Applying Scrum and Organizational Patterns to Multi-site Software Development[1]

Lucas Cordeiro[2], Cassiano Becker[3], Raimundo Barreto[2]

[2]Departamento de Ciência da Computação - Universidade Federal do Amazonas (UFAM), Brazil

[3]BenQ Eletroeletrônica S.A, Manaus, Brazil

lcc@dcc.ufam.edu.br, cassiano.becker@benq.com, rbarreto@dcc.ufam.edu.br

Abstract. *This paper describes a pattern language for managing multi-site software projects which aims at minimizing the main problems present on the multi-site software development context. The practices and patterns of the proposed language were first identified from the literature and adapted according to the authors' experience after running some multi-site software projects. This exercise has led to the identification of two new patterns: "Stories Rework Subsystem", and "Plan Bugs On a Sustainable Pace", as well as to an alternative application of the existing "Inversion of Control" pattern to the organizational context.*

Keywords: *Multi-site Software Development, Scrum Agile Methodology, Lean Software Development, Organizational Patterns, Project Management.*

1. Introduction

Large software projects are usually split into components and developed by different teams, in some cases developed at different places. Software development projects, both large and small, have been consistently difficult to control and manage. Recent studies show that an average project take twice as long to do as its initial plans [Schwaber and Beedle 2002]. Communication overhead and effort to create and update documentation could be pointed as major sources of inefficiency behind project failures. Communication overhead is often introduced by a mismatch in the functionalities required by a given component and the way their development is assigned to separate development teams. In this case, a high rate of communication among teams is introduced, as components developed by one team depend on the services provided by components developed by teams located at different places.

Another problem in large software projects is the increased need for communicating requirements with a higher degree of formality. Requirements are essentially written to describe product characteristics that are proposed in response to a set of business needs. However, customers/users are often not completely sure of what they want, and their mind is likely to change during the time the product is being developed. Moreover, external forces such as competitor's products/services may also lead to changes or enhancements in requirements. Still, many details of what must be produced may be found out only during product development. Therefore, the fact that several development teams may be involved in a project with evolving user requirements calls for practices to efficiently manage the project *(team size and location)* and embrace changes *(scope flexibility),* even late in the development process.

Based on this context, we describe in this paper a pattern language composed of Scrum [Schwaber and Beedle 2002], Lean Software Development [Poppendieck and Poppendieck 2003] and Organizational patterns [Coplien and Harrison 2004] applied to the domain of multi-site software development. In our definition, multi-site software development can be described essentially by characteristics as follows: (i) the project is split into components and assigned to different development teams, (ii) teams are physically separated and may be part of different business organizations, (iii) there is a limited number of teams, such that a two-level hierarchy of coordination is sufficient (between two and five in our experience) (iv) teams are able to physically meet at non-prohibitive cost, if required.

The remainder of this paper is organized as follows: Section 2 provides an overview of the Scrum agile methodology and Organizational patterns. Section 3 introduces the structure of a pattern language in which the proposed patterns are included, and shows how these patterns relate to each other. Section 4 describes the proposed patterns and finally, section 5 summarizes this paper and provides goals of further research.

2. A Brief Look at the Agile Method and Patterns

This section looks briefly at the Scrum method and at the Organizational patterns that were used as basis for the pattern language four our multi-site software environment.

2.1. Scrum

Scrum is a simple and straightforward approach to manage the software development process based on the assumption that environmental (i.e. people) and technical (i.e. technologies) variables are likely to change during the process [Schwaber and Beedle 2002]. In order to manage these variables, Scrum employs the empirical process control model which strongly uses a feedback mechanism to monitor and adapt to the unexpected. Scrum is composed of 14 practices and some of its main practices include: **Sprint** practice which is the iteration work organized in 30-calendar-day. The **Sprint Planning** practice that consists of two meetings as follows: In the first meeting, the product backlog which contains a list of features, use cases, enhancements, and defects of the system is refined and re-prioritized by the product owner, stakeholders and goals for the next iteration are chosen. In the second meeting, the Scrum team figures out how to achieve the requests and creates the sprint backlog that contains detailed tasks to be

accomplished in the current iteration. In the **Sprint Review** practice, the Scrum team presents the results obtained at the end of each iteration by showing working software to the product owner, customers and other stakeholders. In the **Daily Scrum** practice, daily meetings are held at the same place and time with special questions to be answered by the Scrum team.

The Scrum process consists of three roles and the responsibility of each role is described as follows: **Scrum master** is the person responsible for ensuring that Scrum values, practices and rules are followed by the Scrum team. He/she is also responsible for mediating between management and Scrum team, as well as listening to progress and removes block points. **Product owner** is the person who is officially responsible for the project. This person creates and prioritizes the product backlog and ensures that it is visible to everyone. He/she is also responsible for choosing the goals for the next sprint and reviewing the system with other stakeholders at the end of every iteration.

Scrum team is responsible for working on the sprint backlog. The amount of work that will be addressed in the sprint is solely up to the team. They must assess what can be accomplished in the sprint during the sprint planning meeting. Therefore, the team has the authority to make most decisions, and ask for any block points to be removed.

2.2. Organizational Patterns

The organizational patterns described by [Coplien and Harrison 2004] can be combined with Scrum agile methods with the purpose of structuring the software development process of organizations. These patterns are split into four different pattern languages as follows: The **project management pattern language** provides a set of patterns that help the organization manage product development, clarify the product requirements, coordinate project's activities, generate system builds, and keep the team focus on the project's primary goals.

The **piecemeal growth pattern language** provides a set of patterns that help the organization define the overall management structure and amount of team members per project, ensure and maintain customer satisfaction, communicate system requirements, and ensure a common vision for all the people involved in the product development team. The **organizational style pattern language** provides a set of patterns that help the organization eliminate project's overhead and latency, ensure that the organization structure is compatible with the product architecture, organize work for developing products with geographically distributed teams, ensure that market needs will be met.

The **people and code pattern language** provides a set of patterns that help the organization define and keep the architecture style of the product, ensure that the architect is materially involved in implementation, and assign feature development to people in nontrivial projects. The **software configuration management pattern language** is not part of the organizational patterns, but was integrated into the proposed pattern language. These patterns were defined by [Berczuk 2002] and they offer patterns that help the development team define mechanisms for managing different versions of the work products, develop code in parallel, and identify what versions of code make up a particular component.

3. The Proposed Pattern Language

As previously said, the proposed pattern language is composed by patterns identified from languages with complementary concerns: the Scrum Methodology, Organizational Patterns, and Software Configuration Management pattern language. Besides these, the authors also identified from their experience the adoption of practices that pointed to two additional patterns. The resulting pattern language diagram is depicted in Figure 1.

From the resulting set of twenty-one patterns, only a subset was elected for a full description. These obeyed the following criteria:

- A pattern of fundamental importance to description of development process from the multi-site and agile aspect ("Surrogate Customer", "Code Line", and "Integration Build").
- A pattern that had not yet been applied to this context before (c.f. "Inversion of Control").
- A proposed new pattern identified by the authors ("Stories Rework Subsystem" and "Plan Bugs on a Sustainable Pace").

Although a greater number of patterns from the mentioned sources could be indeed mapped to the practices in our cases, we restricted the language to the ones which were more illustrative of the agile and multi-site aspects. It should be noted that these patterns are not intended to be exclusive to the multi-site development context, and will occur in many software development efforts.

The six patterns that will be described are depicted in gray in the pattern language diagram (see Figure 1). In the figure, the relationship PatternA→PatternB can be read as "PatternA can exist once PatternB is in place", that is, PatternA will find a proper context for its application once PatternB has been applied. As an example, the "Sprint Planning" pattern, (when the team sits to plan how to fulfill the goals selected for the next iteration), can be applied and really makes more sense once "Scenarios Define Problem" is in place (when the problem or product being targeted has been decomposed in prioritized stories to be worked). In other words, the resulting context once PatternA is applied can be understood as the initial context for PatternB as the arrows are followed. In addition, the connections simply suggest the probability of patterns occurring together.

Traversing the pattern language diagram vertically also provides a hint on the patterns positioning in the flow of development activities. On the top position, the first pattern is the "Work Queue", which describes the initial set of problems and requirements intended to be addressed by the iterative and incremental development effort. Following the arrows downward will present patterns moving into the solution domain, such as structures for the temporal organization in sprints, multi-site team distribution and the adoption of selected configuration management practices. The traversal concludes at the bottom with the "Integration Build" pattern, which will eventually materialize the results of all processes, practices and tools from each different development cycle into a concrete and valid functionality increment. The patterns are described in the next sections, following the sequence that they appear in the diagram.

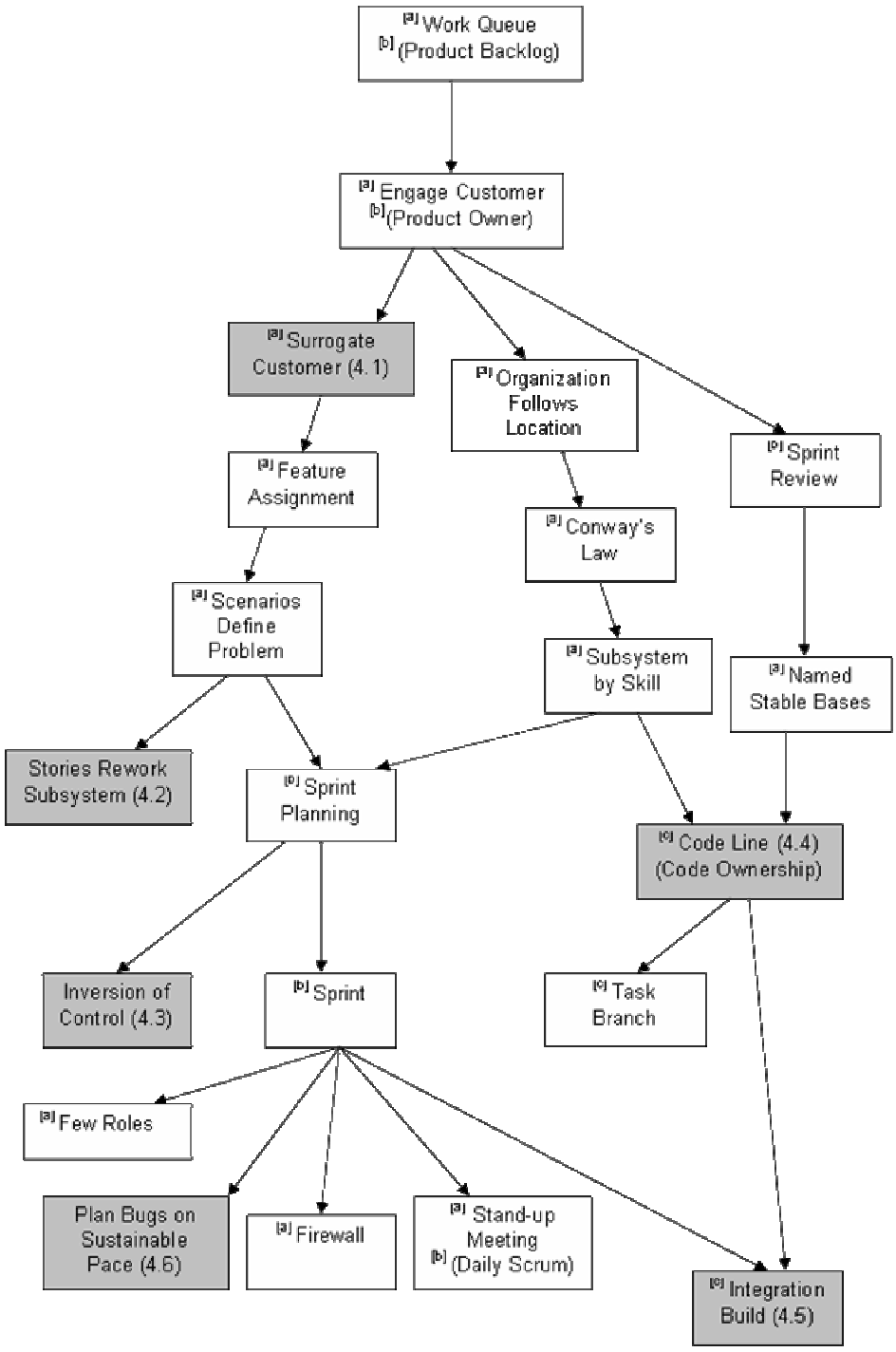

Figure 1. Proposed Pattern Language Structure. Patterns marked with [a] belong to Organization Patterns, [b] to Scrum and [c] to Software Configuration Management Patterns.

4. Patterns for Multi-site Software Development

This section is concerned with describing the patterns presented in section 3 in the following way: the context in which the pattern is applied, the problem that the pattern will solve, the forces that limit the pattern application, the solution of the problem, the related patterns, known uses and finally the resulting context that shows what happens if the solution is applied. The stars after the pattern name indicate the confidence level for the pattern in the multi-site environment. Moreover, we also indicate the pattern origin as follows: "O.P." (Organizational Patterns), "C.M." (Configuration Management Patterns), and "Authors" (the patterns proposed by the authors).

4.1. [**] Surrogate Customer [O.P.]

Alias: Surrogate Product Owner, Feature Leader

Context:

In a project adopting the Scrum methodology, the Product Owner is a central figure. He is the ultimate reference for product content, and his inputs are a major influence on the work performed at each sprint. For larger projects, however, when developed in a multi-site configuration, **a single central Product Owner is not likely to be able to respond to all the demand generated by the distributed development teams** to a satisfactory level of detail.

Problem:

Agile projects rely on close interaction with the customer. Feedback is required at least at each Sprint review and Release planning, but is encouraged to occur throughout the sprint course. With the communication boundaries introduced in multi-site projects, how to maximize information flow and feedback from customers to developers?

Forces:

- The development teams cannot take advantage of constant multi-mode communication channels due to their physical separation.
- Practical solutions usually involve round-trips from requirements to implementation in order to meet time and knowledge constraints.
- Domain knowledge cannot be expected to be fully available in the development team.
- Depending on the project nature (a new solution), a customer might not even exist yet.
- The product owner or customer might not have the necessary available time or detailed knowledge to interact with the development team.

Solution:

Software system functionalities should be split and grouped into features. A "Feature Leader" role is then defined, and will represent the product owner to all teams involved in the implementation of his/her feature set. The set of Feature Leaders can take advantage of closer interaction with the "master" product owner and at the same time will support the remote development teams in specification and decision making in the

sprint planning and throughout its development. The Feature Leader role will influence the development by:

- Defining stories and use case models: Stories and their prioritization are the customer's main contribution to the project in an agile environment. In a multi-site organization, the feature leader will provide more specialized support, in the subsystem or feature level than the product owner.
- Splitting stories: some stories, after their initial estimation, are found to exceed the capacity left for the iteration at hand. The Feature Leader will be able to help establish case by case criteria for decomposing the story (see description in Scenarios Refactor Subsystem).
- Establishing and deciding against trade-offs: when considering different design and implementation for fulfilling a given story, a set of solutions will present different balances on product quality. Although trade-offs might have been laid out clearly at the product-wide level, there might be specific local decisions to consider separately.
- Help establish a domain language: which represents the problem, concepts and solution at hand and which is understandable for both the developer and customer, enabling true two-way communication.
- Providing story acceptance criteria: Defining tests based on real examples for happy-path flows. Additionally, running and looking at partial software releases will usually provide valuable feedback.

Figure 2 describes how a solution for multi-site Scrum teams was proposed in projects the authors participated. In such a set-up, selected team members in the central Product Team were all co-located, and while engaging in ordinary team member activities at that level, acted as Product Owners for the separated subsystem teams.

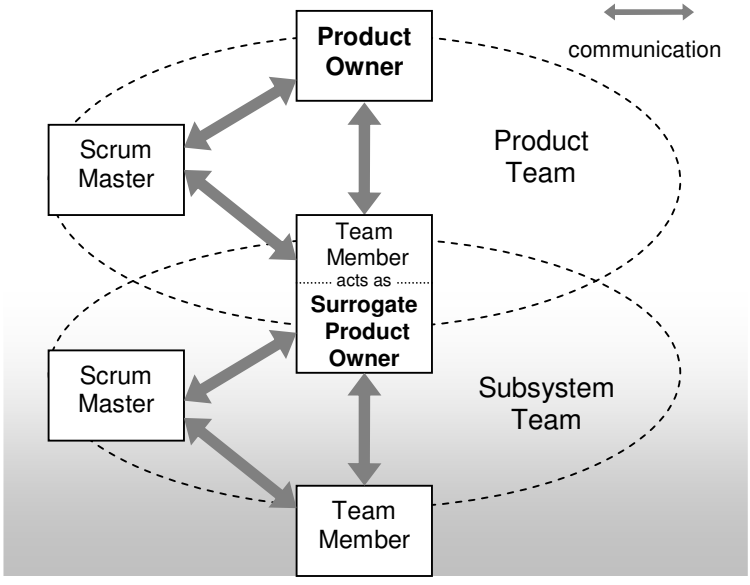

Figure 2. Surrogate Product Owner in a Multi-site Scrum setup

Related Patterns:

The "Product Owner" role, summarized before in this article, and the "Engage Customer" pattern are more general patterns which first described the need for closer interaction and feedback from the customer throughout the entire duration of the development cycle in agile environments.

Resulting Context:

Increased Feature Leader participation raises product perceived integrity, as the stories implemented benefit from a synthesis of the interaction and feedback between the feature leader and the developer.

Developers gain the possibility to discuss and clarify actual design and implementation alternatives in light of product-wide trade-offs. The creation of a common domain language representation is facilitated and is likely to emerge more naturally as a result of the discussions between the Feature Leader and the developer.

However, care must be taken not to over-interact with the development team and cause undesired congestion effects. These would result from an overflow or new requests or changes due to reconsideration, if within a given sprint. In that case, the "Firewall" pattern should be considered.

4.2. [**] Stories Rework Subsystems [Authors]

Context:

In multi-site project, different teams at separated locations will usually define and be assigned different subsystems (see patterns "Conway's Law" and "Organization Follows Location"). For a new story to be fulfilled, usually changes and additional functionalities must be implemented in more than one subsystem (see pattern:" Subsystem by Skill").

Furthermore, when an agile process is applied, stories or feature increments must be integrated and tested in the period **of one time limited iteration.** In the above configuration, *a tension* will generally appear **between the goals posed by a system-wide increment and the goals that each subsystem team** is likely to identify as most important when looking only to their restricted scope.

Problem:

How to coordinate goals and tasks as viewed from the subsystem team standpoint so that the system evolves as a whole and is integrated to fulfill product-wide stories within a given iteration?

Forces:

- An integrated version of working software is expected to be available at the end of each time-limited iteration. Within the course of the iteration, the teams have to make a decision on where to invest their effort at each moment, if on the evolution of the system, on or its stabilization for the integration.
- In a structure defined with "Subsystem per Skills", a separated team will tend to optimize the responsibilities assigned to their components. This will often conflict

with the goals of the whole system for that iteration, which depends on the integration of the functionalities of each subsystem for a given story.

- The problem of suboptimization [Principia] is present: "When you try to optimize the global outcome for a system consisting of distinct subsystems (…), you might try to do this by optimizing the result for each of the subsystems separately. This is called "suboptimization". The principle of suboptimization states that suboptimization in general does not lead to global optimization."
- The more separated or independent the teams working in the system for a given iteration are, more pronounced these forces will be.

Solution:

Introduce the notion to both subsystem and central teams that a **level of rework** should be expected on their subsystems because of the division of the project in sprints. A (perhaps too) simple analogy to this principle is the practice of fencing around a new construction building. The fence will be torn down before the building gets inaugurated, but it is the fencing that allows the construction work to proceed in a controlled way, better integrating the construction to the surrounding environment while work proceeds. Therefore, rework in this case should be understood as activities or code that is produced during the sprint, but which will not be present in the final releases of the product.

From the standpoint of subsystem teams, these activities will usually come in the form of local deviations from what the responsibilities of that subsystem would ideally imply if that subsystem would be the only one being developed. In practice, these local concessions are ultimately caused by the need to converge to integrated stories at the end of each iteration. Examples of activities that could be understood as dimensions of rework are next described:

- Splitting Stories: depending on the story estimates and on the load of each subsystem team in the iteration, a given story can be split to still fit the current iteration. It could be that the amount of work necessary for the split stories is greater than the work for the original [Cohn 2005] provides valuable advice for establishing splitting criteria.
- Splitting Across Data Boundaries: for example, selecting a subset of fields supported for a given form.
- Splitting On Operational Boundaries: for example, selecting a smaller number of operations (CRUD – create, update, delete) or more simple conditions.
- Postponing Cross-Cutting Concerns: for example, leaving out logging, error handling, or security treatment for the iteration being planned.
- Not meeting performance requirements: postponing non-functional requirement aspects.

Because each of these items will probably have to be revisited when the remaining scope is reconsidered, and because there is at least a small volume of code adaptation exclusive to the splitting, these practices might be interpreted as a source of rework. On the other hand, for many larger stories, splitting will be indeed the most efficient way to keep complexity and risk under control.

Coding stubs and mock objects in order to compensate for the absence of subsystem functionality might also be interpreted as unnecessary work for the goals of a given subsystem. Mock objects or stub interface implementations might be interpreted as "inventory" effort, as they will not eventually make it as functionality for that given subsystem. However, when seen from the whole, having such mock objects timely available to other subsystems might be essential for allowing the rest of the system to grow optimally.

Therefore, in order to **enable incremental integration to happen in a multi-site project environment**, the notion that subsystems should expect a level of rework between iterations should be introduced. Project management instruments and measurement tools should be adapted to accommodate for those aspects, for example, acknowledging each local concession causing local under-optimization to the affected team, and focusing measurement on overall progress and performance, rather than local. [Poppendieck 2003] provides good analysis and recommendation on contractual issues that arise in an agile environment.

Related Patterns:

- The "Work Split", "Named Stable Bases" and "Incremental Integration" patterns and the "Thin Slice Story Writing" approach, all describe situations and techniques applicable for incremental and iterative methods that focus on optimizing development output in an environment with complexity and uncertainty
- "Architect Controls Product" has been proposed as a promoter of consensus and conceptual integrity. It acts as a central role that looks at how the subsystems and teams involved in the current iteration can integrate for best fulfilling the goals selected. This integrating role takes the lead for facilitating each subsystem team to see, within their own subsystem, what compromises they can identify so that the stories as a whole are optimized, even if this means subsystem increments depart from ideal. "Surrogate Product Owner" might also fulfill this need, if discussions focus on the splitting of stories between iterations.
- The "Subsystem by Skill" pattern describes a common organizational pattern where "Stories Rework Subsystem" is likely to appear.

Resulting Context:

In a multi-site configuration, having this notion included in the planning and design of solutions at each iteration is a condition for achieving patterns "Named stable bases" and "Incremental integration". Blind denial or avoidance the notion of rework might lead to poor strategies for identifying goals that are manageable within an iteration, and can the prevent system from growing efficiently while maintaining close integration points.

The rework resulting from the compromises taken in each subsystem in a given iteration will have to be considered and re-estimated on the following iterations, reinforcing the need for adaptive planning. Within the limits of a single subsystem and a given iteration, such activities are not generally considered as rework, and are instead understood as regular refactoring.

Also important to take into account, the implications for the measures of performance and quality should be focused first on the feature as a whole, and only secondarily on the performance of each subsystem. Otherwise, subsystem teams will perceive a stronger incentive to optimize their characteristics, which will lead to sub-optimization.

Typical roles that should benefit from the awareness of this pattern are the ones involved in the planning of features at the beginning of each sprint (mostly Scrum Master, Architect and representatives of each distributed team in the planning session). By acknowledging that some level of sub-optimization (in this context that means rework between iterations) is natural and might even be required for the optimization of the system as a whole, conflicting situations might have their causes recognized and discussed more productively.

The more predictable the project is (especially in technology and requirements), the less intermediate integration points will it need, and more work will be able to be performed by teams in parallel, leading to ideally minimum rework. However, for less predicable projects, where a more iterative and adaptive approach is more appropriate, allowing and accounting for rework activities as described in this pattern is likely to lead to increased overall efficiency and lowered risk.

Known uses:

- The lean principle "See the Whole" from [Poppendieck 2003] emphasizes the importance of carefully choosing system-wide variables to measure and optimize, while stating that this will often be accompanied by a relaxation on performance at the local (subsystem) level.
- [Lehman 2000] in his multi-year studies on software evolution proposes eight laws for software evolution planning and management. His "Second Law: Growing Complexity" states that "As an E-type system is evolved, its complexity increases unless work is done to maintain or reduce it" and introduces the notions of Progressive and Anti-regressive work. The rational behind the need for anti-regressive work is closely related to the context and solution here presented.
- The practices of refactoring, as well as the use of stubs and mock objects, are well established in agile software development. They share the notion of work that is revisited or discarded as iterations evolve.

4.3. [*] Inversion of Control [Authors]

Aliases: Don't Call Us We Call You

Context:

In a multi-site organization, communicating and assuring understanding of desired product characteristics to development teams is further complicated by the added communication boundaries. The Product Owner is the ultimate responsible for deciding and prioritizing the stories which make up the solution to the problem. However, depending on the size of the project, a number of details that will eventually affect the perceived integrity of the product are likely to pop up during development, and cannot be expected to be foreseen or discussed with a central product owner timely enough.

If “Surrogate Customer” is applied, as described in this article, the overall team structure is scaled-up and a communication channel for product characteristics can be established between the central product team (see Figure 2) and the subsystem teams.

If a degree of detailed specifications **are expected for each selected feature during each sprint**, this can easily become a bottleneck in the timeframe of a given iteration. The separation of teams occurring in a multi-site environment makes this problem even more important.

Problem:

How to communicate desired product or feature functionality to distributed teams in an agile context, where the selection of stories to be worked is decided at each iteration?

Forces:

- Users and customers are not able to completely state exactly what they want.
- Even if the software developers know all the requirements, many of the details they need to develop the software become clear only as they develop the system.
- Even if all the details could be known up front, it is difficult for a developer to absorb in productive way that many details.
- Even if we could understand all the details, product and project changes occur.

While the software development literature has produced extensive recommendations on the characteristics of well written requirements (concrete, testable, realizable), achieving this in practice is usually easier said than done. Customer state that describing requirements takes too much of their time, and developers often find that they lack in detail or are ambiguous.

Solution:

The pattern “Inversion of Control” has been proposed by [Fowler 2004] as an object oriented design pattern for web application frameworks, in order to eliminate unwanted dependencies in the wiring between framework and application components. In our multi-site and organizational context, the “Inversion of Control” analogy is suggested to describe the way requirements activities can be alternatively handled between the product definition team (Product Owners and it surrogates) and the distributed subsystem development teams.

The solution consists of having the implementing team responsible to continuously refine and revise requirements and solution specification *in the format and level of detail of their preference* (story writing, acceptance tests, schema matrices, verbal and prose descriptions, diagrams). Documentation should only be produced to the level of detail and formality which helps in the communication of the problem and its proposed solution. More recently, developers and analysts have found a reason to move further into each other’s territory in order to cause their language to overlap on top of common domain knowledge representation.

Also, another contribution from agile methods is to promote acceptance tests as the preferred format for requirements. Acceptance test are usually easier to write than requirements because they are based on concrete cases and are written by example,

which also helps eliminate ambiguity. If tests are written in such a way that they allow for automatic execution, they will also provide for instant feedback and progress measurement.

In the "Inversion of Control" pattern, a typical flow of information between the customer and the development team could be described as follows:

1) The Product Owner and its surrogates are initially involved in laying out the initial story description, establishing the prioritization of the quality dimensions, providing examples of happy path tests, and occasionally pointing to existing external standards where applicable.

2) Based on the initial conversation and a subset of the information above, the development team can analyze the problem and write an initial proposal for the solution. In the process of analyzing and proposing a solution, the development team will be in a better position to provide estimates and propose simplifying or splitting criteria in case the estimates values or uncertainty level is too high. If a UI interface prototype has not been given, a sketch can be proposed.

3) The first requirements-analysis-design-validation micro-cycle can be closed a few days after the start of each iteration, when both the developers and product owner surrogates meet to review and discuss with the help of the support material produced.

4) During the course of the sprint, details, alternative flows and corner cases will be identified. The development team is encouraged to constantly feedback its findings and doubts to be revised by the product owners. Each doubt or limitation raised during the sprint refinement can be either accepted as part of the solution space provided or can be fed back to the product backlog in order to be addressed in a further sprint.

Related Patterns:

- "Surrogate Customer", in this article, established the organizational roles on top of which this solution can be applied.
- "Community of Trust" is a pre-condition for the shift in the division of labor in the requirements elicitation and solution creation between product owners and developers to be effective.

Resulting Context:

When "Inversion of Control" is applied to multi-site requirements communication:

- The proposed solution will naturally include the judgment and limitations seen by the implementing team for that iteration (could be reworked on a further it).
- Documentation effort will be prioritized only to the efficient and necessary level of detail and formality which is relevant for the development in the iteration.
- The process of refining the requirements will allow for better estimates and will increase the engagement from the implementing team.
- Early analysis will cause the development team to raise and communicate their external dependencies to other subsystems.

<u>Risks and downsides:</u>

Over reliance on the inversion proposed in this pattern has its danger. The Product Owner role has the ultimate knowledge and responsibility over the problem domain. That is, at least the problem description, major constraints and trade-off dimensions have to be clearly set out by the customer team at the beginning of each iteration, otherwise the expected bootstrapping for the solution might be at risk. As potential risks to the application of this pattern, the following items could be pointed:

1) Having the implementing team to deal with documentation requires analysis capability, which cannot be taken for granted in all teams. In larger projects, however, we felt that a higher number of individuals was willing to step in explore these skills. This was sometimes even felt as a factor of motivation for those individuals inclined.

2) The idea of writing documentation is likely to cause discomfort in an agile environment, and to accommodate for that, the notion of flexibility in both the format and level of detail in the artifacts was introduced. Content produced focused on detailing practical limits, exceptional cases, points of variance and screen refinements; all points that developers felt was key to their technical decisions.

3) The boundary between eliciting requirements and solution providing has to be agreed between product owners (and it surrogates) with developers so that decision making is balanced to the level of detail each side has condition to provide. To the extent of our experience, this balance point varies with team composition, the degree of novelty (uncertainty) of the requirement being worked, and the level of trust between teams. Therefore, for this shifting in balance to be effective, it is necessary that "Community of Trust" [Organizational Patterns] be assured, which is a risk to be analyzed and mitigated in a multi-site (or multi-company) environment.

4.4. [***] Codeline [C.M.]

Context:

Large software systems are usually split into components or subsystems and developed by development teams that may be located at different places. Each development team is responsible for a couple of components or subsystems. They have their own software processes and tools to deal with software configuration management [Louzado and Cordeiro 2005]. Each development team has to implement system tasks (e.g., implement or enhance a requirement and fix a bug) and should not disrupt the activities of other development teams.

Problem:

Components or subsystems making up the system have dependencies, i.e., component B needs the services provided by component A. Changes in the interface or semantics of a component may affect other components of the system. As the components are developed by different development teams, how to keep them synchronized?

Forces:

- Development teams involved in the system development process have different software processes and tools to deal with software configuration management.
- The partition of the system functionalities into components is likely to cause dependencies among components.

- The allocation of these components among different development teams is likely to require a high rate of communication among development teams.
- The work of different development teams must be integrated at least once a week in order to provide feedback on the system functionalities to the customer/user.

Solution:

Components that have dependencies should be allocated to the same development team or at least be allocated to the development teams that are at the same place and/or time zone. Different codelines should be created, one for each development team in order to isolate changes and do not disrupt the work of other development teams. Another development line, called here mainline, should also be created to allow the development teams to integrate their components and generate new system builds. Interface or semantics changes in components must be communicated in advance through the weekly meetings. If describing information is required, then the development team should create an artifact that helps other development teams adapt to the change.

Related Patterns:

- The "Mainline" pattern [Berczuk and Appleton 2002] is applied when there are many people to develop a product and merging must be kept as low as possible. Therefore, it describes a mechanism to keep the number of active development line to a manageable set.
- The "Active Development Line" [Berczuk and Appleton 2002] pattern is applied to developers that want to integrate and test their changes very often during the development process. Therefore, it describes a mechanism to create an active development line by keeping a rapidly evolving development line stable enough to developers.

Known Uses:

- The mainline pattern used by [Louzado and Cordeiro 2005] in a multi-site software development project creates different codelines (one for each partner) and assigns a codeline policy. Moreover, there is a mainline that allows the build manager to integrate the components and generate new system builds.
- An agile codeline management proposed by [Berczuk 2003] creates codeline structures that isolate the components that need to be kept stable from those that are in active development. He also associates policies (how the codeline should be used) for each codeline that is created during the project lifetime.
- The codeline practice proposed by [Wingerd and Seiwald 1998] instantiates this pattern by assigning to each codeline an owner and a policy. They also create a mainline which provides an ultimate destination for changes (e.g., bug fixing, new features) and represents the linear evolution of the software product.

Resulting Context:

Components are grouped into subsystems. Each subsystem is allocated to a development team. Still, there may remain dependencies among subsystems as a higher layer requires services provided by lower layers. Therefore, after creating the codelines, each development team is able to work on its own development line without disrupting the

work of other development teams. The weekly meetings make it possible to synchronize the teams and improve communication. Weekly meetings enables planning which system functionalities, enhancements and bug fixing will be part of the next delivery. On a weekly basis, each development team delivers code to a build manager who is responsible for generating new versions of the system. Each team delivery comes with release notes that states what artifacts have been developed.

4.5. [***] Integration Build [C.M.]

Context:

The software is split into components and developed by teams, at different rates. Each development team is composed by several developers that are responsible for a set of systems requirements. Each developer works on its own *private workspace* and is isolated from the work of other developers [Louzado and Cordeiro 2005]. On the other hand, working software is expected to be delivered on a frequent basis to customers/users. Therefore, a means for integrating code frequently is needed with the purpose of reducing integration problems and providing early feedback to customers.

Problem:

There are several developers working on the production of the software. One developer may depend on the work of another developer. If both developers take long without integrating their code (components) into the product codeline, the number of integration problems might increase substantially. These occur because the system code evolves during the time between the task creation and completion. In this scenario, several tasks are integrated into the main trunk and the code in which the team members started working is different from the code currently available in the main trunk. How to coordinate the contribution from subsystem teams so that changes in one subsystem are integrated in a controlled way, while keeping development pace?

Forces:

- Software integration should occur very often in order to reduce integration problems and provide frequent feedback to customers/users.
- If developers integrate code and generate product builds very often then there is the possibility to spend more time integrating than developing code.
- The most important software functionalities must be implemented and integrated as earlier as possible during the development process in order to provide feedback to customers/users.
- Software development takes months to be accomplished and if it is integrated very often, stable versions of the system should be uniquely identified.

Solution:

Each development team should have a unique window to deliver and integrate the code into the product codeline. For a large system, both daily builds may take place on the codeline of each development team, as well as should one product build per week. For each weekly delivery carried out by the development teams, they should assign a tag in

their codeline and provide the release notes. In addition, they should solve the integration problems that may take place during the integration process.

When different teams share a product codeline, "Integration Build" provides most benefits when performed in a strict sequential mode. That is, only one subsystem team integrates its changes into the main codeline at a time, even if their components logically/physically separated from the remaining subsystems. Only after code increments introduced by one subsystem team are integrated into the product codeline should the next subsystem team by allowed to integrate its contribution. Integration in this sense is typically composed by: (i) check-out (update) of latest version from product codeline (ii) merging it with local changes in the workspace (iii) building and sanity-testing of merged version in the workspace (iv) check-in of integrated version on the product codeline. For the last activity, each development team can appoint an integrator to be responsible for integrating the team's code into the project's mainline (see Codeline pattern). Figure 3 describes a typical workflow with sequential integration.

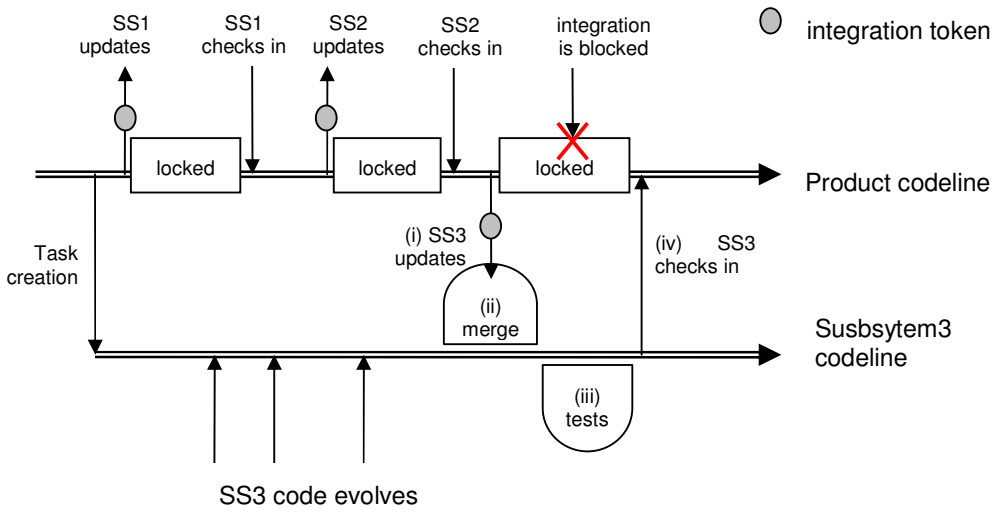

Figure 3. Sequential Integration

Moreover, specific dates/times can be assigned to each development team in order for the integration process to take place. Therefore, this sequential integration always allows a latest version of the system to be regularly identified. It is important to emphasize that the sequential integration does not imply that the development team cannot integrate the latest version of the code in its own codeline.

Related Patterns:

- The "Integration Build" pattern [Berczuk and Appleton 2002] is applied when it is necessary to make sure that components work together in an iterative and incremental approach. Therefore, it allows developers to frequently integrate their code by doing an integration build periodically.
- The "Named Stable Bases" pattern is needed when developers want to integrate software frequently with the purpose of keeping stability and progress. Therefore, it

describes a mechanism to give the stable system a name by which developers can work against.

- The "Build Prototypes" pattern is applied when requirements and design decision must be verified in order to reduce the risk of wasted cost and missed expectations. Therefore, it provides mechanisms to build prototype whose purpose is to help validate requirements and assess risks.

Known Uses:

- The integration build described by [Louzado and Cordeiro 2005] instantiates this pattern by adopting an "integration by stage" approach which provides a progressive integration of the product.
- The incremental integration proposed by [Berczuk 1996] provides a mechanism to allow developers to build the software periodically. This periodic build is also checked for interface compatibility and testing. Therefore, it encourages developers to build from the latest software release and provide time to fix incompatibilities.
- The continuous integration described by [Beck 1999] instantiate this pattern to allow developers to integrate and release code into the repository every few hours. One developer integrates at any time and it takes place only when all unit tests have passed or a smaller piece of the functionality is implemented.

Resulting Context:

If this sequential integration process is adopted in the project, i.e. if one development team has a specific date/time on the week to integrate the code that do not happen at the same date/time of another development team then integration problems may substantially be reduced. Another important benefit is that as the software is built on a weekly basis then it can provide great feedback to customer/users that need working software to clarify system requirements. The software that is produced on a weekly basis receives a unique identification that helps developers identify stable versions of the system. In addition, it allows customers/users to validate only stable versions of the system.

4.6. [***] Plan Bugs on a Sustainable Pace [Authors]

Context:

During the sprint planning, each team member decides which system's functionalities he/she will implement for the next sprint. The system's functionalities are decomposed into activities and are estimated by the team members. At the end of the sprint, the system's functionalities (product backlog items) that were committed to that sprint should be fulfilled by team members in order to be demonstrated to high-level management and customers. The builds generated during the sprint are tested during the same period in order to ensure the product's quality. Therefore, a number of bugs are likely to be found by the test team for the system's functionalities that were implemented in previous or in the current sprint.

Problem:

The test team is constantly testing and identifying bugs, which are added to an existing unsolved bugs list found in previous iterations. Depending on the bugs' criticality, the team members are expected to solve them as soon as possible in order to ensure the product quality. But as team members are committed to the activities of the current sprint, how will they manage to fix these bugs and at the same time ensure that the committed activities will be fulfilled at the end of the sprint?

Forces:

- The global software builds are generated and tested on a weekly basis. The bugs are created and assigned directly to the responsible person through a collaborative development environment tool (CDE).
- The team member responsible for the functionality in which the bug was found should not be interrupted so often because he/she has to complete the activities that were committed to the current sprint.
- The bug that was found at a given functionality might be so important to the customer that it acquires a higher priority than the other activities which are currently running. Therefore, this bug should be fixed as soon as possible by the responsible team member.
- The bug that was found at a given functionality might also impact other important functionalities or might affect the whole system. Therefore, this bug should acquire a higher priority than the other activities which are currently running.
- The development team implements new features in the current sprint and at the same time, it must keep the bug rate as low as possible.

Solution:

Introduce a bug planning process in order to control and manage the product's bugs and avoid project's interruptions. In this process, the test team provides the most critical bugs for each system's component. After that, each feature leader (see Surrogate Customer pattern) reviews the critical bugs, selects them based on the criticality, and informs the project leader. Then the project leader communicates the bugs to be fixed to the development teams. Each development team evaluates the list of bugs and informs to the project leader if the bugs will be fixed in the current sprint. This process is cyclical and its frequency can be higher than the sprint time, as effort for fixing a bug is typically lower than the effort for implementing a new feature. For sprints of one month, the recommended frequency is once a week. Also, as the software builds are generated and tested on a weekly basis (following "Integration Build"), it makes sense for the bug planning process to take place on a weekly basis (sustainable pace). When planning, the bugs, priorities, status, and deadlines should be defined by the project leader or by the person responsible for the feature in which that bug belongs.

The priority may be classified as **critical, high, medium, and low**. The priority level of the bug is according to the feature's importance and the amount of test cases that are blocked because of this bug. In addition, the status of the bug may be classified as **new, started, reopened, resolved, and closed**. After planning the bugs, the leader of each development team involved in the project should analyze if the bugs that are planned can be fulfilled given the workload of its team members. If the bugs can be

fixed without compromising the goals committed to the current sprint then the leader sends an e-mail informing that all the bugs are accepted. Otherwise, he/she commits **only the bugs that his/her team will be able to fix and deliver,** taking into account supporting information as priority, effort and risk. It is of utmost importance that planning and bug-fixing be kept to a sustainable pace during project's sprint. Frequent overtime is usually considered a symptom of serious problems in a team. Therefore, if bugs are planned frequently and according to the team' workload, overtime is substantially reduced. As a result, the correct application of this pattern may contribute to higher code quality as well as happier, more creative, and healthier team.

It is important to emphasize that in case the bug is committed during the bug planning but not delivered on the specified deadline, then the leader of the team should explain the reason why the bug was not fixed and delivered. This situation should not be common, but can take place if the subsystem team does not investigate enough in detail or if it is not able to easily reproduce the bug before it commits to it.

Related Patterns:

- The "Don't Interrupt an Interrupt" pattern can be used when someone is already working in "interrupt mode" on a critical issue of the project. Therefore, this pattern advises that the person who is working on this issue should continue handling it before moving on to the new one.

Known Uses:

- The bug planning described by [Churchville 2006] provides a mechanism to plan bugs in distributed software development projects by defining the risk, frequency, and severity. According to the [Churchville 2006], bugs with high-risk fix, low frequency and severity may not be fixed earlier in the project iterations. Nevertheless, bugs with high severity have always high priority to be fixed. Therefore, for each bug to be fixed, the person who plans the bug should evaluate if the bug fixing provides benefits. On the other hand, the bug fixing should be carried out later in the project.
- The test scripts technique used by [Fowler 2006] represent another approach to plan bugs during the project's iteration. In this scenario, the test scripts are written out before the start of the iteration by a system analyst/tester. These test scripts are written out based on the customer's requirements that should be implemented for a given iteration. During the iteration, regular builds are generated which allows the customer to correct misunderstandings as well as refine their own understandings. As the builds are generated, the customer runs the software and spot the bugs found in the system. After that, the bugs pointed out by the customer are fixed in the same iteration depending on the bug criticality.

Resulting Context:

If the "Plan Bugs on a Sustainable Pace" is adopted, then the goals committed to the sprint by the development teams have a higher probability of being fulfilled. In addition, this bug planning ensures that critical bugs are fixed during the sprint and consequently it keeps the product's quality as high as possible. Therefore, the zero-defect policy is usually not achieved during the sprints. The zero defect policy requires a high effort to fix the bugs which might directly impact the sprint goals. Nevertheless, the software's

bugs should be prioritized according to the features importance, and the decision to work on them should be evaluated in each project's sprint.

Another important result of the application of this pattern is that when the team leader commits the bug then he/she allocates developers to fix it and ensure that the bug will be fixed and delivered as promised at the beginning of the bug planning. Therefore, the development teams concentrate on fixing the bug while carrying out the sprint's activities. Another result is that when a critical bug is found by the test team but not planned, then the development team responsible for that bug is not interrupted to fix it.

5. Conclusions

This paper presented an application of the Scrum methodology, Lean software development, as well as Organizational patterns in the context of multi-site software development. This paper describes the application of six selected patterns, with two of them being proposed as new patterns ("Plan Bugs on a Sustainable Pace" and "Stories Rework Subsystem") and one as an alternative application of an existing pattern ("Inversion of Control"). The first proposed pattern **"Plan Bugs on a Sustainable Pace**" is applied when the project is composed of several project's issues and the level of interruption is very high. Therefore, this pattern describes mechanisms to plan bugs on a sustainable pace in order to control and manage the product's quality and avoid project's interruptions.

The second proposed pattern **"Stories Rework Subsystem"** is applied when development teams are separated by layer (as in pattern "Subsystem by Skill") and stories or feature increments must be integrated and tested within one time limited iteration. Therefore, this pattern provides means to decompose, refine, and prioritize a story in order to fit into one iteration. The pattern **"Inversion of Control"** can be used in a multi-site organization when the need to communicate and assure understanding of requirements is of primary concern. Therefore, this pattern describes a mechanism where the team who will implement the functionality, will be responsible for writing the detailed requirements of that functionality in their preferred format.

As most agile practitioners advocate, we also believe that **co-location is most effective for the majority of software development endeavors**. However, there are still a number of reasons that **require development to be performed in multi-site configuration**, some of them **external to the team's influence**. The main drawback that we found about this configuration is communication overhead. In this case, excessive effort is spent to keep the development teams synchronized and to create and update the documentation. With this paper, we proposed a set of good practices and Software Engineering patterns that we expect can help minimize the main drawbacks present on the multi-site context.

References

Beck, K. (1999). *Extreme Programming Explained – Embrace Change*. Addison-Wesley.

Beedle, M.; Devos, M.; Sharon Y.; Schwaber, K.; Sutherland, J.; (1999). *Scum: An extension pattern language for hyperproductive software development.* In: Harrison, N.; Foote, B.; Rohnert, H. Pattern Languages of Program Design 4. Addison-Wesley.

Berczuk, S. (1996). *Configuration Management Patterns*. In the proceedings of the 1996 Pattern Languages of Programming Conference, PloP'96. Available at http://www.berczuk.com/pubs/PLoP96/. Last visit [3rd June 2007].

Berczuk, S.; Appleton, B. (2002). *Software Configuration Management Patterns.* First Edition, Addison-Wesley.

Berczuk, S. (2003). *Agile Codeline Management*. This paper was published as a StickyMinds Original article.

Bret, T. (2004). *Parallel Development Strategies for Software Configuration Management*. Published at the Summer 2004 issue of Methods & Tools. Available at http://www.methodsandtools.com/mt/download.php?summer04. Last Visit [3rd June 2007].

Churchville, D. (2006). *ExtremePlanner: Agile Project Management for Distributed Software Teams*. http://www.extremeplanner.com/blog/2006/06/biggest-misconception-in-software.html. Last Visit [7th July 2007].

Cohn, Mike (2005). *Agile Estimating and Planning*. Robert Martin Series, Prentice Hall.

Coplien, J. O.; Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development.* First Edition, Prentice Hall.

Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. Available at http://www.martinfowler.com/articles/injection.html. Last visit [28th December 2006].

Fowler, M. (2006). *Using an Agile Software Process with Offshore Development.* Available at http://www.martinfowler.com/articles/agileOffshore.html. Last visit [7th July 2007].

Lehman, M. M. (2000) - Rules and Tools for Software Evolution Planning and Management http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/611_2.pdf

Louzado D. A.; Cordeiro, L. C. (2005). *Aplicando Padrões de Gerência de Configuração de Software em Projetos Geograficamente Distribuídos*. *Proceedings of the 5º Latin American Conference on Pattern Languages of Programming (SugarLoafPlop'2005).*

Poppendieck, Mary and Poppendieck, Tom (2003) *Lean Software Development: An Agile Toolkit*. First Edition, Addison Wesley.

Poppendieck, Tom (2003) *The Agile Customer's Toolkit.* Available at www.poppendieck.com/pdfs/Agile_Customers_Toolkit_Paper.pdf. Last visit [26th December 2006].

Principia Cybernetica. Available at http://pespmc1.vub.ac.be/SUBOPTIM.html. Last visit [26th December 2006].

Schwaber, K., and Beedle, M. (2002)**.** *Agile Software Development with Scrum*. First Edition, Series in Agile Software Development, Prentice Hall.

Wingerd L., Seiwald, C. (1998). *High-level Best Practices in Software Configuration Management*. Springer Berlin, Vol. 1439, pp. 57-66.