ROBUSTIFYING THE SCRUM AGILE METHODOLOGY FOR THE DEVELOPMENT OF COMPLEX, CRITICAL AND FAST-CHANGING ENTERPRISE SOFTWARE

Marcos Vescovi
Finansolo Software, PO Box 847, Aptos, CA 95001, U.S.A.

Flavio Varejão
Departamento de Informática, Universidade Federal do Espirito Santo, Vitória, ES 29000, Brazil

Vagner Cordeiro
Finansolo Software, PO Box 847, Aptos, CA 95001, U.S.A.

Keywords: Agile methods, Scrum methodology, Software design, Software entropy, Change curve.

Abstract: A class of complex enterprise software including financial, taxation and supply chain management software contains mission critical functionality and change requests are substantial and frequent. Agile methodologies provide the adaptability but not the robustness necessary to deal with the criticality and to avoid software entropy. Task analysis shows that a significant effort of analysis and design is required to flatten the change curve. The Robust Agile Methodology, "R-Agile," is proposed with the adaptability to handle fast-changing requirements, and the design and test necessary to handle complexity and criticality.

1 INTRODUCTION

Boehm and Turner (2003) suggest that risk analysis be used to choose between adaptive Agile and predictive Plan-driven methodologies. The authors suggest that each side of the continuum has its own home ground. The Agile methodologies are well suited for low criticality software with frequent change of requirements, and the Plan-driven methodologies are well suited for high criticality software with stable requirements. Over the past decade we have been developing enterprise software for Fortune 500 and e-Commerce companies presenting both challenges: high criticality and frequent change of requirements. Additionally, and more importantly, we had to deal with complexity and the risk of software entropy. Software entropy is a state reached over time as less well designed software becomes harder and harder to change. At a point, the cost of certain changes can become so high that they are not worth undertaking.

In order to cope with these challenges, we have successfully developed an adaptation of the Scrum Agile Methodology. We have taken advantage of the adaptability and have added design and test to robustify the methodology. The change curve says that as the project runs, it becomes exponentially more expensive to make changes. In this paper we focus on design and argue that a substantial effort is necessary to flatten the change curve.

In section 2, we discuss the suitability of traditional Plan-driven and Agile methodologies. In section 3, we present the characteristics of critical and fast-changing complex enterprise software. In section 4, we discuss the key role of analysis and design for flattening the exponential change curve. In section 5, we present the Robust Agile Methodology "R-Agile," a more robust and agile methodology. We close with the conclusions.

Vescovi M., Varejão F. and Cordeiro V..
ROBUSTIFYING THE SCRUM AGILE METHODOLOGY FOR THE DEVELOPMENT OF COMPLEX, CRITICAL AND FAST-CHANGING ENTERPRISE SOFTWARE.
DOI: 10.5220/0003459800700079
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 70-79
ISBN: 978-989-8425-57-7
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

2 SUITABILITY OF METHODOLOGIES

In this section, we briefly discuss and summarize the advantages and drawbacks of Plan-driven and Agile methodologies. More extensive comparisons can be found in the work of Boehm and Turner (2003) and Fowler (2001), who focus on design and discuss planned vs. evolutionary design.

2.1 Plan-Driven Methodologies

Traditional Plan-driven methodologies, such as the waterfall method (Royce, 1970), are based on the principle that time spent early in the software production cycle can lead to greater economy at later stages. McConnell (1996) shows that a bug found in the early stages (such as requirements specification or design) is cheaper in money, effort and time to fix than the same bug found later on in the process.

With Plan-driven methodologies, a substantial effort of analysis and design is performed up front. This is an important advantage of the Plan-driven methodologies, leading towards a flatter change curve and potentially avoiding software entropy. Another advantage relates to the extensive testing phase which is necessary to ensure correct functioning of mission-critical functionality. These advantages are well-suited to handle complexity and criticality of functionality.

The main drawback of the Plan-driven methodologies is related to the long release cycles. Most of the design is performed without learning from later phases of development and operations. With shorter cycles, a lot can be learned from usability and incorporated into the design. Design is usually longer than needed and many unnecessary features are included in the project. The project costs more and takes a much longer time to market, which impacts business negatively. Customers most often get involved later in the process when a full-packaged release is available.

Besides the above inconveniences, the Plan-driven methodologies are not well-suited and adaptable for fast changing requirements software. The design and overall project is rigid and requirement changes are not welcome.

2.2 Agile Methodologies

Larman and Basili (2003) present a brief history of iterative and incremental development, showing that adaptive methods have been proposed since the 1970s. The "Manifesto for Agile Software Development" was published by Kent et al. (2001). The Agile methods promote development, teamwork, collaboration, and process adaptability throughout the life-cycle of the project. Such methods became popular in the past decade, in particular with fast paced web development.

Agile methods propose much shorter release cycles, allowing quick feedback and adaptation. The process involves customers more often, which results in increased customer satisfaction. Developers are empowered, feel more responsible and become better contributors. Agile methods are particularly well-suited for fast-changing requirements software.

The main inconvenience with the Agile methods is associated with decreased analysis and design, which can result in software entropy and exponential change curve. This is especially true in the case of complex and large software. Another drawback is that the reduced testing phase may not be sufficient to ensure the quality necessary for mission-critical functionality.

In summary, Plan-driven methods are better suited for critical and complex software while Agile methods are better suited for software requiring fast changes. A methodology for dealing simultaneously with complexity, criticality and adaptability is sought. This paper proposes such a methodology.

3 COMPLEX, CRITICAL AND FAST-CHANGING SOFTWARE

Over the past decade we have developed enterprise payment and financial software, supply chain management and taxation software for several Fortune 500 and e-Commerce companies world-wide. The specific products and solutions developed were functionally different but shared similar characteristics. The most relevant characteristics are presented below. For simplicity, we are not discussing the additional complexity associated with the enterprise requirements of performance, scalability, robustness and security.

1. Large number of objects for modeling the domain;
2. Complex relationships between the objects;
3. Complex calculations and processing methods;
4. Substantial and frequent change of requirements;

5. Tight deadlines, limited resources and pressure from customers;
6. Mission critical operations dependent on the software.

The first two characteristics have implications on the size and complexity of the objects and data model. Without careful analysis and design, objects and database schemas became large and overly complex. Coding became more difficult and bugs became more numerous and harder to find and fix. Extensions and modifications became exponentially expensive. Changes in data and object models were usually deep and came at a higher cost. In our experience, the tasks of abstraction and generalization of concepts were necessary to simplify the model and required substantial analysis and design. Without this effort, complexity takes over and the change curve becomes exponential.

The third characteristic relates to the complexity of calculations and processes. Billing, revenue sharing and taxation are common examples of calculations in which the effort to define the mathematics required as much effort as coding itself. The consequences of careless analysis and design were often buggy implementations requiring various iterations to fix. The iterations were costly, involving various releases and testing phases. In our experience, non-modular design created the typical scenario of gas plant complexity, which was hard to understand, extend and fix. In order to accommodate for customer requests, branching became the faster and easier solution. But maintenance of the various branches came at a prohibitive cost. The complexity and its related costs were again exponential.

The fourth characteristic is also challenging. In our experience, the change of requirements was frequent and continuous for a single system and each customer wanted something different. If there was no change of requirements we could have envisioned various steps of refactoring, thus improving the design over time. But the continuous change on top of the issues described with the first three characteristics made development harder. Carefully designing a system for extensive change was the key to flatten the change curve. Besides using well architected frameworks, in some cases we used domain models, rule engines and workflow engines to deal with the frequent change of requirements. Up-front design was necessary to develop such architectures.

The fifth and sixth characteristics simply added stress to the development team. This paper describes a methodology combining the best practices of the Agile and the Plan-driven methodologies. We focus, in particular, in adding analysis and design to the Agile Methodology, which relates more directly to characteristics 1 through 4 above. In order to deal with the implications of characteristics 5 and 6, other issues need to be examined. First, there has to be resolution around the deadline-milestone prevailing in the industry. Either the customers need to agree to the more adaptable Agile methodologies or some rigidity and predictability need to be added to the methodologies. Another important issue relates to the stricter testing necessary to ensure correct functioning of critical functionality. Although the proposed methodology contains the necessary testing component, this paper focuses on the need for further analysis and design and how it has been balanced within the proposed methodology.

4 THE ROLE OF ANALYSIS AND DESIGN

We have experienced an exponential change curve on various occasions, in particular when fast development took priority and did not allow for careful analysis and design. This happened despite the fact that our engineering team was composed of talented Silicon Valley developers. In such cases, changing or enhancing the system became prohibitively expensive and some of the systems had to be redesigned and rebuilt.

According to Fowler (2001), Agile methods have rejuvenated the notion of evolutionary design with practices that allow evolution to become a viable design strategy. It also provides new challenges and skills as designers need to learn how to do a simple design, how to use refactoring to keep a design clean, and how to use patterns in an evolutionary style. Simplicity is at the core of the Agile methodologies. We argue that achieving Simplicity in a more substantial way that can avoid software entropy requires a significant effort of analysis and design.

The definition of Simplicity proposed by Beck (1999) is, in order (most important first):

1. Runs all the tests
2. Reveals all the intention
3. No duplication
4. Fewest number of classes or methods

If we further investigate the criteria above we see that Simplicity (as defined by Beck), especially in the case of complex enterprise software, can only be achieved in a more substantial way with careful analysis and design, contrasting with the "quick code and test" premises of the Agile methodologies. Let's examine the criteria.

1. Runs all the tests

This is a Basic Requirement that a system must verify its intended functionality. This is minimal and required for most methodologies so we will not analyze this any further.

2. Reveals all the intention

This is an Adequacy Requirement and should be required for most methodologies as well. Fulfillment of this requirement is often intuitive and is ultimately defined by customer satisfaction. But for a system to reveal all the intention, it must at least have a complete model (i.e. it must cover the functionality) that can lead to accurate enough and precise enough results (i.e. the results must be acceptable). In terms of coding, the underlying model must be intuitive and clean so that communication and collaboration can happen. In terms of usability, careful interaction and graphical design must also be realized. As we can see, this criterion is not obvious to achieve and we find it difficult to be realized without careful analysis and design.

3. No duplication

This is an Optimization Requirement desirable in other methodologies as well. No Duplication requires the creation of modular code so that it can be reused. The idea is to have more fundamental building blocks that are used to build the system.

If we interpret this criterion as not duplicating code in its most strict definition, i.e. not repeating code that is exactly the same, then the criterion is relatively easy to understand and follow. However, a more powerful and interesting interpretation of the criterion is the creation of more generic and abstract code that can be used to replace similar (and more numerous) pieces of code.

If we perform Cognitive Task Analysis (Chandrasekaran, 1986), (Chandrasekaran, 1990), (Pirolli & Card, 2005), we see that the design of modular code that is generic and abstract to be reused, involves the following mental tasks: (1) decomposition of the problem into smaller functional or structural parts, (2) pattern matching to identify sets of similar parts, (3) creativity to create generic and abstract modular code to cope with sets of similar problem parts, (4) synthesis for combining and integrating the modular code, (5) analysis for critiquing the modular code and its integration, and (6) modification capability for adjusting the modular code and its integration whenever necessary.

No duplication, in its more extensive interpretation, is underlying the construction of frameworks and is at the core of software architecture. Except for simpler software, it is difficult to imagine the achievement of this criterion without careful analysis and design.

4. Fewest number of classes or methods

The most basic interpretation of this criterion is that classes and methods that are not necessary should not be developed in advance. Classes and methods that might be needed in the future should not be implemented until they are definitely needed. This criterion accords to the fundamental principles of the Agile methodologies and is not too difficult to interpret and follow.

However, a different interpretation of this criterion has a fundamental impact when dealing with complex software. In order to cope with the complexity associated with the large number of classes, their relationships and their complex methods, it is important to design fewer but more powerful classes and methods. It is necessary to define classes and methods that are generic and often abstract. Fewer generic and abstract classes can replace multiple and more numerous specific and concrete classes, simplifying the overall representation. Fewer modular generic and abstract methods can replace multiple and more numerous specific and concrete methods as well, once again, simplifying a complex problem. Design patterns, for example, although more complex constructs themselves, are powerful mechanisms that simplify the overall design.

The examples above are just a few examples of interpreting the criterion as creating fewer but more powerful concepts. We claim that the design of these fewer but more powerful concepts is crucial to deal with complexity. But this task, once again, requires significant effort. Cognitive Task Analysis (Chandrasekaran, 1986), (Chandrasekaran, 1990), (Pirolli & Card, 2005), indicates that such design involves (1) pattern matching to identify sets of similar classes or methods, (2) creativity to create generic and abstract classes and methods, (3) search on the space of possible design alternatives, (4) definition of criteria for evaluating the design alternatives, (5) synthesis and integration of classes and methods and (6) critique and modification of classes and methods. This is among the hardest tasks in complex software development and requires careful analysis and design.

As a result, in the case of complex software, Simplicity requires a significant amount of analysis and design. The more complex the system, the more analysis and design it requires. In some cases, it is arguable that the analysis and design effort required is comparable to, if not greater than, the coding effort.

5 FLATTENING THE EXPONENTIAL CURVE

Fowler (2001) proposes that the practices of continuous integration, testing, and refactoring provide a new environment that makes the evolutionary design of Agile methodologies become plausible. But he adds that design is important to flatten the curve. In Fowler´s words, "I'm sure that, despite the outside impression, XP isn't just test, code and refactor. There is room for designing before coding. Some of this is before there is any coding, most of it occurs in the iterations before coding for a particular task. But there is a new balance between up-front design and refactoring".

The criteria of Simplicity proposed by Beck (1999) are clearly necessary to flatten the change curve and avoid software entropy. In the previous section, we examined the effort associated with achieving Simplicity and showed that in the case of complex software development, a significant amount of analysis and design is necessary.

The short release cycles and the customer involvement of the Agile methodologies provide significant decrease in overall project effort and increase in customer satisfaction. However, it is important to add more analysis and design to the methodology. But the difficulty is in finding the right balance.

Figure 1 below shows the change curve for options (1) Agile without additional design, (2) Substantial upfront design, and (3) additional design during iterations. Option 1 corresponds to the more radical Agile methodologies with minimal design effort. With option 2, substantial design is performed up-front before coding. This is similar to traditional Plan-driven methodologies in which the software is more fully designed up-front. Option 3 corresponds to adding more design during each iteration (release cycle) of the Agile methodologies.

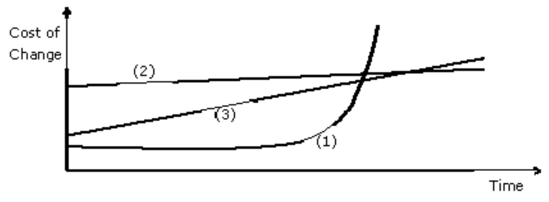

Figure 1: Examples of change curves of Agile methods based on the amount of design performed.

The curves showed in Figure 1 are qualitative and were estimated based on our experience. Curve (1) starts almost flat and then grows exponentially. Curves (2) and (3) are linear approximations assuming design was able to flatten the exponential curve. In reality, these are flattened exponentials and not lines. Option (1) has the advantage of short time to market which is crucial in today's web industry but shows exponential behavior in the long run. Option (2) is the most long term efficient option but has longer time to market. Option (3) appears as a good compromise between time to market and long term efficiency.

The methodology that has worked best in our experience is similar to option (3). Design has been added to each cycle of the Scrum Agile Methodology. Note that in our case we had reference implementations with framework architectures previously developed. For a development effort starting from scratch it may be necessary to develop a reference implementation. The reference implementation defines architecture, application server, framework, domain model and whatever is necessary to cope with complexity. Such reference implementation can be developed either in advance, when time permits, or in parallel with Option (1). In the latter case, an effort of integration is necessary to move the code implemented faster with the Option (1) model into the further architected reference implementation. But we strongly suggest developing the reference implementation using the principles of Agile. If a framework is to be developed then the necessary functionality should be developed first. The remaining functionality should only be developed if necessary. Substantial refactoring of the framework should also be considered, as suggested with Agile methodologies.

6 THE ROBUST AGILE METHODOLOGY

This section presents the Robust Agile Methodology, "R-Agile." For the past few years, we have successfully used this methodology to develop critical and fast-changing complex enterprise software. Applications of our software include: payment, revenue sharing and billing solutions for the telecom and e-commerce industries; and order management, taxation, logistics and financial solutions for the supply chain industry.

The proposed methodology is a modified version of the Scrum Agile Methodology (Beedle et al., 1999). The fundamental change is that we have included additional analysis and design to make the methodology more robust. We have also complemented Scrum by adding a dedicated Testing Team. In this section we will present more details of the methodology, mostly highlighting the points where our methodology differs from traditional Scrum. Notice that the parameters of the methodology, such as sprint duration and chronology, ratio of team members and even more structural elements of the methodology, can be adjusted based on the specifics of the project. For example, the criticality of the software, whether mission or revenue critical, importance of time to market, availability of resources and project budget are factors that affect the parameters of the methodology. The methodology described below is the one that worked best in our overall experience.

6.1 Teams and Roles

We have added dedicated Design and Testing Teams to the more traditional Product and Coding Teams of the Scrum Methodology. We present below the composition and the role of each team.

6.1.1 Product Team

The company can have various scrums running in parallel. We had the experience of running about 2 to 3 simultaneous scrums. Running Agile methods on very large projects can be challenging, requiring special coordination and scaling strategies. These experiences have been reported by Talby et al. (2006) and Chow & Cao (2008). In this paper we will concentrate on small to mid-size projects (< 20 developers) where the methodology works fine without special treatment. We will describe the team composition and roles for each Scrum.

Although the company can have multiple product managers, we used one product manager per Scrum. The product manager is called the "product owner" and is responsible for making sure that the product under development fulfills the business and company requirements. The product manager interfaces with the customers who can be external or internal to the organization.

We had Scrums that required more than one product manager. In such cases, the product managers collaborated within the same Scrum but one of the product managers was considered the product owner for the Scrum. The product manager also coordinates with the product managers of the other Scrums so that the overall product integrates well.

6.1.2 Design Team

The Design Team is dedicated to analysis and design. The goal is to keep the code clean and as simple as possible according to the definition of Simplicity discussed in previous sections. The team is responsible for the analysis and design required to flatten the change curve, which is critical for complex enterprise software development. In addition to designing for Simplicity, the Design Team is responsible for reviewing the code and identifying parts that are candidates for Refactoring.

Although analysis and design are being added, it is important to avoid overdesign. The design task itself should follow the principles of Agile, by designing at the necessary level. We suggest a generalization of the suggestions proposed by Fowler (2001). The Design Team should (1) invest time in learning about analysis and design principles and methods, (2) concentrate on when to apply such principles and methods (not too early), (3) concentrate on how to implement them in its simplest form first, adding complexity later when necessary, and (4) not hesitate in removing overdesigned parts.

We have used one software designer for a team of 4-6 developers. The exact number depends on many factors including the maturity of the project, the maturity of the team, the complexity of the project and other factors. We can cite a few reasons why we were able to design successfully using a relatively small ratio of designer/developer: (1) following the Agile principles and only designing what was necessary, (2) the Coding Team was also responsible for design, (3) often part of the Coding Team is working on less complex or repetitive parts of the code which do not require additional design by the Design Team, and (4) later phases of the project often require less design. Software architects are a key part of the Design Team. In addition, the team can include a few specialists, such as user interface designers, a modeling analyst responsible for data and process modeling, and a database optimization analyst.

We have also experimented with rotating the members of the Design Team. We usually strike a balance considering the following factors: (1) level of design and architecture experience, (2) project phase, and (3) the need to have members of the Coding Team also involved with design. The rotation was interesting in many ways. It caused the developers to feel more responsible and empowered. It also allowed the primary designers and architects to code, keeping them up-to-date with the programming technologies. Learning by design is a powerful learning tool (see research by Pappert & Harel (1990)). In general, the rotation promoted collaboration and avoided hierarchy and other human relational issues.

6.1.3 Coding Team

The Coding Team has the responsibility to deliver the product. The team is mostly responsible for coding, but is also responsible for design, testing (in particular unit testing), technical communication and release.

Continuous integration is an important activity for Agile methods. Scripts are developed to automate code check-out, compilation, release generation and to run test scripts. We were not always able to implement continuous integration, but the activity was very rewarding when implemented. The continuous integration was usually executed on a daily basis. When resources are available, a dedicated Release Team should be responsible for the continuous integration. In our case it was implemented collaboratively by the Coding and Test Teams.

One important member of the Coding Team is the Scrum Master whose primary role is to make sure that the developers are performing according to the plan and do not get stuck. The Scrum Master also ensures that the Scrum process is used as intended. The Coding Team is usually composed of 4-8 developers, plus the Scrum Master. When time permits, the Scrum Master can also code.

6.1.4 Testing Team

The Testing Team is a team dedicated to testing, the same as traditional quality assurance teams. In the case of mission critical systems, it has been challenging to apply the “do only the minimal” principle of the Agile Methodology to testing. We were able to separate parts of the system that were not critical, but the core required rigorous quality assurance. We were also able to reduce the amount of automated testing. Our Testing Team was composed of 2 engineers for a team of 4-8 developers. Once again, this parameter will depend on the specifics of the project at hand.

6.2 Activities and Schedule

One of the most important aspects of the Agile methodologies is the much shorter release cycles than traditional Plan-driven methodologies. In Scrum, such release cycles are called Sprint. We have had experience with both packaged releases and incremental improvements. In the first case, we had contractual obligations to release a particular set of functionalities to external customers. The latter case allowed for the release as improvements were developed and fits more naturally with the Scrum methodology. But even in the first case, we used the Scrum methodology, thus generating shorter internal releases and providing packaged releases at longer cycles. What was important in this case was a close relationship and agreement with the customer to define the appropriate milestones for the packaged releases, as the Scrum methodology avoids defining long term plans in advance.

We have mostly worked with 2-week Sprints, but have experimented also with 3-week Sprints. The 2-week Sprint is more dynamic, allowing faster time to market and market feedback. However, it can feel tight, in particular before project stabilization or when dealing with complex mission-critical functions which are harder to implement and require full testing. We experienced most difficulties and pressure during the last two days of the 2-week Sprint when we performed integration testing and associated bug fixing and regression testing. When we were able to do it, continuous integration definitely helped to ease the process. It is also important to check with the team in order to decide on 2 or 3-week Sprints. The personality and style of the team can affect the decision.

Table 1 shows the activities and schedule of the R-Agile Robust Agile Methodology. It presents the activities associated with each of the Product, Design, Coding and Testing Teams and can be used for either 2 or 3 week Sprints. The second week of a 3 week Sprint is simply a continuation of the activities started in the first week.

A key aspect of the proposed R-Agile Scrum Methodology is related to the temporal relationship and sequencing of the activities. The main activities of the Product and Design Teams, which are Specification and Design respectively, happens one

Table 1: R-Agile Scrum Methodology with 2-3 Week Sprint Release Cycle.

Sprint First Week

Team	Monday	Tuesday	Wednesday	Thursday	Friday
Product	Specification Sprint n+1	→	→	→	→
Design	Design Review Sprint n	Design Sprint n+1	→	→	→
Coding	Design Review Sprint n	Code Sprint n	→	→	→
Testing	Code Test Scripts Sprint n	→	→	Test Sprint n	→

Sprint Last Week

Team	Monday	Tuesday	Wednesday	Thursday	Friday
Product	→	→	→	Review and Sign-off Sprint n	Planning Sprint n+1
Design	→	→	→	Review Sprint n	Planning Sprint n+1
Coding	→	→	Integration Sprint n	Review and Push Production Sprint n	Planning Sprint n+1
Testing	→	→	Final Test Sprint n	Review Sprint n	Planning Sprint n+1

Sprint before Coding. During a 2 (or 3) week period, while the Coding Team is programming the features of Sprint n, the Product and Design Team are respectively specifying and designing for the following Sprint n+1. This gives the Product and Design Teams enough time to prepare their activities before the beginning of Coding.

The Testing Team, on the other hand, performs testing of the features of Sprint n during the same Sprint n. The Testing Team starts developing the test scripts early in the Sprint. Ideally, the test scripts get developed before the corresponding code. This way, the code can be tested as it gets implemented. Code and test during the same Sprint can feel stressful, but it allows the developers to fix bugs while the code is still fresh in their minds. We have experimented with testing executed one Sprint after the Coding Sprint. It certainly provided more ease for the Testing Team, but had the inconvenience of having developers fixing bugs of the previous Sprint n-1 during Sprint n. In addition, we had to manage two releases of the same features: the testing release at the end of Sprint n and the production release at the end of Sprint n+1.

In the remainder of this section we summarize the main activities involved in the Scrum.

6.2.1 Specification

As described above, the Specification activity is performed one Sprint ahead of its corresponding Coding Sprint. Before starting the specification activity, the product owner is responsible for defining and prioritizing the desired product features and adding them to the Product Backlog. Each feature is described by a story and each story is specified with enough details so that the Design and Coding Team can understand it.

6.2.2 Design

Design is also performed in the previous Sprint n-1. The Design Team chooses the stories which require design from the Product Backlog according to their priority. The Design Team can also spend time reviewing the code and identifying areas for redesign and refactoring.

6.2.3 Planning

The Planning activity sets the beginning of the “official” Sprint. Planning is performed on a Friday. One reason for this is so that Sprints can finish with a Production Release on a Thursday. This allows for any emergency repair of a release to be performed on a Friday and not during the weekend.

An entire day is devoted to Planning and is split into two meetings. During the first meeting the engineers learn about the stories and choose which ones to implement, taking into consideration the priority set by the product owner. The chosen stories are placed in the Sprint Backlog. During the second meeting, the engineers break down the stories into tasks that must be coded. The tasks are split among the developers, making sure that they can be completed during the Sprint.

6.2.4 Design Review

The Design Review activity is executed on a Monday and can even be extended if necessary. During this meeting, the Design Team discusses the design (which was performed during Sprint n-1) with the Coding Team. Only the developers involved with the designed parts are mandated to participate in the Design Review meeting. But in some cases we involved other developers. This was important so that more developers could learn about the design. Developers often needed to understand the design for a future Sprint and it saved everyone’s time to have the design review done once. There was also valuable feedback from the developers who, in addition, felt more involved and empowered.

6.2.5 Coding

The Coding Team has about 7 full days (in the case of a 2 week Sprint) to code, check in, unit test, integrate code and fix bugs related to the tasks committed during the Sprint Planning activity. Since the release cycles are short, we decided that only mission critical and revenue-affecting bugs and feature requests could interrupt a Sprint. As discussed, the Coding Team can also perform design of their respective tasks, of course in coordination with the Design Team and following the minimalist principle of Agile.

6.2.6 Testing

The Testing Team participates on the Sprint Planning activity. During the first meeting of the Planning activity, the Testing Team learns about the features to be implemented during the Sprint. This allows the Testing Team to produce test scripts for such features. During the second meeting of the Planning activity, the Testing Team defines and schedules the testing tasks to be executed during the Sprint. These testing tasks are planned and executed in the same way as the coding tasks of the Scrum.

The Testing Team can also participate on the Monday Design review. This is important especially if the Testing Team is to perform white-box testing, which requires testing algorithm paths and their efficiency.

The testing tasks are composed of tasks for coding or for running test scripts. The Testing Team starts coding the test scripts on Monday. Since test scripts are usually coded faster than the actual code to be tested, the code can be tested as soon as it gets implemented. But we tended to spend a few days to create a batch of test scripts before beginning code testing around Thursday. This allowed the Testing Team to focus on either coding or testing and avoided too much back and forth between the two activities. This can be planned based on what the team feels comfortable with and also based on the specific Sprint.

6.2.7 Review

On Thursday of the last Sprint week, an informal Sprint Review is performed to make sure that the implementation meets the requirements. The code is pushed to production if signed-off by the product owner. A Sprint Retrospective can also be organized at times to evaluate the overall methodology and propose improvements.

7 CONCLUSIONS

We performed Cognitive Task Analysis to show that substantial analysis and design is necessary to create powerful code and avoid software entropy and exponential change curve. We have also discussed the more extensive testing necessary for critical functionality and the need for adaptability to handle fast change requirements. We have presented R-Agile which is a robust and agile methodology to handle complexity, criticality and adaptability. We have successfully employed R-Agile to develop financial, taxation and supply chain management enterprise software.

REFERENCES

Beck, K., 1999. *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional;1999.

Beck, K.; et al., 2001. Manifesto for Agile Software Development. Agile Alliance. In website: http:// agilemanifesto.org.

Beedle, M., Devos, M., Sharon, Y., Schwaber, K., Sutherland, J., 1999. Scrum: An Extension Pattern Language for Hyperproductive Software Development. In Pattern Languages of Program Design 4, N. Harrison, B. Foote, and H. Rohnert, Eds. Addison-Wesley.

Boehm, B., Turner, R., 2003. Using Risk to Balance Agile and Plan-driven Methods; In IEEE Computer, Vol. 36 Issue: 6; pp. 57 - 66.

Chandrasekaran, B., 1986. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design; In IEEE Expert 1(3): pp. 23–30.

Chandrasekaran, B., 1990. Design Problem Solving: A Task Analysis; In AI Magazine, Vol. 11, Number 4, pp. 59-71.

Chow, T.; Cao, D., 2008. A Survey Study of Critical Success Factors in Agile Software Projects, Journal of Systems and Software, Volume 81, Issue 6, pp. 961-971.

Fowler, M., 2001. Is Design Dead?, In *Extreme Programming Examined*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, pp. 3–17.

Larman, C., Basili, V. R., 2003. Iterative and Incremental Development: A Brief History; In IEEE Computer, Cover Feature, June 2003, pp. 47-56.

McConnell, S., 1996. Rapid Development: Taming Wild Software Schedules. Microsoft Press.

Papert, S., Harel, I., 1990. Software Design as a Learning Environment; Interactive Learning Environments, v1 n1 pp. 1-32.

Pirolli, P., Card, S., 2005. The Sensemaking Process and Leverage Points for Analyst Technology as Identified Through Cognitive Task Analysis; In Proceedings of International Conference on Intelligence Analysis.

Royce, W.,1970. Managing the Development of Large Software Systems; In *Proceedings of IEEE WESCON*, Number **26** (August), pp. 1–9.

Talby, D., Hazzan, A., Dubinsky, Y., Keren, A., 2006. Agile Software Testing in a Large-Scale Project; In IEEE Software, vol. 23, no. 4, pp. 30-37.