SCRUM: An extension pattern language for hyperproductive software development

Article · December 1998

Source: CiteSeer

4 authors, including:

Martine Devos
TeamGenius
13 PUBLICATIONS **173** CITATIONS
SEE PROFILE

Jeff Sutherland
Institute of Electrical and Electronics Engineers
76 PUBLICATIONS **1,666** CITATIONS
SEE PROFILE

SCRUM: An extension pattern language for hyperproductive software development

Mike Beedle beedlem@fti-consulting.com
Martine Devos mdevos@argo.be
Yonat Sharon yonat@usa.net
Ken Schwaber virman@aol.com
Jeff Sutherland jeff.sutherland@idx.com

Abstract

The patterns of the SCRUM development method are presented as an extension pattern language to the existing organizational pattern languages. In the last few years, the SCRUM development method has rapidly gained recognition as an effective tool to hyper-productive software development. However, when SCRUM patterns are combined with other existing organizational patterns, they lead to highly adaptive, yet well-structured software development organizations. Also, decomposing SCRUM into patterns can guide adoption of only those parts of SCRUM that are applicable to a specific situation.

1. Introduction

NOTE: Throughout this paper we assume the reader is familiar with the other org patterns [OrgPatt], [Coplien95]. Also, the written forms of all pattern names will be presented in bolded-italics types as in: ***DeveloperControlsProcess***, while the things that the patterns are will be represented in italics as in: *Backlog*.

Can a repeatable and defined process really exist for software development? Some think this is not only possible but necessary, for example those that favor the CMM (Capability Maturity Model) approach to software development [1]. The CMM defines five stages of process maturity initial, repeatable, defined, managed and optimizing, and asks its users to define the processes of 18 KPA (key process areas).

However, many of us doing work in the trenches have found over time that the "repeatable or defined" process approach makes many incorrect assumptions, such as:

1) Repeatable/defined problem. A repeatable/defined process assumes that there is a step to capture requirements, but in most cases, it is not possible to define the requirements of an application, because they are either not well defined or they keep changing.
2) Repeatable/defined solution. A repeatable/defined process assumes that an architecture can be fully specified, but in reality it is evolved, partly due to the fact of missing or changing requirements (as described above), and partly because of the creative process involved in creating it.
3) Repeatable/defined developers. The capabilities of a software developer vary widely, so a process that works for one developer may not work for another one.
4) Repeatable/defined organizational environment. The schedule pressure, priorities (e.g. quality vs. price), client behavior, and so on; are never repeatable or defined.

The problem with these assumptions is that they assume non-chaotic behavior. Even small unknowns can have a big influence on the result.

In actuality, the removal of uncertainties is impossible. Many of us have searched for answers beyond the repeatable/defined approach of software development, in a more "adaptive approach".

SCRUM assumes up-front the existence of chaos discussed above as incorrect assumptions, and provides techniques to resolve these problems. These techniques are rooted in complexity management i.e. self-organization, management of empirical processes and knowledge creation.

In that sense, SCRUM is not only an "iterative and incremental" development method but also an "adaptive" software development method.

2. How does SCRUM work?

SCRUM's goal is to deliver as much quality software as possible within a series (3-8), of short time-boxes (fixed time intervals) called *Sprints* that typically last about a month.

Each stage in the development cycle (Requirements, Analysis, Design, Evolution, and Delivery) is now mapped to a *Sprint* or series of *Sprints*. The traditional software development stages are retained for convenience primarily for tracking milestones. So, for example, the Requirements stage may use one *Sprint*, including the delivery of a prototype. The Analysis and Design stages may take one *Sprint* each. While the Evolution stage may take anywhere from 3 to 5 *Sprints*.

As opposed to a repeatable and defined process approach, in SCRUM there is no predefined process within a *Sprint*. Instead, *Scrum Meetings* drive the completion of the allocated activities.

Each *Sprint* operates on a number of work items called a *Backlog*. As a rule, no more items are externally added into the *Backlog* within a *Sprint*. Internal items resulting from the original pre-allocated *Backlog* can be added to it. The goal of a *Sprint* is to complete as much quality software as possible, but typically less software is delivered in practice (***Worse Is Better*** pattern). The end result is that there are non-perfect ***NamedStableBases*** delivered every *Sprint*.

Figure 1. A rugby team also uses *Scrum Meetings* **(not shown here).**

During a *Sprint***,** *Scrum Meetings* are held daily to determine on:

1) what items were completed since the last *Scrum Meeting***.**
2) what issues or blocks have been found that need to be resolved. (The *ScrumMaster* is a team leader role responsible for resolving the blocks.)
3) what new assignments make sense for the team to complete until the next *Scrum Meeting***.**

Scrum Meetings allow the development team to "socialize the team members knowledge" and have a deep cultural transcendence.

This "knowledge socialization" promotes to a self-organized team structure, where the development process is evolved on a daily basis.

At the end of each *Sprint*, there is a *Demo* to:

1) show the customer what's going on (***EngageCustomer***)
2) give the developer a sense of accomplishment (***CompensateSuccess***)
3) integrate and test a reasonable portion of the software being developed (***EngageQA***)
4) ensure **real progress** – reduction of backlog, not just the production of more papers / hours spent (***NamedStableBases***)

After gathering and reprioritizing leftover and new tasks, a new *Backlog* is formed and a new *Sprint* starts. Potentially, many other org patterns (organization and process patterns) may be used in combination with the SCRUM patterns. While Coplien org patterns, are an obvious choice because of their breadth, other org patterns from other sources may also be very valuable [OrgPatt], [Coplien95].

"The system requires it" or the "The system does not allow it " have become accepted (and often unavoidable) justifications for human behavior. What we fear is that current methods do not allow us to build *soft enough* software, because present methods and design paradigms seem to inhibit adaptability. Therefore the majority of software practitioners tend to become experts at what they can *specify in advance*, working with the unstated belief that there exists an optimal solution that can be planned a priori.

Once technology is adopted by an organization, it often becomes a constraining structure that in part shapes the action space of the user. So we build software too much like we build hardware, as if it were difficult to change, as if it has to be difficult to change.

In contrast, SCRUM allows us to build *softer* software, so there is no need to write full requirements up front. The user does not know what is possible and will ask for the pre-tech-paper solution that he perceives to be possible. But not even the software developers know fully what can be built before it is. Therefore, the user has no concept of what is possible before he can feel it, or touch it [Blu96].

Clearly, we need a softer approach for building software. We should recognize that *it is impossible* to have full requirements specified up-front or to freeze the context and environment. Requirements are written in a context. Our system transforms that context. New problems arise in the system and the new context.

This issue is not solved through improved methods for identifying the user requirements. Instead it calls for a more complex process of generating fundamentally new operating alternatives. The empirical way of working in SCRUM is one of the possible alternatives.

3. The SCRUM Pattern Language

The following diagram shows the relationships among the SCRUM patterns and other org patterns.

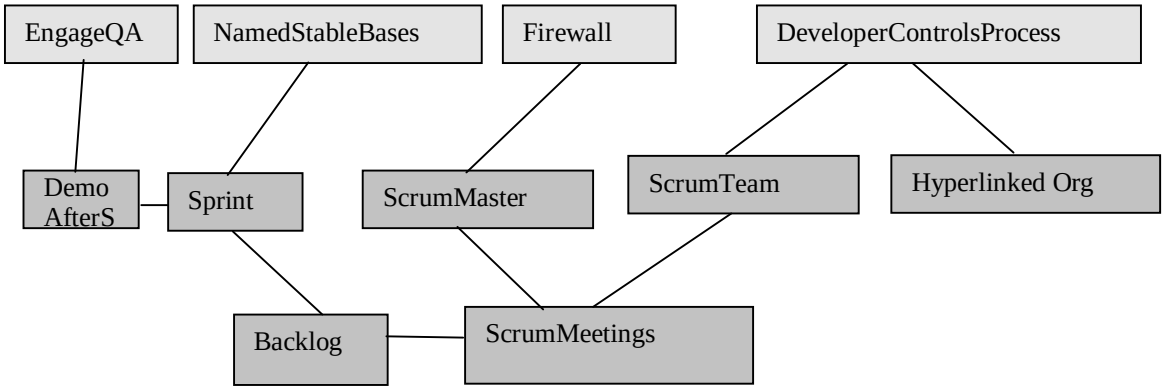

Figure 2. SCRUM Pattern Language Lattice

4. The Patterns of SCRUM

Scrum Meeting

Context

(FROM: ***Scrum Team, Scrum Master)***

Y are a software developer or a coach managing a software development team where there is a high percentage of discovery, creativity or testing involved. For example, a first time delivery where the problem has to be specified, or an object model has to be created, or new or changing technologies are being used.

Activities such as scientific research, innovation, invention, architecture, engineering and a myriad of other business situations may also exhibit this behavior.

You may also be a "knowledge worker", an engineer, a writer, a research scientist, or an artist, or a coach or manager who is overseeing the activities of a team in these environments.

(Misfit variables: estimation, planning, tracking, human comfort)

NOTE: Misfit variables are variables in the context that can be adjusted so the solution can solve the problem.

Problem

What is the best way to control an empirical and unpredictable process such as software development, scientific research, artistic projects or innovative designs where it is hard to define the artifacts to be produced and the processes to achieve them?

Forces (Analysis of Misfit Variables in the Context)

[NOTE: MISFIT examples are referred by some as anti-patterns. I use here some Dilbert-like names to indicate their awry nature.]

Estimation

(+) Accurate estimation for activities involving discovery, creativity or testing is difficult because it typically involves large variances, and because small differences in circumstances may cause significant differences in results.

These uncertainties come in at least 4 flavors:

a) Requirements are not well understood.

b) Architectural dependencies are not easy to understand and are constantly changing.

c) There may be unforeseen challenges with the technology. Even if the challenges are known in advance, their solutions and related effort are not known.

d) There may be hard bugs to resolve in the software, and therefore, it is typical to see project estimates that are several orders of magnitude off. You can't "plan bugs", you can only plan bug handling and provide appropriate prevention schemes based on the possibility of unexpected bugs.

MISFIT example: ***YouGotTheWrongNumber*** In projects with new or changing requirements, a new architecture, new or changing technologies, and difficult bugs to weed out, it is typical to see project estimates that are off by several orders of magnitude.

(-) Estimation is important. One must be able to determine what are the future tasks within some time horizon and prepare resources in advance.

MISFIT example: ***SetMeUpForFailure*** Projects where no estimation is done are difficult to manage.

Planning

(+) Planning and reprioritizing tasks takes time. Using the knowledge workers' in time planning meetings decreases productivity. Moreover, if the system is chaotic, no amount of planning can reduce uncertainties.

MISFIT example: ***ParalysisByPlanning*** Projects that waste everybody's time in planning everything to an extreme detail but are never able to meet the plans.

(+) A plan that is too detailed becomes huge and hard to follow. It may be easier to just apply common sense or call the customer. Also, the bigger the plan, the more errors it will contain (alternatively, the cost of verifying its correctness grows).

(-) No planning at all increases uncertainty among team members and would eventually damage morale.

MISFIT example: ***LostVision*** Projects that never schedule anything tend to lose control over their expectations. Without some schedule pressure no one will do anything, and worse, it will become difficult to integrate together the different parts being worked on independently.

Tracking

(+) Too much monitoring wastes time and suffocates developers.
(+) Tracking does not increase the certainty of the indicators because of the chaotic nature of the system.
(+) Too much data is meaningless - The Haystack Syndrome.

MISFIT example: ***MeasuredToDeath*** Projects that waste everybody's time in tracking everything to an extreme detail but are never able to meet the plans. (You measured the tire pressure until all the air was out!)

(-) Not enough monitoring leads to blocks and possible idle time between assignments.

MISFIT example: ***WhatHappenedHere?*** Projects that never track anything tend to lose control over what is being done. And eventually no one really knows what has been done.

Solution

(A thing, albeit temporary)
Meet with the team members for a short time (~15 minutes) in a daily ***Scrum Meeting***, where the only activity is asking each participant the following 3 questions:

(A process, and what stays the same)
1) What they worked on since the last *Scrum Meeting*. The *Scrum Master* logs what tasks have been completed and what remains undone.
2) What blocks if any they found in performing their tasks within the last 24 hrs. The *Scrum Master* logs all blocks and later finds a way to resolve the blocks.
3) What they will be working in the next 24 hrs. The *Scrum Master* helps the team members choosing the appropriate tasks to work on with the help of the Architect. Because the tasks are schedule on a 24 hr basis the tasks are typically small (***Small Assignments***).

Scrum Meetings typically take place at the same time and place every day, so they also serve to build a strong culture. As such, *Scrum Meetings* are rituals that enhance the socialization of status, issues, and plans for the team. The *ScrumMaster* leads the meetings and logs all the tasks from every member of the team into a global project *Backlog*. He also logs every block and resolves each block while the developers work on new assignments.

Scrum Meetings not only schedule tasks for the developers but can and should schedule activities for everyone involved in the project such as integration personnel dedicated to configuration management, architects, *Scrum Masters*, ***Firewall*** [Coplien95], ***Coach*** [Beedle97], or a QA team.

Scrum Meetings allow knowledge workers to accomplish mid-term goals typically allocated in Sprints that last for about a month.

(what changes)
Scrum Meetings can also be held by self-directed teams, in that case, someone is designated as the scribe and logs the completed and planned activities of the *Backlog* and the existing blocks. All activities from the *Backlog* and the blocks are then distributed among the team members for resolution.

The format of the *Backlog* and the blocks can also vary, ranging from a list of items in a piece of paper, to software representations of it over the INTERNET/INTRANET [Schwaber97]. The SCRUM cycle can be adjusted but typically ranges between 2 hrs, and 48 hrs.

Rationale

It is very easy to over- or under- estimate, which leads to either idle developer's time or to delays in the completion of an assignment. Therefore, it is better to *sample* frequently the status of small assignments. Processes with a high degree of unpredictability cannot use traditional project planning techniques ***only***, such as Gantt or PERT charts because the rate of change of what is being analyzed, accomplished, or created is too high. Instead, constant reprioritization of tasks offers an adaptive mechanism that provides sampling of systemic knowledge over short periods of time.

SCRUM meetings help also in the creation of an "anticipating culture" [Weinberg97], because they encourage productive values:

- increase the overall sense of urgency,
- promote the sharing of knowledge,
- encourage dense communications and
- facilitate "honesty" among developers since everyone has to give a daily status.

This same mechanism, encourages team members to socialize, externalize, internalize and combine technical knowledge on an ongoing basis, thus allowing technical expertise to become community property for the community of practice [Nonaka95]. *Scrum Meetings* are therefore rituals with deep cultural transcendence. Meeting at the same place at the same time, and with the same people, enhances a feeling of belonging, and creates the habit of sharing knowledge.
Seen from the System Dynamics point of view [Senge94], software development has a scheduling problem, because the nature of programming assignments has a rather probabilistic nature. Estimates are hard to come by because:

1) inexperienced developers, managers and architects are involved in making the estimates
2) there are typically interlocking architectural dependencies that are hard to manage
3) there are unknown or poorly documented requirements, or
4) there are unforeseen technical challenges

As a consequence, the software development becomes a chaotic *beer game*, where it is hard to estimate and control the *inventory* of available developer's time, unless increased monitoring of small assignments is implemented [Goldratt90], [Senge90]. In that sense the *Scrum Meeting* becomes the equivalent of a *thermometer* that constantly samples the team's temperature [Schwaber97-2].

From the Complexity Theory perspective [Holland95], [Holland98], SCRUM allows *flocking* by forcing a faster agent interaction, therefore accelerating the process of self-organization, because it shifts resources opportunistically, through the daily SCRUM meetings.

This is understandable, because the relaxation of a self-organized multi-agent system is proportional to the average exchange among agents per unit of time. And in fact, the "interaction rate" is one of the levers one can push to control "emergent" behavior -- it is like adding an enzyme or catalyst to a chemical reaction.

In SCRUM this means increasing the frequency of the SCRUM meetings, and allowing more *hyperlinks* as described below, but up to an optimal upper frequency bound on the SCRUM meetings (meetings/time), and up to an optimal upper bound on the hyper-links or the SCRUM team members. Otherwise the organization spends too much time socializing knowledge, instead of performing tasks.

Known Uses

(Mike Beedle) At Nike Securities in Chicago we have been using SCRUM meetings since February 1997 to run all of our projects including BPR and software development. Everyone involved in these projects receives a week of training in SCRUM techniques.

(Yonat Sharon) At Elementrix Technologies we had a project that was running way past late after about 5 months of development. Only a small part (about 20%) was completed and even this part had too many bugs. The project manager started running bi-daily short status meetings (none of us was familiar with the term SCRUM back then). In the following month, the entire project was completed and the quality had risen sharply. Two weeks later, a beta version was out. The meetings were discontinued and the project hardly progressed since. I don't think the success of the project can be attributed to the Scrum Meetings alone, but they did have a big part in this achievement.

One of my software team leaders at RAFAEL, implemented a variation of *Scrum Meetings*. He would visit each developer once a day, and ask him the 3 questions, and he also managed a backlog. This does not have the team building effects, but it does provide the frequent sampling.

Resulting Context

(TO: ***)***
A structure such as ***DeveloperControlsProcess*** is fully implemented through ***FormFollowsFunction***, or a ***CaseTeam*** in a business environment, is jelled into a highly adaptable and hyperproductive team structure [Coplien95], [Beedle97].

The application of this pattern also leads to:

- highly visible project status.
- highly visible individual productivity.
- less time wasted because of blocks.
- less time wasted because of waiting for someone else.
- increased Team Socialization

Sprint

Context

(FROM: ***NamedStableBases[Coplien95])***

You are a software developer or a coach managing a software development team where there is a high percentage of discovery, creativity or testing involved.

You are building or expanding systems, that allow partitioning of work, with clean interfacing, components or objects.

Problem

We want to balance the need of developers to work undisturbed and the need for management and the customer to see real progress.

Forces

Developers need time to work undisturbed, but they need support for logistics and management and users need to be convinced that real progress is made. (Misfit: Developer without any kind of control will not be productive.)

Often, by the time systems are delivered, it is obsolete or it requires major changes. The problem is that input from the environment is mostly collected at the start of the project, while the user learns most using the system or intermediate releases. (Misfit: Management that is too tight will slow things down.)

Some problems are "wicked", that is it difficult to even describe the problem without a notion of the solution. It is wrong to expect developers do to a clean design and commit to it at the start of this kind of problems. Experimentation, feedback, creativity are needed. (Misfit: Pretending that the requirements can be written down without attempting a solution first is a fallacy.)

We want every person on the team to understand the problem fully and to be aware of all the steps in development. This limits the size of the team and of the system developed. Trust is a core value in SCRUM, and especially important for the success of *Sprints*, so ***SelfSelectingTeams*** is a plus [Coplien95]. (Misfit: Large teams don't work well, because there is a limit as to how many humans can work on something together i.e bounded rationality problem.)

For many people – project managers, customers, it is difficult to give up control and proof of progress as provided in traditional development. It feels risky to do so; there is no guarantee that the team will deliver. Customers and users can seldom give a final spec because their needs as constantly evolving. The best they can do is evolve a product as their needs evolve and as they learn along the process. In our development process we don't use this learning cycle and its benefits. (Misfit: Customer don't really know what they want until they see it, pretending otherwise leads to problems.)

Most systems development has the wrong basis. It supposes that the development process is a well-understood approach that can be planned and estimated. If a project fails, that is considered proof that the development process needs more rigor. If we could consider developers to follow the process more rigorous the project can be completed successfully. But these step by step approaches don't work, because they do not cope with the unpredictabilities (both human and technical) in system development.
At the start of neither a complete, detailed specification and planning of nor scheduling is possible, because of these many uncertainties. (Misfit: Rigid processes are often too constraining and fall short to deliver a system into production.)

Developers and project managers often live, or are forced to live a lie. They have to pretend that they can plan, predict and deliver, and then work the best way that they know to deliver the system. They build one way, pretend to build another way, and as a result are without real controls. (Misfit: Rigid plans are often too constraining and fall short to deliver a system into production.)

Often overhead is created to prove that a process is on track.
We have followed Pert charts and the like, believing that a system would result.
Current process automation adds administrative work for managers and developers and results often in marginally used development processes that become disk-ware.
(Misfit: Activity is not synonymous with results. More often than not a project plan shows activities but hardly ensures progress and results.)

Solution

Each *Sprint* takes a pre-allocated amount of work from the Backlog. The team commits to it. As a rule nothing is added externally during a sprint. External additions are added to the global backlog. Blocks resulting from the *Sprint* can also be added to the Backlog. A Sprint ends with a Demonstration (***DemoAfterSprint***) of new functionality.

Give the developers the space to be creative, and to learn by exploring the design space, doing actual work, undisturbed by outside interruptions, free to adapt their way of working using opportunities and insights. At the same time keep the management and stakeholders confident by showing real progress instead of documents and reports… produced as proof. Do this in short cycles, *Sprints*, where part of the *Backlog* is allocated to a small team. In a *Sprint*, during a period of approximately 30 days, an agreed amount of work will be performed, to create a deliverable. *Backlog* is assigned to *Sprints* by

priority and by approximation of what can be accomplished during a month. Chunks of high cohesion and low coupling are selected. The focus is on enabling, rather than micro-management.

During the *Sprint*, outside chaos is not allowed in the increment. The team, as they proceed, may change course and their way of working. By buffering them from the outside, we allow them to focus on the work at hand and on delivering the best they can and the best way they can, using their skill, experience and creativity.

Each *Sprint* produces a visible and usable deliverable. This is demonstrated in *Demo*. An increment can be either intermediate or shippable, but it should stand on its own. The goal of a *Sprint* is to complete as much quality software as possible and to ensure real progress, not paper milestones as alibi.

Rationale

Developing systems is unpredictable and chaotic. Development is an empirical process that requires significant thought during the process. A method can only supply a framework for the real work and indicate the places where creativity is needed. Yet we tread black-box processes often as fully defined processes. Unpredictable results occur. We lack the controls to measure and respond to the unpredictable.

While building a system many artifacts come into existence, many new insights are gained. These new artifacts can guide future thinking. Increased productivity through good tools or uncovered components may open the opportunity for adding more *Backlog* and more functionality to our system, or for releasing a product early.

Therefore, during a *Sprint*, we optimize communications and maximize information sharing in daily *Scrum Meetings*.

Sprints set up a safe environment and time slots where developers can work undisturbed by outside requests or opportunities. They also offer a pre-allocated piece of work that the customer, management and the user can trust the *Scrum Team* to produce as a useful deliverable, such as a working piece of code at the end of the *Sprint*. The team focuses on the right things to do, management working on eliminating what stands in this way of doing in better.

Known Uses

At Argo, the Flemish department of education, we have been using *Sprints* since January 1997 on a large number of end-user-projects and for the development of a framework for database, document management and workflow. The *Backlog* is divided in *Sprints* that last about a month. At the end of each *Sprint* a working Smalltalk image is delivered with integration of all current applications. The team meets daily in *Scrum Meetings* and

Backlog is re-prioritized after the *Demo* in a monthly meeting with the steering committee.

Resulting Context

(TO: ***Backlog)***
High degree of effective ownership by the participants, including users who stay involved through *Demo*'s and the prioritizing of the *Backlog*).

At the end of a *Sprint*, we have the best approximation of what was planned at the start of the *Sprint*. At the end of the *Sprint*, in a review session, the supervisors have the opportunity to change the planning for the future. The project is totally flexible at this point. Target, product, delivery date and cost can be redefined.

With SCRUM we get a large amount of post-planning flexibility (for both customer and developer).

It may become clear, in the daily *Scrum Meetings* throughout the *Sprint* that some team-members are loosing time at non- or less productive tasks. Alternatively, it may also become clear that people need more time for their tasks than originally allocated by management, because developers may turn out less competent or experienced at the allocated task than assumed or they may be in political or power struggles. But the high-visibility of SCRUM allows us to deal with these problems. This is the strength of the SCRUM method manifested through the *Scrum Meetings* and the *Sprints*.

Difficulties in grouping backlog for a sprint may indicate that priorities are not clear to management or to the customer.

The method is not suitable for people who need strong guidance.

Backlog

Context

(FROM: ***Scrum Meetings, Sprints)***

Your are anyone connected to a software project, or any other project that is chaotic in nature that needs a information on what to do next.

Problem

What is the best way to organize the work to be done next at any stage of the project?

Forces

Project plans captured in Pert charts or Gannt charts often try to capture tasks to be done a priori, but they often fail in their implementations, because they lack flexibility. Tasks are pre-allocated time in Pert or Gant charts but their priorities and number grow or dimish as required in *real* projects**,** and therefore they are not good tools to use where the number of tasks changes drastically over time

Not having a repository of tasks in any shape or form simply translates into project failure. There must be some sort of project control.

Solution

Use a *Backlog* to organize the work a SCRUM team**.**

The *Backlog* is a prioritized list. The highest priority backlog will be worked on first, the lowest priority backlog will be worked on last. No feature, addition, enhancement to a product is worth fighting over; it is simply either more important or less important at any time to the success and relevance of the product.

Backlog is the work to be performed on a product. Completion of the work will transform the product from its current form into its vision. But in SCRUM, the *Backlog* evolves as the product and the environment in which it will be used evolves. The backlog is dynamic, constantly changed by management to ensure that the product defined by completing the *Backlog* is the most appropriate, competitive, useful product possible.

There are many sources for the backlog list. Product marketing adds work that will fulfill their vision of the product. Sales add work that will add new sales or extend the usefulness to the installed base. Technology adds work that will ensure the product uses

the most innovative and productive technology. Development adds work to enhance product functions. Customer support adds work to correct underlying product defects.

Only one person prioritizes work. This person is responsible for meeting the product vision. The title usually is product manager or product marketing manager. If anyone wants the priority of work changed, they have to convince this person to change that priority. The highest priority backlog has the most definition. It is also prioritized with an eye toward dependencies.

Depending on how quickly products are needed in the marketplace and the finances of the organization, one or more *Scrum Teams* work on a product's backlog. As a *Scrum Team* is available (newly formed or just finished a *Sprint*) to work on the backlog, the team meets with the product manager. Focusing on the highest priority backlog, the team selects that *Backlog* that the team believes it can complete within a *Sprint* iteration (30 days). In doing so, the Scrum team may alter the backlog priority by selecting backlog that is mutually supportive, that is, that can be worked on at once more easily that waiting. Examples are multiple work items that require developing a common module or interface and that make sense to include in one *Sprint*.

The team selects a cohesive group of top priority *Backlog*, that – once completed – will have reached an objective, or milestone. This is stated as the *Sprint's* objective. During the *Sprint*, the team is free to not do work as long as this objective is reached.

The team now decomposes the selected backlog into tasks. These tasks are discrete pieces of work that various team members sign up to do. Tasks are performed to complete backlog to reach the *Sprint* objective.

Resulting Context

Project work is identified dynamically and prioritized according to:

1) the customer's needs, and
2) what the team can do.

4. Conclusions

SCRUM is a knowledge creating process with a high level of information sharing during the whole cycle and work progress.

The key to SCRUM is pinning down the date at which we want completion for production or release, prioritizing functionality, identifying available resources and making major decisions about architecture. Compared to more traditional methodologies the planning phase is kept short since we know that events will require changes to initial plans and methods. SCRUM uses an empirical approach to development where interaction with the environment is not only allowed but encouraged, changing scope,

technology and functionality are expected and continuous information sharing and feedback keep performance and trust high.

When SCRUM is combined with other organizational patterns [OrgPatt], specially those by James O. Coplien [Coplien95], it provides with adaptive, yet well structured software development organization.

Their application also generates a strong culture with well-defined roles and relationships, with meaningful and transcending rituals.

Acknowledgements

We would like to thank all of the SCRUM users and reviewers from which we have received feedback over the years. Also, we thank the all of the members of the Chicago Patterns Group that attended an early review session of the ***Scrum Meeting*** pattern (especially Brad Appleton, Joe Seda and Bob Haugen). Finally we thank our PLOP98 shepherd, Linda Rising, for providing us comments and guidance to make our paper better.

(Personal acknowledgement from Mike Beedle.) I'd like to thank both Jeff Sutherland and Ken Schwaber for adapting the SCRUM techniques to software in the early 90s, and for sharing their findings with me. SCRUM has made a significant contribution to the software projects where I used the technique.

References

[Beedle97] Michael A. Beedle, *cOOherentBPR – A pattern language to build agile organizations*, PLoP '97 Proccedings, Tech. Report #wucs-97-34, Washington University, 1997.
[Coplien95] James O. Coplien and Douglas C. Schmidt, *Pattern Languages of Program Design* (*A Generative Development-Process Pattern Language),* Addison and Wesley, Reading, 1995.
[Goldratt90] Eliyahu Goldratt, *Theory of Constraints*, North River Press, Great Burlington (MA), 1990.
[Holland95] John Holland, *Hidden Order – How Adaptation Builds Complexity*, Helix Books, Addison-Wesley, Reading MA, (1995).
[Holland98] John Holland, *Emergence – from chaos to order*, Helix Books, Addison-Wesley, Reading MA, (1998).
[Nonaka95] I. Nonaka and H. Takeuchi, *The Knowledge Creating Company*, Oxford University Press, New York, 1995.
[OrgPatt] Org Patterns web site:
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?ProjectIndex
[Schwaber97] Kenn Schwaber's, *SCRUM web page*:
http://www.controlchaos.com
[Schwaber97-2] personal communication.

[Senge90] Peter Senge, *The Fifth Discipline - The art and Practice of the Learning Organization*, Doubleday/Currency, New York, 1990.
[Sutherland97] Jeff Sutherland, *SCRUM web page*:
http://www.tiac.net/users/jsuth/scrum/index.html
http://www.jeffsutherland.org/scrum/index.html
[Weinberg97] Gerald Weinberg, *Quality Software Management – Vol. 4., Anticipating Change*, Dorset House, New York, 1997.