Even more Patterns for the Magic Backlog

REBECCA WIRFS-BROCK, WIRFS-BROCK ASSOCIATES

LISE B. HVATUM

Agile development processes are driven from a product backlog of development items. This paper adds three more patterns to our Magic Backlog collection of patterns to build and structure the backlog for an agile development effort using the outcomes of requirements elicitation and analysis. RULES defines who should be allowed to do what in managing the backlog, SHARED DEFINITIONS deals with defining and consistently using clear criteria for core decisions across the backlog team(s), and REMODEL is about restructuring the overall backlog contents.

Categories and Subject Descriptors: • **Software and its engineering~Requirements analysis** • *Software and its engineering~Software implementation planning* • *Software and its engineering~Software development methods*

General Terms: Management

Additional Key Words and Phrases: requirements engineering, requirements analysis, agile, product backlog

ACM Reference Format:

Wirfs-Brock, R. and Hvatum, L. 2018. Even more Patterns for the Magic Backlog. 25th Conference on Pattern Languages of Programming (PLoP), PLoP 2018, Oct 24-26 2018, 17 pages.

1. INTRODUCTION

The initial scope of a product and most of the product requirements are generated through requirements gathering using elicitation techniques. This output of requirements gathering is then processed using techniques like story mapping, use cases, and workflows [Hva2015]. There are a number of publications that provide methods and techniques for how to elicit, analyze and process information to reach detailed software requirements [AB2006, Agi2015, Amb2014, BC2012, BKW2018 Got2002, Got2005, GG2012, HH2008, KBN2015, KS2009, KI2012, Mas2010, Mul2016, Pat2014, Rad2016, Rin2009, RR2006, Rot2016, RW2013, SS2013, Sut2014 Wie2009, Wie2006, Wit2007]. Our patterns add to this software requirements engineering body of knowledge by providing practical advice on how to build a good product backlog from these requirements for a large and complex product using an Application Lifecycle Management (ALM) tool.

With long careers in software development, we have observed several problematic product backlogs that did not provide as much value to the development teams as they should. We have also seen backlogs that are thoughtfully created and well maintained. Those backlogs support teams and help them be more efficient and in better control of their development. The patterns presented in this and four earlier papers [Hva2015, Wir2016, Hva2017, Hva2018] provide knowledge around product backlogs to assist software development teams in creating and sustaining high quality backlogs. This paper documents three additional patterns (RULES, SHARED DEFINITIONS, and REMODEL) that we identified when reflecting on our existing set of patterns and running through various scenarios of their use. RULES describes who should be allowed to do what when managing the backlog; SHARED DEFINITIONS deals with defining and consistently using terminology in order to establish clear criteria for core decisions across the backlog team(s), and REMODEL is about restructuring the overall backlog contents to better meet current product development.

In our writing, we aim to be as process agnostic as possible. Still, we have an expectation that the process applied to manage a product backlog is a form of agile/lean, not least because this is the reference for our

personal experience. The reader will find that our use of terminology works better for someone who has exposure to forms of agile development methodologies.

The term "Product Backlog" is part of Scrum terminology and is defined in the Scrum Guide [SS2013]:

"The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. […] The Product Backlog lists all features, functions, requirements, enhancements, and fixes that constitute the changes to be made to the product in future releases."

The items in the ordered list are Product Backlog Items (PBIs), a generic term that allows for any representation of a product "need" to be included in the Product Backlog. A common item type is the user story, but to think of the Product Backlog purely as a list of user stories is too simplistic for any large and complex system. Given that the Product Backlog is the "single source" for driving system development, you want it to give you the full picture of the product requirements. For the remainder of this paper we will use the term "backlog" to mean the Product Backlog.

The initial backlog typically has PBIs of varying granularity, from specific detailed needs to rough ideas on a theme or epic level. As development progresses, the contents are prepared and modified to reflect the current understanding of the product and the efforts required to create it. The Scrum Guide [SS2013] states:

"A Product Backlog is never complete. The earliest development of it only lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. As long as a product exists, its Product Backlog also exists."

In the context of our work, the product backlog is more than a simple to-do list for the team. It captures the "contract" with stakeholders through the detailing of requirements, it supports team planning, and it provides a wealth of information depending on how attributes of the various backlog items are used and maintained. Not least, the backlog serves as the implementation history. By keeping well-documented backlog items that are already completed (for example user stories that are Done), you have a foundation for regression testing with traceability to the original requirements, you have documentation in case of legal implications, and you have history that you can analyze and apply for learning and to contribute to future decisions like estimation and planning. Should your team ever be investigated (maybe a lawsuit involving intellectual property, or because of an accident where your software was in use) it may be a non-negotiable stipulation to be able to document the defined and verified acceptance criteria for key features, and the testing performed with traceability to the software specifications.

Our patterns define role-based activities and responsibilities. We expect individuals to be moving between roles depending on what they are currently doing. A product owner could double as a tester. A project manager might do the work of a business analyst, as well as development and testing. One role that especially needs clarification is that of the analyst. A Scandinavian proverb states that, "A beloved child has many names," and this is true of this role. Business analyst, product analyst, and requirements engineer are frequently used. In essence this role is an expert on requirements engineering, i.e. the elicitation and management of requirements. The analyst is not a domain expert, but a skilled individual who knows how to drive elicitation efforts, how to analyze and structure the outcome, how to translate from the business domain to the technical domain, and how to administer and maintain a requirements collection for a product. The role of the analyst often falls on a project manager or a product owner, and sometimes on the development team as a shared responsibility. But as products are getting larger and more complex, there is an emerging analyst profession, and more frequently teams include a dedicated analyst. The primary audience for our patterns is the analyst role.

2. THE BACKLOG PATTERNS

The eighteen patterns in the Magic Backlog collection give practical help on building a good quality product backlog for a software project or program. The need to structure and manage the backlog and the associated development workflows with some degree of formality increases with project size. The specific context of the Magic Backlog collection of patterns for the development of products of significant scope and complexity, with at least a three-year time frame for gradually delivering the full system. These projects have little choice but to use professional tooling and dedicate time and resources in backlog management.

We initially used the term "Magic Backlog" because agile process descriptions pay relatively little attention to the creation of the backlog – so it appears as if by magic. Feedback we got during the writing about backlog creation made us realize that the term has another meaning: done well with the right contents and structure the backlog can "do magic" to support the team. Readers familiar with the stories of The Magic School Bus [Sch2015] will probably recognize the connection. If you need a submarine, the school bus will transform into one. If you need a microscope, or a fully equipped biology lab, there will be one in the bus. With careful design and preparation of your backlog, it can be as magic as the school bus, supporting your current needs. It provides a technical view of the product to the development team, while the product owner can see a business view. It keeps the current plan for the project manager, and the testing structure for QA. It helps you know where you are and where you should be going next.

In the remainder of this section we give an introduction to the patterns collection. The patterns are provided in short sequences that focus on the purpose of applying the patterns. These sequences are by no means prescriptive but are meant to show a natural and logical progress for a development team in dealing with their backlog. As such they are an example of possible usage. The illustrations in the figures are using house construction as a metaphor to emphasize that many backlog management activities are about creating and repairing meaningful, useful structures.

The first sequence of patterns shown in figure 1 covers the fundamental goal of team planning and monitoring. By creating a FRAME that represents a hierarchy of product functionality you can more easily get an overview of product status and report on progress in a way that makes sense to the stakeholders and users. ALM tools use attributes to help in development planning so by setting these attributes you can see the PLANS for items to go through the stages of readying, implementation, final qualification, and into production. By creating CONNECTIONS from other backlog item types like test definitions, defects, and code revisions you get traceability and can determine the readiness of the functionality items for deployment. Through shared queries and reports you can generate ANSWERS that help the team in planning and assigning work, as well as showing the status of the product under development (for example a list of open defects, or user stories ready for implementation, or passing and failing tests).

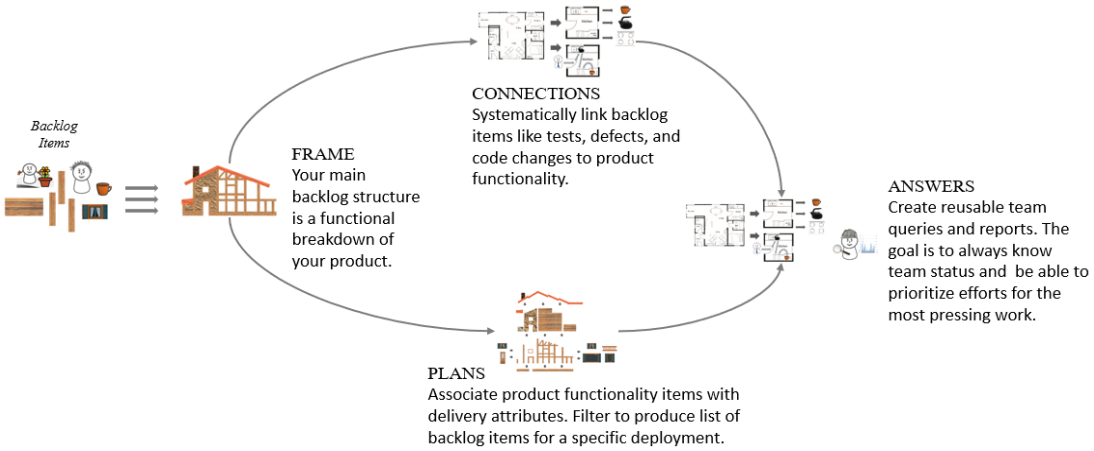

Figure 1: Planning and Monitoring Patterns Sequence

The patterns shown in figure 2 provide richer contents around the users of the product and how their usage is envisioned. PEOPLE adds more detail about the user profiles allows for variation within each user type (for instance experienced users versus inexperienced users, casual users versus frequent/expert users etc.). This information is associated with (linked to) the product functionality of the backlog FRAME. The TALES is prose that describes use of the system, and can focus on specific users, on personas described by PEOPLE, or on transactions or operational scenarios. It is a way to create an understanding of how functionality items in the backlog fit together to create more complex functionality. The USAGE MODELS is an alternative way to bring this understanding, and since a model is more structured than a narrative it better supports planning and prioritization by showing how backlog items like user stories fit together in an operational workflow. The two patterns TALES and USAGE MODELS complement each other, and can each be applied without the PEOPLE pattern. The richer content will improve the ANSWERS.

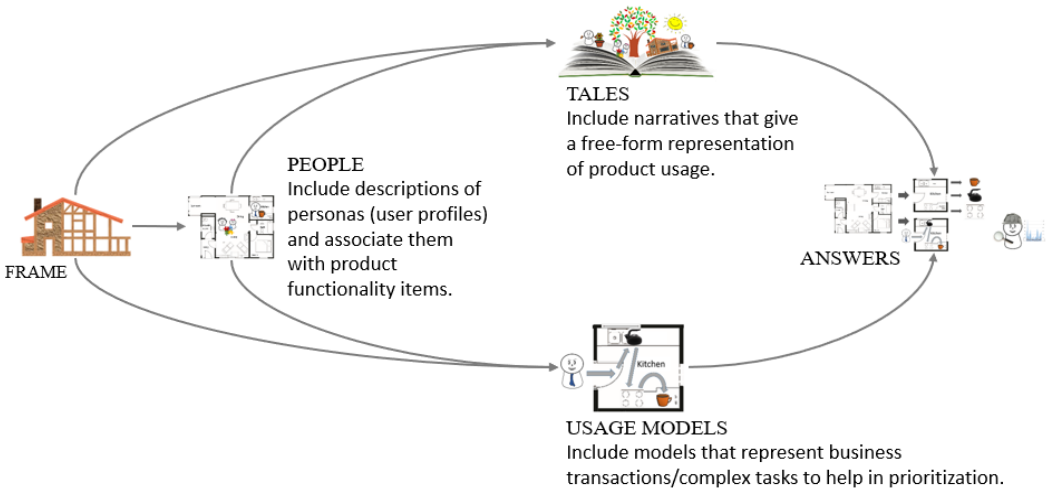

Figure 2: Users and Usage Patterns Sequence

As the project team grows and matures, backlog management may get more sophisticated. Defining and applying SHARED DEFINITIONS helps in making sure contents are managed consistently, while defining and enforcing RULES helps in making sure core contents are consistently kept, for instance keeping a clearly defined structure for the FRAME. The VIEWS create alternative representations of the product, for instance representations that focus on architecture and on testing.

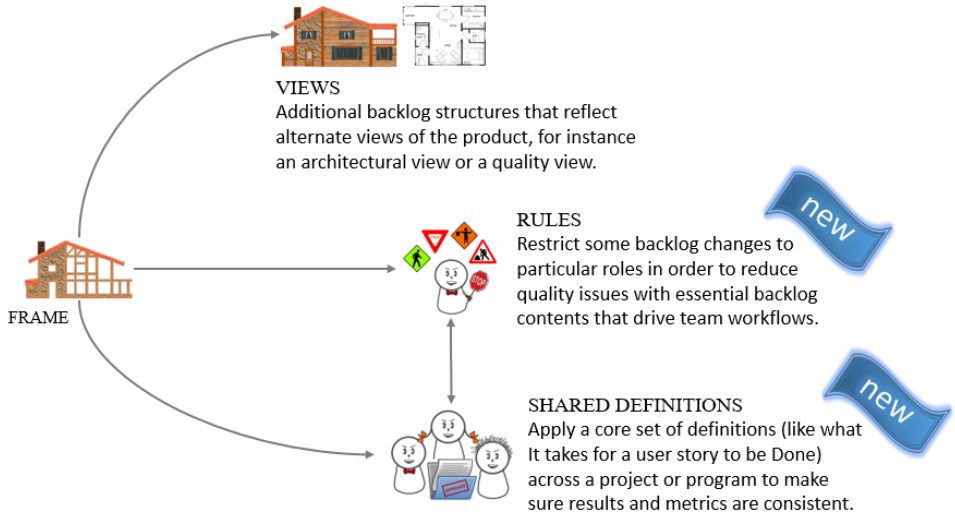

Figure 3: Diversified Team Roles Patterns Sequence

An activity within the development process is the preparation of functional items in the backlog for implementation (for instance making sure a user story has sufficient details and is approved by the business owner and maybe given a priority). The FUNNEL adds product ideas to the backlog with the expectation that only some of these ideas will eventually be implemented, while keeping all product ideas in the same repository (ALM tool) rather than keeping multiple lists and documents. Items at an early ideations stage are added as PLACEHOLDERS to be replaced by more elaborate backlog items later. The PIPELINE describes how the process of going from product idea to implementation ready items is implemented in the backlog.

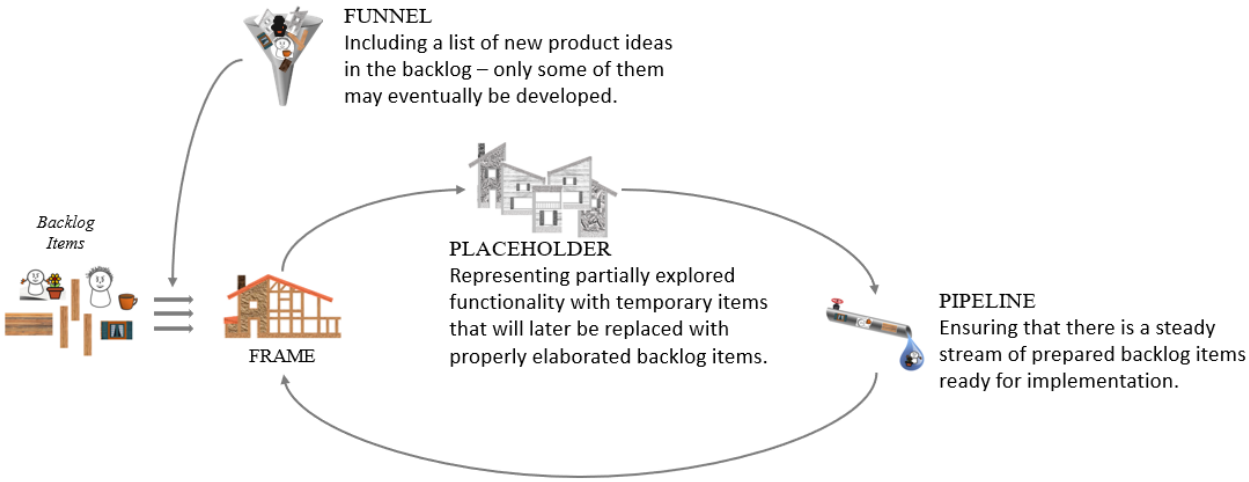

Figure 4: Preparing for Implementation Patterns Sequence

With several team members changing backlog items, over time it becomes increasingly difficult to ensure all the data are consistent and adhering to the FRAME and SHARED DEFINTIONS. To keep the contents up to date requires regular MAINTENANCE of the backlog. More fundamentally, a maturing of the product understanding or changes in the team's development process can lead to the FRAME and other contents no longer serving the needs of the team. A backlog REMODEL may be needed to update the contents and improve the overall structure.

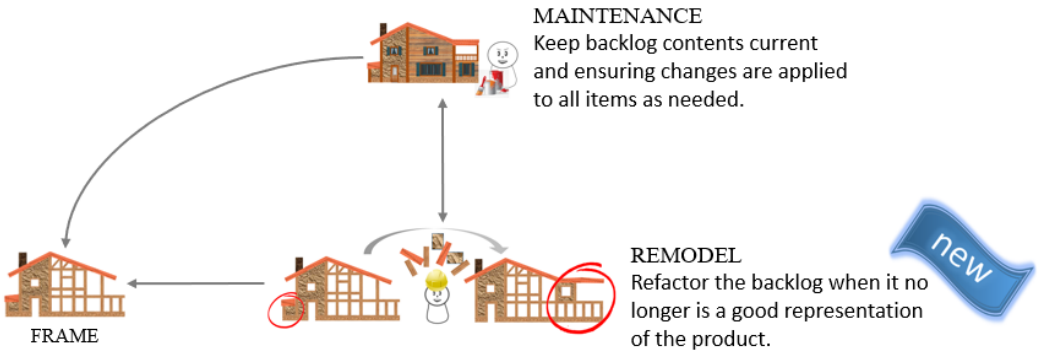

Figure 5: Keeping the Backlog Healthy Patterns Sequence

Three patterns (SINGLE BACKLOG, LINKED BACKLOGS, and AD HOC BACKLOGS) deal with backlogs for programs (or organizations where several teams are contributing to a single product, for example feature teams, squads, or whatever an organization uses to structure their coordinated efforts). These patterns are alternative solutions depending on the program context, and we therefore show these in a table rather than as a patterns sequence. The other patterns in the Magic Backlog collection are equally useful for the selected solution for a product backlog on the program level.

Table 1 shows these patterns with the benefits and tradeoffs of choosing a particular solution.

Name	Description	Benefits	Issues
Ad hoc Program Backlogs	Manage a program through a set of project level backlogs and a separate program level backlog	✓ Existing project level backlogs remains ✓ Program level backlog enables program planning and reporting	- Disconnected backlogs - Reporting is more complex
Linked Program Backlogs	Manage a program through a set of project level backlogs with links to a program level backlog	✓ Existing project level backlogs can remain ✓ Linking work items reduce risk of missing functionality	- Refactoring backlogs - Reporting across multiple sources
Unified Program Backlog	Manage a program through one backlog	✓ Planning is easier ✓ Reporting is simplified	- Rework to merge current backlogs - More maintenance

Table 1: Product Backlog Patterns for Programs

As stated in the introduction, the expectation for these patterns is that the team is using an ALM tool to maintain the backlog. The pattern solutions utilize the functionality of typical ALM tools (e.g. TFS/VSTS, JIRA, Doors NG, and several others) to link objects (FRAME, CONNECTIONS), set attributes (PLANS, PEOPLE), and create queries for generating outcomes (ANSWERS). We try to ensure that our solutions are generic and can be implemented in any ALM tooling, but we do not have personal experience with every tool on the market so depending on the tooling some solutions may need to be tweaked for implementation.

Since our first paper in 2015, our patterns collection has gradually been growing more complete. This paper is an attempt to complete the collection by documenting three additional patterns: RULES, SHARED DEFINITIONS, and REMODEL. The full descriptions of the earlier patterns can be found in the following papers:

FRAME, VIEWS, PEOPLE, TALES, USAGE MODELS, PLACEHOLDERS, PLANS, CONNECTIONS, ANSWERS are documented in "*Patterns to Build the Magic Backlog*" [Hva2015] workshopped at EuroPLoP 2015

PIPELINE, FUNNEL, MAINTENANCE are documented in "*More Patterns for the Magic Backlog*" [Wir2016] workshopped at PLoP 2016

SINGLE BACKLOG, LINKED BACKLOG, AD HOC BACKLOG are documented in "*A Program Backlog Story with Patterns*" [HVA2018] workshopped at EuroPLoP 2018

Patterns sequences exploring use of the patterns was presented in "*Pattern Stories and Sequences for the Backlog*" [Hva2017] workshopped at PLoP 2017

3. INTRODUCING THE RUNNING EXAMPLE

The example case used throughout the patterns in this paper is based on an imaginary development effort since confidentiality issues block the authors from using any real-world example. The flow of activities is still realistic and based on our combined 50+ years of system development experience. The requirements in the example were first developed to evaluate requirements/backlog management tools.

Example: The Benson Automated Pool Cleaning System

Living in the southern part of the US, I have a pool. I also had a pool boy, a trustworthy and hardworking high school kid who started his own pool cleaning company after working as a lifeguard at the community swimming pool one summer. My pool was crystal clear, and life was good. But nothing lasts forever. High school completed, my "perfect" pool boy left for college in another city. A few months and several mediocre pool cleaning companies later I found myself in a permanent role of being my own pool boy. And I started dreaming of an automated pool cleaning system. Here is the story of building the backlog for the perfect pool cleaning system and yes, it is named after my perfect pool boy…

A first round of elicitation activities consisted of interviewing friends who maintain their own pools, a couple of professional pool cleaners, and the owner of a company that would sell and operate the automated pool cleaning system. This produced a vision statement, some high-level goals, and unstructured data on user profiles, system parts, functionality, cost models, and legal aspects. And some funny stories from my friends' successful pool cleaning activities.

***Vision**: "The Benson Pool Cleaning System keeps your pool water crystal clear and correctly balanced with no effort from the owner. Equipment and Chemicals are monitored remotely and replenished and serviced based on automated system reports."*

***Main goals**: Remove Debris, Maintain Water Quality, Remote Monitor Equipment Operation, Schedule Maintenance, Low Cost, Safe Operation.*

The example is continued in the individual patterns.

Shared Definitions

Using the same criteria to inform the team and enable decisions.

Your product is created by a large team, possibly structured into a multi-project effort (e.g. a program). There is a certain degree of freedom in how work is performed that you want to maintain, but you also want to make sure the team is acting on a consistent set of information.

How do you ensure that key information in the backlog that is used to drive internal team processes and to communicate with stakeholders is correct enough to be meaningful?

Some backlog items have attributes which values are used to drive team processes, to make decisions, and to communicate with stakeholders. If individual team members or members of sub-teams within the overall team (for instance one project within a program) are applying different values or interpreting values differently, this may cause problems that make it harder to manage the product development. The following examples, although not exhaustive, should provide an appreciation of the problem:

- Closing a user story should tell you that the functionality it represents is ready for deployment. John, one of your developers, is setting his user story to *closed* as soon as he has completed the coding and submitted it to the code repository. Sarah, another developer, sets the user story to *closed* when she has committed her code with automated tests. A third developer, Noel, never closes user stories but expects the QA team to close them when they are done with the final acceptance testing and all acceptance criteria have been verified. So, if you attempt to list the product user stories targeted for a product increment, what do you know about how complete the product increment is?
- Defect severity and the number of defects within each severity level is a measure of product quality. A large product is developed through a program with six projects. Two of the projects share a defect definition that has 4 severity levels. Another three projects use a different severity attribute with 7 levels. And the last project does not use the severity level at all but keeps the default value set by the system. So, if you review the defect statistics, what can you discern about the quality of the product?
- The outcome of running manual system regression tests is another indication of product quality. Fred, a domain expert, passes manual tests if he is reasonably happy with the system behavior. Joe, a junior tester, will only pass a test if it is 100% in accordance with specifications. Fred and Joe will pass or fail tests, they never use the *block* state. But Hannah, the test manager, uses the *block* state to indicate tests that cannot be run for some reason (missing functionality). So, if you review the test results, what do you learn about the quality of the product?

User stories, defects, and tests with test results are all part of the core product information in your backlog. For developers and other team members working with individual items the consistency of an attribute value across items is not necessarily important. But anyone working on larger sets of items or trying to get a bigger picture needs to have consistent attribute values to provide insights into the state of the product, and to be able to drive the team processes. The list of actual attributes and their values can grow long. Here are a just a few to start. Some attributes are important for assessing state – like the number of open versus closed user stories, while others are important to drive progress – like the business value of a user story. Some values are explicit – like the estimate of implementation effort, and some are embedded in an attribute – like the user role and the action found in the title of a user story or test case.

Therefore, develop and share a core set of definitions across the project or program so that the attribute values of your backlog items are consistent and can be used for decision making and reporting purposes.

Keep these definitions in a shared space that is easily accessible by all team members, like a project or program wiki. But even more important, make sure that these definitions are actively used because the team members contributed to defining them, agree with their definition, and know where to find them.

Here are some definitions that you will most likely need for a project:

User story – definition of ready (DoR) and definition of done (DoD)
Ready means what it takes for a user story to be ready for implementation: enough details in the description, clear acceptance criteria, and sign-off by product owner for example.
Done might mean when automation tests pass, or when both the automation tests and manual testing on a production (or close to production) system pass, and/or when the product owner has accepted the user story as complete.

Defect – severity level
Define the meaning of each level so it is clear versus priority to fix, and impact on system
For instance a HIGH defect level may mean must be fixed in a week, and you cannot deploy with more than 3 HIGH defects.

Test case – pass/fail/blocked
Do you pass with any remaining defects found when running this test? Or just as long as you have no HIGH defects? Do you fail for environment problems like a bad network connection? What does blocked mean – you cannot run the test because of missing functionality, or you cannot run the test because another failing test causes this test not to be applicable before a defect is fixed?

User roles and actions
Your user roles may be defined in a documented set of ROLES kept with your backlog. Another set of definitions can be very helpful in getting a consistent user interface design – ensuring that user actions have a clear terminology so that words have a specific user related meaning (for example view, monitor, or update) and that you do not get a confusing mix of synonyms to describe user actions.

If system qualities are specified in acceptance criteria, it is also important to have a consistent definition of what the quality attributes mean, the way that they are measured, and what it means for the story to be complete [YWW2015].

These shared definitions must of course be presented to new people as they join the team. And they should be revisited at intervals as the product and the team processes mature. They need to remain useful and respected by the team members so the ongoing collective ownership of these definitions is key to making them used and useful.

Example

During the implementation of a new product version of the Benson system, the test lead started noticing that defect reports did not seem to have the correct results. After investigating, she discovered that the ALM tooling they were using had two different attributes for defect severity. There was an attribute called Severity, which was used by the team developing the water quality system, while the team working on maintenance scheduling used another attribute called User Impact.

The quality reports were pulled using the User Impact attribute. The default value for this attribute was "medium" indicating a non-critical defect. After correcting the situation and having all teams use the same attribute with the same definition of the severity indicator, the reports now accurately showed the critical issues that had to be fixed before deployment. Before the correction, all critical defects from the water quality system team were in effect hidden and not contributing to the overall quality status.

Rules

Restricting some backlog changes to particular roles.

Your product backlog is mature with a lot of contents. It is shared by team members with various roles – developers, testers, architects, a product owner, subject matter experts, and managers. It supports the work of all these roles and their collaboration.

How do you protect the backlog from changes that risk adversely affecting key team processes?

The whole team owns the backlog. Everyone contributes contents and updates product backlog items that they are involved with. As work is performed, backlog items change state from *new* to *active* to *resolved* to *closed*. Discussions and additional contents are recorded. And structures are growing as new items are added. It is important that all team members take care to keep backlog contents up to date, in order to enable clear communication of the product and to enable dashboards created from extracted backlog information to reflect the reality of the project or program.

The challenges in keeping a lean, efficient and consistent backlog increases with the size and complexity of a team. With a large number of people adding and modifying contents in the backlog, there is a risk of the contents deviating and becoming poorly structured and less accurate. For instance, when adding user stories, wording may be inconsistent. This can lead to the same user role having multiple names, or the same action having various (mostly synonymous) verbs, which then causes confusion for the developers and increases the risk of poor usability.

If poorly coordinated editing of the main structure of the backlog causes it to deteriorate, it will bring issues for backlog navigation and product clarity, e.g. it will be harder to "read" the product functionality out of the backlog. As this structure breaks down, branching and unwanted additions will be made to the FRAME as people create new nodes for contents because they do not understand how their new work items fit in, thereby bloating the overall backlog structure. Extracting good ANSWERS from a poorly structured backlog is difficult and there is an increasing risk of getting inaccurate results as the backlog quality deteriorates.

Some contents of backlog items are needed to drive team workflows. Without acceptance criteria for user stories, developers will not know when an item is implemented, nor will the testers know how to verify the items in their testing efforts. Without enough information to reproduce a defect, or not knowing the version of software where the defect is found, providing a fix can be hard.

Some changes will impact the metrics used by the team to make decisions. Allowing user stories and defects to be re-opened because of a later realization of missing or faulty functionality will impact burn-down charts and measures of time to implement or resolve issues, and can create strange spikes or anomalies in graphs that need to be explained.

Therefore, create role-based rules for backlog changes and only permit specific roles to make those changes that impact the overall team.

Restrictions defined by these rules should be only for those backlog items that impact the ability of the project and/or program workflows to run efficiently, and to items and attributes that are part of the

commitment to stakeholders. Individual team members should have full control of all other items and their attributes that relate to their own work.

The following paragraphs serve as guidance for how to define these rules. One should be aware of differences in tooling and the tool terminology when trying to apply these guidelines.

Backlog structures and values that serve as the foundation for the internal team workflows are typically created and maintained by a program/project manager, product owner, and/or business analyst. For example, the product owner may be the only role that is allowed to update the acceptance criteria. Or, if the product owner works closely with the business analyst, both roles might be permitted to make these changes.

Typically, when sharing a common FRAME with CONNECTIONS, the higher level of this structure should be managed by a very limited set of roles because you do not want this to be changing without understanding the impact of the changes.

The project manager role would typically be the owner of the structural information used for planning (e.g. in VSTS, the iteration path). Architects or technical leads would own the structural composition of the product (e.g. in VSTS, the area path). Depending on how QA is incorporated into the team, this role may be the only role allowed to close a user story (e.g. owning the done decision) or approve a defect resolution, while any team member can create and fix bugs, implement a user story, and create new user stories. The product owner may own the priority or user impact attributes and also be in charge of the acceptance criteria for user stories. And the business analyst is probably able to create, modify, and delete any contents as the keeper of the backlog quality.

It is important to restrict only the items that really need to be managed carefully. If rules are overly restrictive, then team members may stop updating the backlog or do additional work that isn't tracked and managed in the backlog, thereby undermining efforts to keep an accurate accounting of the product, its requirements, and its status. It is also important that the entire team feels collective ownership of the backlog, see the relevance of all the backlog contents, and take responsibility for keeping their individual contributions to it up to date.

Two questions that the team must address are when should rules be put in place and when they might need to change. Ideally, in the early days of a project or program, it is good to establish a few simple rules. As the project or program grows and the backlog structure becomes more complex, it becomes important to revisit these simple rules to make sure that they continue to support the optimized workflow of the team. For example, in the early days of the project perhaps only the product owner adds acceptance criteria to a story. But later on, an analyst may support the product owner in defining acceptance criteria, or the testers may be permitted to add quality-related acceptance criteria to a user story.

Sometimes the team may need to define rules for what is permitted to be changed in the backlog and when. Once a user story is complete, is it permissible to add or remove acceptance criteria? If not, a rule might be that a completed user story cannot be re-opened. Another question arises about what happens if not all acceptance criteria are met and you want to include that story (even if only partially complete) in a release. In this case you may need to define a rule for when it is permitted to split a user story and what role should be responsible for doing so. Most likely the product owner would be the only role permitted to split a story. And when that happens, perhaps another rule defines that this new user story is linked to the story that was split.

The team may also have rules for when a technical story needs to be written, and by whom. For example, when a specific system quality spans multiple user stories, such as the aggregated performance for multiple business transactions, then a rule might be that the technical story should be added to the backlog that represents that overall quality requirement which spans multiple items [YWW2015].

ALM tools have functionality to support particular roles having special privileges for updating the backlog. The example from VSTS in figure 6 shows that the project administrator can set permissions for individual activities for user groups defined in the tool.

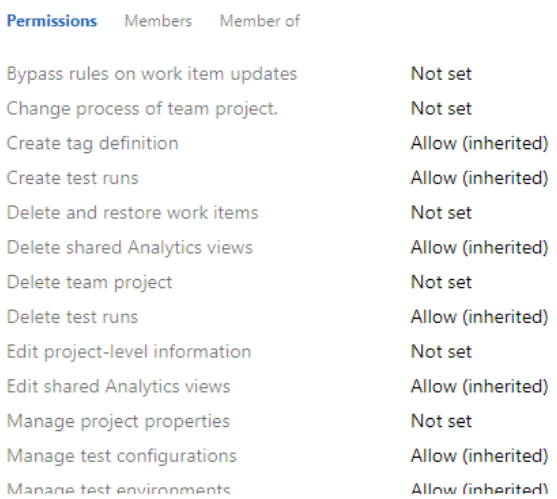

Figure 6: Permissions settings for a user group in a VSTS project

There is a strong relationship between RULES and SHARED DEFINITIONS. If you have crisp definitions that are consistently used by all team members, you can be more relaxed about RULES about who is permitted to make backlog changes. For example, if there are detailed criteria for Definition of Done for user stories, and everyone on the team knows and respects them then any team member should be able to close a user story. The optimal situation is when you need no or few rules and you have a fully aligned team. But as the team size increases, the natural progression is to gradually, and only as needed, introduce RULES that limit who can perform certain backlog actions that could affect its integrity.

Example

The team developing the Benson system is a multi-disciplinary team including mechanical, electrical, and software engineers. They have a project manager, a product owner, and a systems architect, a testing team that includes pool maintenance technicians and a couple of experienced pool builders, and an analyst. With help from the product owner and the systems architect, the analyst designed and implemented a FRAME representing the system functionality, and special VIEWS for the electrical, mechanical and software development sub-teams.

As more contents are being added to the system, the analyst spends more and more of her time correcting contents and reminding team members to stick to agreed practices. An unknown "role" keeps creeping into the new user stories – for some reason some team members have started calling the PMT (pool maintenance technician) the MO (maintenance operator). This is causing confusion for the user interface development. The analyst has also started realizing that user stories are set to complete while there are critical defects pending and tests that are not passing. Looking into this, she realizes that the developer who did the implementation closes several user stories immediately.

After bringing these problems up with the project manager, they decide to have a team meetup to discuss them. In the meeting the team decides to put in place the following RULES for their backlog management:

- *Before going to implementation the analyst reviews each user story for correctness.*
- *Only QA will set user stories to closed. To close a user story they apply a SHARED DEFINITION of "done": approved by product owner, no major defects remain open, and tests are defined and pass.*

Remodel

Refactoring the backlog structure.

The original FRAME for the backlog was created in the early days of the project, based on the product understanding at the time. Since then the project has grown, and the full scope of the product functionality is now better understood. Not surprisingly, the product functionality has also evolved and morphed from the original vision. All in all, the original backlog structure is no longer a good choice for the current understanding of the product.

How to you deal with a mature product backlog whose structure is no longer efficiently supporting the development team?

The team processes may have changed over time, and the current backlog implementation is no longer optimal to support them. Typical symptoms of a backlog in need of reworking are a large number of tags used as band aids because backlog attributes are missing or are too basic, and additional structural items being added that duplicate or conflict with existing items but still help part of the team with navigating the backlog (e.g. the core structures are insufficient so sub-teams start building their own additional structures). In many ways, the situation is similar to what can happen to the system architecture of a larger legacy system whose integrity erodes over time.

Some changes that should be reflected in the backlog may not require a restructuring, but rather a modification of a large number of existing items. For example, if user roles or other SHARED DEFINITIONS are modified, this change must be applied throughout the backlog. If ignored, extracting information out of the backlog will get complex and increase the risk of ignoring user stories in the metrics that were created with older obsolete definitions.

But a mature backlog has a lot of contents and the team will be weary of changing its contents. For a team doing continuous integration, even a few days spent on a backlog remodel can seriously interrupt the team progress. And changes may lead to information being lost and current processes breaking.

Therefore, remodel the backlog to better represent the new understanding of the product while keeping core backlog items largely unchanged.

Changing the backlog FRAME through modifying the way items are linked to each other still fully preserves the definition of each backlog item. So an update to better represent the system functionality is most likely an exercise in creating new/updated top-level items while keeping the contents of users stories untouched and linking them to this new structure. Even for a good size backlog this effort can be done in a day with proper upfront planning.

If new attributes are needed, most likely you do not need to update existing items to have this new attribute if the items are already completed/no longer represent new work, but are being kept only for historical and regression test purposes.

Changing user roles and terminology for user functionality will mean changing text in a number of backlog items. This may be a lot of work but it is not really a high risk, and should not disrupt development.

Some tools have hierarchical attributes, like the area path and iteration path in VSTS. Changing these must be handled as part of structural changes. The iteration path is of course modified regularly as new sprints are being added, but the overall structure may need modifications if the team makes process changes. The area part is normally used to represent system or functional components, and as a product grows it is natural to have to update this structure as well.

If the backlog remodel involves several types of changes, we recommend to start by changing the FRAME, then updating the CONNECTIONS, followed by making any other structural changes, before updating single backlog item types by adding attributes to them. There should rarely be a reason to modify the content type definitions of the user story, the test case, or the defect items types, except for adding a new attribute or modifying the linking model. These items are typically the ones with the richest content and thereby also contain most of the development history.

A REMODEL is not necessarily a discrete event, but may be performed by ALM administrators (possibly the business analyst and product owner) who frequently adjust and improve, and ensure newly added contents conform to the ALM strategy and the RULES laid out by the team.

Example

The Benson system has been on the market for three years, and it has been a success with good sales and a growing market share. The product owner is now considering a major upgrade of the cleaner robots to add a model that can empty baskets when the ground is uneven. This requires several changes to the overall system for monitoring of the operation.

When trying to add this new functionality it becomes clear to the product owner and the analyst that the current structure is no longer reflecting the product as it has evolved, and they also want to do some changes to the backlog structure to better incorporate new functionality. Their changes are primarily modifying the way the backlog items are linked together, so the new split into pool cleaning and filter cleaning and restructuring the FRAME with CONNECTIONS has a minimal effect on the development teams.

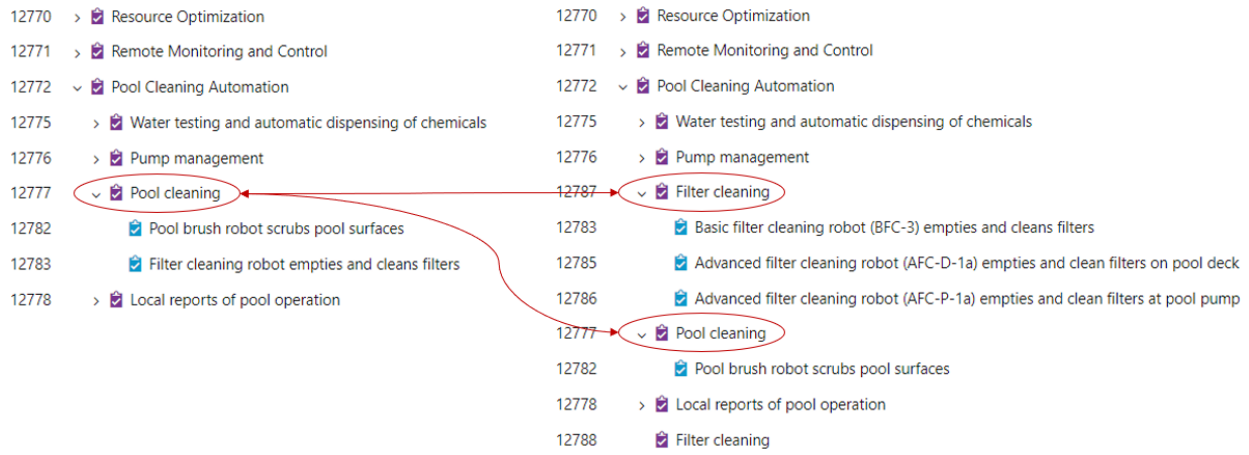

Figure 7: Remodeling the FRAME by splitting pool cleaning into two requirement groups

4. COMMENTS

As stated in the introduction, this paper is part of a larger work on Product Backlog creation and management [Hva2014, Hva2015, Hva2017, Wir2016]. Our efforts started with a desire to help software engineering teams create better backlogs, and it was a result of working with several teams that struggled with their planning and backlog management because their backlogs were not created with an understanding of good backlog practices. For large systems and large teams, the backlog cannot be an unstructured to-do list, but needs to be designed and implemented to support the team's development process. Common mistakes that we have seen are mixing product tasks and project tasks thus making it hard to report progress on the product, backlogs created by architects where the backlog structure looks like a modular breakdown of the system but more or less hides the product functionality, and confusion on the granularity so user stories are anything from epics that take months to deliver to small tasks that are not user stories at all – all mixed up in the same backlog.

We now think it is time to collate our work over the last four years into a structured collection – possibly as a book. In doing so, we will focus on more examples and try to better cover the "known uses" through stories from a broader audience. We are also thinking about ways to better cover the usage of ALM tools and possibly do case studies using a few of the most common ALM tools on the market.

5. ACKNOWLEDGEMENTS

Many thanks to our shepherd Stefan Sobernig who provided us with valuable comments that really helped us improve the paper, and even more valuable comments that will aid us in future work on this material. We would also like to thank our PLoP 2018 workshop participants for their feedback. To gain a better understanding of software requirements and the processes around requirements engineering we have consumed a lot of literature, and we especially appreciate Karl Wiegers' writings on Software Requirements, Jeff Patton's work on Story Mapping [Pat2014], the Scrum Guide by Ken Schwaber and Jeff Sutherland [SS2013], Ellen Gottesdiener's and Mary Gorman's workshops and books [Got2002 Got2005, GG2012], and Johanna Rothman's writing on program management.

REFERENCES

[AB2006] ALEXANDER, I. and BEUS-DUKIC, L. 2006. *Discovering Requirements: How to Specify Products and Services*. Wiley (ISBN: 978-0-470-71240-5).

[Agi2015] AGILE ALLIANCE 2015. *Agile Alliance Glossary: Backlog*. https://www.agilealliance.org/glossary/backlog/ (visited on January 3 2019)

[Amb2014] AMBLER, S. 2014. http://www.agilemodeling.com/artifacts/userStory.htm (visited on January 3 2019)

[BC2012] BEATTY, J and CHEN, A. 2012. *Visual Models for Software Requirements*. Microsoft Press (ISBN 978-0-7356-6772-3).

[BKW2018] BITNER, K, KONG, P., and WEST, D. 2018. *The Nexus Framework for Scaling Scrum: Continuously Delivering an Integrated Product with Multiple Scrum Teams***.** Addison-Wesley (ISBN 978-0-13-468266-2).

[Got2002] GOTTESDIENER, E. 2002. *Requirements by Collaboration*. Addison-Wesley (ISBN 0-201-78606-0).

[Got2005] GOTTESDIENER, E. 2005. *The Software Requirements Memory Jogger*. GOAL/QPC (ISBN 978-1-57681-060-6).

[GG2012] GOTTESDIENER, E. and GORMAN, M. 2012. *Discover to Deliver: Agile Product Planning and Analysis*. EBG Consulting (ISBN 978-0985787905).

[HH2008] HOSSENLOPP, R. and HASS, K. 2008. *Unearthing Business Requirements: Elicitation Tools and Techniques*. In Management Concepts (ISBN 978-1-56726-210-0).

[Hva2014] HVATUM, L. 2014. *Requirements Elicitation using Business Process Modeling*. 21st Conference on Pattern Languages of Programming (PLoP), PLoP 2014, September 14-17 2014, 9 pages.

[Hva2015] HVATUM, L. and WIRFS-BROCK, R. 2015. *Patterns to Build the Magic Backlog*. 20th European Conference on Pattern Languages of Programming (EuroPLoP), EuroPLoP 2015, July 8-12 2015, 36 pages.

[Hva2017] HVATUM, L. and WIRFS-BROCK, R. 2017. *Pattern Stories and Sequences for the Backlog: Expanding the Magic Backlog Patterns.*. 24th Conference on Pattern Languages of Programming (PLoP). PLoP 2017, October 23-25 2017, 26 pages.

[Hva2018] HVATUM, L. and WIRFS-BROCK, R. 2018. *Program Backlog Patterns: Applying the Magic Backlog Patterns.* 23rd European Conference on Pattern Languages of Programming (EuroPLoP). EuroPLoP 2018, July 4-8 2018, 22 pages

[KBN2015] *What is Kanban?* http://kanbanblog.com/explained/ (visited on January 3 2019)

[KS2009] KANNENBERG, A. and SAIEDIAN, H. 2009. *Why Software Requirements Traceability Remains a Challenge*. CrossTalk: The Journal of Defense Software Engineering. July/August 2009.

[KI2012] KNIBERG, H, and IVARSON, A. 2012 *Scaling Agile @ Spotify* https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf (visited on January 3 2019)

[Mas2010] MASTERS, M. 2010. *An Overview of Requirements Elicitation*. http://www.modernanalyst.com/Resources/Articles/ /115/articleType/ArticleView/articleId/1427/An-Overview-of-Requirements-Elicitation.aspx (visited on January 3 2019)

[Mul2016], MULDOON, N. 2016. Backlog grooming for Kanban teams in JIRA Agile. http://www.nicholasmuldoon.com/2016/02/backlog-grooming-for-kanban-teams-in-jira-agile/ (visited on January 3 2019)

[Pat2014] PATTON, B. 2014. *User Story Mapping*. O'Reilly (ISBN 978-1-491-90490-9).

[Rad2016] RADIGAN, D. 2016. The Product Backlog: *Your Ultimate To-Do List*. https://www.atlassian.com/agile/backlogs (visited on January 3 2019)

[Rin2009] RINZLER, J. 2009. *Telling Stories*. Wiley (ISBN 978-0-470-43700-1).

[RR2006] ROBERTSON, S. and ROBERTSON J. 2006. *Mastering the Requirements Process*. Addison-Wesley (ISBN 0-321-41949-9).

[Rot2016] ROTHMAN, J. 2016. *Agile and Lean Program Management: Scaling Collaboration Across the Organization*. Practical Ink (ISBN 978-1-943487-07-3).

[RW2013] ROZANSKI, N. and WOODS, E. 2013. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* (2nd Edition). Addison-Wesley (ISBN 978-0321718334).

[Sch2015] SCHOLASTIC, 2015. *The Magic School Bus*. https://www.scholastic.com/magicschoolbus/books/index.htm (visited on January 3 2019)

[SS2013] SCHWABER, K. and SUTHERLAND, J. 2013. *The Scrum Guide*. http://www.scrumguides.org/ (visited on January 3 2019)

[Sut2014] SUTCLIFFE, A. G. (2014): "Requirements Engineering" in Soegaard, Mads and Dam, Rikke Friis (eds.), *The Encyclopedia of Human-Computer Interaction*, 2nd Ed.," Aarhus, Denmark: The Interaction Design Foundation. Available online at https://www.interaction-design.org/encyclopedia/requirements_engineering.html (visited on January 3 2019)

[Wie2009] WIEGERS, K. 2009. *Software Requirements* 2nd Edition. Microsoft Press (ISBN: 0-7356-3708-3).

[Wie2006] WIEGERS, K. 2006. *More about Software Requirements*. Microsoft Press (ISBN: 0-7356-2267-1).

2016. *Kanban Board.* https://en.wikipedia.org/wiki/Kanban_board (visited on January 3 2019)

[Wir2016] WIRFS-BROCK, R. and HVATUM, L. 2016. *More Patterns for the Magic Backlog*. 23rd Conference on Pattern Languages of Programming (PLoP), PLoP 2016, Oct 24-26 2016, 18 pages.

[Wit2007] WITHALL, S. 2007. *Software Requirement Patterns*. Microsoft Press (ISBN: 978-0-735-62398-9).

[YWW2015] YODER, J.W, WIRFS-BROCK, R. and WASHIZAKI, H. *QA to AQ Part Four Shifting from Quality Assurance to Agile Quality "Prioritizing Qualities and Making them Visible"*. 22nd Conference on Pattern Languages of Programming (PLoP), PLoP 2015, October 24-26 2015, 14 pages.