A Method for Service Agile Construction

Felipe Carvalho[1,3], **Leonardo Guerreiro Azevedo**[2,3], **Gleison Santos**[3]

[1]Petrobras – Petróleo Brasileiro S/A – Brazil

[2]IBM Research – Brazil

[3]Graduate Program in Informatics (PPGI)
Federal University of the State of Rio de Janeiro (UNIRIO)
Rio de Janeiro – Brazil

felipe.carvalho,gleison.santos@uniriotec.br,lga@br.ibm.com

Abstract. *Service-Oriented Architecture (SOA) and agile methods share common drivers. However, there is a lack of guidelines a SOA team should pursue in order to develop services considering best practices, acceptance tests, distributed teams, contract refactoring, among other issues related to SOA principles and agile practices. This work presents a new method that addresses team concerns and needs aiming at a systematic approach for service development using XP's agile practices and SOA principles. We provide best practices, phases and activities that specifically address XP's core practices and service-oriented best practices. We also provide an example of our proposal in order to demonstrate its applicability.*

1. Introduction

SOA (Service-Oriented Architecture) has emerged as a concept that fosters business agility, responsiveness to changes, reuse of corporate assets, cooperation among stakeholders ([Josuttis 2007], [Erl 2008], [Lankhorst and Janssen 2012] and [Gu and Lago 2007]). In parallel, in late 1990's, many lightweight methods were proposed to overcome problems in software development due to documentation driven development and heavyweight processes. In 2001, the Agile Manifesto was published emphasizing collaboration, responsiveness to changes, business understanding, simplicity and agility in software development [Beck et al. 2001]. Nevertheless, despite the common concerns that guide both service-orientation and agility, there is no consensus on how to make use of agile methods in combination with service-oriented development [Carvalho and Azevedo 2013].

Service-Oriented Architecture is an approach for software construction by the composition of loosely coupled services, which capabilities are exposed to consumers via their interfaces [Marks and Bell 2006]. Once a service is made available, modifications on its interface may impact several consumers. Hence, software changes requires a careful design. Nevertheless, predicting reuse is not an easy task in any application context; it is even harder when crossing enterprise boundary [Fowler 2008]. Therefore, the service-oriented principle of designing an interface for maximum reuse and flexibility does not follow the evolutionary design principle, which is the base of agile methods.

We conducted a systematic mapping study searching for a method that proposes a combination of SOA and Agile methods. The results demonstrated that most authors

provide insights/guidelines on SOA concepts and agile practices combination. There is a lack of details on using agile practices for incrementally building service-oriented solutions. This work evolves and systematize in a method the proposed guidelines and best practices for service construction in a SOA environment using XP presented in a previous work [Carvalho and Azevedo 2013].

This work proposes a new method for service development. It aims to fill in the open gaps and offer an approach that is closer to XP practices and concepts, enabling the construction of service-oriented solutions in an agile way. This is a critical phase for product quality and customer satisfaction, since, ”working software is the primary measure of progress” [Beck et al. 2001]. Valuable working software delivered in a continuous, flexible and adaptive manner allows customer to keep competitive advantage.

The remainder of this work is organized as follows. Section 2 presents an analysis of the related works. Section 3 presents our proposal. Section 4 presents and an exemplary implementation. Finally, Section 4 presents a discussion of the main characteristics of a XP and SOA method and an exemplary use of our proposal. Finally, Section 5 presents the conclusion and proposals of future work.

2. Related work

This section presents the systematic mapping study conducted in this work, an analysis of the main related work and the gaps found in the literature.

2.1. Systematic mapping study

We conducted a systematic mapping study in order to assess the existence of methods that address XP and SOA combination in a proper level of detail. Besides, we also aimed at confirming that combining SOA and agile is still an open gap. The research protocol was based on [Kitchenham and Charters 2007], and a complete version of it is available at `http://www.uniriotec.br/~azevedo/SystematicMappingStudyProtocol.pdf`.

In summary, the following string (in a search tool independent syntax) was used to search for articles that addressed SOA development via XP practices: *((soa or service oriented) AND (xp or ”extreme programming”))*. The search was executed on the following search engines: Scopus, Compendex, ACM Digital Library, DBLP Complete Search, IEEExplore. Proceedings of conferences that address service development and/or agile methods were analysed, such as: SBSI, ICEIS, AMCIS, CAiSE, ICIS, ECIS, SOSE, SBCARS, ESELAW and XP Conference. After protocol tests and adjustments, 154 articles came up to be analysed. The abstract of 119 were not in consonance to our research focus, while 4 others were not available for download. So, we analysed the proposals of the remaining 31 works. We found that the articles often do not define phases, roles, responsibilities, deliverables etc. Most of them provide insights, general guidelines or mappings between XP and SOA concepts, but without a focus on a concrete method. Most authors have not either focused on service construction phase. Only one paper has provided best practices on that phase. The next section presents the most relevant articles.

2.2. Relevant works

The most relevant works are described as follows:

[Karsten and Cannizzo 2007]: focus on distributed communication to keeping a fluid communication, such as, presential events where the team establishes work relationships, share know-how, identify shared dependencies and restrictions, and reshuffle among other teams. The goal is to maintain intra team communications and keep members challenged. Besides they emphasize the use of automated acceptance tests to demonstrate story delivery and as a source of documentation for integration purposes.

[Lee et al. 2006]: contribute with best practices related to service construction. They specify that service development projects should be concerned, for example, with the right level of service granularity, interoperability, interaction mode to fit SOA principles and address functional & non-functional standpoints.

[Lee et al. 2005]: describe a method for web services development, composed by phases, roles, activities and deliverables, which goes deeper and proposes an extension to the original method [Tan et al. 2005] to fit XP principles and practices. They do not introduce any new concept, but rather translates those into phases that map to XP's phases, e.g., Requirement and Analysis phases are joined into the Planning phase, while Deployment phase is renamed to Releases.

[Krogdahl et al. 2005]: suggest ideas to enable Lean [Poppendieck 2003] principles in service development. For instance, they propose the usage of an enterprise wide backlog globally prioritized. They also propose open service interface, quickly deployed to production via a working pilot. When other applications start to reuse services, both the interface and internal logic can be constantly refactored and refined to fit business actual needs. They also advise frequent meetings with team representatives for synchronizing knowledge and shared impediments.

[Ivanyukovich et al. 2005]: depict the impacts incurred by XP practices on service development. For example, Planning Game can be used to allow the team to see the services backlog as a set of features to be developed. Refactoring can be used for service continuous redesign and improvement. Test-driven development would lead to automatic coordination of dependencies among services, thus enabling complex and reliable orchestration of those services at runtime.

[Maranzato et al. 2012]: presents a framework developed for improving communication among all stakeholders involved in the development of a product. The product was divided by features addressed by different teams and customers. All teams shared a "mega backlog", where items were denominated "themes", which granularity was fit to allow customers to maintain a vision of the whole product. Every three months, the "mega backlog" was revisited to ensure an up-to-date product view and to make it easier to agree on business priorities. Besides this event, several other regular meeting took place including people in distinct roles aiming to maintain communication flow.

Table 1 presents the characteristics addressed by relevant works. The symbols indicate the extent each characteristic was addressed: (-) the characteristic was cited, but no examples or guidelines have been provided; (+) some examples were provided to follow the characteristic; (++) the characteristic was addressed in a good level of details. Those works were considered in the phase definitions of our proposal (Section 3). The last column of the table relates characteristics to gaps presented in the next section.

Table 1. Characteristics addressed by authors

Characteristic	Reference	Level of Detail	Gaps
C1. Frequency of iterations	[Maranzato et al. 2012]	+	G1
C2. Acceptance tests	[Tan et al. 2005] [Karsten and Cannizzo 2007]	- +	G2
C3. TDD	[Tan et al. 2005] [Ivanyukovich et al. 2005]	- -	G3
C4. Refactoring	[Karsten and Cannizzo 2007] [Ivanyukovich et al. 2005]	+ -	G4
C5. Service construction principles and best practices for distributed and co-localized teams	[Lee et al. 2006]	++	G5

2.3. Gaps in the literature

We identified the following gaps from the most relevant works presented in Section 2.2.

G1. Iterations: Most authors provide ideas about how to use XP practices for service development, but they do not discuss how iterations fit in a service development model, although XP is an iterative and incremental method [Wake 2002]. The XP lifecycle is divided into 5 iterative phases [Beck and Andres 2004]. The WSIM-XP method [Tan et al. 2005] includes phases, activities, roles and deliverables. However, it does not define where iterations take place, or how iterative is each phase.

G2. Acceptance tests: None of the authors discusses in depth the usage of acceptance tests (also known as ”functional tests”). These tests are written by the customer aiming at assessing software adherence to business. A given feature cannot be considered as ”done” until it fits all acceptance tests ([Beck and Andres 2004] and [Wake 2002]). The tester is responsible for helping the customer in writing the acceptance criteria, but those are not actually translated into automated tests, or used in any phase or activity [Tan et al. 2005].

G3. Unit tests: Despite the importance of unit tests being written before production code [Beck and Andres 2004], none of the authors mentions the usage of tests during services development, neither do they discuss best practices or issues inherent to test-driven development of services. Unit tests should be written before production code, method by method, with several purposes (e.g., define the expected behavior of new code, clarify the behavior of existing code, among others) [Beck and Andres 2004].

G4. Refactoring: Refactoring is also not handled in detail by the related works, despite being a routine practice in a XP project ([Beck and Andres 2004] and [Wake 2002]). It aims at improving design and fostering reuse, based on the support provided by unit tests. A service interface is a corporate asset. Reuse of corporate assets is a major driver for SOA and Web Services [Josuttis 2007]. Being refactoring such an ordinary and primordial activity in a XP lifecycle, one could expect services refactoring issues to be discussed by authors; more specifically, the issues inherent to interfaces refactoring, due to the complexity of impacting multiple customers and business partners.

G5. Service construction best practices: Most authors do no mention to issues or differences inherent to service development, although there are several differences between service oriented software engineering and ”traditional” software engineering [Gu and Lago 2009]. The exceptions are [Karsten and Cannizzo 2007] and [Lee et al. 2006] works, which address service construction at some level of detail from both team communication and product robustness standpoints. However, service construction phase was often not the focus of the related works. Many authors provide insights, mapping of concepts, indications of possible ways to combine SOA and XP, but a few have referred to construction best practices.

3. A SOA-XP method

Considering the SOA and XP concepts, the following main characteristics should be addressed by methods that combine both approaches: (i) it should define how frequent are each activity, since XP is an iterative and incremental method ([Beck and Andres 2004] and [Wake 2002]); (ii) it should address the usage of Acceptance Tests as the starting point of any coding activity [Beck and Andres 2004]; (iii) it should not address testing and coding activities separately; instead, tests should be written prior to execution code, being automated and incorporated to a test suite in order to provide feedback on the product quality and readiness at any time [Beck and Andres 2004]; (iv) it should address refactoring - a core XP practice [Beck and Andres 2004] - especially, contract refactoring, which is not a trivial activity given the impact on multiple customers once a service is made available [Gu and Lago 2007]; (v) it should address service-oriented principles and best practices [Erl 2008], providing examples that make easy for teams, either distributed or co-localized, to execute service construction.

The focus of our work is specifically on activities that comprise the construction phase, and not on an entire service development lifecycle. We propose six activities to be performed during the construction phase of a service development lifecycle (Figure 1). These activities are further detailed as follows. The four phases show on Figure 1 are an example of a four-phase service development lifecycle where our proposal could fit.

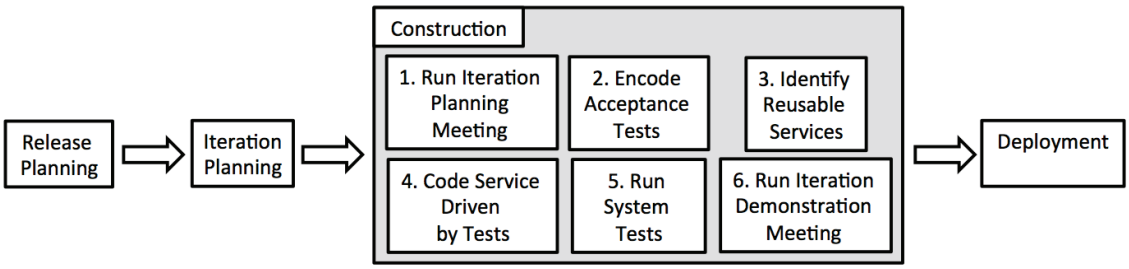

Figure 1. Our proposal of a construction phase

The **input** for the construction phase is the iteration backlog (a prioritized list of features to be constructed during the iteration) and the acceptance criteria (a set of criteria written by the customer or its representative that indicate when a feature is adherent to business needs). The **output** resulting from the construction phase are the services deployed and running in a neutral environment, where the iteration product can be used and validated by the customer. The next sections detail each activity, using the following

notation: ***i*** denotes the artifacts that serve as input to the activity, ***o*** denotes outputs and ***f*** indicates activity's frequency.

1) Run Iteration Planning Meeting *(**i**: prioritized features / services, acceptance criteria (text), **o**: iteration backlog, **f**: once per iteration)*:

The construction phase starts with the Iteration Planning meeting where Customer or its representative present the prioritized services to the team. Services should be documented using story cards [Patel and Ramachandran 2008] with information about its business purpose and the expected behavior. When presenting the story, the customer also defines how to measure the service adherence to business through acceptance criteria. The story card may define both functional and non-functional information. A story will be considered as finished or delivered when all of its acceptance criteria are met.

2) Encode Acceptance Tests *(**i**: prioritized features (services), **o**: acceptance tests (executable code), **f**: once per feature)*: Tests should automatically and periodically run in a neutral continuous integration environment [Beck and Andres 2004]. Acceptance tests should be periodically executed along with the entire test suite to provide feedback to all stakeholders about the services adherence to business needs at any given time. Thus, the acceptance criteria, which are primarily written in a textual manner, need to be translated into executable code [Karsten and Cannizzo 2007]. There are several tools that allow features to be specified textually and automatically translated into executable tests, e.g.: JBehave (`http://jbehave.org`), Cucumber (`http://cukes.info`).

3) Identify Reusable Services *(optional) (**i**: reusable services documentation, **o**: documentation of identified reusable services, **f**: once per feature)*:

Reuse of corporate assets is a major driver in SOA [Erl 2008]. Hence, the method must have an activity related to identifying existing services that can be reused, so the solution being worked on can built on a pre-existing piece of software that solves the business need, either completely or partially. Besides the programmer, the customer is another stakeholder of this activity.There may be cases where his/her organization requires a given set of services to be used (e.g., services that handle security according to business legal demands). Customer provide information about those services.

Besides, software design emerges as the simplest response to the acceptance tests [Beck and Andres 2004] - upfront design specifications do not apply. Therefore, the artifacts involved in this activity would be documentation about the reusable Web Services, instead of a software design specification for the entire product.

4) Code Service Driven by Tests *(**i**: acceptance tests (executable code), **o**: services deployed and running in a neutral environment for system tests, **f**: once per feature)*:

Coding and testing activities should be performed together, in an incremental and iterative way ([Beck 2001] and [Wake 2002]). Therefore, instead of focusing solely on production code, this phase is made of both coding and testing activities, at both unit and integration levels, comprising several steps as follows.

The first step is to execute the tests with no production code [Beck and Andres 2004]. The very first execution of such tests usually fails. Then, iteratively, design and implement the simplest code that makes the test pass, while keeping other tests still running (XP's practice of "Simple Design" [Beck and Andres 2004]),

until the implementation fills the requirement. When possible, simplify the design (XP's practice of "Refactoring").

In a first moment, just the service endpoint is created. Nevertheless, usually this is not the only layer to be built. The feature internal design is created incrementally, throughout the strive to make the acceptance tests to pass, following the same steps [Beck and Andres 2004]: (i) create an unit test; (ii) design and implement the simplest code that makes the test pass, while keeping other tests still running; (iii) repeat previous steps; (iv) if possible, simplify the design;

All code is written based on acceptance criteria, which derive integration tests, which derive unit tests, which derive production code. During the construction phase, programmers often refer to customers (On-site Customer) to provide guidance and clarifications on the expected behavior of the service or of a given layer [Abrahamsson 2003].

Such steps must be followed for functional and non-functional requirements. Acceptance criteria may also include non-functional requirements, especially if those provide a competitive advantage for the organization [Patel and Ramachandran 2008]. XP teams measure progress by the feature's code adherence to acceptance tests [Wake 2002].

As an example, if an airline company is exposing ticketing services to be used by cheap flights searchers (e.g., Fare Compare), fast responses become very important to the business; therefore, it should be included as an acceptance criteria. A separate set of integration tests becomes necessary to ensure that response time is kept below an upper threshold above which business is harmed. Ensuring adequate functional response from third-party services may be a challenge, especially on scenarios where such services respond differently to the same request depending on given variables. For instance, a stock quoting service provided by a business partner is expected to respond differently to a request for a given company quoting along the day. Hence, a test would have to be relaxed from the functional standpoint to consider as valid not a response with an exact value, but a response that is well-formed and with a non-negative number.

Another example related to third-party services is billing. For example, on the stock quoting example, if each request is charged for a given price, and the suite of integration tests requests stocks several times a day, these integration tests may quickly become expensive, from a financial standpoint. A solution would be to put those integration tests in a separate suite running in a lower frequency, periodically ensuring the service is adherent to specifications. A service composition testing may go under the same problems, as each service integrating the composition may have different behaviors. To overcome this issue, a test-enabled ESB (Enterprise Service Bus - [Josuttis 2007]) can be used [Ribarov et al. 2007]. In this case, it is possible to "hook" mock services replacing given parts of the composition to provide a more predictable response from the composition's entry point.

5) Run System Tests *(optional) (**i**: acceptance tests, **o**: defects information, **f**: once per feature)*:

The overall system functionality must be assessed considering all components running together. Non-automated acceptance tests must be performed in order to evaluate if all requirements are covered by the solution, when all conditions cannot be automatically tested. This activity focuses on acceptance tests and defects are artifacts to be addressed.

In the stock quoting service example, responses may vary along the day and require human intervention for validation. There may also be cases where a service should respond to a request by providing a file attached to the response message. In a case where the acceptance criteria refers to the contents of such attached file, human intervention may be necessary, due to lack of tooling support for automatically validating is contents.

6) Run Iteration Demonstration Meeting *(**i**: executing services, **o**: customer sign-off, list of changes or adjustments to be made to the delivered product, **f**: once per iteration)*:

Upon the end of the iteration, the features are presented to the customer in the Iteration Demonstration meeting. The team presents the features running and meeting the acceptance criteria, and the customer is responsible for signing off the feature. When customer realizes the delivered service does not exactly match what he/she had in mind or a business change demands the service to work differently, he/she is responsible for clarifying the desired behavior, prioritizing it over the other features and scheduling its inclusion in another iteration's scope.

It is important to emphasize that the iteration length in weeks is an agreement between Customer and Team, e.g., 1-3 weeks [Beck and Andres 2004] or 4-weeks iterations [Maranzato et al. 2012] that fits stakeholders' schedules. There is not really a rule about iterations length, although authors agree that all iterations should have the same length for the team to be able to assess its velocity.

4. Discussions and example

We have previously identified five characteristics that a SOA and XP method should address for service construction (Table 1). Considering those items, we consider **C4** and **C5** to be the most critical, for the other items involve concerns mostly inherent to a single team, whereas these two often impact multiple parallel teams and/or customers, possibly bringing losses to business. They are discussed in the following sections. Besides, an example of the method use is presented.

4.1. Best practice: Distributed teams

SOA teams are often geographically dispersed and end up facing communication problems. This is a fundamental difference from XP, which recommends face-to-face communication to address this exact problem. Therefore, it is not possible to use a XP practice to handle such situation. Authors suggest different approaches to bridge this gap.

Loosely coupled interfaces can be used to minimize dependencies among distributed teams [Karsten and Cannizzo 2007]. They propose the use of a common integration area for the updated service upon a change in its interface. This area is used by other teams to update their references to the updated service. Besides, we emphasize the importance of using contract-first development [Erl 2008]. If distinct teams depend on a given contract yet to be defined, we recommend the contract definition to receive a high priority during the construction phase, so that teams can start working independently as soon as possible, without harming any team's speed. Nevertheless, a service full up-front analysis may prove itself non-viable or useless, given its characteristics are likely to change with time ([Erl 2008] and [Fowler 2008]). Instead, we recommend a minimal effort spent on the contract definition, with the purpose of defining the very least teams need to work

in parallel, leaving contract details open for as long as possible [Krogdahl et al. 2005]. Notice that assigning top priority to the contract definition is not contrary to spending minimal effort on this activity. We recommend a minimal definition of the contract to be done early, in order to allow teams to progress independently.

Developers should be divided into small teams, either collocated or distributed. As to improve communication among teams, as recommended by [Karsten and Cannizzo 2007], local teams should be reshuffled at each planning session, as a way for people to get to know other parts of the system, and also to keep teams challenged and motivated. Communication can also be addressed by using Web-based technologies such as Wiki, Portal, forum and issue tracker. These serves both for teams to establish internal communication, as well as documenting information important to customers. Additionally, RSS can also be used to let people know about changes. When dealing with distributed teams, it is important to have some sort of high-level regular communication among teams.

4.2. Best practice: Contract refactoring

At the end of iteration, customer expects to see the feature passing acceptance tests [Wake 2002]. Therefore, acceptance criteria derive the whole implementation of the feature. In fact, it is possible to say that acceptance criteria derive the business purpose being pursuit on a specific feature. They represent the business needs on a specific matter. Acceptance criteria derive the service's contract, which is one part of the simplest solution (XP's Simple Design) that passes the acceptance tests. Therefore, the contract reflects the business needs on a specific matter. On the other hand, how to deal with contracts in a XP approach to service-oriented development is an important issue to be addressed. The general perception is that XP's orientation towards continuous refactoring would drive a service-oriented initiative to a scenario of constant overhead and rework of distributed teams, due to constant contract refactoring.

Nonetheless, there is not a conclusive answer towards the contract's behavior in either agile or non-agile approaches. In fact, as stated before, a contract reflects an organization's needs over a specific feature at a given time. Therefore, all service construction work is done to meet business needs at that time, regardless of method (waterfall, RUP, agile). So, if organization's needs changes, there's no way to avoid a contract being changed, regardless of the employed development method. In such case, there are several strategies documented in literature to address impact on consumers [Karsten and Cannizzo 2007].

In regards of XP usage of Refactoring, it is important to improve code quality and design without modifying its external behavior [Beck and Andres 2004]. So, there is no evidence indicating that contract refactoring in a XP approach would take place more or less often than in a non-agile scenario. Contracts reflect business needs, and, when they change, it is inevitable to expect modifications on contracts and impact on consumers.

There is a fundamental difference, though, from the flexibility standpoint. SOA generally dictates contract design towards reuse, as to reduce time-to-market. On the other hand, XP aims at the current iteration needs, while taking also into consideration flexibility needs that are known, concrete and unlikely to change [Beck and Andres 2004]. Unknown or unclear flexibility is disregarded [Poppendieck 2003].

4.3. Example project using our method

We developed a sample web services project (available at `https://github.com/felipecao/xsoa-example`) to exemplify and perform an analysis of our proposal.

The example is based on the following scenario. A fictitious company called ”SmartBrick” has developed a solution to assess productivity on building sites. It is based on a web service that sends data to a PDA (Personal Digital Assistant) based on user roles: administrator users are able to download data from all sites; non-administrator users are only allowed to see information regarding sites they have been assigned to. The project's home page presents the acceptance criteria for ”sites download” feature.

We use the terms ”feature” and ”service” interchangeably [Ivanyukovich et al. 2005]. Following our proposal, the acceptance criteria are the starting point for the service construction. So, as far as ***Run Iteration Planning Meeting*** goes, all artifacts are available. In our example, ”sites download” is the top priority (fitting ”prioritized features”), and its acceptance criteria are available.

The next step is to run ***Encode Acceptance Tests*** and translate those criteria into acceptance tests. This step starts breaking the acceptance criteria into textual acceptance tests. There are several ways to write acceptance tests. We have used the GIVEN-WHEN-THEN syntax, which is supported by Cucumber, a BDD tool we chose for this example. Once the acceptance tests are ready, we transpose them in the project, into *.feature* files, which are used by Cucumber generate executable code. An example of *.feature* file and executable code are available at `http://goo.gl/VPGFYg` and `http://goo.gl/bsols9`, respectively. Notice that Cucumber annotations are responsible for matching each line from acceptance tests to test methods. Cucumber also binds the parameter values defined in each acceptance test (written between ””) to actual Java types in test methods.

The third phase is to ***Identify Reusable Services***. Nevertheless, this example considers a scenario where there is not a pre-existing service to be reused, thus, this phase is not executed. Notice that our proposal does not state all phases are mandatory; instead, we consider that some phases may be skipped, according to the situation at hand.

The next activity is to ***Code Service Driven by Tests***. Following our proposal, after acceptance tests development, unit tests should be created for each part being built. An example unit test is available at `http://goo.gl/KMaJmI`. These tests should be built prior to writing service code. When test is ready: run the test; let it fail; write the simplest code that passes the test; check if it succeed; and, then, refactor. Other unit tests, created for other layers, are available within the project. Integration tests have also been created (`http://goo.gl/6446nP`).

The ***Run System Tests*** are not illustrated, since it requires human intervention to explore the feature in areas where automation is not possible.

For the final step, ***Run Iteration Demonstration Meeting***, the features need to be deployed and available for demonstration to the user or customer. This demonstration is usually based on the acceptance criteria and/or acceptance tests, as to demonstrate the product fits the expectations the customer has previously registered on the same criteria. Cucumber reports can also be used to make customer confident that acceptance criteria are being ensured at all times.

This example helped to indicate the applicability of the proposal. Although the example was not conduct by a real team, the acquired knowledge allowed the method evolution towards its adequacy for usage in real scenarios. For example, when compared to an execution of the method proposed by [Tan et al. 2005], our proposal indicated the existence of fewer defects during user tests. We also had front-end developers working on the development of an application to present data coming from the services on the PDA, in parallel to service developers. This parallel working indicated the importance of an early minimal contract definition to allow teams to progress. The execution of both methods (ours and [Tan et al. 2005]) was done by having the same team implementing the example system described in the beginning of this section and strictly following the instructions outlined by each step of each proposal.

5. Conclusion

Service orientation and agile methods are important paradigms for system development with similar concerns. Despite this similarity in fundamental concepts, there is no consensus on how to use agile methods in service-oriented system development.

We have presented how authors face the combination of agile concepts to service oriented solutions, and the protocol of a mapping study from which was possible to conclude that most authors do not provide a method that defines enough details to be followed by a team. This work proposes a method for the construction of service-oriented solutions, with phases, activities and artifacts, addressed in an iterative-incremental manner.

We have also presented a sample project that illustrates the use of our proposal. Our proposal ensures meeting of customer's acceptance criteria through an automated test suite running regularly. In the case of non-functional requirements, if a source code modification drives to a violation of a given threshold, the test suite fails and feedback is immediate. Given the results of our systematic mapping study, it is possible to notice that such concerns are not address by other authors. Besides, following our proposal, software product will only be made available for system tests, and subsequent customer validation, once all acceptance criteria are met. Finally, a Test-Driven approach leads to a more decoupled code, which in turn is easier to refactor, since tests ensure the systems external behavior is maintained after a change [Beck and Andres 2004].

As future work, we point out to expand the scope of this proposal to include other activities of a service development lifecycle, e.g., planning and deployment. Another important future work is evaluate our proposal in a distributed team environment using a real scenario, and compare the use of our proposal to other approaches.

References

Abrahamsson, P. (2003). Extreme programming: First results from a controlled case study. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 259–266. IEEE.

Beck, K. (2001). *Planning extreme programming*. Addison-Wesley Professional.

Beck, K. and Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., et al. (2001). The agile manifesto. *The agile alliance*, 200(1).

Carvalho, F. and Azevedo, L. G. (2013). Service agile development using xp. In *7th Intl Symposium on Service-Oriented System Engineering (SOSE 2013)*, pages 254–259.

Erl, T. (2008). *SOA: principles of service design*, volume 1. Prentice Hall.

Fowler, M. (2008). Evolutionary soa. `http://martinfowler.com/bliki/EvolutionarySOA.html`.

Gu, Q. and Lago, P. (2007). A stakeholder-driven service life cycle model for soa. In *2nd International Workshop on Service-Oriented Software Engineering*, pages 1–7. ACM.

Gu, Q. and Lago, P. (2009). Exploring service-oriented system engineering challenges: a systematic literature review. *Service Oriented Computing and Applications*, 3(3):171–188.

Ivanyukovich, A., Gangadharan, G., D'Andrea, V., and Marchese, M. (2005). Towards a service-oriented development methodology. *Journal of Integrated Design and Process Science*, 9(3):53–62.

Josuttis, N. M. (2007). *SOA in Practice*. O'reilly, 1st edition.

Karsten, P. and Cannizzo, F. (2007). The creation of a distributed agile team. In *Agile Processes in Software Engineering and Extreme Programming*, pages 235–239. Springer.

Kitchenham, B. A. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

Krogdahl, P., Luef, G., and Steindl, C. (2005). Service-oriented agility: An initial analysis for the use of agile methods for soa development. In *Services Computing, 2005 IEEE International Conference on*, volume 2, pages 93–100. IEEE.

Lankhorst, M. and Janssen, W. (2012). Agile service development. *New York: Springer*.

Lee, E. W., Tan, P. S., ChenG, Y., et al. (2005). Web service implementation methodology. *Organization for the Advancement of Structured Information Standards (OASIS)*.

Lee, S. P., Chan, L. P., and Lee, E. W. (2006). Web services implementation methodology for soa application. In *2006 IEEE Intl. Conf. on Industrial Informatics*, pages 335–340.

Maranzato, R. P., Neubert, M., and Herculano, P. (2012). Scaling scrum step by step:" the mega framework". In *Agile Conference (AGILE), 2012*, pages 79–85. IEEE.

Marks, E. A. and Bell, M. (2006). *Service Oriented Architecture (SOA): a planning and implementation guide for business and technology*. Wiley. com.

Patel, C. and Ramachandran, M. (2008). Acceptance test driven story card development for xp (agile software development). In *Proceedings of the International Computer Science and Technology Conference*.

Poppendieck, M. (2003). *Lean software development: an agile toolkit*. Addison-Wesley.

Ribarov, L., Manova, I., and Ilieva, S. (2007). Testing in a service-oriented world.

Tan, A., Ang, C. H., Lee, E. W., and Haines, M. (2005). Web service implementation methodology: Case example using extreme programming. *Organization for the Advancement of Structured Information Standards (OASIS)*.

Wake, W. C. (2002). *Extreme programming explored*, volume 1. Addison-Wesley.