Pay to Play or Requirements Prioritization in Collectives

Michael Weiss
Systems and Computer Engineering
Carleton University
Ottawa, Canada
Email: weiss@sce.carleton.ca

***Abstract*—This paper presents two patterns for requirements prioritization in a collective. A collective is a group of stakeholders with a common need. The stakeholders join the collective to create an infrastructure that they can leverage to develop their own products more effectively. This new organizational model differs from the traditional value chain, and changes the way requirements are identified and prioritized.**

***Keywords*-requirements; prioritization; pattern; collective**

I. Introduction

Requirements define what a system "needs to do, but not how" [21]. A requirement is a single objective that a system must satisfy [21]. It must be possible to measure whether a requirement has been met. A requirements specification documents all the requirements (functions and capabilities) for a system, and also includes any background information necessary to understand them [12], [21].

Requirements prioritization refers to the process of selecting the requirements to implement in a system [13]. It is usually impossible to accommodate all desired functions and capabilities. Projects need to make compromises to ensure that the most critical and most timely requirements are implemented [20]. Typically, the requirements selected will be the ones that maximize customer value [19].

A collective is a group of stakeholders with a common need [2]. The stakeholders join the collective to create an infrastructure that they can leverage to develop their own products more effectively. This new organizational model differs from the traditional value chain. In a value chain, products are developed in response to customer or perceived market needs [12]. In a collective, products are jointly created between the stakeholders. Stakeholders share the cost and risk of developing these products, but also the profits.

Existing research on requirements engineering has mainly focused on the value chain [12]. The goal of this paper is to examine how requirements are prioritized in a collective of stakeholders. In this case, the customers of the system are the members of the collective themselves. Insights on this question were obtained through a combination of action research with an analysis of examples from the literature. In action research, the research is done by or in collaboration with practitioners [9]. The author is an active member of a collective to create a collaboration framework (TFN 200). The findings are documented in the form of patterns.

The paper is organized as follows. Section II provides background on requirements engineering, collectives, and patterns. Section IV describes the patterns for requirements prioritization in collectives. Section V concludes the paper.

II. Background

A. Types of requirements

Wiegers [19] identifies three levels of requirements: busines requirements, which represent the business value created by a system; user requirements, which describe the capabilities to be provided to users; and detailed functional and non-functional requirements. A recent review [12] distinguishes three types of requirements to be managed: stakeholder, system, and design requirements. Stakeholder requirements can be defined in a solution-independent manner, whereas the latter two are specific to a solution.

B. Types of organizations defining requirements

Hull et al. [12] identify three types of organizations in which the requirements are collected:

1) Acquisition organizations that request systems. They create and manage stakeholder requirements.
2) Supplier organizations that respond to requests from acquisition organizations or other supplier organizations. They receive input requirements and develop system requirements and designs in return.
3) Product companies that develop and sell products. They collect stakeholder requirements from the market instead. They develop products in response to perceived stakeholder requirements. They are, in a sense, both acquisition and supplier organizations.

Collectives appear to be a fourth type of organization. They combine the roles of product organizations with those of acquisition and supplier organizations. The members of a collective are all stakeholders in the same system. They collaborate to establish common requirements, while also individually interacting with their own customers.

C. Collectives

A collective can achieve things that its individual members cannot achieve on their own [2]. For example, as a collective, a group of startups can deliver a complete solution to a customer, whereas individually they are only

able to deliver pieces of the solution, which the customer has to integrate. Joining forces makes the group of startups much more competitive against large system integrators. Collectives can also collaborate to address common needs, allowing their members to focus on features of their products that differentiate them. The more members a collective has, the more its members are able to share the load of meeting common needs. However, such collaboration is also fraught with problems, for example, the coordination overhead that results from dependencies between subtasks [17].

A key characteristic of collectives is that they are voluntary organizations. Membership in a collective is a function of how well the collective helps its members meet their business goals. As contributors to the collective, members gain access to the total value generated by the collective. As long as the total value is higher than the cost of contribution, previous research [3] has shown that members benefit from joining. Conversely, existing members of a collective are not interested in members who do not add value to the collective. Thus, collectives often impose conditions on membership such as asking members to commit resources.

D. Patterns

A pattern describes a recurring problem that occurs in a specific context and its solution [1]. Each pattern describes the situation when the pattern can be applied in its context. The context can be thought of as a precondition for the pattern. This precondition is further refined in the problem description with its elaboration of the forces (trade-offs) involved. The solution describes a way of resolving the forces. Some forces may not be resolved by a single pattern. In this case, a pattern often includes references to other patterns, which help resolve forces that are left unresolved by the current pattern. Together, patterns connected in this way are often referred to as a pattern language [1].

Two common pattern representations are the Coplien form, and the Alexandrian form. The Coplien form [5] includes explicit sections for forces and consequences, in which the forces and the implications of using the patterns are presented in bullet form. This provides quick access to the reasons for applying a pattern. The Alexandrian form [1] resolves the trade-off between the needs to have structure on the one hand, and the desire to create more easily readable pieces of literature, on the other. In practical use for documenting software designs, the Alexandrian form has been adapted to include the concept of explicit lists of forces and consequences from the Coplien form.

III. Case studies

This section provides an overview of the case studies. More details on the case studies are available in [17].

A. Eclipse

Eclipse is an open source community focused on building an open software development platform [14]. The Eclipse project was founded in 2001 as a spin-out of technology that IBM had acquired from Object Technology International. We use the term "spin-out" to refer to a case where a company externalizes an internal development project [18]. Initially, the Eclipse community was primarily driven by IBM and other software vendors. With the creation in 2004 of an independent, non-profit governance body, the Eclipse Foundation, IBM relinquished its control over the project and allowed other players, including IBM's competitors, to become equal members of the community [15].

Eclipse has a well-defined process for how members can engage with the collective [8]. Three councils, responsible for requirements, planning and architecture, guide the projects. The requirements council collects, reviews, and prioritizes incoming requirements. The planning council manages the release train. The architecture council defines and evolves the architecture of the Eclipse platform. Individual projects are overseen by project management committees. The councils are comprised of strategic members and representatives of the project management committees.

B. TFN 200

The TFN 200 is a next generation collaboration system released under an open source license. It is designed by a collective that comprises academics and students, several startups, an economic development agency, and early customers. Its goal is to provide a shared platform that can be extended by members to develop their own products in a shorter time frame and to deliver a more comprehensive solution than they could provide on their own. Achieving alignment with the business goals of the members was an important design goal of the TFN 200. The collective was launched by Carleton University in March 2011.

The governance structure of the TFN 200 involves three councils responsibile for architecture, opportunity development, and infrastructure. The architecture council guides the development and evolution of the platform. The opportunity council identifies opportunities – products or services that can be derived from the platform – and extracts and priorities requirements for the shared platform from the opportunities. Opportunities need to be backed up by commitments; they only get to move forward, if there are members of the collective willing to fund or assign resources to them. The infrastructure council provides a shared testing and development platform that is available to all members.

IV. Patterns

The format for describing the patterns for requirements prioritization in collectives follows the Coplien form.

A. Pay to Play

1) Context: A group of stakeholders with a common need jointly create an infrastructure, which they then leverage to develop their own products more effectively.

2) Example: Eclipse is an infrastructure for the development of applications [8]. Unlike other development environments, Eclipse is jointly developed by a group of companies and individuals with a common interest. Various subprojects of Eclipse support different vertical domains.

3) Problem: **How does the group of stakeholders decide on which requirements to implement?**

4) Forces: The solution needs to balance these forces:

- Stakeholders, if given the option, would like to see all their requirements implemented.
- There are always more requirements than time or resources to implement them.
- Stakeholders have limited resources.
- Stakeholders have different skills. Some have domain experience. Others are good at attracting customers.
- Some stakeholders may have more urgent needs than others, which make them care more about them.
- Stakeholders are prepared to share the results.

5) Solution: **Only accept requirements that stakeholders are willing to pay for.** Start by identifying opportunities for new products to be built on top of the common infrastructure. Then, ask the stakeholders to commit money or developers to those opportunities. Only opportunities that are backed up by commitments should be selected. Note that this pattern is not about requirements the stakeholders' (external) customers are willing to pay for, but commitments that the stakeholders themselves are prepared to make.

Stakeholders can "pay" for requirements in different ways. They may contribute to the implementation, or offer to pay someone to do so. They can also provide unique domain knowledge, or experience with other systems. Finally, stakeholders can provide access to paying customers.

Some requirements are not visible to users, and it may be difficult to link them directly to stakeholder needs. However, these requirements need to be met as other visible requirements depend on them. You need to budget for these invisible requirements as described in *Wallflowers*.

Some members of the collective may be in a better position to implement a specific requirement, because the skills required are not generally available, or they may have a more urgent need than other members for a specific requirement to be in place. One advantages of driving the implementation of a requirement is that the infrastructure will be better aligned with the member's needs.

6) Consequences: Applying this pattern ensures:

- System will implement the most valued requirements.
- Scope of the requirements implemented matches the amount of resources available.
- Implemented features are available to all stakeholders.
- Stakeholders' perspectives complement each other.
- Stakeholders who invest more time and resources in implementing the requirements will benefit more.

7) Known uses: The Eclipse project only allows strategic members of the collective to influence the direction of the project. To become a strategic member, a company has to pay a membership fee and commit resources to the development of the common infrastructure.

In the TFN 200 project, an opportunity council identifies products or services that can be derived from the infrastructure, and extracts and priorities requirements from these opportunities. To be moved forward, opportunities need to be backed up by funding or resource commitments.

8) Related patterns: Requirements are driven by business needs. These break down into user and functional/non-functional requirements. It is best to think of these as a *Requirements Pyramid* [10]. *Buy a Feature* [11] has been proposed as a technique for getting customers to prioritize requirements. *Planning Poker* [4] in agile methods serves a similar purpose. In both cases, a virtual currency, not actual resources, is used to assign priorities.

9) Source: This pattern is based on the author's observations about the case studies. The name refers to a term used in the investment field that means that you cannot achieve a high return on your investment without taking risks.

B. Wallflowers

1) Context: Not all requirements are visible to stakeholders. *Pay to Play* only deals with visible requirements.

2) Example: Many non-functional requirements are not visible. For example, login into a web conferencing system needs to be secure, but this is not a very visible feature.

3) Problem: **How do you ensure that all essential requirements are implemented?**

4) Forces: The solution needs to balance these forces:

- Stakeholders only want to pay for visible requirements that are directly meaningful to them.
- Visible requirements often depend on other, invisible requirements that need to be met as well.
- Some requirements are more interesting to implement than others, which may never get done.

5) Solution: **Build the cost for invisible requirements into the cost of visible requirements.** Increase the cost for visible requirements to include invisible requirements (a.k.a. "wallflowers") that need to be implemented as well. These requirements may be as critical to the operation of the system as visible ones, but they remain hidden to most stakeholders, and are not recognized as important.

As this may be difficult to achieve on a per-requirement basis (for example, it may not be possible to accurately estimate the dependencies on invisible requirements without a detailed design), you can impose a "tax" (sometimes also referred to as project overhead) on visible requirements. The tax can be a portion of the cost for implementing the visible requirement. Stakeholders then share the cost of implementing the invisible requirements.

Wallflowers can be identified based on architectural knowledge. This is the distinction made in [6] between what the system *is* and what the system *does*. What the system *is* includes all the visible system features. Stakeholders are well aware of those requirements. What the system *does* may only be visible to those with experience in the domain.

6) Consequences: Applying this pattern ensures:

- Invisible requirements (a.k.a. "wallflowers") are budgeted as part of visible requirements.
- Dependencies between requirements are articulated and can be better accounted for in the future.
- All necessary requirements get implemented.

7) Known uses: The Eclipse platform contains core projects related to the runtime. These components are not visible to subprojects focus on particular vertical domains. In terms of *Wallflowers*, the runtime projects implement invisible requirements. Eclipse members need to pay a membership fee as well as commit developers to the project. The membership fee can be interpreted as a kind of tax; it is not assigned to any project in particular.

In the TFN 200, the architecture council ensures that the implications of visible requirements on the architecture are understood, and that all invisible requirements that need to be implemented are uncovered. A potential weakness of the TFN 200 project is that, at this early stage, it does not yet charge a membership fee. Thus the implementation of invisible features relies on creating a common understanding among the members of the collective.

8) Related patterns: None.

9) Source: This pattern is based on the author's observations about the case studies. The name refers to a stock that has fallen out of favor and is trading at a low price. The need to include requirements dependencies in requirements prioritization has also been identified in [7].

V. Conclusion

The process by which collectives of stakeholders prioritize requirements has not received sufficient attention in the literature. Based on insights the author developed through the active involvement with an open source collective (TFN 200) and from examples in the literature, this paper identified two patterns for requirements priorization in collectives. These pattern are only the beginning of a pattern language for requirements prioritization that awaits to be written.

Acknowledgment

I thank my colleague, Tony Bailetti, for suggesting the name for the Pay to Play pattern. He does not know that he did, but he introduced this expression to me.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language: Towns, Building, Construction*, Oxford University Press, 1979.

[2] T. Bailetti, Keystone Off-The-Shelf, *Open Source Business Resource*, September, 9-15, 2010, www.osbr.ca.

[3] C. Baldwin, and K. Clark, Architecture of participation: Does code architecture mitigate free riding in the open source development model?, *Management Science*, 52(7), 1116-1127, 2006.

[4] M. Cohn, *Agile Estimating and Planning*, Prentice Hall, 2005.

[5] J. Coplien, *Software Patterns*, SIGS, 1996.

[6] J. Coplien, and G. Bjornvig, *Lean Architecture for Agile Software Development*, Wiley, 2010.

[7] M. Daneva, and A. Herrmann, Requirements prioritization based on benefit and cost prediction: A method classification framework, *Euromicro Conference on Software Engineering and Advanced Applications*, 240-247, 2008.

[8] Eclipse Foundation, About the Eclipse Foundation, http://www.eclipse.org/org (last accessed in May 2011).

[9] K. Herr, and G. Anderson. *The Action Research Dissertation*. Sage, 2005.

[10] A. Hoffmann, Requirements pyramid, *European Conference on Pattern Languages of Programs*, 2010.

[11] L. Hohmann, *Innovation Games: Creating Breakthrough Products through Collaborative Play*, Addison Wesley, 2007.

[12] E. Hull, K. Jackson and J. Dick, *Requirements Engineering*, Springer, 2011.

[13] Z. Racheva, M. Daneva, K. Sikkel, R. Wierenga, and A. Herrmann, Do we know enough about requirements prioritization in agile projects: Insights from a case study, *Requirements Engineering Conference*, 147-156, 2010.

[14] D. Smith, and M. Milinkovich. Eclipse: A premier open source community. *Open Source Business Resource*, July, 2007, www.osbr.ca.

[15] S. Spaeth, M. Stuermer, and G. v. Krogh. Enabling knowledge creation through outsiders: towards a push model of open innovation. *International Journal of Technology Management*, 52(3/4), 411-431, 2010.

[16] M. Weiss, Performance of open source projects, *European Conference on Pattern Languages of Programs*, CEUR, 566, 2009, http://ceur-ws.org/Vol-566.

[17] M. Weiss, Economics of collectives, *International Workshop on Quantitative Methods in Software Product Line Engineering* at the *Software Product Line Conference*, 2011 (to appear).

[18] J. West, and S. Gallagher. Challenges of open innovation: the paradox of firm investment in open-source software. *R&D Management*, 36(3), 319-331, 2006.

[19] K. Wiegers, Karl Wiegers describes 10 requirements traps to avoid, *Software Testing & Quality Engineering*, 2(1), 2000.

[20] K. Wiegers, *More About Software Requirements: Thorny Issues and Practical Advice*, Microsoft Press, 2005.

[21] S. Withall, *Software Requirements Patterns*, Microsoft Press, 2007.