditional software development classifies requirements into functional and non-functional, Agile does not make this distinction in USs. In fact, most USs are related to functional requirements due to the fact that they are those requirements that the customer perceives as the result he/she requires. However, it often happens that non-functional requirements, such as distribution or security, have to be implicitly considered and implemented to meet the functional requirements, i.e. customer needs. As a result, the architecture team must keep in mind non-functional requirements to identify them in the USs. In consequence, we understand that the architecture team must analyze USs to identify:

1. *PPCs:* Units of basic functionality, also known as major software components [10]. They are *candidate components* of the software architecture of the working product. They make up the *working architecture*.
2. *Features:* Features represent *non-crosscutting features* (see section 2.2). They are usually functional requirements that are not relevant enough for being major software components. They constitute additional functionality of the final product, which is susceptible of being removed over time. Thus, they are part of the functionality that a PPC provides.
3. *Aspects:* Aspects represent *crosscutting features* (see section 2.2). They are usually non-functional requirements. They are part of the functionality that one or more PPCs provide.
4. *Architectural Connections*: Connections to coordinate PPCs and configure a working architecture.

From this iteration, a first version of the *working architecture* is obtained.

– **Subsequent Iterations**

As in the first iteration, the customer, the architecture and development teams select the USs that are going to be developed during the current iteration. However, in this case the selection can be also guided and supported by the *working architecture* obtained from the previous iteration. This is due to the fact that

software architectures not only can help us study the feasibility of the development of software systems, but also can help us determine which requirements are reasonable and viable [30], and by extension which USs could be selected. There are different criteria of selection that can be assisted by the architecture knowledge such as scalability, reusability or the impact of changes. Therefore, the knowledge of a *working architecture* is a value which may enrich the agile process.

Once the customer, architecture and development teams have selected the USs for the iteration, the architecture and development teams can start the architecture design and implementation of the selected USs as in the first iteration. Finally, after completing the last iteration, a *final software architecture* is obtained as part of the *final product*.

It is important to emphasize that the unique difference among the first iteration and the rest of them is the fact that the first iteration starts from scratch the software architecture. We do not define a ZFR (Zero Feature Release) where the customer does not participate as other approaches propose [16,10]. We consider that the investment of time and cost in this ZFR does not guarantee that the decisions taken will be definitive and the ZFR architecture will be preserved. So, our first iteration is just one more, where the customer participates. In each iteration, PPCs, Aspects, Features or Connections from the software architecture are updated, added and removed in a flexible way and without any restriction.

4.3 Analysis of PPCs and *Working Architectures* from the Agile Perspective

PPCs facilitate to meet the agile principles and values and to carry out some of the agile practices. PPCs and *working architectures* match with the four agile values: (i) *Individuals and interactions over processes and tools*: The architecture team is part of the Agile team and participates in its meetings; (ii) *Working software over comprehensive documentation*: PPCs are part of a working architecture, which is software that is delivered in each working product; (iii) *Customer collaboration over contract negotiation*: The architecture team interacts with the customer throughout the agile process; and (iv) *Responding to change over following a plan*: PPCs easily accept changes by adding or removing features and/or aspects and they are connected between them to configure working architectures.

With regard to the twelve agile principles, next we detail those that could be enriched with our proposal.

- *(P1). Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*: PPCs help get closer and closer to customer needs over iterations, and by extension the working architecture.
- *(P2). Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage*: PPCs are plastic. They are ready to be extended or modified at any moment.
- *(P3). Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale*: Working architectures are part of the delivery.

- *(P4). The most efficient and effective method of conveying information to and within a development team is face-to-face conversation*: The architecture team participates in every meeting of the project and is welcome to the new customer needs. The architecture team shares architectural information among its members and with all others in every scheduled meeting. The architecture team is open minded to the changes that their feedback may imply. This is thanks to the adaptation facilities that PPCs and working architectures provide.
- *(P5). Working software is the primary measure of progress*: the working architecture is part of the working software that is delivered in each iteration.
- *(P6). Continuous attention to technical excellence and good design enhances agility*: PPCs supports to intuitively modularize and scale software by using its variability mechanisms, which can be advantageously used for adding or removing pieces of software throughout the different iterations that comprise the development of a working product. In addition, PPCs help us to easily apply a major technique used in Agile to cope changes: refactoring. Refactoring is a process and a set of techniques to reorganize code while preserving the external behavior of a working system [31]. PPCs help us to extend and reorganize code by its encapsulation into features and aspects, which avoids the inherent tangled-code that crosscutting concerns generate.

5 Applying PPC for the Agile Development of a Testing Framework for Biogas Plants

In this section, we illustrate the use of PPCs in Agile through our experience of developing a framework for monitoring, testing and operating biogas power plants. This development has been performed following SCRUM [18] and in cooperation with the software company Answare-Tech, which is operating in the software and system engineering arena. This industrial collaboration has taken place within *FLEXI ITEA2 project*, in which both UPM and Answare-Tech have worked closely together.

5.1 A Test and Operation Framework for Biogas Plants

Biogas power plants produce electric energy from the anaerobic digestion of the animal meat/vegetable waste. The process of biogas production is composed of four main stages: shredding, pasteurization, homogenization, and anaerobic digestion. Each stage is performed in tanks that must be monitored, tested and controlled.

It is common that customers monitor and control several biogas plants distributed in a geographical area. In addition, they require that the language and framework to monitor, test and operate the plant will be specific of the biogas domain. To satisfy these needs, we decided to evolve an existing domain-specific framework for testing and operating environments, called TOPENprimer [32]. TOPENprimer is devised for testing and operating systems from various domains. Therefore, UPM and Answare-Tech work together to update TOPENPrimer to test biogas plants, i.e. TOPEN Biogas.

5.2 Developing a Flexible Architecture for TOPEN Biogas

TOPEN Biogas was developed through 6 SCRUM *sprints* during 15 weeks (see P3, section 4.3). The Scrum Team was composed of 10 engineers: a Product Owner, a Scrum Master, two architects (architecture team), and six developers (development team). During Pre-game phase, we created the *Product Backlog* with the USs identified by the customer (see P4, section 4.3). Later in the process, the USs were prioritized in each sprint in a Sprint Backlog. The development results of 3 sprints are described below.

- **Sprint 1:** The Sprint 1 was focused on the TOPEN Biogas main functionalities. Following, some selected USs from the Sprint Backlog:

(US1). *Test engineers specify a test case utilizing a user interface and with the biogas plant specific language.*
(US2). *Test engineers compile and execute a test case from the user interface. The results of the test case executions must be shown to them.*
(US3). *Test engineers remotely test/monitor the biogas plant.*

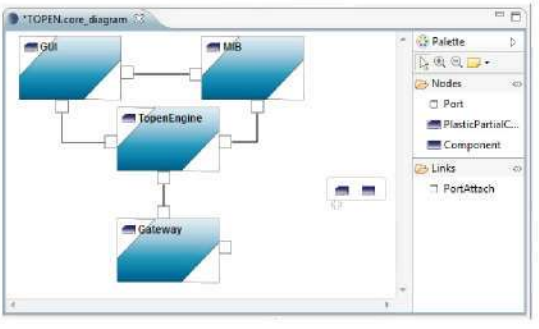

Fig. 3. Topen Biogas Working Architecture of the First Sprint

From these USs, the architecture team identified some non-functional requirements with regard to the distribution, security and the critical nature of the data. The architecture team decided on a distributed architecture and identified 4 PPCs, i.e. 4 incomplete components that had to be completely defined in following sprints. These PPCs are: kernel functions (TopenEngine), graphical user functions (GUI), data management (MIB), and the communication with the biogas plant (Gateway), as shown in Figure 3[3] (see P1 and P6, section 4.3). From now on, our focus will be the PPC *TopenEngine* to illustrate how a PPC can be iteratively updated by considering each sprint working architecture. In the first sprint, to support distributed communication among components, the use of Java Remote Method Invocation (RMI) was decided. As a result, the architecture team defined a crosscutting feature (aspect), which implemented a distribution concern based on RMI. In addition, it was implemented an AVP, which hooked the aspect to the *TopenEngine* (see Figure 4.a).

[3] Figures 3 and 4 are snapshots of Flexible-PLA tool.

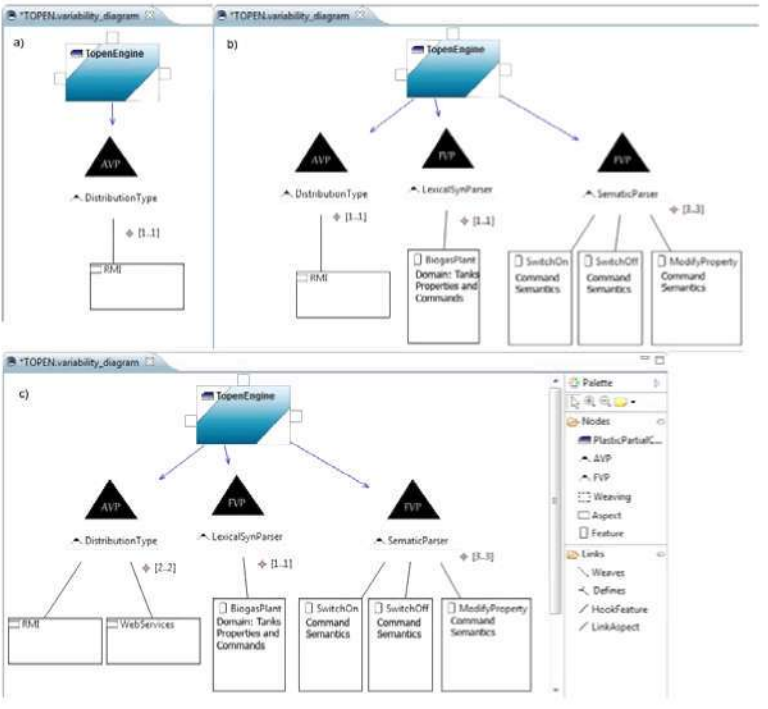

Fig. 4. Topen Engine PPC Adaptation through Sprints

– **Sprint 2:** The Sprint 2 was focused on supporting the operation of the biogas plant. Below, some selected USs from the Sprint Backlog:

(US4). *Test engineers operate biogas tanks and modify the value of their properties.* As a result of this US, the properties and operations supported by each tank were developed. This US was divided into sub-USs as:

(US42). *Test engineers switch on/off a tank, and/or modify the tank properties value by sending commands.*

(US44). *Test engineers be notified about temperature excess in tanks by means of alarm reception.*

From these USs, the architecture team decided to add two features to the TopenEngine PPC. One feature implements the lexical-syntactic parser to analyze the commands/alarms that test engineers send/receive to/from the tanks of a biogas plant (e.g. parsing of references to tanks, properties and operations). The other feature implements the semantic parser to provide of meaning to these commands/alarms (see Figure 4.b). These parsings validate both lexically and semantically the commands and alarms that test engineers send or receive to/from the tanks of a biogas plant. Adding these new User Stories by means of adding new features that hook the TopenEngine PPC implies more effort in specifying the pointcuts and the weaving operators. However we gain in code modularity,

scalability and reusability because these fragments of code are unaware of the linking context (see P5 and P6, section 4.3).

- **Sprint 3:** The Sprint 3 was focused on the physical communication with the biogas plant.

Since the Gateway had a Service Oriented Architecture based on Web Services technology, TopenEngine had to be changed to support Web Services (see P2, section 4.3) . Therefore, the TopenEngine supported both RMI communication with the GUI and MIB, and Web Services communication with the Gateway (see Figure 4.c). This new requirement was solved by hooking a new distribution aspect, which provided Web Services technology. It is important to emphasize that this aspect was not only hooked to the PPC TopenEngine, but also to all the PPC components that needed to communicate with the Gateway. As a result, the distribution of our application was modified just by adding a new single fragment of code. We did not need search in all the components the scattered distribution capabilities of the components previously implemented to modify them. The incremental design of the architecture in subsequent sprints by means of the use of PPC allowed us to flexibly modify the distribution capabilities with the minimum impact both in cost and effort, and to inherently refactor the distribution code (see P5 and P6, section 4.3).

6 Conclusions and Further Work

The deployment of PPCs is an attempt to provide a solution for architectural design of large software systems in Agile. They are malleable components that we have advantageously used for adding or removing pieces of software throughout the different iterations that comprise the development of our *working product*. As a result, we realized that PPCs were *working components* of a *working architecture* that was designed in time to our working framework for testing and operating biogas power production plants.

In cooperation with our industrial partners we have managed to show that it is possible to focus on architecture without suffering from practices that move away the customer from the architecture and development teams. In addition, we realized that non functional requirements can be allocated into the architecture while the discussion/communication with the customer is mainly focused on the set of functional requirements. Therefore, the impact of introducing non-functional requirements in Agile is minimized. From an architectural technical point of view, scaling up can be achieved and, as important, the need of refactoring was sized down: the impact of introducing new features was often less dramatic thanks to the flexibility and reusability provided by PPCs.

As future work, we plan to work in larger size projects to understand what is still missing and to obtain measures and empirical results. Systematization of use of PPCs and automation are two issues that will have to be faced for the approach deployment. Together with Answare-tech we intend to use PPCs in a project for security and safety of intelligent buildings and IT for energy

management systems. In addition, it is necessary to analyze how the use of PPCs can facilitate the maintenance and evolution of software architectures.

Acknowledgments

The work reported here has been partially sponsored by the Spanish MEC (DSDM TIN2008-00889-E), MICINN (INNOSEP TIN2009-13849), and MITYC (FLEXI ITEA2 6022 FIT-340005-2007-37 TSI-020400-2009-066) and by UPM (Researcher Training program). Authors are indebted to Answare-tech and BiogasFuelCell SA for their participation and support during the development of the project.

References

1. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
2. Garlan, D.: Software architecture. In: Wiley Encyclopedia of Computer Science and Engineering (2001)
3. Beck, K., et al.: The Agile Manifesto (2001), `http://www.agilemanifesto.org` (accessed July 2010)
4. Bowers, J., May, J., Melander, E., Baarman, M., Ayoob, A.: Tailoring xp for large system mission critical software development. In: Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002, London, UK, pp. 100–111. Springer, Heidelberg (2002)
5. Dybå, T., Dingsøyr, T.: Empirical studies of agile software development: A systematic review. Inf. Softw. Technol. 50(9-10), 833–859 (2008)
6. Babar, M.A., Abrahamsson, P.: Architecture-centric methods and agile approaches. In: Agile Processes in Software Engineering and Extreme Programming (XP 2008), pp. 242–243 (2008)
7. Cockburn, A.: Agile Software Development. The Cooperative Game, 2nd edn. Addison-Wesley Professional, Reading (2006)
8. Kruchten, P.: On software architecture, agile development, value & cost. Keynote SATURN, Pittsburgh, Pennsylvania, USA (2008), `http://www.sei.cmu.edu/architecture/saturn/2008/keynotes.html`
9. Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, Reading (2004)
10. McMahon, P.: Extending agile methods: A distributed project and organizational improvement perspective. CrossTalk: The J. Defense Software Eng. 18(5), 16–19 (2005)
11. Babar, M.A., Ihme, T., Pikkarainen, M.: An industrial case of exploiting product line architectures in agile software development. In: Software Product Lines Conference, SPLC (2009)
12. Pérez, J., Díaz, J., Costa-Soria, C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: WICSA 2009: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, ECSA (2009)

13. Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Germany (2005)
14. Shore, J., Warden, S.: The Art of Agile Development. O'Reilly Media, Inc., Sebastopol (2007)
15. Vilki, K.: Juggling with the paradoxes of agile transformation. Flexi Newsletter 2(1), 3–5 (2008)
16. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional, Reading (November 2004)
17. Poppendieck, M., Poppendieck, T.: Implementing Lean Software Development: From Concept to Cash. Addison-Wesley Professional, Reading (2006)
18. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice-Hall, Englewood Cliffs (2002)
19. Assmann, U.: Invasive Software Composition. Springer, New York (2003)
20. Bachmann, F., Bass, L.: Managing variability in software architectures, pp. 126–132. ACM Press, New York (2001)
21. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj, pp. 327–353. Springer, Heidelberg (2001)
22. Beydeda, S., Book, M., Gruhn, V.: Model-Driven Software Development. Springer, Heidelberg (2005)
23. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. In: Model-Driven Engineering (2006)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, Reading (2009)
25. Ambler, S.W.: Agile architecture: Strategies for scaling agile development, `http://www.agilemodeling.com/essays/agileArchitecture.htm` (accessed July 2010)
26. Kruchten, P.: Software architecture and agile software development an oxymoron? Keynote Software Architecture Challenges in the 21st Century, USC (June 8, 2009)
27. Erdogmus, H.: Architecture meets agility. IEEE Software 26(5), 2–4 (2009)
28. Madison, J.: Agile architecture interactions. IEEE Software PP(99), 41–48 (2010)
29. Kornstadt, A., Sauer, J.: Tackling offshore communication challenges with agile architecture-centric development. In: WICSA 2007: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, Washington, DC, USA, p. 28. IEEE Computer Society, Los Alamitos (2007)
30. Andrade, L.F., Fiadeiro, J.L.: Architecture based evolution of software systems. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 148–181. Springer, Heidelberg (2003)
31. Fowler, M., et al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
32. Magro, B., Garbajosa, J., Perez, J.: A software product line definition for validation environments. In: Software Product Lines Conference (SPLC), pp. 45–54 (2008)