Ten Misconceptions about Product Software Release Management explained using Update Cost/Value Functions

Slinger Jansen, Sjaak Brinkkemper
Information and Computing Sciences Institute
Utrecht University
Utrecht, the Netherlands
{s.jansen, s.brinkkemper}@cs.uu.nl

Abstract

The decision for a young product software vendor to release a version of their product is dependent on different factors, such as development decisions (it feels right), sales decisions (the market needs it), and quality decisions (the product is stable). Customers of these products, however, are much more cost oriented when deciding whether to update their product or not, and will look mainly at the cost and value of an update. Product software vendors would gain tremendously if their release package planning method was supported by a similar cost/value overview. This paper presents cost/value functions for product software vendors to support their release package planning method. These cost/value functions are supported by ten misconceptions encountered in seven case studies of product software vendors that these vendors had to adjust during their lifetime. Finally, a number of cost saving opportunities are presented to enable quicker adoption of a release and thus shorten release times and customer feedback cycles.

1 Introduction

Product software release planning has been characterised as a "wicked" [4] and "complex" [1] problem for which no perfect solution exists. One part of release planning, release package planning, is often underestimated due to its seemingly innocent and uncomplex nature. Product software vendors that do not have much experience in release planning often publish their release packages because a team of experts within the organization deems the release good-enough, which results into some releases that are hardly adopted by customers, whereas others are much more popular.

Simultaneously, release packages are created often during the lifecycle of a product, which suggests that processes such as release package creation, release package publication, informing the customer of a new release, and updating are repetitious processes that must be automated as much as possible, to ease both customer update effort and vendor release package creation effort. Decreasing this effort results into customers that are more willing to update, and vendors who are more willing to release regularly, as suggested by the agile development methods, such as extreme programming [2]. However, from a number of case studies performed in the past it is found that product software vendors generally do not sufficiently plan their releases [10].

We define *Software product release management* as the storage, publication, identification, and packaging of the elements of a product. *Release package planning*, which is part of the release planning process, is the process of defining what features and bug fixes are included in a release package and the process of identifying these packages as bug fix, minor, or major updates, taking into account releases that have been published in the past and the possible update process required to go from one release of the product to another release. To illustrate, figure 1 displays a release snapshot from a recent case study [7], in which major, minor, feature, and bug fix releases are shown.

An *update package* is a package that promotes a customers configuration to a newer configuration. Secondly, a *bug fix update package* contains only bug fixes, a *feature update package* contains only new features, and *minor* and *major update packages* contain both bug fixes and new features. The distinction between minor and major update packages is usually that major update packages change structural parts of a product, such as the architecture or the data model. Our view of software evolution described here is similar to Rajlich and Bennet's staged model [3], which addresses evolutionary changes (minor and major update packages) first, and then continues to see patches (bug fix packages) until a release is phased out and closed down.

The objective of this paper is to create release package planning awareness within software product management research. This is achieved by the presentation of cost/value functions that support misconceptions found in seven real-life cases about software product release management. Two

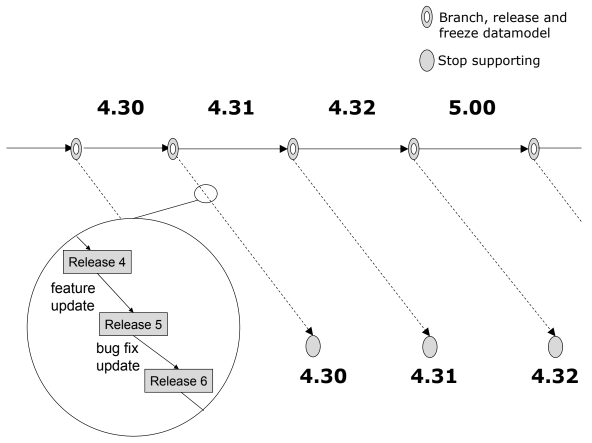

Figure 1. Typical Versioning Example

complementary cost/value functions are presented that enable a product software vendor to estimate whether a release package will actually be downloaded and installed by its customers. Also, two complementary Cost/Value functions are presented to help a software vendor decide whether the next release package will be marked a bug fix, minor, or major release.

The presented Cost/Value functions provide an extra check before publishing a release package for product software vendors. In that, the presented decision method is a useful extension to product road mapping methods, such as the one presented for small product software businesses [15], the method that supports the product software knowledge infrastructure [17], and other methods that support release planning [14]. Section 2 presents and describes the Cost/Value functions. Section 3 describes ten misconceptions encountered in seven case studies that support the Cost/Value functions as a valid release package planning method. In section 4 methods are described to save either customer update costs or vendor side release costs. Finally, in section 5 we discuss the presented method and describe the conclusions.

2 Defining the Cost/Value functions

This section describes the Cost/Value functions for both the customer (see figure 2) when updating its software and the vendor (see figure 3) when releasing a new version. These functions separately describe the cost of an update for a customer, the value of an update for a customer, the value of a new release for a vendor, and the cost to create the release package for the vendor. The functions are based on case studies performed at seven organisations [10]. Also, the customer functions are based on different papers from Enterprise Resource Planning (ERP) application updates and migrations [13], and a recent case study we performed at a content management systems vendor that also does updates and migrations for customers [8]. The Cost/Value functions are similar to the profit functions developed by the Research Triangle Institute [16]. However, these profit functions are used to calculate the impact of software testing inadequacies to the software business and not specifically for update release timing.

2.1 Customer Functions

A customer will base its decision to update a software product on a number of factors. First and foremost, the customer is interested in the value the update represents for her. This value can be of many different forms, such as the addition of a new level to a game providing the customer with more entertainment or a complete new production planning module to an ERP package saving the customer many millions. Simultaneously the customer will take the cost of updating into account. Such cost can be the downright effort of downloading and installing the new level for the game or downtime of the ERP system during the update costing the company many millions.

A customer's value of an update is defined as *Cval* (1). The function defines the value of an update to a customer as the value of new features the customer will use plus the value of the removal of previous workarounds. The new features include those features that have been added to the new release, but also those that simply did not work in the previous release and for which a workaround was not available.

Before a customer decides to update, however, the customer will calculate the cost of an update to see if it is really worth it. The cost function *Ccost* (2) defines the cost of an update as the cost of downtime of a product, the cost of training for the people using the new/changed functionality, the cost of effort put into the update process, the cost of functionality that was removed from the release or the cost of customisations that can no longer be used after the update, and finally the cost of the payments to the vendor for the update.

For a customer to make the update decision, *Cval* must exceed *Ccost* (3), especially when taking into account that the resources that are required to perform the update normally perform other value adding tasks. It is quite surprising to see that many vendors do not invest structurally into reducing these costs for the customer, especially since in most of our case studies up to 70 percent of revenue was coming from existing service contracts and only 30 percent from new customers. Also, when a vendor sees that *Ccost* exceeds *Cval* for a large number of customers, releasing an

$$Cval(update) = value(newFeatures) + value(removalOfWorkarounds) \tag{1}$$

$$Ccost(update) = \begin{cases} cost(downtime) & + \quad cost(training) & + \quad cost(updateEffort) \quad + \\ cost(lostFunctionality) & + \quad cost(paymentToVendor) \end{cases} \tag{2}$$

$$Cval(update) > Ccost(update) \tag{3}$$

Figure 2. Customer Cost/Value Functions

$$Vval(newUpdatePackage) = \begin{cases} newCustomers & * \quad priceNewRelease & + \\ oldCustomers & * \quad priceOfUpdate & + \quad costReduction(support) \end{cases} \tag{4}$$

$$Vcost(newUpdatePackage) = \begin{cases} cost(development) & + \quad cost(updateCurrentCustomers) & + \\ cost(increasedSupport) & + \quad cost(marketing) & + \\ cost(deliveryToCustomers) & + \quad cost(packageCreation) \end{cases} \tag{5}$$

$$Vval(update) > Vcost(update) \tag{6}$$

Figure 3. Vendor Cost/Value Functions

update becomes essentially useless, unless the vendor hopes to attract a large number of new customers. This seems improbable though, since if current customers are not interested in the product, why would new ones be?

2.2 Vendor Functions

For a vendor the value of an update is much harder to calculate, especially because it involves estimating how many new customers are attracted with the new release and how many customers are actually prepared to update. A vendor's value of a new release are new customers attracted by the release that specifically targets a new market, the reduction in support calls due to a bug fix to a commercial operating system, or a customer that pays the vendor for an update of their ERP product.

The function for a vendor's value *Vval* (4) describes the value of a new release as the number of new customers times the price of a new release, the current customers who are prepared to update against reduced cost, and finally the cost reduction in support calls due to the fixes in the new release package. Calculating the *Vval* is hardest, mostly because it involves estimating the number of new customers and estimating how many customers are willing to update from any previous version, which might introduce different prices for the different updates. If the release package contains a large number of bug fixes there might be a cost reduction in support costs. However, if many new features have been introduced, this reduction might be cancelled out by the cost increase in support.

The *Vcost* (5) function is defined as the cost of development of the functionality and bug fixes for the new release package, the cost of updating the current customers, the cost of increased support questions relating to the new release, the cost of marketing, the cost of delivering the new release package to customers, and finally the cost of packaging the release. The cost of updating current customers includes such things as update tools [11], renewing their licenses, and possible support questions that arise during the update process. The cost of marketing includes informing current customers of the new release, the marketing campaign, creating release notes, and maintaining the product's website. The cost of delivering the update to customers encompasses the creation of the delivery medium (CD, DVD, floppy, USB-stick, website, etc.), the assembling of all artifacts, the possible translations of the products language files, and completeness checking of the release.

The *Vcost/Vval* functions are used to evaluate whether it is time to create a release package. This is generally the case when *Vcost* exceeds *Vval* (6), i.e., when the potential value of releasing an update is higher than the cost that was required to create the update. Automation of the processes that make up release package creation and publication can potentially reduce *Vcost*, enabling a software vendor to release more often. This is similar to condition (3), where automation of the delivery and deployment processes can decrease *Ccost*, thus making it more attractive for customers to update.

The functions shown in this section tend to change largely when looking at either a bug fix, a minor, or a major release package. In the case of a bug fix package that is released on-line, contrary to a major release, no new storage media need to be created by a vendor. The decision to release either a bug fix, minor, or major release package can be made using these functions. If the reduction in support costs justifies the effort put into fixing a number of bugs, a bug fix release is justified. If the reduction in support costs does not justify the effort put into fixing a number of bugs and the addition of functionality, you might want to earn it back by making the next release a minor release. If the vendor feels that the next release should generate more revenue from new customers and old customers as well, this might be a justifiable case for a major release package. Of course this is not a hard science. Especially in the case a bug cost a disproportionate amount of time to fix, it might not be justifiable to publish a minor release package. A question the vendor must ask itself then is whether it was worth it to try and fix the bug in the first place.

3 Ten Misconceptions about Product Software Releasing

All product software vendors undergo series of paradigm shifts during their lifetime leading to radical changes in earlier established principles [6]. These misconceptions are generally strategic misconceptions that beginning software vendors can easily have about product software management and release management specifically. Here ten misconceptions are presented that were encountered in seven case studies of product software vendors. These product software vendors have been the subject of study from 2004 until 2006, and include Dutch software organizations with between 60 and 1500 employees [7] [8]. The main focus of research were the vendors' release, delivery, and deployment processes. For a further description on how the case studies were undertaken we refer to the case study reports and a paper describing all seven cases [10]. The value/cost formulas presented in this paper support the lessons learnt presented here.

1. Customers want to stay up-to-date - It is important to realize that a customer of a software product uses it only to make life better. If a newer release package does not provide the customer with new functions, why would she update? When, for instance, was the last time you updated a computer game? Or your ftp client? To quote one of the case study participant's customers "Their software supports our business process perfectly. Some of the workarounds are strange, but as long as we don't have to invest in the ghastly process of updating, we're happy." This is a clear example of where *Ccost* exceeds *Cval*.

2. Customers must stay up-to-date - To guarantee success of a product software vendor it is often assumed that customers must stay up to date. The misconception is demonstrated by the example of a content management system product software vendor, where customers use versions from years back who never updated due to the large number of customisations and complex update process. These customers, however, don't feel limited in their use of the product however, and will update when they require new functionality. Once again *Ccost* exceeds *Cval*. The difference between the first and second misconception is that they are discovered at different times in the product lifecycle. The first misconception is discovered once a new version of a product is released and is not adopted at all by customers. The second misconception is discovered once a vendor has many different versions out in the field, without encountering life-threatening problems.

3. Release *n + 1* is better for a customer than release *n* - Many of a bookkeeping software vendor's customers were still using the MS-DOS based version of their product until 2005 when the vendor declared it would no longer support the DOS version. When attempting to update all these small entrepreneurs to a GUI based version the main complaint was that the graphic interface was *less* intuitive than their previous DOS versions. The bookkeeping software vendor ended up implementing all the same keystroke combinations that were typical of the DOS era, into their GUI based client. Even though from the vendor's point of view their update to the GUI based version was necessary, customers could have worked with the DOS version for at least the next ten years and considered *Ccost* to be larger than *Cval*.

4. Fixes can be postponed to the next major release - A typical mistake to make is to postpone bug fixes for later releases, hoping to save the effort of having to implement the fix into multiple releases. This works fine if customers are eager to update, and the next major release is around the corner. However, in one of the case studies performed in 2004 we encountered a vendor who postponed many bug fixes to its next major release package. The major release package, planned for early 2005, still has not been released mid 2006. Many of the bug fixes had to be back ported to keep customers satisfied. This is a clear example where *Vcost* seemed to be lower than *Vval*, but actually was not.

5. Workarounds must be avoided at all costs - Once again, as long as *Ccost* exceeds *Cval*, workarounds are a nice solution to a problem that would otherwise require a large investment from an organisation or person. An example of this is the Internet Explorer workarounds for style sheets. Quite often style sheets will look different on Internet Explorer 6 than other browsers, due to a bug. It is common knowledge, however, that Internet Explorer's interpreter can be fooled by adding specific characters to the code of a style sheet. Microsoft has chosen not to fix this bug until Internet Explorer 7, mainly due to the fact that everyone is aware of the workaround and too many customers would need to be updated.

6. Customers always want new features - This common misconception is that any release package can contain new features, since the customer should be happy with (possibly) free new features. An example encountered is a point of sale product software vendor, whose users typed more or less blindly into the system and checked only every ten seconds to see if the screen was showing the desired result. The simple displacing of a button in the user interface raised so many complaints (*Ccost* exceeds *Cval*) that they decided to freeze the user interface to their application in between minor releases as much as possible.

7. Releasing too often is bad - The aforementioned bookkeeping product software vendor started releasing on a weekly basis at some point, to shorten the feedback cycle to developers. The vendor did receive more bug reports, but product experience in general, declined. The vendor decided that this was not caused by the fact that they released too often, but that they released to their final customers too often. The frequent releases were maintained, but only for internal use, quality assurance, and pilot customers. Also, customers are required to stay up to date to reduce the number of support calls.

8. A quiet customer is a happy customer - An informal survey amongst a number of customers of a plug-in software vendor showed that customers who contacted the helpdesk in the early phases of its use were much more content with the product than those customers who had not called the helpdesk in the early adoption phases. Another example encountered was a software vendor who called up a customer for a yearly check-up, and heard that they had recently decided to buy a competitors product, even while the customer still had a contract with the current vendor. This demonstrates the importance of regular customer contact. The customer would still have been a customer if the vendor had made the customer aware of the fact that *Ccost* is smaller than *Cval*.

9. Customers read release notes - Especially system managers of large software products are well accustomed to browsing through release notes, trying to find that one fix to a bug or that one new feature that justifies a customer's investment into updating the product. Clearly, this is a proactive customer that is looking to optimize the value of the software product's latest release package. These system managers, however, would be much more interested in information about new releases that specifically targets them. One software vendor [12] is currently experimenting with a system that filters release notes for specific customers, such that they do not receive information that is irrelevant to them. An example of this is a bug fix to a component a customer has not purchased.

10. Having many different releases out in the field is bad - The earlier example of the content management systems product software vendor shows us that having many different releases out in the field is not necessarily a bad thing, as long as it is part of the business model. This vendor, for instance, charges its customer for all services in the form of a service contract, especially to those customers with very old versions. To the software vendor these customers present more of a knowledge management problem, since many of the solutions built in the past have to be reused for customers experiencing similar problems now. The vendor does agree that this is only possible due to its small "manageable" number of customers. The difference between the second and this misconception is that the second misconception addresses the "happy" customer, whereas this misconception concerns the successful product software vendor.

Some other misconceptions encountered were "our next release must contain less bugs than our previous release to satisfy customers" and "we shouldn't build an automatic updater because the customer will feel they're not in control". These misconceptions are proven wrong by our Cost/Value functions as well, but we simply encountered them less often than the ten mentioned here. It is our firm belief that taking the profitability approach with regards to release package planning in a commercial environment is the way to go. An interesting question of validity is whether this type of anecdotal evidence is enough to prove that our Cost/Value functions are correct. It is part of our future work to further evaluate the validity of the Cost/Value functions based on historical (cash flow) results from both software implementations at customers and software release history.

4 Reducing Costs of Release Management

Besides using the Cost/Value functions for daily decisions, they allow us some thought experiments. Product software vendors generally adhere to bug fix/minor/major release scoping. When looking solely at version numbers, an open source project such as Mambo/Joomla, has had three major releases since 2001, approximately 10 minor releases, and approximately 120 bug fix releases. These numbers show that bug fix updates are released much more often than major updates. Also, when looking at customer behaviour, they are more inclined to regularly update to a new bug fix release package than they will perform a costly major update.

When looking at bug fix updates and the functions presented earlier the Cost/Value calculation impact factors change compared to major updates. In the case of a major update, the cost of development will largely exceed all other costs, making those less important from a financial point of view. For a major release, for instance, the completeness checking of artifacts will be a relatively small step in the release package creation project. When looking at a bug fix project, however, the development might have taken only a couple of days developing effort, whereas the creation of the release package might take an equal amount of time and effort. If we then take into account that these bug fix releases generally do not generate profit and only improve product quality and reduce the number of support calls, other costs are suddenly much more drastic.

Besides the scope of a release, the number of customers who update to a new release determines how much effort must be put into reducing the cost of release management.

For Exact Software and its 160,000 customers [9], for instance, the reduction in cost by introducing a combined software configuration management system and customer relationship system was huge. By combining these two systems they enable customers to automatically download and deploy bug fix and minor updates. However, if a vendor only serves twenty customers and is not planning to extend their customer base beyond one hundred customers, it must consider whether it is worth investing much into automatically releasing, delivering, and updating releases at its customers.

A product software vendor can reduce its costs in a number of areas. This cost reduction in turn enables a vendor to release more often. Releasing more often generates feedback about new releases quicker, which enables a vendor to improve its product and make better informed decisions on development and fixing plans. Cleary, this theory supports the agile camp, in its "Release early and often" viewpoint [2].

4.1 Vendor Side Cost Reduction

To begin with a vendor must strive to **release often**, if not continuously [18]. The more a product under development is in the shape it will be in when finally released, the less chance there is for errors to be introduced during release package creation. After all, any party within the vendor organization, be it pilot customers, other developers, or the quality assurance department, will use this latest release for internal evaluation. The parties responsible for the final release will also have less work in the final stages of release package creation, a process that takes place often. This process is hampered by a product that supports different languages, since quite often these language files are translated shortly before the final release date.

The process of **release package creation must be automated** as much as possible to eliminate simple (error sensitive) manual tasks. If a release package is checked for completeness automatically each time a release package is created, it does not need to be checked extensively by quality assurance, eliminating a large part of this process.

The cost of software delivery is greatly minimized if **all delivery is done through a network** instead of expensive media, such as CDs or DVDs. The releases stored on these media are never as up-to-date as the ones stored in the vendor's release package repository, which could be accessible through a network or secure Internet connection.

4.2 Customer Side Cost Reduction

Whereas the vendor might be reducing costs internally, it must invest in making the deal to update to a new version as attractive as possible. Though this seems like a large investment at first, the payoff comes quickly when customers become more eager and better informed with regards to releases a vendor offers.

Software deployment costs can be reduced for the customer by **automating the update process**. This requires the software vendor to seriously invest into an update tool and to develop its architecture in such a way that customisations remain functional after an update. Even though this seems like a large investment up front, it makes the decision for a customer to update easier, and as such makes them more eager to update often. The same holds for the reduction of downtime, since customers will be much more eager to update if downtime is reduced to a minimum.

Before customers can update to a new release, however, they need to be informed about the new release package. Currently, most product software vendors inform their customers through information news letters, customer days, e-mails, and many other ways. A higher rate of release penetration can be reached, however, if the vendor **uses the software itself to inform the customer**. This can range from a small pop-up when the application starts up, to an automatic pull of an update, such as Mozilla's Firefox currently does.

With regards to informing customers, release notes are an essential part of release management. When customers are looking for a bug fix, for instance, they will browse through the release notes looking for that specific piece of information. Clearly these **release notes need to be indexable**, such that customers who previously requested information concerning a problem are informed as soon as a fix for that problem has become available.

5 Discussion and Conclusions

This paper presents cost and value functions that product software vendors can use to evaluate whether it is profitable to release a version of their software. Simultaneously, functions are provided that assist a customer in making the decision to update a vendor's software product. These functions support ten changed viewpoints that were encountered in seven case studies. Finally, these functions show that costs can be saved for both product software vendors and customers on commonly occurring patch and minor updates, which can shorten feedback cycles from end-user to product software developer.

The process of release package planning is greatly simplified with the use of the provided Cost/Value functions. These functions also defend that product software vendors invest into automating processes such as release package creation, release package publication, informing the customer of a new release, and updating. The fact that this does not happen in practice raises a number of questions, such as why the vendors do not invest more into these processes. An answer often given when product software vendors were confronted with this question was that they are busy creating their specific software solution already, but that they would be happy to buy a tool that helps automating these tasks.

A weakness of the Cost/Value functions is that being obsessed with short-term profits will lead any product software vendor without a long-term vision to the abyss. Vendors must take into account customers will always be prepared to offer large amounts of money to small vendors if they just build one little feature that is extremely valuable to them. The vendor must always keep in mind that it is creating software for a market and not one particular customer. The functions must only be used once the prioritization of requirements for the next couple of releases has been finalized.

These calculations provide a decision method for updating and releasing, but only in case all costs and prognoses are exact. Knowing that this is impossible, we leave it to the practitioner to perform data gathering [5] and implement a risk factor for unforeseen costs (and unforeseen value). Currently the Cost/Value functions are still in an experimental state even though we feel they are of great value to the field of release package planning. Thus it belongs to our future work to evaluate the functions in real world scenarios with historical release and cash flow data. We do recommend using a currency as the unit of measurement, since both sales and full time employment units can be expressed in money.

Part of the work thus is to find methods and tools that assist product software vendors in automating the tasks of release creation, release publication, informing the customer of a new release, and updating a customer's configuration. In earlier work the lack of tools for software deployment was identified [11] and possible solutions were presented. With respect to continuous software releasing the tool Sisyphus was built to support product software vendors with automatically creating their software releases [18]. Work recently has started on the Pheme prototype, a communication infrastructure that assists product software vendors in sharing software, data, feedback, licenses, and commercial information with its customers.

References

[1] A. J. Bagnall, V. J. Rayward-Smith, and J. M. Whittley. The next release problem. In *Information and Software Technology*, volume 43, pages 883–890, 2001.

[2] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley, 2001.

[3] K. Bennet and V. Rajlic. A staged model for the software lifecycle. In *IEEE Computer, July*, 2000.

[4] P. Carlshamre. Release planning in market-driven software product development: Provoking an understanding. Springer-Verlag, 2002.

[5] C. Ebert, R. Dumke, M. Bundschuh, A. Schmietendorf, and R. Dumke. Chapter 1. In *Best Practices in Software Measurement*, 2004.

[6] I. Heitlager, S. Jansen, S. Brinkkemper, and R. Helms. Understanding the dynamics of product software development using the concept of co-evolution. In *Second International IEEE Workshop on Software Evolvability (at the International Conference on Software Maintenance)*. IEEE, 2006.

[7] S. Jansen. Software Release and Deployment at Planon: a case study report. In *Technical Report SEN-E0504*. CWI, 2005.

[8] S. Jansen. Software release and deployment at a content management systems vendor: a case study report. Institute of Computing and Information Sciences, Utrecht University, Technical report UU-CS-2006-0XX., 2006.

[9] S. Jansen, G. Ballintijn, S. Brinkkemper, and A. van Nieuwland. Integrated development and maintenance for the release, delivery, deployment, and customization of product software: a case study in mass-market erp software. In *Journal of Software Maintenance and Evolution: Research and Practice*, volume 18, pages 133–151. John Wiley & Sons, Ltd., 2006.

[10] S. Jansen and S. Brinkkemper. Definition and validation of the key process areas of release, delivery and deployment of product software vendors: turning the ugly duckling into a swan. In *proceedings of the International Conference on Software Maintenance (ICSM2006, Research track)*, September 2006.

[11] S. Jansen, S. Brinkkemper, and G. Ballintijn. A process framework and typology for software product updaters. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 265–274. IEEE, 2005.

[12] S. Jansen and W. Rijsemus. Balancing total cost of ownership and cost of maintenance within a software supply network. In *proceedings of the IEEE International Conference on Software Maintenance (ICSM2006, Industrial track), Philadelphia, PA, USA, September, 2006*, 2006.

[13] C. S. P. Ng, G. G. Gable, and T. Chan. An erp maintenance model. In *36th Hawaii International Conference on Systems Sciences (HICSS)*, 2003.

[14] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. A linguistic-engineering approach to large-scale requirements management. *IEEE Software*, 22(1):32–39, 2005.

[15] K. Rautiainen, C. Lassenius, J. Vahaniitty, M. Pyhajarvi, and J. Vanhanen. A tentative framework for managing software product development in small companies. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.

[16] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, 2002.

[17] I. van de Weerd, S. Brinkkemper, R. Nieuwenhuis, J. Versendaal, and L. Bijlsma. Towards a reference framework for software product management. In *Proceedings of the 14th International Requirements Engineering Conference (Accepted for publication)*, 2006.

[18] T. van der Storm. Continuous release and upgrade of component-based software. In *Proceedings of the 12th International Workshop on Software Configuration Management (SCM–12)*, 2005.