**Customer Configuration Updating**
**in a**
**Software Supply Network**

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht op gezag van
de rector magnificus, prof. dr. W. H. Gispen,
ingevolge het besluit van het college
voor promoties in het openbaar te verdedigen op
8 oktober 2007 des ochtends te 10.30 uur
door

Slinger Remy Lokien Jansen

geboren op 1 juni 1980,
te Rotterdam

Promotoren: Prof. Dr. S. Brinkkemper
Prof. Dr. P. Klint

Dit onderzoek is deel van het Deliver project, betaald door NWO/Jacquard, projectnummer 638.001.202.

The work in this thesis has been carried out at Utrecht University and the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam. The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems. SIKS Dissertation Series No. 2007-20.

# Acknowledgements

Before being thrown into the depths of customer configuration updating I would like to acknowledge the people who have been there for me the last four years. First and foremost I would like to thank the girl going through all the ups and downs, the down and outs, and the high and low points of a researcher. If you will stand by me for the rest of my life as you have done these years, Merel, I am guaranteed a happy future. My family is next, my mother Carrie, my father Geertje, and my sister Blixa, thanks for the trips, the meals, the christmases, and the overall entertainment during those laborious times. I specifically have to thank Blixa for keeping me young (and giddy).

I must thank Vedran, even after his treacherous move to Venlo, for providing me with at least 50% of the brainpower for this thesis. As our Judo trainer once said: "you guys look as if you know just that little more about life." Every day I hope he is right, and every day I hope we get to have our American style Barbeque when we are 45.

I would like to thank our Matriarch Doortje Wisse and all my aunts and uncle, for helping me out when stuff went horribly wrong (Dorotee, Birgitta, Julia, Yvje, Clemens, and the rest), their partners who put life into perspective when necessary (Ab), and of course Prof. Dr. Marian, who will be a motivating force for the rest of my career. Furthermore, I owe great thanks to Diana Emmink, Adriaan, Harald, and Rogier van Geest, for being the best in-laws a guy can wish for.

My friends as well as colleagues have helped loads, either personally or professionally. I am of course speaking of my esteemed friends Tijs, Magiel, Jurgen, Eva, and Inge. Furthermore I would like to thank Arie, Rob, Gerco, Lidwien, Ilja, Remko, Ronald, and Johan. Also, I must thank my dear friends and colleagues overseas, being Sue Black, Anthony Finkelstein, Andy Maule, and all others in London. Finally I want to thank those that have taught me life's lessons in no particular order, Karel Philipsen, Miss Ewings, Marko Steenbergen, (soon to be Professors) Kosters and Hogeboom, Aad van Polanen, Meneer van de Broek, Rene de Jong, dokter Meijer, Harry Ottink, Theo Ham, and Erik Bulk. Finally I'd like to thank Sjaak Brinkkemper and Paul Klint, for being mentors, friends, colleagues, and bosses on-demand.

# Contents

# Part I

# Introduction to this Thesis

# CHAPTER 1

# Introduction

The product software industry is flourishing. Computer games, enterprise resource planning products, and navigation systems are just some examples of successful products, nationally and internationally. The international product software industry has had a sustained growth of around 14% for a number of years, making it one of the most successful industries at this time. In 2001 (1999) the total market of the product software industry was estimated to be 196 (154.9) billion USD, which is just 9% of the overall ICT spending of 2.1 trillion USD worldwide. "The product software sector is among the most rapidly growing sectors in OECD countries, with strong increases in added value, employment and R&D investments" [44]. More specifically, the Netherlands exported 1.6 billion Euro worth of software products in 2005, putting the Netherlands in fourth place on the list of largest exporters of product software in the world.

Though product software vendors have a large body of knowledge available to them about generic software development, product development, and engineering, none of it is specific to the development of product software. Authorative works such as the the SoftWare Engineering Body Of Knowledge [1] only sparsely address the issues that are specific to product software management. Also, in product lifecycle management and product data management literature physical products are preferred over software products, with few exceptions.

One area of product software development that requires more attention are the release, delivery, deployment, and usage and activation processes, also known as customer configuration updating. This "ugly duckling" of product software development is the subject of this thesis, mostly because so little research has been done in this area, even though it affects product software developers on a large scale.

The overall question of this research is whether customer configuration updating can be improved for product software vendors by explicitly managing product software knowledge. This question is answered using a mixed method multi-theory approach. The results are a detailed description of the Customer Configuration Updating (CCU) processes that are obtained by conducting case studies into practices of product software vendors, by tool evaluations, surveys, prototype building and evaluation in industrial size case studies, and design research.

The overall result of this research is the development of four contributions that improve customer configuration updating and the unveiling of the importance of

customer configuration updating. The first contribution is a customer configuration updating process improvement model that enables product software vendors to make strategic improvement decisions. The second contribution is a tool evaluation method that enables customer configuration updating support tool builders to establish what features are required for such tools. The third contribution is a tool infrastructure for software knowledge delivery, enabling product software vendors to share knowledge with their end-users in a software supply network. Finally, the fourth contribution is a tool that facilitates correct evolution between configurations of components while keeping complexity within manageable borders. The importance of customer configuration updating is uncovered by nine case studies of product software and a survey, showing that customer configuration updating improvements have had a large influence on product success.

## 1.1 Research Area

### 1.1.1 Product Software

Product software is a packaged configuration of software components or a software-based service, with auxiliary material, which is released for and traded in a specific market [132]. In contrast, product software is different from embedded software because product software is sold separately from the hardware on which it will be installed. Furthermore, it is different from customly built software for one customer, in that because product software is delivered to a large number of customers and deployed on a wide variation of hardware components.

Some examples of product software are games, components-off-the-shelf, database management systems, and ERP packages:

- TomTom [129], the manufacturer of navigational software and embedded devices, develops the software product TomTom navigator. This product is a software product because it can be deployed on a large range of devices (PC, PDA, and embedded devices) and is sold separately. TomTom has demonstrated an explosive growth throughout the last years and is now active in eighteen countries.
- Zylom games [27] is a manufacturer and service provider of games delivered over the Internet. Their target customers are casual gamers who wish to play a quick game. For a small amount of money gamers can play all games without limitations. Furthermore, gamers can participate in international tournaments. This Dutch founded company currently has 10 million subscribed casual gamers.
- Exact Software [26] is a manufacturer of product software and one of the largest product software vendors in the Netherlands. Their ERP products, making bookkeeping for small to medium companies simple, are being sold all over the world. Currently their customer base exceeds 160,000 customers.

Besides illustrating what product software entails, these three Dutch examples show that product software vendors can be extremely successful. Furthermore, these three

provide only a minor part of the 1.6 billion euros in exports from the Netherlands. Besides being a booming business, the OECD [44] claims that product software is one of the most rapidly growing sectors in OECD countries.

### 1.1.2 Product Management Research Perspectives

Product software management can be seen from three perspectives [132], being the social perspective, the company perspective, and the development perspective. The social perspective views all external factors that influence a product software vendor, such as laws and regulations and the economy. The company perspective concerns all non-development processes, such as marketing, sales, quality control, etc. Finally, the development perspective concerns all processes that eventually produce software products that can readily be deployed at the customers. The product software research framework is displayed in figure 1.1 [132].

The focus of this thesis is on the development perspective, and more specifically the development, release, delivery, and deployment of product software (found in the bottom right corner of figure 1.1). The development process determines how bugs are resolved, feature requests are satisfied, the development methodology, etc. Furthermore, the release process determines how and when new releases are published, keeping in mind the releases that are already installed at customers. The delivery process concerns the delivery of software artefacts from customer to vendor, again keeping in mind the releases that are deployed at customer sites. The deployment process concerns not only the technical deployment of products, but also the implementation of these products into the customers' organization.

### 1.1.3 Customer Configuration Updating

One area specific to product software vendors is that they have to release, deliver, and deploy their products on a wide range of systems, for a wide range of customers, in many variations. Furthermore, these applications constantly evolve, introducing versioning problems. An increasingly important part of product software development thus is CCU. CCU is *the combination of the vendor side release process, the product or update delivery process, the customer side deployment process, and the activation and usage processes*. Product software vendors encounter particular problems when trying to improve these processes, because vendors have to deal with multiple revisions, variable features, different deployment environments and architectures, different customers, different distribution media, and dependencies on external products. Also, there are not many tools available supporting the delivery and deployment of software product releases that are generic enough to accomplish these tasks for any product. For a complete description of CCU we direct the reader to section 2.2.

This thesis does not stand alone in its attempt to improve CCU for product software vendors. The work on evaluating product updaters is largely based on an earlier evaluation model provided by Carzaniga et al [21]. Furthermore, the entrepreneurial aspects of this work were inspired by Xu and Brinkkemper's work on product software [132]. The tool Pheme was largely inspired by the Software Dock [52] and can be considered a next generation of it.


Figure 1.1: Product Software Research Framework

An integrated view of CCU is rarely presented in literature. There is, however, a large body of knowledge about the separate CCU processes.

**Release management** is divided into two areas, being release support tools and release planning. Release planning [122, 6, 20] is often seen as a wicked problem. This thesis only touches the surface of release planning, and attempts to find a generic process description for release planning, instead of detailed solutions for feature and bug prioritization [92] and solutions that describe new release methods [124]. The solutions presented focus on software products as blank artefacts. With respect to release tools this thesis does present a fairly simple release tool. However, the release tool presented in this thesis assumes the product is ready for delivery, whereas others [125], do not.

In regards to **software delivery** no research specifically addresses delivery of product software knowledge, with the exception of the works of Farbey and Finkelstein [42]. Other works on software delivery [61] focus more on the technical problems of delivery, such as trying to reduce overhead and delivery cost.

The work on **deployment** of components in multidimensional configurations could not have been carried out without the work of van der Storm revealing that these configuration spaces can be reduced to binary decision trees [123]. Other works on deployment and updating generally only look at technical aspects of deployment. An inspirational example of this is Nix [35], a tool that ensures complete and correct deployment by storing a customer's complete component configuration in a versioned repository. This tool elegantly updates component configurations but does not at all focus on the organizational aspects of deployment. Another example is the thesis of Ajmani [2] who has developed a theoretical model for run-time updates of distributed systems. Though this system is highly advanced and elegantly performs runtime updates, it does not describe any of the organizational deployment issues, such as dependencies on other applications and evolution of customer specific solutions.

With regards to **usage and activation** a research trend can be observed that focuses more on software quality and profiling of software products in their operational environments. Some examples of research projects focusing on usage and error feedback are Skoll project [77], EDEM [56], and GAMMA project [96], which uses a technique called *software tomography* to get valuable information from running deployed software. The techniques proposed by Elbaum and Diep [41] have been inspirational to this thesis, however, these techniques focus solely on where a software application must be probed to gather information on a deployed software application.

### 1.1.4 Product Software Development and Maintenance

Product software development is the activity of development, modification, reuse, re-engineering, maintenance, or any other activities resulting in packaged configurations of software components or software-based services. Product software development shares many processes and concepts with software engineering, although software engineering is all encompassing for the building of software, whereas product software development specifically looks at software products released for a market.

One of the main drivers for this research is the need to deliver working software quicker and to deliver software of higher quality. Customers expect better software and

shorter iterations between releases. Software developers are able to satisfy this demand by constantly testing, releasing, and generating more code. This type of software development is known as agile software development [10]. Some of the development methodologies originating from the agile camp are Extreme Programming [11] and Scrum [105] and are suitable methods to apply to product software development.

As product software vendors grow larger they find that product software knowledge management is a critical success factor. Sales personnel must know when the next release is coming out, what features are being developed at the moment, and what they can and cannot show customers. Software developers need to know when the next release is due, what customer concerns are, and what requirements deserve priority. Also, helpdesk personnel need to know what issues are being fixed, for which issues workarounds are available, etc. These are just some of the examples of knowledge that needs to be managed by product software vendors.

Product software knowledge management is the driving factor behind high quality CCU processes. For these processes knowledge is required about the product, such as product features, relationships to third-party products, and product licensing possibilities. Not only must this knowledge be managed internally, such that stakeholders can acquire this knowledge at any time, but this knowledge also needs to be shared with customers, third-party component providers and third-party implementation partners. For the CCU processes, different types of knowledge are required.

To release software a vendor needs to know when a product release will be finished, what product releases have already been released, the bill of materials for a release, and a products relationship to other products. Furthermore, a product software vendor must clearly define its debug and release policies, which together define release planning. Also, a vendor must define policies on how the release policy is shared with external parties.

To deliver software a vendor needs to determine what customers already have, how product artefacts are transported to the customer, and how often. A vendor needs to clearly specify how product releases are published. Furthermore, a vendor must deliver products to customers when they want products, not when the vendor feels like it. For the delivery process the customer also plays a part in knowledge management, because they decide when and how product usage and error feedback is delivered to the vendor.

To deploy software a vendor must make sure that all product requirements and dependencies have been made explicit, preferably in a human and computer readable format, and that this knowledge is at the customer side at the time of deployment. Furthermore, the vendor needs to know exactly how a customer's configuration can be updated to contain a vendor's new features and products. Also, vendors potentially need to train customers and keep them up-to-date about product developments.

Finally, the usage and activation process are for a large part the customer's responsibility. They decide when a license is used to activate a product, when knowledge about the configuration is delivered to the vendor, and how bug reports and feature requests are sent to the vendor.

### 1.1.5 Overview of Concepts

To improve the readability of this thesis the following definitions are provided. Please note that these definitions are in part a contribution of the research.

- **Customer Configuration Updating -** Customer configuration updating (CCU) is the combination of the vendor side release process, the product or update delivery process, the customer side deployment process, and the activation and usage processes. These processes all influence knowledge interaction between a vendor and a customer. A customer configuration is its current configuration of a product, the hardware on which it runs, and the services required to activate and use a product [143, 151].
- **Release Management -** We define *Product software release management* as the storage, publication, identification, and packaging of the elements of a product.
- **Release Package Planning -** 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.
- **Delivery -** Product software delivery is the delivery of software products, licenses, and software product knowledge from vendors to customers and from customers to vendors.
- **Deployment -** The delivery, assembly, maintenance of a particular software system at a customer site [21].
- **Usage and Activation -** The usage and activation process describes the customer-side processes in between deployment and removal of a software product. Processes included are the activation of the software product, the usage, the generation of feedback, the delivery of feedback to the software vendor, and billing of usage of the product.
- **Product Update -** A product revision released to enhance an older version of a software object.
- **Product Updater -** A product updater is an application that evolves a customer configuration by deploying a product update.
- **Software Product Management -** Software product management is portfolio management, product roadmapping, requirements management, and release planning [119]. Though CCU is not explicitly mentioned in this definition, it contributes to all parts. In regards to portfolio management a vendor must know what versions customers are running. Furthermore, product roadmapping is directly related to bug policy making. Also, requirements management contains customer feedback such as error reports and feature requests. Finally, release planning includes release package planning which is a CCU practice.

- **Continuous CCU -** Continuous CCU (C-CCU) is defined as being able to continuously provide any stakeholder of a software product with any release of the software, at different levels of quality. This way developers, testers, and even end-users can always be fully up to date, if desirable. C-CCU requires clear policy definitions for release, delivery, deployment, logging, feedback, and bug resolution. Please find a detailed description in chapter 7.
- **Software Supply Network -** A Software Supply Network (SSN) is a series of linked software, hardware, and service organizations cooperating to satisfy market demands. Whereas in the past product software vendors used to be monolithic organizations dealing with their own customers only, trends such as components-off-the-shelf and plug-in architectures have lead to software vendors forming into complex networks of suppliers.
- **Update Package -** A package that promotes a customer's configuration to a newer configuration.
- **Bug Fix Update Package -** A package that contains only bug fixes and no new functionality.
- **Feature Update Package -** A package that contains only new features.
- **Minor Update Package -** A package that contains bug fixes and new functionality that does not change the product structurally.
- **Major Update Package -** A package that contains bug fixes and new functionality that changes large aspects of the product, such as the architecture and underlying data model.
- **Software Product -** A packaged configuration of software components or a software-based service, with auxiliary material, which is released for and traded in a specific market.
- **Software Product Lines -** Engineering techniques for creating a portfolio of similar software systems from a shared set of software assets using common means of production.

## 1.2 Research Setting and Industrial Embedding

This research has been conducted in both an academic and an industrial setting. In the following subsections is explained what part different organizations played in the research. Furthermore, the reasons are provided why this research was conducted in such close connection with industry.

### 1.2.1 Academic Research Centers

Much of the research was conducted at three academic institutes, being the Centrum voor Wiskunde en Informatica (CWI) [128], the Center for Organization and

Information [94], which is part of the Beta Faculty of Utrecht University, and the Software Systems Engineering Group, which is part of the Computer Science department of University College London [107].

At CWI the case study protocol was created and used during for nine case studies presented in this work. Furthermore, the tool evaluation model for software product updates was developed there, and a large part of the tool evaluations presented in chapter 5 were conducted at the CWI as well. Finally, the Meta-Environment, a framework for language development, source code analysis and source code transformation developed at the CWI, [113] is used as a case study in chapter 7. The CWI provided a fertile ground for the more technical aspects of this research.

At the Center for Organization and Information the research was conducted into the industrial aspects of this research. The CCU model was created there, as well as the survey to evaluate the practices of a product software vendor as well. Furthermore, the Pheme tool was conceived at the center. The larger part of the thesis was written here. Also, the mixed method multi theory approach was devised here. The Center for Organization and Information has provided a useful academic network for cooperation with different organizations, such as the Platform for Product Software and VivaCadena [137]. Also, some work was conducted with students from this institute [136].

Finally, the Pheme prototype, described in chapter 7, was built for the larger part at the Software Systems Engineering Group, which is part of the Computer Science department of University College London [107]. Also, in cooperation with Anthony Finkelstein, we further developed the modeling method for software supply networks [151]. The Systems Engineering Group helped us with an industrial case to prove that the modeling method contributes in discovering business threats and opportunities.

### 1.2.2 Platform for Product Software

The Platform for Product Software is a Dutch group of product software vendors who have united under this name to share non-competitive business knowledge. The Platform is unique in the Netherlands, and attempts to share knowledge between product software vendors and academic institutions. The Platform arranges several meetings and working groups and is planning to grow into a formal organization with membership fees, regular product software specific courses, and a yearly convention.

The Platform for Product Software has provided different information sources for this research. To begin with, several companies were selected from the platform to do case studies with. Also, the survey, presented in chapter 4, was held amongst several of the Platform's members. Furthermore, organizing and attending the delivery workgroup meetings brought awareness of problems in the field. Finally, the workgroup assisted in prioritizing the questions of the afore mentioned survey.

The companies visited for this research were Exact Software, GX, Planon, Stabiplan, Chipsoft, Nedstat, and Tribeka. These companies provided literature, design documents, and update tools, and participated in case studies and tool evaluations.

### 1.2.3 Industry Context and Applicability

Clearly this research has been strongly embedded in the product software industry. There are a number of reasons for this.

The CCU processes are described to a limited extent in common literature, resulting in a lack of process descriptions, from which both industry and academics could profit. Furthermore, because product software vendors were the first to encounter problems with CCU manageability, academia has not yet developed equally advanced solutions. The industrial partners have also provided us with a number of critical success factors for the implementation of these processes.

The second reason why this work is strongly embedded in industry is that the industrial partners enabled tool evaluation from a different perspective. The product software vendors provided us with reviews and change requests for the Pheme prototype that is presented in chapter 7. Also, the product software vendors enabled evaluation of proprietary CCU support tools, providing a richer dataset. These proprietary support tools are discussed in chapter 5.

The third reason is that the product software vendors have enabled evaluation of the process models and improvement propositions. Each case study was finalized with an advisory report indicating strengths and weaknesses of their processes and that proposed a number of improvements, based on the process models. These improvements were praised and criticized by the product software vendors.

This research is, besides being highly relevant academically, a contribution to the product software industry. Product software vendors can use this thesis as a set of guidelines when designing CCU support tools and implementing these tools and CCU processes into their organizations.

## 1.3 Research Description

Product software vendors are impeded in their growth because they cannot adequately share knowledge with customers about products. Furthermore, these product software organizations increasingly cooperate in complex software supply networks, where one software vendor is the customer of another. These situations require an even larger amount of knowledge to be shared over a web of vendors and customers. To improve this situation software vendors need to know what knowledge they need to share with customers and how to share it.

Knowledge about products is shared when vendors release, deliver, deploy and when customers use a software product. These CCU processes have not yet before been properly modeled and evaluated. Furthermore, there are few tools that support large parts of the CCU processes. When software vendors attempt to improve CCU three problems become apparent:

- There are no adequate process descriptions for CCU,
- There is a lack of tools to support CCU,
- Each software vendor spends a lot of time automating CCU tasks, even though these tasks are similar for all software vendors.

The results of these three problems is that product software vendors, customers, and end-users experience many problems when releasing, delivering, distributing, deploying, and using product software.

For example, the Dutch product software and embedded devices vendor TomTom released a version of their product with a virus in 2007. Another example is when Microsoft released an old pre-release version of Vista to all its large volume customers on the 22nd of November 2006. Furthermore, delivery of software and knowledge, though seemingly trivial through the Internet, is often blocked by Firewalls.

When a customer organization or company purchases a software product, it often needs to be distributed to large numbers of workplaces. This is a complex activity, especially when these end-user configurations need to be updated or require one resource, such as a pool of licenses, or a database management server.

In 6 industrial case studies and a survey it was discovered that between 5% and 35% of the deployments of new products and product updates do not proceed as planned and require unplanned extra support from the software vendor. These organizations are impeded in their growth, due to the fact that they cannot handle larger customer bases, since it would result into more configurations that require maintenance and updates.

When a product has been deployed, it still needs to be activated and used. When a product is being used, usage and error feedback reports need to be sent back to the product software vendor. Currently there are no sufficiently adequate tools that can send, receive, and process error and usage feedback. To illustrate the use of customer feedback: when Microsoft implemented automatic error reporting for Microsoft Windows they quickly discovered that 1% of the bugs caused 50% of all the errors [18].

There clearly exists a need for CCU tools, process descriptions, and research.

### 1.3.1 Research Questions

The research is based on two main research questions. The first research question is

**RQ1: What are the concepts and is the state of affairs of customer configuration updating for product software?**

Delivering the correct configuration of components to a customer is a complex task for a software vendor, especially when taking into account subsequent releases of components. Before this research, there were no complete definitions that defined the practices of CCU. Authorative works, such as the SoftWare Engineering Body Of Knowledge [1] only address these processes in a scattered fashion, leaving a need for concrete process descriptions. Also, developers of tools supporting these processes generally develop tools that are restricted to only one development technology or process fragment. There are three subquestions (SQs) to this research question.

*SQ1.1: What is the state-of-the-art of customer configuration updating and who are the stakeholders?* The state of the art and the stakeholders provide insight into current problems being addressed by the scientific community. Furthermore, the

state of the art provides us with typical problems and measuring tools to evaluate the presented research.

*SQ1.2: What is the state-of-the-practice of customer configuration updating for product software vendors?* The state of the practice provides us with a view of how CCU is currently implemented at most product software vendors. This enables us to compare "ideal" theoretical solutions with solutions that actually work in practice. Furthermore, descriptions from the state of the practice enable us to test research results in practical settings and the model descriptions from sub question 1. By looking at product software vendors' best practices can be uncovered for the product software industry and software engineering. These best practices can be used to inform both the scientific community and other product software vendors on what best practices exist regarding knowledge sharing.

*SQ1.3: What parts of the CCU processes are currently supported by product update tools?* Many practical problems from CCU processes are caused by the absence of adequate tools. Before making such judgment and acting upon that by building new prototype tools, however, an inventory is needed of what features are currently provided by product software update tools. Such an inventory will uncover gaps in CCU processes and tools, and enable product software vendors to make informed decisions in make-or-buy situations.

**RQ2: Can customer configuration updating be improved by explicitly managing and sharing knowledge about software products?**

The scientific motivation for the research question is to develop processes and tools for the fully automatic consistency checking and web-based deployment, upgrading and integration of product software. The goal of the research is to provide academics and practitioners with comprehensive process models and tools that support the CCU processes, enabling both groups to improve the area of product software engineering, maintenance, and management. The economic motivation for the research question is to strengthen the manufacturing and deployment of product software in the Netherlands even more.

*SQ2.1: What aspects of customer configuration updating can be improved by explicitly managing and sharing customer configuration updating knowledge and can these improvements be measured?* The first step in answering this research question is to discover which parts of the product software development process are potentially improved. To do so a number of measures need to be developed to quantify these improvements. Such measurements enable the explicit definition of the contribution of this research.

*SQ2.2: Are product software vendors who explicitly manage customer configuration updating knowledge more successful?* A first insight into what

areas of product software development can be improved is provided by examples of product software vendors who are more successful because they explicitly manage software knowledge. Detailed descriptions of how these processes are improved and what type of knowledge is being managed and distributed in complex networks of product software vendors and customers enable further validation of the hypothesis that explicit knowledge sharing contributes to product software development.

*SQ2.3: What functionality is required from tools managing and reusing customer configuration updating knowledge to support product software development and the customer configuration updating processes?* Product software vendors can only explicitly manage product software knowledge with tools that can store software facts (in both a computer and human readable format) due to the large amount of data. To build such tools an inventory needs to be made of software knowledge that can potentially be managed by such tools. Furthermore, there needs to be a demand for such tools from a business perspective. Finally, the tools need to be evaluated in different settings to establish their contribution.

### 1.3.2 Research Approach

There are three views of CCU, being the practice view, the tool view, and the process view (see figure 1.2). The practice view concerns how CCU is applied in practice by product software vendors and open source products. The tool view looks at CCU support tools and their specific characteristics supporting the CCU processes. The process view concerns CCU process design and process models enabling any stakeholder of a software product to evaluate its CCU processes.

To answer the first research question about the current state of the practice of CCU, two studies were conducted; one through case studies at product software vendors and one through tool evaluations found in literature and used by the case study subjects. These case studies and tool evaluations enabled the forming of hypotheses about the Dutch product software industry, and about CCU support tools. It was soon found that the product software vendors encountered problems while implementing CCU processes and that the tools did not provide all features required by these product software vendors. This was confirmed when the tools and cases were evaluated using two specially designed evaluation models.

The CCU evaluation model was used to establish whether product software vendors can be compared and whether their CCU processes are a source of new problems and solutions. The CCU evaluation model enabled the forming of hypotheses about the Dutch product software industry. These hypotheses were then tested using the CCU survey. We then used the survey to establish empirical evidence for further theory building.

The CCU support tool evaluation model was used to establish what features are usually missing in these tools, what processes still provide technical challenges, and how these tools compare to each other. Some missing features also came from demands made by the product software industry. To improve the product software

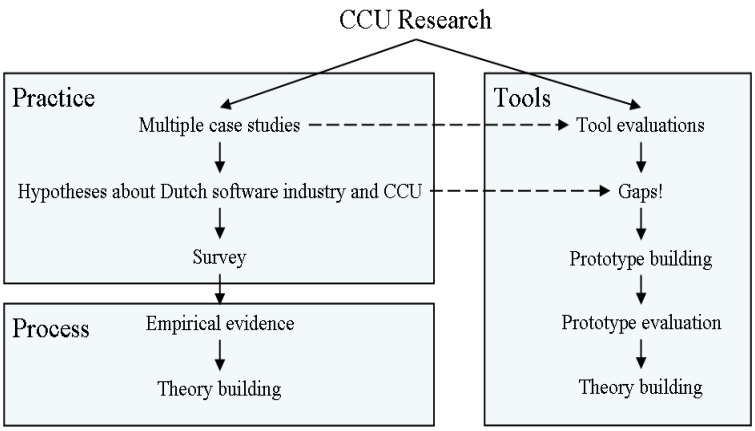



Figure 1.2: CCU Research Approach

CCU processes two tools were developed. These tools were evaluated in two different scenarios. The tools and evaluation enabled further theory building.

### 1.3.3 Research Methods

This research has been conducted as a mixed method multi-theory study of CCU processes. The methods used are case studies, survey research, design research, tool evaluation, and prototype evaluation. The applied research methods per chapter and publication are shown in table 1.1.

| Chapter | Publication | Case study | Survey | Design research | Tool evaluation | Prototype evaluation |
|---|---|---|---|---|---|---|
| 2 | [143, 139, 144] | X | | X | | |
| 3 | [141, 148] | X | | X | X | |
| 4 | [146] | | X | X | | |
| 5 | [147] | | | X | X | |
| 6 | [142] | | | X | | X |
| 7 | [149] | | | X | X | X |
| 8 | [145] | X | | | | |
| 9 | [151, 152] | X | | X | | |

Table 1.1: Applied Research Methods

**Case Studies -** Nine case studies have been conducted applying the case study method developed by Yin [133]. The case studies have contributed to defining the problem area, defining the state of the practice, and finding tools in the field. In chapter

3 one of these cases is described in further detail. Also, from these nine case studies ten misconceptions about customer configuration updating are highlighted and clarified using cost/value evaluation in chapter 8. In chapter 9 a case study is used to evaluate a modeling method for software supply networks. The case studies were performed with care and rigor, using Yin's case study method and guidelines [133]. To avoid common pitfalls we have used guidelines and pointers from Kitchenham and Flyvbjerg [69, 43].

**Survey Research -** The case studies, though useful for initial investigation and finding new phenomena, did not enable full generalization of the conclusions drawn about product software vendors in the Netherlands. To counter this, a survey was held among 74 product software vendors in the Netherlands. Their results enabled generalization of our conclusions and provide a clear image of the CCU processes of Dutch product software vendors.

**Design Research -** Some of the conducted research is design research [118], where solutions are designed and evaluated in different settings. Some examples are the CCU model in chapter 2, the product development cycle time model and the Pheme Knowledge Delivery Infrastructure described in chapter 7, the software deployment modeling technique and tool in chapter 6, and the software supply network modeling technique presented in chapter 9.

**Tool Evaluation -** Tool evaluation in this research is seen as a specific case study method to determine properties and features of CCU processes support tools. To establish what tools are appropriate to support (C-)CCU processes, tools were evaluated by testing and applying them to real-life examples in chapter 7. The result of this research is a list of features that are currently insufficiently provided by (C-)CCU support tools.

**Prototype Evaluation -** Two tools were built to demonstrate feasibility and to establish that they actually improved (C-)CCU. These tools were evaluated the same way as the other academic and commercial tools in chapters 6 and 7.

## 1.4 List of Acronyms

**PDM** Product Data Management

**SCM** Software Configuration Management

**CRM** Customer Relationship Management

**SKB** Software Knowledge Base

**CCU** Customer Configuration Updating

**C-CCU** Continuous Customer Configuration Updating

**SSN** Software Supply Network

**COTS** Components off the Shelf

**ASP** Application Service Provider

## 1.5 Thesis Outline

The different research questions and subquestions are answered throughout the chapters. The focus of this thesis is on two topics: the state of the practice and the designing of new solutions that support and improve the CCU processes, which is reflected by the two research questions. The chapters can (crudely) be divided over the two questions as well. Chapters 2, 3, 4, and 5 consider the state of the practice and demonstrate best practices. Chapters 6, 7, 8, and 9 introduce new processes, techniques, and technology to support and improve the CCU processes.

Readers new to the field of CCU are recommended to read chapters 1 and 2. Furthermore, once the foundations for CCU are laid, one should move on to chapter 4, to read more and find some of the empirical evidence for the claims made in this thesis. Should one wish to know more about the case studies, chapters 2 and 3 describe nine case studies and provide a detailed case study approach. These chapters make up part II of this thesis, describing the current practice of CCU in product software and providing evaluation models for CCU.

For those interested in CCU support tools, chapters 5, 6, and 7 provide tool evaluation models, evaluations, implementations, and descriptions. Together these chapters make up part III of this thesis.

Finally, for those looking for anecdotal evidence and process improvement proposals, chapter 8 describes some process improvement proposals about release management. Furthermore, this part describes the effect of software supply networks on CCU. These two chapters make up part IV, describing process improvements for CCU.

**Chapter 1** introduces the thesis work and provides an overview of the thesis, the research questions and methods, and the concepts used in the chapters.

**Chapter 2** is based on joint work performed with six product software vendors and CWI [128], and describes case studies into the CCU processes of product software vendors. This work establishes CCU as a new area and firmly puts it onto the software product development agenda, hence the title "Turning the Ugly Duckling into a Swan". The work was published in part at the doctoral consortium of the International Conference on Software Engineering in 2006 [139], in part at the Workshop on Interdisciplinary Software Engineering Research in 2006 [144], and presented in full at the International Conference on Software Maintenance in 2006 [143].

**Chapter 3** describes one of the case studies in full detail and shows that integration of product data management, software configuration management, and customer relationship management enables a product software vendor to serve 160,000 customers with minimum overhead. This work was published as a technical report at CWI in 2004 [64], as a paper at the International Conference on Software Maintenance in 2005 [148], and in the Journal for Software Maintenance and Reengineering in 2006 [141].

**Chapter 4** describes the result from a benchmark survey amongst product managers active in the product software industry. The survey was used to benchmark products from different product software companies in regards to their CCU processes. The work has recently been submitted [146].

**Chapter 5** describes an evaluation model for product update and CCU tools.

Furthermore, fourteen tools are evaluated and missing features are described. The work was published in 2005 [147] at the European Conference on Software Maintenance and Reengineering.

**Chapter 6** describes, based on van der Storm's theory that all differences between two component configurations can be calculated using binary decision trees [123], how this theory can be used on component configurations with different versions and features. An early version was published at the Workshop on Development and Deployment of Product Software in 2005 [138] and the full version was presented at the Workshop on Component Deployment in 2005 [142].

**Chapter 7** describes improvement in development speed and time to market for two experimental case studies in which different CCU support tools (such as Pheme) are applied. The work has recently been accepted for publication in the Proceedings of the ERCIM Workshop on Software Evolution 2007 [149]. A short paper describing Pheme has also been published at the conference on software maintenance in 2007 [140].

**Chapter 8** describes ten misconceptions based on evidence from the case studies. These misconceptions describe process improvements for release management. The work was published at the First International Workshop on Software Product Management in 2006 [145].

**Chapter 9** has been published in part at the Caise Forum [150] and at the industrial session of the International Conference on Software Maintenance in 2006 [152]. A full version has been accepted for publication [151] in the Proceedings of the 8th IFIP Working Conference on Virtual Enterprises. The work describes a modeling method for software supply networks and describes the case study of Tribeka, an English firm that provides hardware that "prints" software on-demand in a computer or software store.

**Chapter 10** provides answers to the research questions and lists the contributions of this work.

# Part II

# Current Practice

# CHAPTER 2

# Turning the Ugly Duckling into a Swan

*For software vendors the processes of release, delivery, and deployment to customers are inherently complex. However, software vendors could greatly improve their product quality and quality of service by applying a model that focuses on customer interaction if such a model were available. This chapter presents a model for Customer Configuration Updating (CCU) that can evaluate the practices of a software vendor in these processes. Nine extensive case studies of medium to large product software vendors are presented and evaluated using the model, thereby uncovering issues in their release, delivery, and deployment processes. Finally, organizational and architectural changes are proposed to increase quality of service and product quality for software vendors.*[1]

## 2.1 Introduction

With the advent of increased amounts of bandwidth, the communication between software vendors and their customers can greatly be improved by introducing automatic error feedback reporting, usage feedback reporting, electronic customer feedback, and license, patch, and update distribution. Whereas in the past customers and vendors could only communicate by mail and phone, the World Wide Web can now function as a lifeline between customers and software vendors, allowing automatic license retrieval, deployment and error feedback, automatic updates, and automatic provision of commercial information to customers. Product software vendors, however, generally do not implement any of these key practices.

To date product software is a packaged configuration of software components or a software-based service, with auxiliary materials, which is released for and traded in a specific market [132]. Product software vendors encounter many problems when

[1] This work was originally published in the proceedings of the 22nd International Conference on Software Maintenance, entitled "Definition and Validation of the Key Process Areas of Release, Delivery and Deployment for Product Software Vendors: turning the ugly duckling into a swan" in 2006 [143]. The work is co-authored with Sjaak Brinkkemper.

attempting to improve customer configuration updating of their product software. Customer configuration updating is defined as the combination of the vendor side release process, the product or update delivery process, the customer side deployment process, and the activation process. To begin with, these processes are themselves highly complex, considering vendors have to deal with multiple revisions, variable features, different deployment environments and architectures, different distribution media, and dependencies on external products (see chapter 6). Also, there are not many tools available that support the delivery and deployment of software product releases that are generic enough to accomplish these tasks for any product (see chapter 5). Finally, CCU is traditionally not seen as the core business of software vendors, and seemingly does not add any value to the product, making software vendors reluctant to improve CCU.

A number of sources show that CCU is often underestimated and that it requires more attention in the quickly changing software industry. First, the quality of deployment and upgrade processes can increase customer perceived quality of a software product significantly [82], making it important that these processes are managed explicitly. Also, field research has shown that by explicit management of CCU, software vendors are able to handle large amounts of customers as is described in chapter 3. Finally, Niessink et al. have shown that the development of software should be seen as product development, whereas maintenance should be seen as a customer service, thereby improving customer interaction [87], the latter being stressed again by the introduction of the Software Maintenance Maturity Model [5].

Even though the previous sources call for more attention to CCU, it is underemphasized in literature. The SWEBOK, for instance, gives a generic description in the Software Configuration Management (SCM) chapter of the processes of release and delivery. The Capability Maturity Model (CMM) [59, 95] also does not provide adequate descriptions for CCU, which can be explained by the fact that the CMM does not focus on product software specifically. Attempts have been made in the release candidate of the IT Service CMM [86], although the IT Service CMM does not provide an elaborate description of the processes of release, delivery, and deployment either. Clearly, even though there is a need for process definitions, there are no adequate process descriptions available for product software vendors. This chapter attempts to satisfy that need by shedding light on the ugly duckling that is customer configuration updating.

The contribution of this chapter is twofold. First, it attempts to answer the need for adequate process descriptions by presenting a model describing and identifying CCU. Secondly, eight case studies performed at medium to large software vendors into their development and CCU processes, are presented. These case studies provide practical knowledge and specific process descriptions which are, similar to the presented model, focused on customer interaction. The cases are evaluated using the model, which reveals that several key practices are left completely uncovered, due to the implementation effort involved, the lack of sufficient process descriptions, and the lack of sufficiently equipped CCU support tools.

Section 2.2 describes the CCU model and its process areas, along with the key practices belonging to each process area. The approach taken in the case studies and the eight studies that were performed and evaluated using the CCU model are reported

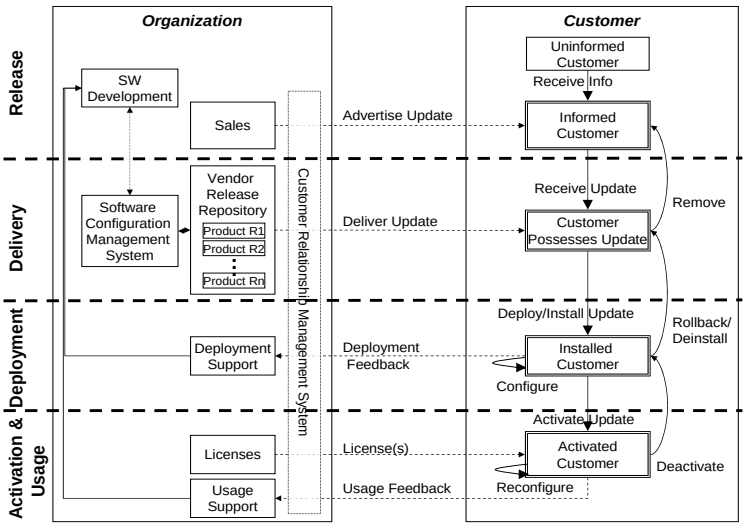



Figure 2.1: CCU Model

in Section 2.3. A description of the results per case study is also provided there. The key practices and combined results of the case studies are discussed in Section 2.4, where we also defend the claims made in the chapter. Finally, our conclusions and future work are presented.

## 2.2 Process Areas for Customer Configuration Updating

In this section the key process area of customer configuration updating is modelled. This model explicitly defines customer actions, enabling a software vendor to better manage and predict the key practices that need extra focus. Much akin to the CMM [98], the model uses the concepts of key practices, features, and process areas. Key practices are practices of a software vendor that enable features. Features are defined as properties of a process that improves product quality and quality of service. Each process area identifies a cluster of related features that, when performed collectively, achieve a set of goals considered important for enhancing process capability. A software vendor possesses a feature within a process area, once it responds correctly to one of these customer triggered actions.

To describe the key practices for CCU, its process areas need to be established. These process areas are found using a model for software updaters that focuses on the customer, which is described in chapter 5. Due to the fact that software maintenance and deployment focuses solely on the customer, the model is extended

with the organizational interactions that are required to fully support a customer's actions after an update is released. The CCU model, as seen in Figure 2.1, displays the states a customer can move through after a product or update release on the right side. On the left side, the organizational structures that facilitate interaction are displayed. Within the CCU model four process areas are distinguished, being release, delivery, deployment, and the activation and usage process areas. The process areas are separated by dotted lines in figure 2.1 and are further described in the sections below. Both the process models of the Software Dock [21] and SOFA [99] are contained in the presented model.

Processes in the model are triggered by customer actions. These actions are becoming aware of, downloading, deploying, reconfiguring, activating, and deactivating the release. When a vendor receives a customer request, the Customer Relationship Management (CRM) system is used to identify the customer. The vendor then handles the request and interacts with the customer. The customer moves through a number of states when about to update his configuration. At first the customer is unaware of the update, until the customer requests information about a product. Once received, the customer hopefully downloads, deploys and activates it for use, in the mean time communicating with the vendor in the form of software, licenses, feedback, and product knowledge.

### 2.2.1 Release Process Area

The release process area describes the release of a software product for a specific vendor and the interaction with its customers. The features within the release process area are:

- Release process management
- Product knowledge management

With respect to release process management a primary key practice is a formalized release procedure describing step by step how a release is created. Another key practice is the sharing of knowledge within the organization about the next release, such that all employees whose jobs are in some way related to the new product release are aware of the functionality in the next release, the release date, and the policy on sharing such information. Such awareness creates transparency within the software development organization, improving the relationship between the sales and development departments. This is related to the key practice that the sales, development, and support departments must all be aware of the product's relationships with other components, such that no late surprises at a customer site are possible. For example, if a product comes in simplified Chinese, it might not be compatible with a large number of commercial database management systems, even though the original release of the application in English did work.

One key practice of the release process area with respect to product knowledge management is that all versions of the software that have been released by an organization, must be stored in a release repository that mirrors the releases in the software configuration management system. This enables customers using older

versions to reinstall and update their product at any time. The same way releases must be managed explicitly, the software vendor must manage explicitly all internally used development and CCU support tools. Finally, the vendor must manage all external components that are included and packaged with the product.

The vendor must make a conscious effort to keep its customers updated on the latest news and product releases using any channel of communication, such that customers are not lagging behind in either product releases or product release information. Sales and lead management includes the use of pilot customers that pre-evaluate and test the software before an official release. Also, customer communication in the release process area is most interesting to the sales department of a product software vendor. A sales department must have insight into the purchasing guidelines and processes of a customer organization. One relevant aspect to determine product quality is strategic planning of product releases and updates. Customer organizations utilize product software in such an intensive manner that an update is a costly matter, due to down time, system instability, and the number of systems that require the update. A software vendor must establish the best time when an update is published and what the possible consequences are of deploying the update [5]. Microsoft, for instance, releases its security updates for all its products on the second Tuesday of the month and they have communicated this with their customer base.

### 2.2.2 Delivery Process Area

The delivery process area concerns the delivery of software, licenses, and product knowledge to customers. The key practices belonging to the delivery process area are focused on the following features:

- Delivery methods to customers
- Customer side delivery

To begin with software vendors must enable customer organizations to perform deployment using whichever medium a customer chooses, such as DVD, CD, a local area network, or the Internet. Secondly, customers must be able to remotely deploy applications and updates onto a user system without physically having to touch it. Thirdly, the product must supply a mechanism for automatic pull of updates, such that the customer can check for updates and download them automatically on a regular basis. The customer must be able to abstract from the download site of the vendor, allowing the customer to use an internal download server. If possible, the product must send back a deployment report after a customer has deployed the product, to inform the vendor whether the deployment was successful or not.

### 2.2.3 Deployment Process Area

The **deployment process area** contains key practices that enable a product to be deployed, removed, and updated. The key practices in the deployment process area are categorized into:

- Environment checking

- Local configuration management
- Deployment process automation

The key practices related to local configuration management are prone to many issues, such as missing (external) components, incomplete downloads, erroneous deployments, and overwritten customizations. To improve the deployment a deployment tool must inspect the local configuration, to see whether external components are missing and whether the local system provides enough resources, such as disk space. Also, downloaded packages must be checked for integrity and completeness. In the cases of missing components and files that do not pass their integrity checks, some automatic resolution must be implemented. Finally, it must be possible to rollback from an update or deployment to return to the previous configuration.

Customizations are widely applied for specific business domains and for specific customers. In many cases these customizations account for a large portion of their total revenue, which proves that explicit customization management is vital to many software vendors. A key practice for a software product with many different customizations at different customers is that the main product is updated without overwriting local customizations.

Once these issues have been tackled [138], the software vendor can make these processes as quick and easy for the customer as possible by implementing semi-automatic deployment, update, and rollback procedures. Another key practice is that updates do not require downtime when performing an update, allowing the customer to use the product without interruptions.

Table 2.1: Some Statistics on each organization

| Software Vendor | Employees | CCU employees | Customers | Technology | CMS |
|---|---|---|---|---|---|
| ERPComp | 1500 | 15 | 160.000 | ASP+ Delphi | Proprietary |
| CMSComp | 65 | 5 | 140 | Java | SubVersion |
| FMSComp | 160 | 3 | 900 | Delphi + Java | VSS + CVS |
| OCSComp | 115 | 2 | 20 | C++ | CVS |
| CADComp | 60 | 3 | 4.000 | Delphi | PVCS |
| HISComp | 100 | 2 | 40 | Delphi | VSS |
| Mozilla | 710 | 5 to 10 | 1.000.000+ | Java | CVS |
| Apache | 388 | NA | 1.000.000+ | Java | SubVersion |

Customer organizations often use different testing and acceptance stages according to the IT Infrastructure Library (ITIL) [22] before actually implementing software in the entire organization. This requires that deployments are done quickly, and that configuration settings and data files are moved separately from the software. This key practice is related to the externalization of all user and configuration data, which enables a transparent configuration environment [36]. Within such an environment all configuration and user data is accessed externally from the product, which allows for

relationships to be established between configuration data between products, enabling sharing of user configuration data such as e-mail account settings between e-mail clients, font sizes between applications, or even appearance settings between operating systems. Such externalization allows for the product to perform product data backups as well, enabling quicker and more reliable backup retrieval actions.

### 2.2.4 Activation and Usage Process Area

The **activation and usage process area** concerns the activation and working of a product at the customer site. The activation and usage process area focuses on the following features:

- License management
- Feedback management

License management enables a customer organization to manage licenses explicitly, and activate the product with a different license on each start-up, allowing customers to use test and development versions, and to provide different functionality to different user profiles. Another key practice belonging to license management is that licenses need to be stored in some coded fashion, to hinder piracy of products. Finally, to have maximum commercial flexibility, the licenses should control large parts of the software, such that any functionality is activated or deactivated using the licenses.

The vendor must also explicitly manage its customer licenses. To begin with, a vendor must be able to automatically renew a license for a customer, such that the vendor can renew or prolong a license without much effort. To achieve this, it must be possible to generate licenses from contracts automatically.

Feedback management allows a vendor to gather large amounts of data about its customers and its product as it acts in the field. Feedback can come from either automatic sources or manual customer triggered sources. Feedback is used, in the automatic case, to provide knowledge to the vendor about product usage and knowledge about the customer's configuration. Finally, the user should be able to report errors and questions to the software vendor through the software product. This allows users to state questions and report bugs about specific screens and unclear functions in the product.

## 2.3 The Cases and their Key Practices

In this section the anonymized cases are described. Some generic information is provided on each software vendor and the reasons why the case was included in this research are stated. A description is also given on how the case studies were performed. Table 2.1 provides some statistics on each organization that is part of our research set. Tables 2.2, 2.3, 2.4, and 2.5 show the key practices these software vendors have implemented. Each of the key practices has been evaluated using a list of criteria, which have been left out for the sake of brevity. An open circle shows that the vendor has implemented the facilities that provide a key practice, yet it has not implemented the key practice. These can be considered "quick wins" for the vendor.

### 2.3.1 Case Study Approach

To produce these results six descriptive case studies [133] were performed at Dutch software vendors. These case studies resulted into six case study reports [19]. During several months of doing the case studies, facts have been collected from several sources:

- **Interviews -** To study the cases and confirm our hypotheses, interviews were held with the people responsible for the development and usage of the studied product.
- **Studying the software -** Academic licenses were granted to the products. These licenses helped to gather many facts by examining the products, using the products, and experimenting with the product and its updating capabilities.
- **Document study -** Document study was performed to evaluate the development process and cross check the answers provided by the other sources of information.
- **Direct observations -** Since our research took place at the development departments (of the non-open source cases), we were able to directly observe and document day to day operations.

The interviews consisted of two sessions, one to explore and elaborate, and one to cross-reference answers from other interviews. The second session was also used to cross-reference documentation and confirm the facts stated in these documents. Besides these reviews we also created a case study protocol and a case study database. To ensure reliability, the case study report was reviewed by key informants. Two open source organizations were included to evaluate their key CCU practices. For these two cases all on-line material was used, including the source code of the products and the products themselves were tested extensively. The open source cases' high numbers of employees can be explained by the fact that open source developers are not working on a product full-time. The open source cases can therefore not be compared to the commercial cases in terms of size. The open source cases have been added to show that the CCU model can be used for any type of software vendor or distribution organization. Also, the open source cases contrast with the commercial cases in a number of interesting ways. CCU model coverage looks different for an open source product than for the other products presented. To begin with, licensing is an underrepresented aspect of open source products for obvious reasons and bugs tend to be reported using other channels than the product itself (Bugzilla, for instance).

The validity threats to our case studies are construct, internal, external, and reliability [133] threats. With respect to construct validity, the same protocol was applied to each case study, which was guarded by closely peer reviewing the case study process and database. To create a complete and correct overview, both the development and CCU processes have been documented extensively. The internal validity was threatened by incorrect facts and results from the different sources of information. By crosschecking these results and observing the processes as they were going on a complete view could be created. With respect to external validity, the cases are representative for the Dutch software vendor market domain because each software vendor has a different number of customers and is active in a different problem domain.

Also, the general information about these vendors has been compared to other vendors that are active in the Platform for Product Software [45], an organization that aims to share knowledge between research institutes and software vendors in The Netherlands, with over 100 members. The comparison shows that the six cases are a cross-section of the Dutch software industry. Finally, to defend reliability we would gather the same results if we did the case studies again, with one major proviso, which is that many of the case study reports, published after the case study, lead to improvements in each of the software vendors' organizations.

### 2.3.2 Hospital Information Management System

*HISComp* business activities are the production and sale of medical information systems, the customization of their products for customers, and the reselling of all required third-party hardware and software. *HISComp* currently has a customer base of approximately 40 international hospitals and currently employs approximately 100 employees.

*HISComp* is a typical software developer with a traditional and straight-forward way of distributing software via CDs. Patches are released on a website and the customer's system manager is responsible for deploying the patch, using a detailed list of instructions. Each customization that is built for a customer is included in a separate customization branch, which is merged with the trunk later on. Such variable functionality is activated using a coded license file. *HISComp* does not gather automatically any technical information on customer sites and the working of the product heavily depends on the customer's system manager. [7] *HISComp* releases patches and service packs containing multiple patches irregularly and main releases periodically.

### 2.3.3 On-line ERP Information Portal for Large Businesses

*ERPComp* is a manufacturer of software for accounting and enterprise resource planning (ERP) that has established an customer base of over 160,000 customers, mainly in the small to medium enterprise sector. Through autonomous growth and acquisitions the number of employees has grown to 2,025 in 2004. The International Development department employs 365 developers on different international locations. *ERPProd*, *ERPComp*'s product is a front office application that provides organizations with financial information, multi-site reporting, and supports relationship and knowledge management. Employees, customers and company partners are provided with real-time on-line access to information across an entire organization.

In chapter 3 the results of the extraordinary integration a company has achieved within its Product Data Management (PDM), CRM, and SCM. The main lesson learnt was that a company can serve many customers as long as it focuses on making CCU effort as low as possible. *ERPComp* applies the KISS (Keep it Short and Simple) principle to such an extent that they have abolished version management. The use of a proprietary product data management system for software products allows *ERPComp* to reason and store information about their software and share knowledge about product items throughout the company, such as compatibility information. The integration of

their SCM and CRM systems allows customers to log into the *ERPComp* customer portal and download software the customer has purchased, including a license file for that customer. This license file is managed on both the customer and vendor side and must periodically be renewed by the customer. Furthermore *ERPComp* has developed its own product deployment and update tool, and reports the version number of the latest download by the customer to the CRM software, such that the support department can always see what version of the software the customer is currently using.

### 2.3.4 Content Management System

*CMSComp* is a web technology company that focuses on content management, online application development and integration of backend systems into web portals. The services of *CMSComp* include consulting, development, implementation, integration and support of interactive web applications. These services are supported by *CMSComp* products. *CMSComp* attempts to find a personified solution for each customer organization. *CMSComp* currently employs 65 people. *CMSComp* has been experiencing such rapid growth over the last years that they have had to limit growth to keep it manageable at 6%. To serve the growing amount of customers with this restriction, *CMSComp* has started a partner program, where partner companies can provide the same services as *CMSComp*, using the *CMSComp* product.

*CMSComp* has only recently started focusing on their product, instead of the services the company used to provide. The content management and display product is generally deployed on a web server, where it will remain unchanged, until updated manually by customers. The product is checked with an unencoded XML license file that is accessible to the customer. License files cannot be generated automatically. Due to the large amount of customization that is implemented during the building of a site, the content management product has a transparent software architecture especially adjusted to enable such customizations. Due to the complexity of the software, deployment is a complex two hour process per web server. Due to the fact that *CMSComp* generally has access to their customers' web servers, remote deployment and updating are possible.

### 2.3.5 Providing a Counter Service On-Line

*OCSComp* is an application service provider that provides commercial organizations web statistics. They provide page count solutions to any type of customer, from small counters on personal websites, to large navigation tracking counters on e-commerce sites. *OCSComp* currently employs around 100 people, based on multiple European locations. *OCSComp* does not deliver software to customers, since customers visit the *OCSComp* portal to see the data that was gathered while people surfed their sites.

The Application Service Provider (ASP) case adds some interesting data to our research. To begin with *OCSComp* is much more capable at local configuration management and deployment processes, due to the fact that their servers are freely accessible by the organization itself. This explains *OCSComp*'s presence in local configuration management, and product data and SCM features, and can therefore not be compared to other product software companies in this area. Due to the fact

that customers log into *OCSComp*'s website on at least a weekly basis, *OCSComp* uses this channel to communicate the product information and new functionality to its customers. Finally, licensing has not been connected to CRM and requires an employee to copy the information from a contract into the license management system.

### 2.3.6 Facility Management System

*FMSComp* is an international software vendor that produces facility management and real estate management software for organizations. *FMSComp*'s products are marketed through four international *FMSComp* subsidiaries and eight international partners. At present *FMSComp* employs 160 full time employees. Recently, they have started testing a new version of their software, which has been completely reimplemented using J2EE technology.

*FMSComp* is an extremely good tool builder and has built many tools that are not managed explicitly, sometimes resulting in loss of knowledge about the source code or even the source code itself. These tools, however, have improved their development and CCU processes. They are very strong in product development and provide services to many large companies. They provide different types of deployment for their product, as to allow both *high network traffic, low deployment effort* and *low network traffic, high deployment effort* scenarios. *FMSComp*'s weakest area is licensing, even though they have a (semi-)automatic license generation process. The software has an in-built function to create a feedback report that is used to inform *FMSComp* of problems in their software. However, this report must be e-mailed to *FMSComp* manually by the customer.

### 2.3.7 CAD plug-in for Building Design

*CADComp* currently employs 60 employees. *CADComp* produces software plug-ins for AutoCAD that support building services and building management consultants in the Dutch industry, by creating drawing libraries, tools, and enhancements for two three-dimensional drawing tools, being AutoCAD and IntelliCAD. *CADComp* and its 60 employees at present serve 4000 customers.

Due to the nature of their product, *CADComp* must deliver its products to customers by unpacking a common CAD application and repacking it with their plug-in, using InstallShield for the deployment process. They use both software and hardware licensing mechanisms. Due to the size of their final deployment package they use CDs for distribution. *CADComp* makes no assumptions about the customer's network connection and therefore does not do any user or deployment feedback. Backups of user configuration data and files are complex, due to the fact that such knowledge is stored in many different formats, databases, and files, spread out over the complete deployment. This complexity is caused by a complex software architecture that allows *CADComp* to deliver its plug-in for different CAD applications.

### 2.3.8 Mozilla Firefox

Mozilla currently owns *Firefox*, one of the most successful open source development projects available. The *Mozilla* internet browser, created by the Mozilla Foundation provides a viable alternative to other browsers such as Opera and Internet Explorer.

*Mozilla* has implemented some update key practices in their product, such as an automatic update function that is used to update the local product installation. Mozilla does not, however, keep strong ties with each customer due to its large number (75 million downloads, according to the *Mozilla* website). Also, *Mozilla* does not report any information back to the Mozilla Foundation, by use of feedback servers (such as *Apache*'s TraceBack) or another form of automatic post-installation feedback.

### 2.3.9 Apache's HTTP Server

*Apache*'s development began in February 1995 as a combined effort to coordinate existing fixes to the NCSA http program. It soon became a popular and successful open source product [81]. At present it is the most used HTTP server software for servers on the world wide web. The product is mostly used by web server maintainers with some technical knowledge, and therefore *Apache* does not have many of the key practices for the features of local configuration management and feedback management. Another reason for the absence of these key practices is that the *Apache* HTTP server is used for public websites, where automatic deployment and feedback could compromise security.

## 2.4 Evaluation of the Process Areas and Features

This section discusses and describes the impact and effort required for making improvements in each process area. These results have been generalized for the eight cases and are summarized in Table 2.6. Each of the following paragraphs describes the problems and the availability of tools per feature.

The **release process management** feature describes the skills of a company to plan and manage their product and update releases. The maturity of a software vendor can often be established by looking at the key practices for this feature, because it is essential to all other vendor side process areas. Primarily, to have all key practices for this feature, the vendor should manage its software with a PDM system. By doing so the vendor is forced to manage all secondary artefacts, such as manuals, boxes, and DVDs, as explicit as the product itself. Both open source cases do not have a release planning that is adjusted to customer requirements, due to less market pressure for early releases. The tools used to support the key practices in this process area are numerous, and it contains tools that support software configuration management and many proprietary tools that support software artefact and product management.

The **product knowledge management** feature is strongly represented for all cases. The vendor must manage the relationships of its products to other products in both a human readable format, for the support, sales, and development department, and a computer readable format to allow for automatic conflict detection and even automatic

dependency resolution. Also, the availability of past product releases is required such that customers can download older versions. *Mozilla* does not provide such functionality, due to the fact that the source of their products is always available. The downside of this is that a customer can never download older versions of the software automatically to be used with a set of other applications, without having to build the source code. Another example is *ERPComp*, which only provides the latest version of the software and no other, such that users will always use the latest version. The question remains whether a vendor wishes to provide customers with more flexibility, or whether this simplification and therefore cost saving method does not scare off customers. Many tools are available for knowledge management and distribution, however, each organization has its own channels for distribution.

The **delivery methods to customers** feature is dependent on many different factors, such as bandwidth, network policies, security, and infrastructure. Coverage of all key practices in this process area is rather weak, with the automatic pull key practice as an extreme. To improve in this process area, a software vendor must carefully review whether the software architecture and the vendor organization itself do not restrict customer communication. Integration of the CRM system throughout the complete organization is required to gain serious improvements in this area. There are some tools available that are already supplying such integration, though a lot of customization is required [148].

The **customer side distribution** feature is dependent on the format of deployment and installation packages, the product software architecture, and possible storage locations. The key practice to allow a customer to use any medium for deployment enables customer organizations to freely deploy software using its proprietary methods of deployment. An example encountered in the cases is when customers request for Microsoft Installer packages (msi) because their internal deployment and distribution tools require msi packages. Since the customer does not allow each user system to go on-line individually and download the latest updates from the vendor but forces them to download patches and products locally, much bandwith is saved. Tool support is found in package managers, such as rpm-update, Portage, and Microsoft's open source project Wix [130].

**Environment checking**, a feature of deployment, requires the deployment application or software product to first scan the system on which the update will take place, for the availability of required components and possible resource constraints such as disk space. If such constraints or missing components are encountered, these issues must be resolved automatically. There are not many tools available (besides package managers) that can support these key practices, mostly due to the complexity and number of different deployment environments.

The **Local configuration management** feature is highly dependent on the operating system and deployment tools used by the customer. Some deployment tools have integrated the build process into the management of software packages [33], whereas other tools are primarily focused on copying of files from one location to another, such as InstallShield [58]. Implementing the key practices in this process area requires large development efforts and changes to the software architecture, such as the rollback key practice, which is generally not implemented because changes to the data model cannot be rolled back [64] [62] without a versioned database

management system. Subsets of the key practices in this process area are often covered by the operating system, such as the deployment capabilities of Gentoo's Portage or the registry and the deinstall key practices for Microsoft Windows. Customization key practices are hardly represented in the presented case studies, showing a large opening for product and service quality improvements. Customization management requires heavy development effort and integration with the CRM system to store customer configuration settings, such as network architecture, used database system, and operating system. This information is used to deliver the appropriate updates and fixes to specific customers and to perform market and requirements research. The backup of data key practice is usually provided through commercial database management tools, and therefore the results presented in Table 2.4 might be misleading. However, providing a mechanism to backup all external data and configuration information with the press of a button is a valuable key practice, because customers are allowed to perform quicker and more reliable backups. Some of the key practices of **customization management** are supported by development platforms such as J2EE that forces developers to store configuration information in external XML files.

To improve the feature of **deployment process automation** the two previous features must be combined. Both automatic dependency resolution and local configuration management must be automated to perform automatic updates and deployments. Tools that support such automation are not widely available and an automatic update and deployment solution requires a specially adjusted software architecture.

**License management** consists of both license management on the customer side and the vendor side. Customer side license management is usually easy to implement, since many license management mechanisms, such as to renew the license, are already implemented under the bonnet. To provide the key practices within customer side license management development effort is required. Vendor side license management requires changes to the CRM system, such that it can store and distribute licenses, and requires organizational changes, such that license generation and renewals are done automatically by the sales department. Improving vendor side license management requires little effort, due to the fact that some type of CRM and license management is usually already present in an organization. Customer side license management solutions exist, such as ManageSoft's [75] software management suite. Dedicated vendor side license management systems, such as Hasp [4], provide many key practices that are required in this process area.

Improving in the area of **sales and lead management** may require changes to the product, such that the products are used to communicate with the user, by form of a daily pop-up, or a message to the sales department if a user attempts to use an unpurchased feature a number of times. The reasons for this key practice are numerous. Often a customer will have an old version of the software running, requesting outdated and expensive support on old (and even buggy) functionality. Also, when customers are not aware of the newest functionality within a product, they might opt for a competitor who simply told the customer first about one market sensitive characteristic of their product. Pilot customers can pre-evaluate the software and have a say in the final set of requirements and in commercial cases use the product at a discount price. Pilot customers increase market awareness for the vendor and improve relationships with

some of its primary customers. New functionality can also be made available to customers using temporary licenses, which allows customers to test new functionality before actually purchasing it. This process area is limited by trust and network infrastructure issues. It will require some changes to the CRM system to get messages to the right customer organizations. Some commercial tools are available in the form of PDM and CRM systems, but once again integration and customization effort are large, and structural communication between the sales and development departments about new product features is required.

The **feedback management** feature is valuable to a software vendor because it will introduce new requirements on the product, show what the most used functions of a product are, and where most errors occur. Though improvements in this process area requires a lot of effort, the products discussed in this chapter already implemented different error logging mechanisms, sometimes even with a "send error report to vendor by e-mail"-button. No commercial tools were found to handle such feedback although some tools such as Mozilla's TraceBack and the components presented in the work by Renaud et al. [102] provide similar functionality. Network infrastructure, privacy, and security should be taken in consideration carefully when improving these areas. Effort to improve this feature is low, whereas the implementation of feedback is highly profitable. Such feedback reports can even be linked to customers, informing the vendor of its customers' configuration. This information is used by the support department to determine a customer's configuration, but also to inform the development department of "proven" configurations. A well-known example of feedback error reporting is the feedback function in Microsoft's Windows XP. However, other mechanisms are imaginable [41], such as usage reports (which can also be used for pay-per-usage scenarios) that can help improve the knowledge about which functionalities are most used by customers.

## 2.5 Discussion

Now we put the key process area of CCU up for discussion. The first question that needs to be answered is whether a software vendor's success relies on its customer relationships. In the commercial cases encountered and presented in this chapter 50%-70% of their yearly revenue was coming from existing customers, which in our view shows that customer retention and the maintenance of relationships is essential to survive in the current industry. In the case of open source products, where many of the users of the product are also developers, testers, and quality assurance team members, the same premise on customer relationship management holds. Since CCU is a customer focused process, the improvement of these processes will lead to better customer relationships and possibly a higher customer retention rate. By applying the CCU model onto the eight presented cases, Tables 2.2, 2.3, 2.4, and 2.5 lead to the following observations:

- Software vendors focus insufficiently on customer side configuration management
- Licensing and contract integration is rare

- Software vendors do not focus on deployment and usage feedback
- Software vendors neglect explicit product knowledge management

With these observations and customer retention, product quality and quality of service in mind, a number of conclusions can be drawn.

Even though some of the cases reported that up to 15% of their deployments failed at the customer side, Table 2.4 shows that software vendors do not implement key practices in the area of **customer side configuration management**. The most commonly reported causes for deployment problems are faulty configurations, incompatible updates, and customizations. By implementing the key practices stated for the deployment process, these problems can partly be avoided [142].

Also, vendor side **license management**, which includes contract registration and automated license creation, is not sufficiently represented in the cases. This area leaves open an opportunity for an integrated contract and license management tool that plugs into any CRM system. For obvious reasons license management is not such a large issue in open source software, although some sense of consciousness throughout the industry about open source licenses would improve customer organizations' awareness of their acquired (open source) products. Often redistribution rules are not respected, simply because customer organizations are not aware of them. Another example where licensing is not such a big issue are the B2B (business-to-business) software vendors we researched. *CMSComp*, for example, provides unencoded XML license files to its customers that could easily be altered, and defends that choice by saying that trust in B2B markets is more important, since *CMSComp* will simply offer the functionality to them if the customer chooses to change the license file.

All cases do not sufficiently implement the key practices of **usage and deployment feedback**. Such feedback, however, is used to gather essential product knowledge, such as product incompatibilities, common user errors, and usage statistics of product functionality. This knowledge is translated into requirements for future products and product fixes. For the two open source cases customer feedback seems to be underrepresented, whereas deployment and user feedback seem to be integral parts of the open source development process. Open source software products, however, can improve their development process by implementing automatic usage and error feedback as well.

To process customer usage feedback, to store product compatibilities, and to handle the huge volume of requirements on a product, a software vendor must have a **high-level product knowledge infrastructure**. Such a product knowledge infrastructure is used to communicate product information throughout the development department, the organization, and even its customers.

Interestingly enough, many key practices in the areas of customization management, internal product relationship management, and product data and software configuration management are an integral part of software product line development [16]. Especially the explicit manner in which products and product configurations are managed by the PuLSE approach [9] sets an example for product software companies. The combination of the concepts of this research and PuLSE paves the way to an integrated software product data management system that manages all artefacts and information for a software product family.

In our search for tools that can provide the key practices presented in this chapter and in chapter 5, undiscovered product niches have been encountered. To begin with it seems that there are no product data management systems that explicitly manage licenses, software products, their fixes, and their patches, in such a way that customers can log in and download them. Secondly, feedback sending and feedback analysis applications seem to be in short supply. Finally, operating systems and deployment tools [21] generally do not support the key practices for local software configuration management.

If anything can be learned from this research, it is that software vendors must integrate their CRM, PDM, and SCM [148] systems to automate the processes related to CCU. Such automation provides more efficient methods to perform repetitive tasks such as license creation, license renewal, product updating, error reporting, usage reporting, product release, and manual configuration tasks, such as backups. The second main lesson is that usage of feedback reports supplies software vendors with the largest test bed imaginable, and therefore deserves more attention. The presented CCU model can be used as a guideline for software vendors or for the development of a software manufacturing and software product data management system.

## 2.6 Future Work

The presented material allows for a larger evaluation of the customer configuration updating process. Next to the case studies we will perform in the future, we are planning to build a benchmark site where software vendors can evaluate their own key practices and position themselves in the market. This evaluation technique, however, requires a new classification of software product companies, which is used to further analyze the results from such research. As a continuation on one of the cases we have been offered the opportunity to implement a subset of the presented key practices within that organization. We will investigate the implementation of these key practices and use it to validate the results of this research. We are currently negotiating with several software vendors whether the concepts shown by Elbaum et al. [41] can be implemented within their products, to evaluate the usefulness of such functionalities and to get more practical experience with field data gathered from customers.

| **Release Key Practice** | **Software product vendors** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ERPC | CMSC | FMSC | HISC | CADC | OCSC | Mozilla | Apache |
| The organization has pilot customers to test early releases | ● | | | | | ● | ● | ● |
| Release planning is published internally | ● | ○ | ● | ○ | ○ | ● | | ○ |
| Restrictions on configurations due to internal components are managed | ○ | ● | ● | ○ | ○ | ● | ● | ● |
| Restrictions on configurations due to external components are managed | ○ | ○ | ● | ○ | ○ | ● | ● | ● |
| The tools that support release, delivery, and deployment are managed | ● | ● | ○ | ○ | ○ | ● | ● | ● |
| There is a formalized release procedure | ● | ● | ● | ● | ● | ● | ● | ● |
| Releases are stored in repositories (that mirrors the SCM) | | ● | ● | ● | ● | ● | | ● |
| The formalized release planning is adjusted to customer requirements | ● | ● | ● | | | | | |
| Sending information regularly to customers | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ |
| Vendor uses all possible channels for informing customers | ○ | | | | | ● | ○ | ● |
| Legend: ●: Currently implemented; -: Not applicable; empty: Not implemented<br>○ Requires some manual steps, yet would be easy to automate; | | | | | | | | |

Table 2.2: Release key practices

| **Delivery Key Practice** | **Software product vendors** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ERPC | CMSC | FMSC | HISC | CADC | OCSC | Mozilla | Apache |
| Automatic pull is available | ● | | | | | - | ● | |
| Delivery through any medium (Internet, DVD, Floppy) | ● | ○ | ○ | ○ | | - | ● | ● |
| Download site abstraction | ● | | ○ | ● | ○ | ● | | ● |
| Sending information regularly to customers | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ |
| Vendor uses all possible channels for informing customers | ○ | | | | | ● | ○ | ● |
| Legend: ●: Currently implemented; -: Not applicable; empty: Not implemented<br>○ Requires some manual steps, yet would be easy to automate; | | | | | | | | |

Table 2.3: Delivery key practices

| **Deployment Key Practice** | **Software product vendors** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ERPC | CMSC | FMSC | HISC | CADC | OCSC | Mozilla | Apache |
| Configuration completeness checking of external components | ● | ○ | ○ | | ○ | ● | ○ | ● |
| Automatic resolution of dependency issues | | | | | | | ○ | |
| (Semi-)automatic local update process | ● | | | ○ | ○ | ● | ● | |
| Rollback from an update is possible | | | | | | ● | | ● |
| Rollback from an install is possible | | | | | ○ | ● | | |
| Updates require no downtime | | ○ | | | ○ | ○ | | |
| Test, acceptance, production environments | ○ | | | ○ | | ● | ○ | ○ |
| Updates can cope with local customizations | ○ | ● | ○ | ○ | ○ | - | ○ | |
| External configuration to allow trace | | ○ | | | | ○ | ○ | ● |
| Integrity checks of SW artefacts at customer | ● | ● | | | | - | ● | ● |
| Detection and exploration of customer environment | ○ | | | | | - | ○ | ○ |
| Data backups are done through the product | ○ | ○ | ○ | ○ | | ○ | | |
| Legend: ●: Currently implemented; -: Not applicable; empty: Not implemented<br>○ Requires some manual steps, yet would be easy to automate; | | | | | | | | |

Table 2.4: Deployment key practices

| **Activation and Usage Key Practice** | **Software product vendors** | | | | | | |
|---|---|---|---|---|---|---|---|
| | ERPC | CMSC | FMSC | HISC | CADC | OCSC | Mozilla | Apache |
| Licenses are coded | ● | | ○ | ○ | ○ | ○ | | |
| Licenses have an effect on software | ● | ● | ○ | ○ | ● | ○ | | |
| Licenses are managed explicitly by customer | ● | | ○ | | ● | | | |
| Possible to renew licenses automatically | ● | | ○ | | ○ | ● | | |
| Licenses are generated using contracts | ● | ○ | ○ | | | | | |
| Temporary licenses are distributed | ● | | | | | | | |
| Awareness of customers configuration | ○ | ○ | ○ | ○ | ○ | ● | | |
| Feedback from a user is possible through the product | | | | | | ● | ● | |
| Usage reports are created | | ● | | | ○ | ● | | ○ |
| Legend: ●: Currently implemented; -: Not applicable; empty: Not implemented<br>○ Requires some manual steps, yet would be easy to automate; | | | | | | | | |

Table 2.5: Activation and usage key practices

| | Software Architecture | Development Effort | CRM Impact | organizational Impact | Available Tools | Change Project Size |
|---|---|---|---|---|---|---|
| **Release** | | | | | | |
| Release process management | none | none | large | large | SCM systems, no PDM | medium |
| Product knowledge management | none | none | medium | medium | SCM systems, no PDM | medium |
| **Delivery** | | | | | | |
| Delivery methods | some | none | some | some | none | Small |
| Customer side distribution | some | medium | none | none | package deployment tools | medium |
| **Deployment** | | | | | | |
| Environment checking | medium | medium | some | none | some deployment tools | medium |
| Local configuration management | large | medium | none | some | some tools, operating systems | medium |
| Deployment process automation | large | large | some | some | none | large |
| **Activation and Usage** | | | | | | |
| License management | some | medium | large | medium | some | medium |
| Feedback management | some | heavy | medium | some | some | medium |

Table 2.6: Process area impact assessment

# CHAPTER 3

# A Case Study in Mass Market ERP Software

*The maintenance of enterprise application software at a customer site is a complex task for software vendors. This complexity results in a significant amount of work and risk. This article presents a case study of a product software vendor that tries to reduce this complexity by integrating Product Data Management (PDM), Software Configuration Management (SCM), and Customer Relationship Management (CRM) into one system. The case study shows that by combining these management areas in a single software knowledge base, software maintenance processes can be automated and improved, thereby enabling a software vendor of enterprise resource planning software to serve a large number of customers with many different product configurations.*[1]

## 3.1 Integrated Development and Maintenance

The complexity of the maintenance, release, and deployment processes of product software is a result of the enormous scale of the undertaking. There are many customers for the vendor to serve, who all might require their own version or variant of the application. Furthermore, the application itself will consist of many (software) components that depend on each other to function correctly. On top of that, these components evolve over time to answer the changing needs of customers. As a consequence, the release and deployment of these applications take a significant amount of effort and is a time consuming and error-prone process.

To date product software is a packaged configuration of software components or a software-based service, with auxiliary materials, which is released for and traded in a specific market [132]. Customer Configuration Updating (CCU) is defined as the combination of the vendor side release process, the product or update delivery

[1] This work was originally published in the Journal of Software Maintenance and Evolution: Research and Practice, entitled "Integrated development and maintenance for the release, delivery, deployment, and customisation of product software: A case study in mass market ERP software" in 2006 [141]. The work is co-authored with Sjaak Brinkkemper, Gerco Ballintijn, and Arco van Nieuwland.

process, the customer side deployment process, and the activation process [143]. To alleviate these processes I envision a Software Knowledge Base (SKB) that contains facts about all product artefacts together with their relevant attributes, relations and constraints [19]. In this way, high-quality software configurations can be calculated automatically from a small set of key parameters. It also becomes possible to pose "what-if" questions about necessary or future upgrades of a customer's configuration [142]. The need for a distributed software knowledge base comes from literature. Meyer is the first one to introduce the concept of a centrally available software knowledge base [80]. Others, such as Klint et al. [70] and Robillard et al. [104] emphasize the need for explicit knowledge management during development and maintenance.

Exact Software (ES)[2], a Dutch software manufacturer that serves 160,000 customers worldwide, has implemented an SKB to manage and improve its software maintenance, release, and deployment processes. The SKB used by ES is implemented in their own commercial product called e-Synergy. In this article I show that ES successfully supports its large customer base with an integrated product data management, software configuration management, and customer relationship management system, thereby alleviating the process of software product maintenance. The article describes how the processes of development, release, and deployment have been improved by integrating processes that were previously managed by utilizing different isolated systems. The article also demonstrates how a central software knowledge base, containing all the relevant knowledge about software products, is implemented and used to support the processes of software maintenance. Finally, the article describes four principles employed by ES to deal with general complexities in the software engineering discipline with respect to software maintenance.

Figure 3.1 displays the overall architecture of the integrated CRM, PDM, and SCM systems in e-Synergy. The integration of these systems enables efficient maintenance of software configurations at a customer site. The CRM system contains a contract module, in which all products that have been sold to a customer are stored. Each contract applies to a product that can be downloaded and activated by a customer. The products stored in the PDM system are associated with their corresponding artefacts in the SCM system, which enables a customer to download the correct files that are required for a product update or deployment. Finally, the PDM system generates a list of files on the vendor side that is compared to the list of files on the customer system. When differences are encountered the required files are downloaded by the customer automatically to update the customer configuration.

The software maintenance processes on the vendor side also have improved by the integration of different systems. The integration of the SCM and PDM systems allow product managers to quickly oversee whether work still needs to be done on deliverables before the next release. The integration of the SCM and workflow management system enables traceability of changes on deliverables, thereby improving product quality. The integration of the SCM and PDM systems also allows for quick deployment and testing of test versions for the quality assurance department. These

[2]Please note that this research took place in 2003 and 2004. Some of the presented results no longer represent daily ES practice

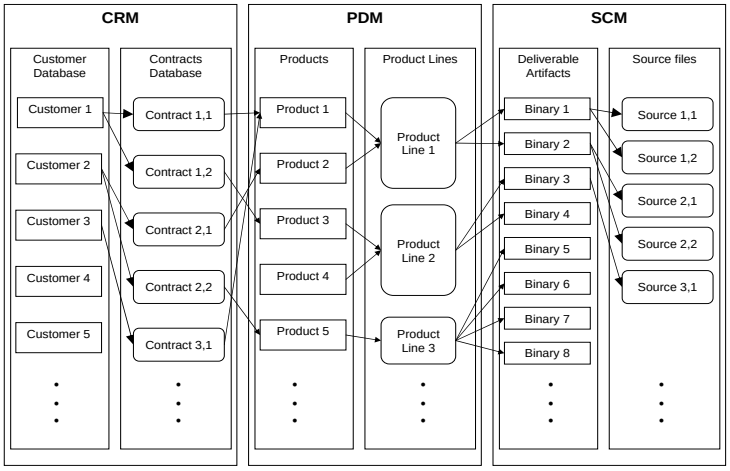



Figure 3.1: Integration of CRM, PDM and SCM

and other improvements are discussed further in Section 3.3.

The rest of this article is structured as follows. Section 3.2 describes the objective of our research at ES and a motivation. Section 3.3 describes ES and the tools it uses to integrate its SCM, PDM, and CRM systems. Section 3.4 describes the maintenance processes of the products at the vendor site and of the configurations at the customer site. Section 3.5 discusses the lessons learned from ES and what functionality I feel is lacking in Es's SKB. Related work is presented in Section 3.6 and finally Section 3.7 concludes our article with a discussion.

## 3.2 Research Approach

### 3.2.1 Problem Overview

Two important parts of the maintenance process are release and deployment of software. The release and deployment processes for a software product involve a large amount of risk and effort for a software vendor. These processes, however, have been documented insufficiently in literature. The SWEBOK [1], for instance, gives a generic description in the SCM chapter of the processes of release and delivery. The Capability Maturity Model (CMM) [98] also does not provide adequate descriptions for these processes, which is explained by the fact that the CMM does not focus on product software specifically. Attempts have been made in the release candidate of the IT Service CMM [86], although the IT Service CMM also does not provide an elaborate description for the processes of release, delivery, and deployment. Clearly, even though

there is a need for process definitions, there are no adequate process descriptions available. The goal of our research is to simplify the software release and deployment effort. I propose to do so by managing all the knowledge about a software product explicitly. The explicit management of software knowledge enables the evaluation of "what if" scenarios, such as, what will happen to the current configuration of customer X, if she upgrades application component Y? These evaluations help in assessing the risk of the deployment process, and these assessments, in turn, improve interaction between customer and software vendor because the vendor can guarantee whether a combination of components can function correctly together.

Managing software knowledge is, however, only part of the story. The software still has to be delivered to customers. I aim to support dynamic delivery of software via the Internet, both in the form of upgrades and of full packages. The previously mentioned product and component knowledge is used to compute the difference between the existing software configuration at a customer and the desired configuration. This difference is used to create required upgrades [124]. Central to the maintenance activities I envision, is the software knowledge base. This software knowledge base can be seen as an integrated SCM/PDM/CRM system that stores all information about all the artefacts that are part of the applications life cycle. The SKB stores the information of all available applications in all available versions at the vendor site, whereas at the customer site the SKB stores information about the installed applications, application settings, and configurations. Both the vendor and the customer can request or receive information from the configuration of the other party, such as regular updates, product information, usage and error reports, product knowledge, and licenses.

As part of our research, I have been conducting case studies [62] [7] at product software companies to evaluate the state-of-the-practice of software vendors in the Netherlands, such as ES. ES is relevant to our research because ES has implemented one of its own products, e-Synergy, to support the processes of release and deployment and to function as an SKB, which partly validates the theory that an SKB can improve software release and delivery.

### 3.2.2 Exact Software

ES is a manufacturer of software for accounting and enterprise resource planning (ERP), based in Delft, the Netherlands. Since its founding in 1984, ES has established an international customer base of over 160,000 customers, mainly in the small to medium enterprise sector. Through autonomous growth and a number of acquisitions the number of employees has grown to 2025 in 2004 (see Table 3.2.2). Twenty percent of these employees are active in the development of software on several international locations with most of them (180 employees) working in Kuala Lumpur.

A typical application sold by ES is Exact Globe, a back-office application that integrates business processes, such as finances, logistics, PDM, and CRM. A recent product is e-Synergy, a front office application that provides organizations with real-time financial information, multi-site reporting, and relationship and knowledge management capabilities. Employees, customers and partners are provided with real-time access to information across an entire organization.

Based on more than 20 years of experience in developing software products for

| Department | FTE | Percentage |
|---|---|---|
| Support | 546 | 27,0 |
| Services | 263 | 13,0 |
| Sales and Marketing | 445 | 22,0 |
| Finance and Administration | 142 | 7,0 |
| Staff and General Management | 223 | 11,0 |
| Development | 294 | 14,5 |
| Quality Assurance | 96 | 4,7 |
| Release and Deployment | 15 | 0,7 |
| Total | 2024 | 100 |

Table 3.1: ES full-time employment (2004)

the small to medium enterprise market, ES enforces four main principles for product development:

- **Uniform architecture -** All software developed by ES has a three layered architecture. The user application layer (a browser or a standalone client), the application server layer (containing the business logic), and the database layer.
- **One-X -** ES has developed a strategy for developing its ERP software, called One-X, which aims to develop all software around one single instance of truth, making the data available to all ES applications, such that the data can be created and provided to all the stakeholders. The idea behind One-X is that data needs to be entered just once and that extensive navigation is possible through integration.
- **KISS -** To support such a large customer base within such a complex problem domain, ES follows the principle of KISS (Keep It Small and Simple) for its development process. The use of KISS within ES has resulted in a development cycle where a fully functional prototype is produced by a spearhead team first. Once the prototype is released, the product enters a maintenance cycle and the product can then only be changed by the maintenance team through well-defined maintenance procedures. All procedures are monitored by a large quality assurance team, as seen in Table 3.2.2. These procedures allow ES to keep the maintenance of its products simple and controlled.
- **Eat your own dogfood -** ES uses its own software products to support internal processes, which is called “eat your own dogfood” by Microsoft [28]. This internal use provides the maintenance department with early bug reports and feedback.

These principles are enforced with one fact in mind. Revenue statistics from ES show that since 2003, the largest stream of income cines from maintenance contracts, whereas previously money came for the larger part from license sales. This made ES management realize that investment in the maintenance process was required.

### 3.2.3 The Case Study

There were three reasons for performing the case study at ES [64]. To begin with, I wished to prove the hypothesis that explicit management of software knowledge on both the customer and vendor site can improve the CCU processes. Secondly, ES provided us an example SKB and showed how an SKB can be applied. They also allowed us to review the reasons for implementing an SKB to support its processes. Finally, ES gave us an opportunity to see the advantages and disadvantages of using an SKB in daily life. During the three month case study, facts have been collected using several sources:

- **Interviews -** To study ES and confirm our hypotheses, interviews were held with the people responsible for the development and usage of the e-Synergy product.
- **Studying the software -** ES granted an academic license for the e-Synergy software. This license helped to gather many facts by examining, using, and experimenting with the software.
- **Document study -** Many of the documents found in the document management system of e-Synergy supported the research and gave an in-depth view of the ES maintenance processes.
- **Direct observations -** Since our research took place at ES's International Development department, I was able to directly observe and document day to day operations.

The validity threats to our descriptive case study are construct, internal, external, and reliability [133] threats. With respect to construct validity, the protocol used for this study was applied to a number of case studies [62] [7], which was guarded by closely peer reviewing the case study process and database. To create a complete and correct overview, the development and release, delivery, and deployment processes have been documented extensively. The internal validity was threatened by incorrect facts and incorrect results from the different sources of information. The interviews that were held consisted of two sessions, one to explore and elaborate, and one to cross-check documentation found in the document management system of e-Synergy and to confirm facts stated in other interviews.

With respect to external validity, a threat is that this case is not representative for the Dutch software vendor market and the ERP domain. This threat was dealt with by comparing general information from ES to other vendors that are active in the Platform for Product Software [45], a national organization that aims to share knowledge between research institutes and software vendors, with over 100 members. The comparison shows that ES, though being one of the largest ERP manufacturers in the country, is similar to other ERP manufacturers. Finally, to defend reliability similar results would be gathered if the case studies were redone, with one major proviso, which is that many of the improvements proposed in the case study report, published after the case study, were implemented by ES.

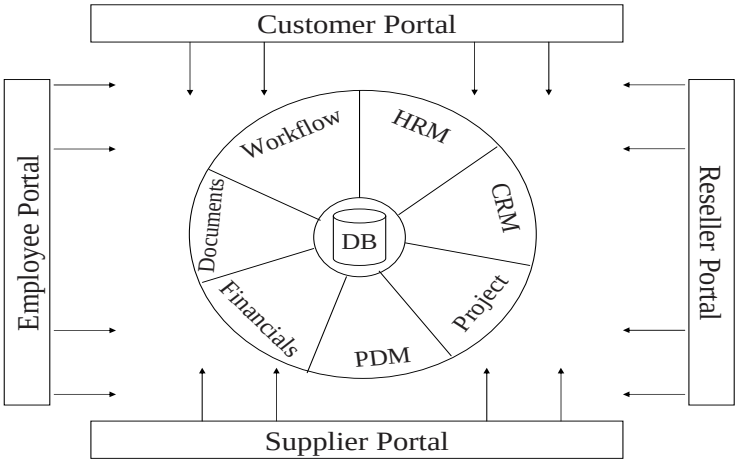



Figure 3.2: Abstract Architecture of e-Synergy

## 3.3 The SKB and its use within ES

ES uses its proprietary product e-Synergy, to support all its business processes. ES uses all e-Synergy modules for its activities, such as document management, workflow management, and financial accounting. An implementation of e-Synergy provides four optional Internet portals (see Figure 3.2), which are used to provide customers, employees, resellers, and suppliers with their specific views on the data. With respect to maintenance e-Synergy is used to support two forms of maintenance. On the one hand e-Synergy is used to support the maintenance department in performing the product composition, development, bug fixing, and workload division amongst developers. On the other hand e-Synergy is used to supply customers with an interface to the latest releases of products.

The SKB used by ES, e-Synergy, is a front office application that integrates seven modules: project management, workflow, human resource management, document management, CRM, logistics, and financial activities, as seen in Figure 3.2. Through the One-X architecture, each module can use the data in other modules enabling users to easily navigate from one item to another. The logistics module of e-Synergy is a PDM module that manages conventional products, the CRM module stores information about customers, and the project and workflow modules are used to distribute activities among personnel. Development workflow activities are classified as bug reports and change requests and can be attached to other workflow, documents, and deliverables. These attachments are used to quickly produce reports on how many tasks are still attached to a deliverable or a document. Since all tasks have different defined levels of

impact, projections can be made about the amount of work and the cost associated with that work, which enables status reporting.

Before e-Synergy was implemented, ES was utilizing different isolated systems for the processes of software maintenance, release, and delivery, such as daily build servers and conventional concurrent versioning management tools for SCM. ES experienced many problems within the setting of isolated systems. To begin with, many of the tasks performed included the duplicate entry of data into different systems. A second problem experienced by ES was that deliverables were not managed explicitly, delaying deadlines and often producing incomplete sets of deliverables for customers. The final problem relevant to this case study was that multiple worldwide departments needed access to the software repositories twenty-four hours a day. To solve these problems ES implemented their own web based e-business product, e-Synergy.

### 3.3.1 SCM

The SCM system in e-Synergy consists of five repositories, in which five concurrent releases of all deliverables and corresponding source code are stored, as shown in Figure 3.3. Each of the five repositories contains a release of all the source files, help files, binary files, executables, resources, and SQL scripts, for one product, such as e-Synergy or Globe. Periodically, depending on the quality criteria for each repository, the full repository is manually promoted (copied) from one repository to another.

All 294 developers perform their operations, such as committing, on the development release stored in the D repository. When all uploaded bug fixes and new functionalities have been checked in by the programmers on the release stored in the D repository, that release is promoted to the C repository on a weekly basis, overwriting the release previously stored there. The quality assurance department checks the release and reports errors back to the development department. Every 10-20 weeks QA freezes the C repository for three weeks to check that release intensively. After approval from quality assurance, that release is copied to the B repository. The release stored in the B repository is, if possible, used internally by all ES personnel and is thereby thoroughly tested. This testing again generates new bug reports or functionality requests.

When the release stored in the B repository is deemed stable enough by primary internal users, such as the director of ES Finance and Administration, the release is copied to the A repository, which is open to external pilot customers who report their experiences back to an experienced support employee. After the release in the A repository has been used for a minimum of eight weeks, the release is copied to the NULL repository containing the official product release, which is sent out to customers on CD-ROMs or through the Internet. To remove the complexities introduced by the concurrent versioning systems, ES now uses one single development version to manage software artefacts. Versioning of files is therefore not possible, which is different from common practice, but one of the results of the KISS strategy of ES. New functionality releases occur approximately four times a year.


Figure 3.3: Repository Promotion Scheme and Promotion Periods

### 3.3.2 PDM

When ES started designing e-Synergy, ES decided that their commercial PDM system could just as well manage its software products and control its (software) product deliverables. PDM systems generally implement a classification of artefacts to support reuse [60]. The PDM system implemented in e-Synergy makes use of atomic entities called "items" by ES. An item is used to represent any business item, such as a promotional sweatshirt, a printout of a manual, or a software product's executable. Items are categorized, of which the relevant categories for this case study are sales items, source items, and deliverable items.

- **Sales items -** ES uses sales items to encapsulate all sellable goods. A sales item is a service agreement, a manual, a software product (including its paper manual and CD-ROM), or any other good sold by ES. From each sales item a bill of materials can be generated, stating what items are necessary to complete the product (such as the deliverables for a software product).
- **Deliverable items -** Deliverable items depend on source items, even if they are simply direct copies of those source items. Deliverable items include digital manuals, resource files, library files, and executable files.
- **Source items -** Source items are source files that are required to create a deliverable. Source items are source code files, resource files, etc. Companies producing conventional products use the source items to store their basic raw materials and resources with which they create their products.

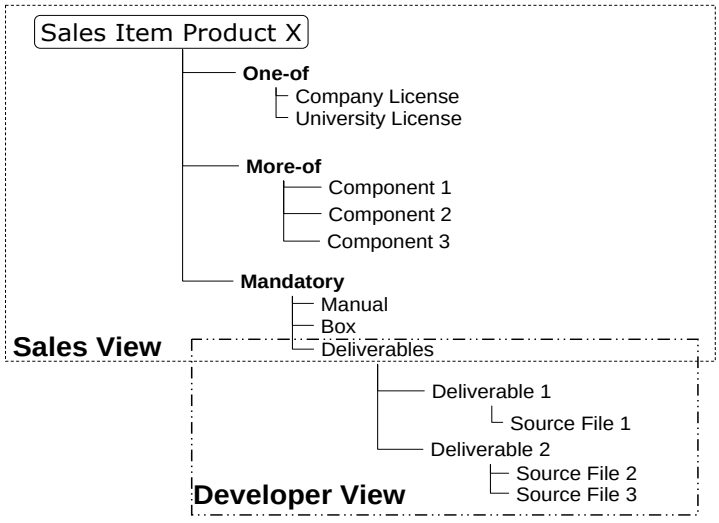



Figure 3.4: Sales and Developer Product View of an Example Product

ES's products are represented in e-Synergy in different ways at the development sites of ES, whereas instantiation information for products at customers is stored in the contract management facility, the product data management facility, and locally at the customer's site. These two different views are based on the assumption that sales personnel do not need to know all the implementational details, whereas developers are not concerned with sales knowledge. An example is that the sales department sees products as decomposable modules that can be sold separately, whereas development sees the product as a large set of deliverables containing all modules, which are later activated or deactivated at runtime by the license file. Figure 3.4 displays a generic product structure and the two different views of that structure:

**Sales View -** From the point of view of the sales department it is not relevant what deliverables and sources look like. For this view it is required to know what a product costs, what options there are to a product, what kind of sales agreements are possible, and what materials make up a product. Sales personnel thus share no interest in source files. A product, consisting of sales items connected by the one-of, more-of, mandatory, and optional relationships, is instantiated by binding these relationships. The binding of these relationships corresponds with the product information stored in a license file. The relationships defined here are similar to the relationships defined for feature diagrams [67, 123].

**Development View -** Developers are concerned with deliverable files and source files. As developers always work in the context of a product, and they know the complete structure of that product, however, developers usually do not use sales items. A developer considers a product as a complete set of deliverables, therefore there are

only two relationships available for the development view, the Sources relationship, meaning the file is a source file for some parent deliverable item, and the Deliverables relationship, meaning the file has corresponding sources.

The ES PDM system stores lists of all the mandatory deliverables for a product and these deliverables have sources as their children. This results into the fact that dependencies amongst deliverables are not explicitly stored. The ES Product Updater [147], a proprietary tool that is used to update all ES products, uses the list of deliverables for products to compare that to the list of installed files at the customer. Also, even though the PDM relationships such as more-of and one-of are used to deliver packages of only the purchased components to a customer, all deliverables for a product are delivered to a customer. The e-Synergy product currently is represented by approximately 8000 source items, tools, and deliverable items in ES's PDM system. The largest part consists of source files that are not delivered to customers. Other items are tools that are required to produce the product, but are not delivered to the customers, such as developer tools. The deliverable items are executables, dll files, sql definitions, manuals, boxes, CDs, and promotional material. ES's PDM system stores artefacts in product lines. Product lines are collections of products that share items (files) to support reuse.

### 3.3.3 CRM

With respect to customers ES has attempted to increase customer contact, scale down support, reduce the complexity of the delivery process of software products, and reduce piracy of its products. ES believes that product (experience) improvement by intensive customer contact will retain more customers [68]. Customers log into the ES portal to see their contract information, to see the status of their support calls or bug reports, and to download (renewed) license files. Because customers are expected to visit the ES customer portal regularly, customers are notified of new NULL releases and other products through the ES e-Synergy customer portal.

Customers can use the information portal of e-Synergy to access the CRM system and see the status of their contracts, see their support questions, find new products, and find where customers can download license files to activate their purchased products. e-Synergy's contract management facility stores a link to the customer information, purchased product information, the license file for each product, the version number of the latest sent out version, and a link to customer support calls and service status. The purchased product information lists what variants of products have been purchased. The corresponding license files are only available for download by customers. License files are generated every twenty-four hours, depending on whether a new contract has become available or needs to be renewed. The license file is published on the customer portal of e-Synergy so that customers can download the periodically renewed license file.

## 3.4 Maintenance and the SKB

### 3.4.1 Maintenance at the Development Site

Before the systems of SCM, PDM, and CRM were integrated, the concurrent versioning system RCS [110] was used to support development. Besides RCS, ES used daily build servers, so that the quality assurance department could check the work of the day before. During the implementation of e-Synergy, ES drew up standard requirements for change control, team support, status reporting, process control, and audit control. Aside from these standard requirements, ES had the following non-generic requirements [76] in the area of SCM:

- **Version control -** In the past too many resources were absorbed by legacy support and customers were confronted with complex upgrades and bug fixes. As a consequence of the KISS principle, ES decided to reduce complexity for the customer and development by no longer supporting and storing multiple versions of a product.
- **Configuration support -** Because bandwidth and disk space are relatively cheap and development is expensive, ES concluded that installing the full set of deliverables for a product and activating purchased modules according to a license is more efficient than doing partial delivery. Also, to improve service and product quality, ES wished an automated check of the validity of a product configuration before it is deployed.
- **Build support -** ES previously used separate build servers to build products overnight. That build was tested the next day by quality assurance. As ES grew larger internationally and developers were working on code twenty-four hours a day, there was no down time left for servers to build the software. A new way of partially building was required to facilitate these needs.

ES promotes some key starting points for its development process. To begin with, developers have private ownership of deliverables and source code and they include their compiled deliverables when comitting their source code. This introduces pessimistic locking (where no two developers can be working on one source file) and enables management to assign responsibility for deliverables to one developer specifically. ES uses very strict maintenance procedures and role based workflow for each task such as functional requests or bugs. First the development manager evaluates the task. Once evaluated the developer executes the task. Finally, quality assurance tests whether the task was successful and approves the task.

ES has development sites in multiple time zones covering twenty-four hours, which eliminates the option to perform nightly builds. This caused the decision for building the end product on the developer's workspace. The solution where developers upload their compiled deliverables creates a repository that always contains the most recent build of the software, removing the need for nightly builds. This solution also enables ES to have a latest running version available twenty-four hours a day at all departments worldwide.

- **Manage deliverables -** Previously, source files were the focus of management instead of deliverables. A compiled version of the software created on the build

server and a manual from the manual department were the only deliverables. As product complexity grew, however, ES desired to be able to manage the deliverables individually and attach workflow and documents to deliverables to increase its traceability. Also, previously developers determined, depending on the requirements, what the variabilities of a product would look like. ES decided that sales departments should be able to influence at what level a developer introduces variability.

- **Ease of delivery -** ES wished to automatically update its products with an evolving set of deliverables because ES consultants had often been confronted with complex manual update procedures.

To manage the deliverables and ease the maintenance process at the customer site, e-Synergy's PDM module was employed. The PDM functionalities were implemented as a central system to the process of maintenance. e-Synergy connects the PDM system and the human resources management system to enable ES to assign deliverables to specific developers and hold developers responsible for the quality of their deliverables. Before e-Synergy was introduced, when deliverables were mostly unmanaged, automatically reusing modules for different products was impossible. Since deliverables can now be linked together to form new products, ES can create new products from a standard set of components. Because currently all deliverables are stored explicitly in the PDM system, the Product Updater can automatically retrieve a list of deliverables for a product from the PDM system and install them if necessary. The fact that all deliverables are retrieved this way eases the process of software delivery to customers.

Combined SCM and PDM support is provided by the logistics module of e-Synergy because it stores the product data, the deliverables, and the source code. ES has combined the PDM and SCM systems in e-Synergy because ES believes that building a software product is fairly similar to building physical products and can be done using a commercial and generic PDM system. Developers see the SCM system primarily as one repository in which both the source and their corresponding deliverables, such as executables, are stored. For sales personnel the PDM system primarily consists of physical objects and software objects, to which documents, software deliverables, and workflow activities can be attached. Finally, the PDM system supplies customers with a link to the published repository from which they can download the most recent versions of the artefacts that are part of the products the customers have purchased.

ES uses e-Synergy to manage all tasks with the concepts of tasks and projects. A task has four states, being open, approved (in progress), realized, and processed. These states can be changed by both the assigner and the assignee when appropriate. Projects, which are collections of tasks and subprojects, enable a project manager to assign tasks within a project and view the status of a project by looking at task overviews. A typical task is setup when a bug report is processed by quality assurance to a bug fix request. The request will be assigned by the quality assurance team member to a developer, in such a way that the task comes back to the quality assurance team member after the developer finishes the task.

A typical recurring project is the promotion of a product from the B release to the A release. The release to the A repository requires special care due to the fact

that the A repository is open to pilot customers. This requires the product and all its artefacts to be available and up to date, before a release can actually be promoted. A release project thus consists of different subprojects, such as approval steps, artefact testing, and quality checking projects. An interesting aspect of a product release is internationalization. ES uses generic replaceable terms for its applications, which are replaced by dictionaries for each language in which a product is available. Obviously, these dictionary artefacts need to be complete and available at release time. One product release subproject thus is the translation of the dictionaries. Also, since manuals are delivered with the product, the PDM system must contain a valid manual for each language the product is released in.

### 3.4.2 Maintenance at the Customer

Figure 3.1 depicts a small subset of the information stored in the integrated system. The CRM system stores information on customers and their contracts. The contracts are used to generate licenses and store what product(s) a customer presently has purchased. The products, as stored in the PDM system, are linked to the artefacts that make up that product and that must be deployed at the customer site when the customer owns the rights to that product. Finally, the artefacts are linked to source artefacts that are used to build the deliverables. The SKB implemented by ES consists of the SCM, PDM, and CRM modules in e-Synergy and are integrated through the One-X architecture.

ES has chosen to deliver the full set of deliverables for a product to a customer, abolishing the need for elaborate dependency information among software modules. The reasons for this approach are that development costs for a partial delivery system are high while disk space and bandwidth are relatively cheap. Besides e-Synergy, there is one external tool that performs actions on the repositories. The Product Updater downloads and installs all deliverables for a release from a selected repository. The Product Updater establishes what deliverables are present at the client site and downloads (new versions of) the required files. The Product Updater also has some scripting capabilities to install the application and create and transform the tables in the database.

Before e-Synergy was implemented, ES only used a proprietary product Globe for CRM. However, when maintenance matters had become too complex the overall goals became to reduce complexity of delivery, intensify customer contact, and reduce the cost of the support department. These lead to the following non-generic requirements, which were met by e-Synergy:

- **Facilitate custom solutions -** The ES customer base still depends for a noteworthy part on custom solutions to extend current functionality in ES products. They wanted to reduce complexity of delivery, yet still facilitate custom solutions and extensions to its products, and ES wanted to remove the expensive need for consultants at the customer site to perform an update of the product.
- **Unify licensing and CRM -** ES realized its software was being copied and distributed illegally. To trace back illegal licenses ES wished to link a license

directly to a customer, whereas in the past its licensing was done through license numbers provided with the distribution CD-ROM.

### Custom Solutions

Ever since ES produced standard out-of-the-box applications a Custom Solutions department has been needed to create specific solutions for customers. To do so in e-Synergy a specific messaging architecture has been created to enable the addition of extra components. Custom solutions are created using a dedicated e-Synergy software development kit (SDK). Custom Solutions produces two types of customisations, being building blocks and customer specific solutions. Building blocks are standard customisations that are applicable to specific market niches, such as equipment rental companies or educational organizations. Customer specific solutions are requested by customers and are not generalizable to other customers. Finally, customers can purchase the option to create customisations themselves.

All three types of customisations are built using the same SDK and are facilitated by a messaging architecture. The messaging architecture created by ES is kept as stable as possible, such that custom solutions that worked with older versions of e-Synergy do not break down due to an update. If a customisation changes a file that belongs to the product itself, such that an update would remove the customisation, the customisation developer can mark the file as unfit for update. The Product Updater will then simply skip the file during the update process. Obviously, it is impossible to guarantee that the product remains fully functional after an update when changes have been made to the product itself and therefore ES focuses on communicating regularly with its customers that implement customisations.

When the Custom Solutions department creates a customisation for a customer, that customisation is stored in a Custom Solutions SCM. The dedicated SCM system enables customers to update a custom solution automatically when the Product Updater is run. When a product for which a customisation is built is updated, Custom Solutions cannot cost effectively test the customisation that was made for one specific customer in each possible updated configuration and only building blocks are tested for each new (maintenance) release. However, due to the fact that updates do not break compatibility in the SDK, customer specific customisations generally do not break down after an update.

### Contracts and License Files

Some of the results of the integration of the PDM and CRM systems are the contract management functionalities and the license files. The version number that is stored in the contract management facility for each customer's product is changed automatically, when a customer downloads an update, or manually, when a CD-ROM is sent out to a customer. The version number is used for support purposes, telling the support department what version a customer is currently using. The link to the customer support calls and service status is used to see how many calls a customer has made, and whether a customer is still allowed support. Using the support information leads to a stricter way of dealing with support and results in less support calls.

At the customer site a data file and license file are stored. The data file contains a list of all the deployed deliverables and the license file states all the modules that have been purchased by a customer. The license file also contains information on the expiration date of the license, since licenses need to be refreshed periodically (yearly for most products). Finally the license contains, if available, a pointer to a download location for a custom solution. The download location is used by the Product Updater to update the custom solution for a customer. The deployed data file stores the version number and the install location for each deliverable. When updating the data file is used by the Product Updater, by comparing the deployed data file to the list of available deliverables for a product. After an update the data file is updated to contain all the newest information. The local settings for a product are stored in the database of the client.

In an attempt to reduce piracy a license checking mechanism was implemented. Periodically the license file is renewed and must be downloaded again to keep the product active. Currently there is no data available to prove whether piracy has effectively been reduced. ES is unique because they provide customers a direct link to their individual contracts and their license files. ES is also unique because e-Synergy stores the version number of a product deployed at the customer, improving support. The usage of license files to encapsulate product instantiation information is common in the software industry.

## 3.5 Discussion

The customer base of ES has shown a constant growth over the last 15 years. Related to the area of maintenance and development ES has dealt with this growth by integrating its CRM, its PDM, and its SCM systems. The solution implemented by ES teaches us three lessons.

- **Integrated support systems for maintenance -** The first lesson is that integration of these three systems is highly profitable. The integration has resulted in a reduction of effort required for the processes of maintenance. Explicit management of deliverables with the PDM system has enabled ES to attach workflow to them, thereby providing a software maintenance process which is easy to manage and enables quicker releases. The integration of the CRM contracts facilities and PDM enable ES to quickly and automatically manage the delivery of software and licenses to customers through the Internet.
- **Integrated maintenance of customer configurations and published releases -** Secondly, ES teaches us that by mapping maintenance of configurations at customers onto the published deliverable repositories simplifies customer configuration updating, because this process is reduced to comparing the list of local artefacts against the list of released artefacts.
- **Development simplification -** The third lesson lies in the fact that ES attempts to simplify all processes, thereby eliminating complexities that would normally result in more effort. Their decision to deploy the full set of deliverables has removed the complexities of partial delivery and removed the need for

dependency tracking among modules, enabling ES to focus more on the delivery process.

According to our experiences, delivering the full set of deliverables is common practice for software developers practicing KISS. Another decision with major impact is the decision to remove build servers that would perform daily builds, and introduce the concept of developers committing deliverables with their sources. ES has also chosen to build all its products on one universal data model, the One-X architecture, enabling applications to share data among each other. Finally, ES uses a maintenance cycle instead of a development cycle to improve its software. There are two advantages to the maintenance cycle. First, it reduces complexity for developers and ensures quality because developers do their activities in a maintenance cycle with predefined workflow. Secondly, the workflow module stores the processed workflow, making activities traceable.

There are downsides to the strategies employed by ES as well. ES keeps track of dependencies among source modules only by adding textual notes to sources, which are hard to maintain. Another downside of the simplification strategy is that ES performs destructive updates, disabling customers to move back to older versions of the software. Finally, ES does not allow developers to branch development in its SCM system to simplify the maintenance process, thus restricting concurrent development.

ES's implementation of e-Synergy can be seen as a software knowledge base (SKB). The SKB consists of (A) a vendor side SKB that stores all product and component knowledge and (B) a local software knowledge base consisting of the data file containing all deployed deliverables, the configuration information of the tools stored in the database, and the license file storing a list of all activated modules and licensing information. It is the SKB that has allowed ES to expand its customer base. Before the implementation of the SKB, too many resources were used up by maintenance tasks to allow growth to 160.000 customers.

The case study has influenced our research in the following ways. First, the ES solution is not easily applicable to products in domains that do not wish to send out all deliverables, wish to provide incremental updates that can be rolled back, or wish to maintain a high level of reusability among products. In contrast, I wish to create a generalizable solution. Secondly, the case study shows us that integration of software knowledge, as I suspected, is a powerful tool in the maintenance of software. ES, however, also showed us this software knowledge is effectively used in other processes, such as workflow management, human resource management, and customer support. As such, the ES case shows that a simplified instance of the concept of an SKB improves a company's ability to handle large amounts of customers.

ES manages six large product lines, with each product line containing approximately ten solution areas (that in turn can contain large amounts of products and product modules), consisting of a total of one million lines of source code. The source code is managed by 294 software developers, who commit their sources a number of times per day. Customers connect to the release repository of ES approximately 250.000 times per year to download updates. I thus strongly believe that the integration of the PDM, CRM, and SCM system is the best way to automatically manage an ever growing amount of customers in need of updates, licenses, and support.

## 3.6 Related Work

The techniques applied by ES to integrate SCM and PDM are similar to those described by the book on the integration of PDM and SCM systems in Finnish industry [60]. The execution of case studies performed by Tiihonen et al. [111], Bosch et al. [17], and Davenport et al. [31] are similar to the way in which the ES case study was performed. All three describe one or more case studies with qualitative, rather than quantitative results for software products and development processes. The work presented by Tiihonen et al. and Bosch et al. is different from our research because it focuses on the state of the practice of software product configuration in software product lines and because both apply a problem focused approach. Davenport et al. describe a model of knowledge management that is later put into practice at Siemens, and differs from our work in that their model is first compared to the current practices and then implemented to provide extra grounds for evaluation.

The aim of the presented research is to provide qualitative results to the current body of knowledge of knowledge management and case studies in product software companies in software maintenance in general. The techniques applied by ES are clearly centered around the SCM system, and have been integrated upwards with the CRM and PDM systems. With respect to the SCM system, I have positioned it in the framework created by Conradi and Westfechtel [25]. The SCM system under study, by their definition, is a state based tool-kit that applies a database to store coarse grained extensional product compositions for any problem domain. The product configurator applies functional rules to provide an interactional model to its users. Finally, the case studies performed by us [62] and Ballintijn [7] are of the same format as the work presented here. These two case studies show, similar to the ES case but to a lesser extent, that explicit software knowledge management can improve the CCU processes. The main differences between the ES case and these cases is that they describe much smaller organizations (160 and 100 employees, respectively) and that they have not yet integrated PDM, SCM, and CRM to such an extent as ES.

## 3.7 Conclusion

If anything can be learned from this research, it is that software vendors must integrate their CRM, PDM, and SCM systems to automate the processes related to CCU. Such automation provides more efficient methods to perform repetitive tasks such as license creation, license renewal, product updating, error reporting, usage reporting, product release, and manual configuration tasks, such as backups. The second main lesson is that usage of feedback reports supplies software vendors with the largest test bed imaginable, and therefore deserves more attention. The presented research can be used as a guideline for software vendors or for the development of a software manufacturing and software product data management system. In our search for tools that can provide the key practices presented in this chapter, an undiscovered product niche was encountered. It seems that there are no (other than ES's e-Synergy) commercial product data management systems that explicitly manage licenses, software products, their fixes, and their patches, in such a way that customers can log in and download

them.

This article describes a case study of a software knowledge base at Exact Software. The case study helps to provide evidence that the complex maintenance tasks of enterprise application software for a vendor is best managed with an SKB. Our contribution is twofold. First, I showed that explicitly managing software knowledge improves the processes of release and deployment for a software vendor selling different enterprise resource planning software products. Secondly, I showed that integrating the knowledge with other systems, such as PDM and CRM systems optimizes the processes of maintenance and delivery, and enables vendors to serve a large customer base. I also use the results of this case study in comparisons to other case studies I am performing at other software manufacturers [141] and I will use the results to build, in cooperation with industry, prototype tools related to SKBs.

CHAPTER 4

# A Benchmark Survey into the Customer Configuration Processes and Practices of Product Software Vendors in the Netherlands

*Product software vendors do not invest enough effort into release, delivery, deployment, and usage and activation of their software products. Not spending effort in these customer configuration updating processes leads to high overhead per customer, which impedes growth in customer numbers. This chapter presents the results of a survey that provides product software vendors with an overview of their customer configuration updating processes and practices, and benchmarks their practices against competitors using similar technology, of the same size, and active in the same market. These benchmarks contain customized advice to the respondent company that can be used to strategically improve their customer configuration updating processes to gain efficiency and effectiveness. The survey was held in the Netherlands, and 74 software vendors responded. Amongst other conclusions, a significant positive correlation was found between success of a software product and a vendor's recent investments into customer configuration updating.* [1]

## 4.1 Introduction

The product software industry in the Netherlands is flourishing. Computer games, ERP products, and navigation systems are just some examples of successful products, nationally and internationally. Though these businesses have a large body of knowledge available to them about generic software development and engineering, none of it is

[1] The work has recently been submitted [146].

specific to product software development. One area that is specific to product software vendors is the fact that they have to release, deliver, and deploy their products on a wide range of systems, for a wide range of customers, in many variations. Furthermore, these applications constantly evolve, introducing versioning problems. This chapter presents a benchmark survey into the customer configuration updating processes of 74 product software vendors.

To date product software is a packaged configuration of software components or a software-based service, with auxiliary materials, which is released for and traded in a specific market [132]. Customer configuration updating is defined as the combination of the vendor side release process, the product and update delivery process, the customer side deployment process, and the usage and activation process (see chapter 2). The release process is how products and updates are made available to testers, pilot customers, and customers. The delivery process consists of the method and frequency of update and knowledge delivery from vendor to customer *and* from customer to vendor. The deployment process is how a system or customer configuration evolves between component configurations due to the installation of products and updates. Finally, the activation and usage process concerns license activation and knowledge creation on the end-user side.

Vendors encounter many problems when attempting to improve customer configuration updating. To begin with, these processes are themselves highly complex considering vendors have to deal with multiple revisions, variable features, different deployment environments and architectures, different distribution media, and dependencies on external products (please see chapter 6). Also, there are not many tools available that support the delivery and deployment of software product releases that are generic enough to accomplish these tasks for any product (please see chapter 5). Finally, Customer Configuration Updating (CCU) is traditionally not seen as the core business of vendors, and seemingly does not add any value to the product, making vendors reluctant to improve CCU.

This chapter presents the results from a survey of 74 product software vendors. The respondents are product managers for one product from one product software vendor. The object under study is the release, delivery, deployment, and usage and activation processes from each vendor for one of its products. These four CCU processes consist of two to four practices. Each practice consists of capabilities. A vendor's capabilities are measured using between three and five questions per capability. The aims of these scores are to establish a vendor's position compared to competitors. The overall goals of this research are to see what the CCU landscape in the Netherlands looks like, to establish whether CCU process scores are directly related to product success, and whether a survey can be used as a knowledge dissemination method.

In the following section the processes and practices of customer configuration updating are described in detail. In section 4.3 the research design, consisting of the hypothesis and research approach, is presented. In section 4.4 the survey results and respondents are presented, including the results for each process area and for the hypotheses. Finally, in section 4.6 we present our conclusions and discuss our future work in regards to CCU process improvement.

## 4.2 Customer Configuration Updating

The four process areas and their corresponding practices used in the benchmark survey are defined using the SPICE model for process assessment [93]. The SPICE model, which enables self analysis for vendors, defines a process as "a statement of purpose and an essential set of practices that address that purpose". These practices are software engineering or management activities that contribute to the creation of the output (work products) of a process or enhance the capability of a process. In this section CCU and practices are defined, together with their relationship to the CCU model and current literature.

### 4.2.1 Processes and Practices

We define CCU as the release, delivery, deployment, and usage and activation processes of a software vendor. These processes consist of two to four practices, each with a number of elementary capabilities. For instance, the release process is made up of four practices. One of these practices is release frequency and quality. The capabilities falling under the release frequency practice are that a vendor must frequently release major, minor, and bug fix releases, that a vendor must synchronize these releases with customer demand, and that releases are tested by pilot customers before they are made publicly available.

The **Release process** is based on four release practices. The first practice is how often versions and updates of a product are released and how this is planned within the organization. The second practice is how releases are shared within the company and between customers and the software vendor. Thirdly, all dependencies between components, be they products that have been built by the vendor or purchased from another, must be managed by making explicit dependencies between these products and components. Finally, versions of external components, such as components-off-the-shelf, must be managed explicitly to maintain high quality releases.

With regards to the **delivery process** there are two practices. The first practice prescribes that vendors must use every possible channel for the distribution of products and product updates [52]. The second practice states that every possible method for delivery must be applied, such as automatic push or pull.

There are four practices for the **deployment process** of a vendor. To begin with, a product must be removable without leaving any remnants of data on a system. This is required because a new installation preferably must not be contaminated with old data. Secondly, if issues are encountered during the deployment of a software product, automatic resolution must take place to resolve these issues. Such resolution mechanisms are capabilities such as automatic downloading of missing components, freeing up resources when required, or even automatic renewal of licenses. The third practice for the deployment process is that updates and installations must be able to cope with customizations made by customers or third parties. A vendor supports this practice when a special software architecture is in place that enables customizations. The fourth practice is deployment reliability, which is ensured by capabilities such as validity checks, feedback reports, and externalization of customer specific changes and data [33].

Finally, a vendor's **activation and usage process** is based on three practices. First, a vendor must (semi) automatically handle all license requests and distribute licenses with a maximum amount of flexibility within the organization. A vendor supports this practice once customers can explicitly manage their licenses, once licenses expire, once temporary licenses can be generated for sales and test purposes, and once licenses are generated automatically as soon as a sales contract is signed. Secondly, vendors must make use of feedback to gain as much knowledge about the product in the field as possible. A vendor is considered adequate for this practice once it makes use of both usage reports and error feedback. The third practice is that vendors must be aware of their customers' configurations. Vendors scores for this practice when they are aware of the software and hardware components that are used by its customers.

## 4.3 Research Design

This research has been conducted to find out more about the CCU practices and processes of product software vendors, to benchmark a product software vendor's practices, and to generalize some of the conclusions of earlier work we applied in a multiple case study (please see chapter 2).

### 4.3.1 Hypotheses

The work has been conducted for two reasons. The first reason is to perform a benchmark for product software vendors about their release, delivery, deployment, and usage and activation practices. Secondly, we wished to prove or disprove the following hypotheses, which have been formulated in earlier research [143]:

**H1: A product software vendor's scores for the CCU processes are positively correlated to the age of the company, the age of the product, the number of natural languages in which the product is available, the number of developers working on the product, and the number of customers of the product.** Many different tools are built around software products during their lifecycle. Furthermore, customers come and go, making the need for good CCU processes and tools even larger. Also, as a vendor's customer base grows, this needs increases further. Also, with more developers on board, more time can be spent on CCU improvement. We therefore expect H1 to be true.

**H2: A younger technology platform will result into weaker performance of the CCU processes.** When a product software vendor starts using a new technology there are not many CCU support tools available for that technology. This implies that older technology platforms will have better CCU support, improving a vendor's CCU score.

**H3: When a vendor explicitly manages CCU knowledge they are more successful.** When a product software vendor changes the CCU process it improves customer experience and enables a product software vendor to spend less resources per customer. This frees up resources to further develop the product or do more maintenance. This hypothesis is two sided, however, due to the fact that a more successful product software vendor will have more resources available to spend op CCU processes.

| **Process** | **Practice** | **Average score** | **Maximum score** |
|---|---|---|---|
| Release | Releases are frequent and of high quality [143] | 6.08 | 17 |
| | The vendor maintains an explicit release planning [142] | 2.26 | 7 |
| | A formalized release scenario that describes what happens on release day [143] | 8.73 | 18 |
| | Manage external products, such as commercial-off-the-shelf components [62] | 4.73 | 9 |
| | **Total** | **21.8** | **51** |
| Delivery | Vendors must use every possible channel to stay into frequent contact with customers [52] | 11.91 | 30 |
| | Every possible method for delivery must be applied [143] | 11.28 | 38 |
| | **Total** | **23.19** | **68** |
| Deployment | Explicit dependency management for correct deployment [21, 121] | 3.35 | 7 |
| | The product can be uninstalled and rolled back [147] [89] | 2.7 | 4 |
| | The vendor uses update tooling and manages customizations [37] [147] | 11.8 | 26 |
| | Update reliability and semi-automatic problem resolution [142] | 14.49 | 32 |
| | **Total** | **32.34** | **69** |
| Usage and activation | Licenses are can be renewed and trialled and activate components of the software [143] | 6.97 | 7 |
| | Licenses are explicitly managed within the organization and generated from contracts[143] | 2.65 | 9 |
| | Vendors know how customers use products and take advantage of this knowledge [135, 148] | 9.24 | 14 |
| | **Total** | **18.86** | **30** |
| **All** | **Cumulative Total** | **96.19** | **218** |

Table 4.1: Scores for Practices and Processes

### 4.3.2 Approach and Survey Design

The research hypotheses are proved or disproved using a web survey that establishes scores of CCU processes and different maturity indicators for product software vendors and their products. The benchmark survey consists of eleven parts. Each part contains between three to twelve questions. There are closed as well as open ended questions in the benchmark survey. The closed questions are used to establish scores for practices of vendors and consist of yes/no questions and multiple choice questions. The open questions establish the generic or numeric information on the vendor, such as the vendor's name, the amount of customers of the product, the amount of users of a product, etc. Three open ended questions have been added to the end of the survey to find out in which parts of CCU the vendor will invest in the future and what tools they feel are missing in the range of CCU support tools.

Each of the practices stated in Section 4.2 has three to five questions that assesses the vendor's **practice score**. The practices have been derived from the CCU model, as described in Section 4.2. When all practice scores for one process are added up we obtain the **process score**. As is suggested by Saywell [30], methods that force your public to evaluate their own process actively by participation (in a benchmark survey, for instance) are a valid method for knowledge dissemination. When sending out the benchmark report we attached a small paper survey, with a stamped return envelope, to evaluate Saywells claim. We received 26 responses of which only one was negative. All others responded positively to the usefulness of the benchmark report as both a knowledge dissemination tool and as being useful to use in future improvement projects.

### 4.3.3 Sample Selection

The respondents have been selected based on 2 criteria: First the submitter must be a product manager or a development manager who is close to the process and can answer each question. Secondly, the software product must specifically be a software product that is delivered to customers and executed at the customer's site. These requirements are specified in the invitation e-mail and at the beginning of the benchmark survey itself.

The vendors have been selected through the Platform for Product Software [45], the yellow pages, and the Netherlands business index for ICT [84]. The potential respondents have been approached by e-mail twice. No two respondents belonged to the same product software company.

### 4.3.4 Survey Tool

The open source PHP/MySql tool used for this benchmark survey is called PHP Surveyor [47]. The tool is still under development and the fact that it is open source has enabled some final customizations, such as the addition of question types and custom data analysis methods. The survey tool has been selected specifically because the submitter can save the survey results up to a certain point, to allow him or her to

| **Distribution of respondents** | | | | | |
|---|---|---|---|---|---|
| **# Employees** | | **Product Categories** | | **Development Technologies** | |
| # Employees | # Respondents | Category | # Respondents | Development Technology | # Respondents |
| 1 | 16 | Business productivity | 51 | Java | 18 |
| 2-4 | 10 | Internet | 36 | C++ | 14 |
| 5-7 | 8 | Health care | 5 | dotNet | 14 |
| 8-10 | 6 | Home user | 5 | Pascal | 14 |
| 11-20 | 9 | Financial | 4 | PHP | 12 |
| 21-50 | 10 | Embedded | 4 | Application Service Provider (ASP) | 9 |
| 51-75 | 8 | Extensions | 3 | C | 8 |
| 76-100 | 3 | Media | 2 | Basic | 8 |
| 101-300 | 2 | Games | 1 | Clarion | 4 |
| 301-600 | 2 | | | Perl | 3 |
| | | | | Lisp | 1 |

Table 4.2: Distribution of Respondents

find answers to questions he or she does not know at the time of asking. The previous answers can then be reloaded at a later point.

### 4.3.5 Benchmark Survey

The benchmark survey is an initiative of the Platform for Product Software [45], a scientific initiative that attempts to unite Dutch product software vendors to improve the industry by disseminating knowledge from the academic world. The benchmark was created to serve both our academic interests and the interest of the platform. To be of use to the platform the results from the survey are used to automatically generate a customized benchmark report for each of the respondents. Such a report includes comparisons of a respondent's practices to other product software vendors using similar development technology or are active in the same market. Besides providing a dataset for research, the report spreads knowledge and provides new insights to product software vendors. The survey can be found in section 4.7.

One question asks the software vendor to prioritize six reasons for CCU improvement. These CCU improvement priorities were to serve more customers, serve customers more cost-efficiently, reduce deployment problems, reduce the time in which bugs are found, shorten release cycles, and apply a more flexible pricing model.

Furthermore, the customized benchmark reports include specific advice for each of the product software vendors. The advice, such as "You must introduce a formal release scenario" included a full description (1 paragraph) and a relevance indicator. This relevance indicator (*low*, *medium*, and *high*) is based on the above mentioned prioritization question. For example, if the respondent put "Shorten release cycles" as a top priority, the release scenario advise example received a *high* relevance rating. Each advice was assigned to between one and three CCU improvement priorities, based on the CCU evaluation model.

To evaluate whether the benchmark reports are a useful tool for knowledge dissemination, a second survey was sent with the benchmark report. This second survey contains questions such as "Has the report provided you with new insights into the release process?"

### 4.3.6 Validity Threats

To make sure that we draw the right conclusions from the results, we defend construct validity, reliability, and external validity.

**Construct validity -** In this study concepts come directly from literature and from our earlier practical case study work. The questionnaire was pre-tested by a small working group of four software vendors and by five researchers. Their comments lead to approximately 10% of the survey having been changed. These four software vendors were excluded from participation.

**Reliability -** To stimulate correct answers the submitters were all promised a full report, complete with customized advice to specifically fit their product. Furthermore, we promised the submitters that the results would only be published anonymously.

**External validity -** We strongly believe that the results of this survey are generalizable to other countries. The OECD states that the Netherlands is the fourth

largest exporter of product software in the world [44]. It must be noted that three of the larger software vendors in the Netherlands replied (in a country where there are only a handful). Even though we estimate that our dataset covers 6 percent of the total number of software vendors, we expect this number to be much higher when looking at the number of employees active in the product software industry.

This work is specific to product software vendors. During the research we were approached by a number of on-line application service providers, asking whether they could participate. These requests have been declined, because too many questions were about concepts that are not applicable to software providers, such as delivery of a product and its updates, usage and feedback reports.

## 4.4 Results

In table 4.1 the average process and practice scores are listed for CCU. The first column listst the process for which this practice is applicable. The second column describes each practice and provides references to other work in which these practices are described and discussed. In the third column the average score is provided for each practice, obtained by the vendors. The final column lists the maximum score that could have been earnt. The processes and practices of the CCU evaluation are based on the CCU model [143], the SOFA model [99], and some capabilities of the Software Dock [52] and have been recorded into the SPICE based evaluation model. Please note that the total scores for each process cannot be compared to the scores of other processes. We believe that the four processes are equally important. In tables 4.3, 4.4, 4.5, and 4.6 the average scores for the release, delivery, deployment and usage and activation practices and processes are listed. The tables describe the questions that are posed to determine each practice score for the processes. The scores that can be earned per question have been determined by a committee of five representatives from four product software vendors. The scores for each practice are determined by between two and five questions each.

### 4.4.1 Respondents

74 product managers submitted the survey, from software vendors ranging from 1 to 460 employees (see table 4.2). Three were excluded from the dataset on the grounds of being from the wrong country or for being a web service provider. Statistics Netherlands estimates that there are 1400 software vendors in the Netherlands, giving a 5 percent coverage of the total product software industry. Furthermore, when asked the product managers to categorize their business by checking at least two categories (see table 4.2), by far the largest part of product software vendors is building business productivity tools and enterprise resource planning systems. The product managers were also asked to provide the development technologies they used (results in table 4.2). Please note that for the technology and business category questions respondents were allowed to submit more than one answer.

| **Practice** | **Question or description** | **Average Score of all vendors** | **Highest possible score** |
|---|---|---|---|
| A1 | How often do you publish a major release of your product? | | 5 |
| | How often do you publish a minor release of your product? | | 5 |
| | How often do you publish a bugfix release of your product? | | 5 |
| | Are releases timed in sync with customer requirements? | | 2 |
| **Total** | **A1: The vendor addresses release package planning to maintain high quality releases[143]** | 6.08 | 17 |
| A2 | Does your organisation use a formal release planning that states the times until the next major, minor, and bugfix releases? | | 5 |
| | Is this planning published in such a way that all related personnel can access it? | | 1 |
| | Is there a formal publishing policy towards the outside world in regards to this document? | | 1 |
| Total | **A2: The vendor maintains an explicit release planning [142]** | 2.26 | 7 |
| A3 | Is there a step-by-step description (release scenario) of what happens on the day of a release? | | 2 |
| | Releases are stored in a versioned repository? | | 6 |
| | All major, minor, and bugfix releases can be downloaded by all product stakeholders? | | 6 |
| | The latest release can be downloaded by all stakeholders? | | 4 |
| Total | **A3: A formalized release scenario that describes what happens on release day [143]** | 8.73 | 18 |
| A4 | All tools that are built to support CCU are managed as if they are externally acquired products. | | 2 |
| | All other tools (such as development tools) are managed explicitly as well? | | 2 |
| | Are these components stored in a versioned repository? | | 5 |
| Total | **A4: Manage external products, such as commercial-off-the-shelf components [62]** | 4.73 | 9 |
| **All** | **Release Process** | **21.8** | **51** |

Table 4.3: Scores for Release Practices and Processes

| **Practice** | **Question or description** | **Average Score of all vendors** | **Highest possible score** |
|---|---|---|---|
| B1 | All major, minor, and bugfix releases can be downloaded by all product stakeholders? | | 6 |
| | The latest release can be downloaded by all stakeholders? | | 4 |
| | You inform your customers using different channels? | | 10 |
| | Does your product automatically send back error reports? | | 3 |
| | Customers report bugs via different channels. | | 5 |
| | Does your product generate usage reports and send these back to you? | | 2 |
| **Total** | B1: Vendors must use every possible channel to stay into frequent contact with customers [52] | 11.91 | 30 |
| B2 | Your products can be delivered using different types of media? | | 15 |
| | You inform your customers often and regularly? | | 5 |
| | Your product can be delivered through automatic pull/push methods? | | 16 |
| | Your product updater can download product updates from any location? | | 2 |
| **Total** | B2: Every possible method for delivery must be applied [143] | 11.28 | 38 |
| **All** | **Delivery Process** | **23.19** | **68** |

Table 4.4: Scores for Delivery Practices and Processes

| **Practice** | **Question or description** | **Average Score of all vendors** | **Highest possible score** |
|---|---|---|---|
| C1 | Are external relationships managed between your product and other products? | | 2 |
| | Are these products stored in a versioned repository? | | 5 |
| **Total** | C1: Explicit dependency management for correct deployment [21, 121] | 3.35 | 7 |
| C2 | Is it possible to uninstall your product without complex manual steps? | | 2 |
| | Is it possible to undo an update? | | 2 |
| **Total** | C2: The product can be uninstalled and rolled back [147] [89] | 2.7 | 4 |
| C3 | Please indicate whether your update tool can be used for major, minor, and/or bugfix releases. | | 8 |
| | Does your product use an update tool? | | 5 |
| | Can the update tool deal with customer data and customizations? | | 5 |
| | Can the product be deployed in a Development, Test, Acceptance, Production (DTAP) environment? | | 3 |
| | Can your product be updated at runtime? | | 5 |
| **Total** | C3: The vendor uses update tooling and manages customizations [37] [147] | 11.8 | 26 |
| C4 | Does the product installer or updater extensively check a customer's configuration before updating? | | 7 |
| | Are any of these problems automatically resolved? | | 6 |
| | Which of your updates are checked by pilot customers? | | 4 |
| | Can a customer run a tool to check the completeness of its deployment of your product? | | 3 |
| | Are all product specific data and customer settings stored separate from the product? | | 3 |
| | Are you well aware of how customers generally use the product? | | 3 |
| | Are you well aware of the hardware customers use? | | 3 |
| | Are you aware of all customer specific solutions for your product? | | 3 |
| **Total** | C4: Update reliability and semi-automatic problem resolution [142] | 14.49 | 32 |
| **All** | **Deployment Process** | **32.34** | **69** |

Table 4.5: Scores for Deployment Practices and Processes

| **Practice** | **Question or description** | **Average Score of all vendors** | **Highest possible score** |
|---|---|---|---|
| D1 | Do you use license agreements? | | 5 |
| | Can an end-user choose which license he/she will use at startup? | | 1 |
| | What type of data is stored in your license files? | | 1 |
| **Total** | D1: Licenses are can be renewed and trialled and activate components of the software [143] | 6.97 | 7 |
| D2 | Can a customer renew its license without your intervention? | | 3 |
| | Do your licenses expire? | | 1 |
| | Do you often supply temporary licenses? | | 2 |
| | Are licenses generated from contracts automatically? | | 3 |
| **Total** | D2: Licenses are explicitly managed within the organization and generated from contracts[143] | 2.65 | 9 |
| D3 | Are you aware of how customers use your product? | | 3 |
| | Are you well aware of the hardware customers use? | | 3 |
| | Are error reports automatically sent when a product crashes? | | 3 |
| | Does your product create usage reports? | | 2 |
| | Are you aware of all customer specific solutions for your product? | | 3 |
| **Total** | D3: Vendors know how customers use products and take advantage of this knowledge [135, 148] | 9.24 | 14 |
| **All** | **Usage and activation process** | **18.86** | **30** |

Table 4.6: Scores for Usage and Activation Practices and Processes

### 4.4.2 Hypotheses Results

**H1: A product software vendor's scores for the CCU processes are positively correlated to the age of the company, the age of the product, the number of natural languages in which the product is available, the number of developers working on the product, and the number of customers of the product.**

There exists a strong positive correlation between the number of customers a product software vendor has and its CCU scores, for all CCU processes. The strongest positive correlation is found between deployment process scores and the number of customers. Clearly, the higher the number of customers, the more likely a vendor is to spend time on minimizing overhead by managing and improving CCU processes. Unexpectedly, there is hardly any correlation between product age and process scores. Though the scores vary greatly, older products do not necessarily have higher CCU scores. The product age (ranging from 0 to 22 years) only showed a slight upward trend for the deployment process scores, possibly meaning that only the deployment process score is influenced by the age of a product. In regards to FTE developers there exists a strong positive correlation between all processes except usage and activation (where there is a hint of a negative correlation). An explanation for this phenomenon has not yet been found. The hypothesis thus only holds partly true.

**H2: A younger technology platform will result into weaker performance of the CCU processes.** To prove or disprove this hypothesis, we have listed the technologies, the average CCU scores, the average separate scores per process, and the technology age.

Some technologies do obtain much higher scores than others. At the top of the list are Visual FoxPro, C#, Visual Basic, and Java. At the bottom of the list are C, C++, and the scripting languages PHP and ASP. Furthermore, there is a strong negative correlation between development technology age and the deployment process score, suggesting that newer technologies have better deployment support.

**H3: When a vendor explicitly manages CCU knowledge they are more successful.** A series of three questions was asked to establish a causal relationship between explicit CCU knowledge management and product success over the last two years. To the first question, whether the product was more successful, 62 answered they were more successful, confirming our belief that product software is booming business. Of course there are many more factors that influence the success of a software product. To remove this variable the respondent was asked to value the relationship between product success and recent CCU improvements, from "of no influence at all" to "mostly caused by". We found a direct relationship between the success of a software vendor and its investment into the CCU process over the last two years ($r = 0.26, p \leq 0.05$), taking into account the answers to the question on the relationship between product success and CCU improvements.

### 4.4.3 Open Questions

To the open question "Into what area of CCU will your organization soon invest" most respondents answered they would invest into delivery methods of updates (40%) and into update tools and methods (35%). In regards to delivery the respondents explained

they wished to deliver through different media such as FTP sites and portals and with different methods, such as scheduled automatic updates. Regarding update tools some answered they were investing into Microsoft Windows Vista and some that they were using commercially available installers such as InstallShield and Wix. The third problem that was discussed frequently was release planning and scheduling (12%).

When asked how recent CCU improvements had affected the product most respondents answered that *higher product quality had been reached* (23%) and that the product had become *more reliable* (21%). A close third and fourth were *less deployment problems* (16%) and *lower development cost* (16%). The less popular choices were *less time to find bugs* (11%), *more knowledge about customers* (5%) and *shorter release cycles* (9%). The submitters answered questions on how many people were active in the areas of CCU (release managers, version system managers, etc) and development (quality assurance, programmers, etc). When dividing the number of employees supporting or managing CCU processes by the number of development personnel, an average of 16% is found. This tells us that research contributions in CCU can potentially affect on average 16% of software development personnel on average at product software vendors.

According to this survey the most important reasons for CCU improvement are to serve more customers and to serve them more cost-efficiently on a shared first place. The runner-up was to reduce deployment problems. In contrast, these three were at the first place of software vendors' lists 19 times each, whereas the other three ended up at the top of the list six times each. The scores are computed by awarding 6 points for priority 1, 5 points for priority 2, etc. Please note that all columns and rows (excluding the scores) add up to 74.

### 4.4.4 Suggestions for CCU Improvement

Each of the 74 product software vendors received a personalized and customized report, with up to eight CCU improvement suggestions per process area. These improvement suggestions were then annotated with a relevance rating, based upon the ordering of a product software vendor's reasons why they would improve CCU. The submitters received an average of thirteen improvement suggestions per survey report. The frequency of improvement suggestions displays relevant issues in CCU.

For the release process vendors generally do not use a formal release planning. Furthermore, many product software vendors set shortening release cycles or reduce the time in which bugs are found as a prime priority. This led to the advice "Introduce a formal release planning and share this within the organization" being issued to 47 of the 74 product software vendors. The advice "Store external components and development tools in a versioned repository with the product" was provided least often (13 times out of 74) due to the fact that most of the vendors already do this.

For delivery the advice issued most frequently (47 times) was "Seek contact with your customers more often through alternative channels", especially to those vendors who set "Serving more customers" and "Reduce the time in which bugs are found" as their first and second priority. This advice was especially provided to those who do not use the product itself, for instance through pop-ups and in-product messages, for knowledge delivery.

In regards to deployment the advice "Explicitly manage all relationships between external products and yours" given 46 times, especially to those vendors who set "Reduce deployment problems" and "Serve customers more cost-effectively" as their top priorities. Another improvement suggestion was 46 times as well, but with a lower average relevance rating, was "Enable your product update tool to facilitate custom solutions by customers and partners" because many product vendors do not leverage the advantages of a software supply network yet [152, 151].

Finally, in regards to usage and activation "Send automatic error reports" was advised 53 times to those software vendors with the top priorities "Reduce deployment problems" and "Serve customers more cost-effectively". Furthermore, improvement suggestions often concerned licenses, such as "Enable the customer to renew their license without your manual intervention" and "Generate licenses automatically after entering a sales contract" (43 and 44 times). With the report sent back to the product software vendors a small survey is included that is used to evaluate the applicability of the advice.

We received 19 responses from the short survey. All short surveys reported that the benchmark had changed their view of the CCU processes and that they increasingly saw the CCU processes as interconnected. Of the 19 surveys two reported that they planned no new changes based on the benchmark report, two reported that they would make large changes to their CCU processes, and the other 15 reported they would make small changes based on the report. The advice was rated per process on a Likert scale. 17 surveys reported that the advice given in three of the four processes were useful. All recommendations for the processes were rated equally and averaged between useful and very useful. Some of the open suggestions were that we change the survey for smaller companies and that we include other subjects such as support and user interface in our benchmark.

### 4.4.5 Exploring Relationships

The benchmark survey delivered us with a large set of data, consisting of answers to yes/no questions, multiple choice questions, and a large amount of open ended questions. We attempted to find new relationships between yes/no questions using two-sided Pearson's goodness of fit chi-square tests. We used an algorithm to find whether relationships existed between all yes/no questions. We (obviously) found strong relationships between conditional questions, where the submitter could only provide an answer if the submitter had answered yes or no to the previous question. Some of the relations of interest are described below. Please see all the found relationships in table 4.12.

Many obvious relationships surfaced: first, there exists a relationship between whether a product's licenses expire and whether a product software vendor can provide temporary licenses ($r = 16.1141, p \leq 0,0002$). Secondly, the ability the update at runtime and the use of a commercial update tool are related ($r = 4.6423, p \leq 0.0313$). Thirdly, the ability to check a customer's configuration for completeness is related to the use of components off the shelf that are included in the product ($r = 7.0652, p \leq 0.008$). Finally, the capability to send feedback and usage information from customers to the vendor is related to the expiration of licenses ($r = 9.6038, p \leq 0.002$), sending

of automatic error reports ($r = 3.9422, p \leq 0.0472$), and the checking of a customer's configuration for completeness ($r = 5.8311, p \leq 0.0158$).

We have defined two major practices, the use of a concrete release planning that states release dates for major, minor, and bugfix releases, and the practice to have a release scenario that states exactly what needs to happen on the day of a release. These two practices are related to eachother ($r = 6.9163, p \leq 0.0086$). Furthermore, the release planning practice is related to explicitly managing dependencies between a vendor's product and external components ($r = 7.9383, p \leq 0.0049$). The release scenario practice is strongly related to the practice that the software vendor explicitlymanages CCU and development support tools created by the vendor ($r = 8.1318, p \leq 0.0044$), the product Components off the Shelf (COTS) ($r = 4.5867, p \leq 0.0323$), and the practice that the product can be deployed in a development, test, acceptation, and production (DTAP) environment ($r = 7.7168, p \leq 0.0056$). We find the relationships between the release planning practice, the usage of COTS, and the explicit management of development tools easy to explain. As the software supply network surrounding a software product grows, so do its dependencies [151]. These dependencies must be managed explicitly, including a specific planning of when the software product will include or support newer versions of COTS or other products. The use of a DTAP environment is beneficial when a product must be deployed in complex environments with many dependencies, including when a product uses COTS.

The ability to undo an update is related to the fact that the product can deal with customizations and extensions ($r = 5.8681, p \leq 0.0155$) and the fact that the product can be deployed in a DTAP environment ($r = 6.6069, p = 0.0102$). We do not find these results surprising because if an update can be undone it is much easier to use in a DTAP environment ("if update is not approved undo"). The undo update feature is beneficial when a product is full of customizations and one needs to be sure that the update does not damage any of the customer specific functionality.

From these statistics a number of things can be concluded. First, we see that once a vendor periodically initiates contact with its customers for one purpose, other purposes soon follow. Secondly, when a vendor starts using COTS, release planning is required to further enable the product to be maintained. Thirdly, the ability to undo an update provides customers with more flexible methods of deployment and enables a vendor to make the update process more reliable.

## 4.5 CCU Practice Results

### 4.5.1 Release

Typically, software vendors will have their major releases checked by an average of 5 pilot customers. Furthermore, bug fixes are published every one to six months. 10 respondents publish daily bug fixes. Minor releases are published every two to twelve months. Major releases are published every one to three years. Please note that only seven respondents did not publish all three types of releases. We can safely assume that the major, minor, and bug fix release schedule is universal. 37% of organizations use a formal release planning with dates attached. Of this 37%, 70% have a formal

publishing policy towards this planning. Furthermore, of these 37%, 45% of the vendors take into account when the release of a product or update suits customers.

Only 52% of the respondents have a formal release scenario or plan that describes the steps taken for a release. Releases are stored most of the time on a shared network drive within the organization, with a versioned repository as a close second. Frequently respondents answered that their old releases are available to customers. In 55% of the cases tools that have been built by the respondent's organization are only managed as if they were commercially purchased. 69% of the respondents use COTS integrated into their products. In 75% of the cases these products are saved alongside the product in a release repository, ensuring version compatibility.

As product software vendors and their products become more mature, more knowledge becomes available about the product and needs to be shared with larger numbers of stakeholders (assuming that product software vendors grow). We see that product software vendors implement less than half of the release process practices. This is largely caused by the low number of product software vendors who frequently publish releases, who maintain a release planning, and who use a release scenario. Many of the software vendors see the release process as a low priority process that does not require formal planning documents. The many failed deployments and published bugs can be drastically reduced if product software vendors put more effort into the release process. As an example, a company that builds navigation software for PDAs and embedded devices, recently released a version of their software containing a virus [129] due to low quality of the release process.

### 4.5.2 Delivery

Product customers are contacted most frequently by e-mail and the software vendor's website. Only 12 of the 74 respondents inform customers using their own product. Bugs are reported most frequently through e-mail, by phone, and by an on-line bug system. Furthermore, 15 of the vendors send automatic feedback reports when the product encounters an error. Customers are kept up to date yearly, every three months, or monthly in regards to product information.

Regarding delivery methods software is mostly delivered through a website, with CD-Rom as a close second. Furthermore, 12 of the respondent's products are delivered by USB stick. 47 of the respondents' products are delivered when a customer manually pulls it from the vendor. Only 12 of the 74 respondents have developed their product to automatically download updates on a regular basis. In 55% of the cases software products can be downloaded by the customer from any location, instead of from a preset site, encouraging customers to just download an update once from the vendor for any amount of workstations. 75% of the products are sent to customers in full, and later on parts are (de-)activated by a license file. The other 25% deliver the product to customers on a *get-what-you-paid-for* basis.

These numbers show that many software products and updates are still being delivered using CD-Roms and e-mails. CD-Roms and e-mails as a type of delivery, which needs to be managed by hand, is time inefficient and informal. Product software vendors must improve these processes to lower delivery costs, and to ensure that customers get the right updates and products. Product software vendors only

implement one half of the practices of the delivery process. The low scores can be explained by the fact that product software vendors stick to just one method for software and knowledge delivery and that products are not delivered using (semi-)automatic push and pull delivery. We strongly believe that customer retention is a critical success factor for product software vendors, due to the fact that often more than half of their yearly revenue comes from existing customers. To retain customers, however, vendors must seek more creative methods to contact customers about their products.

### 4.5.3 Deployment

Respondents estimate that on average 4.26% of deployments fail and require extra assistance. Some respondents, however, report up to 35% of failed deployments.

Approximately one fourth of the respondents use InstallShield to deploy their products. Furthermore, MSI (19%) and ZIP (24%) are popular formats for delivery. Only 3 of the cases deliver their products as open source. 52% of the respondents uses an update tool to update their customers' configurations. Customer data and content is separated from product files by 68% of respondents. Furthermore, 40% of respondents has functionality for correctness checking of a customer's configuration. Product update and installation tools check most often for harddisk space, with the operating system and already present customer data as a close second and third. The end-user's hardware is checked least often (6%). 21 respondents report that their deployment tool, if any, cannot automatically (attempt to) resolve deployment problems. For those tools that can resolve deployment problems, the detection and use of data from previous installations is most common.

In 49% of the respondents' products, the product update tool can deal with customizations. Update tools are most commonly used for major and minor updates. The tools are less commonly used for patch updates and content updates. In 34% of the cases the product update tool can update the product at runtime. Furthermore, 85% of the tools can be deinstalled without complex manual steps. Furthermore, in 51% of the cases minor updates can be undone. Finally, in 84% of the cases the product can be installed in a Development-Test-Acceptation-Production environment.

In regards to tools product software vendors generally do not apply their update tool to all release types and their tools generally cannot be used at runtime (although we expect this to be situational). In regards to update reliability many product software vendors are unaware of their customers' configurations and do not store product data separate from the product executables. We believe that this is caused by a lack of product update tools and technologies that provide these capabilities. We expect product software vendors to grow over time, both in customer numbers and employees. As they grow, product software vendors will try to reduce deployment problems and increase overall product experience.

### 4.5.4 Usage and Activation

95 percent of the respondents use license files, containing information on the purchased modules, the number of users, and the customer name and address. Customers pay

most often per floating user or per user name. Pay per usage is quite popular as well, with 19 of the 74 respondents using it to bill their customers. 45% of the cases uses licenses that expire. 13 product software vendors have mechanisms in place that enables customers to renew a license without the intervention of the vendor. 65% of the cases provide temporary licenses on a regular basis. In 20% of the cases licenses are automatically generated from contracts.

28% of the vendors make use of usage feedback reports. 78% of the vendors is aware of all customizations that have been built on top of their products. In 30% of the cases software vendors send back error reports in case of a crash or error.

The average scores per practice are quite high, which explains that product software vendors perform well when it comes to license management. Furthermore, they provide many different reporting features in their products (which are sent back to the vendor in only 30% of the cases, mind). Product software vendors perform well when it comes to license management and product usage reporting. This does not necessarily mean that this area does not deserve attention, though. We believe there is still much research to be done when it comes to the mining of data from feedback reports.

## 4.6 Conclusions and Discussion

This chapter presents the results of a survey of 74 product software vendors in the Netherlands. The survey was created using the CCU process evaluation model, which is presented in detail. The survey allows us to generalize conclusions from earlier research that CCU processes of product software companies are generally implemented to a very limited degree. Furthermore, the results from the survey show that CCU improvements have a significant positive effect on product success. Finally, the results demonstrate weak points in CCU processes, such as a lack of CCU tools.

In earlier work the CCU evaluation model was presented [143], and validated at nine product software vendors. This work, however, could not easily be generalized to be applicable to a larger group of product software vendors. The survey enabled a further detailing of the model, adding weights to practices and capabilities using an expert panel. Furthermore, the survey enables us to validate our hypotheses from the nine case studies and to show that CCU is an area that urgently requires more research.

The survey results show that CCU is an underdeveloped area that requires more attention and tooling. The areas that deserve most attention are the release, delivery, and deployment processes, due to the fact that product software vendors on average only implement between one third and one half of the capabilities of each practice. The main results of such research will improve customer retention rates for product software vendors, product success, and product update and deployment reliability.

CCU research and tools should be geared towards release planning, knowledge delivery, component updating, and feedback reports. This is indicated by four facts. First, only 37% of the software vendors uses a formal release planning. Secondly, only 20% of the software vendors use their own product to share knowledge with customers, such as product knowledge and product news. In the area of deployment many product software vendors do not perform any configuration correctness checking or provide any checks before installation of a product. Finally, only 30% use automatic error reporting

from customers for the usage and activation process.

One of the most important contributions of the survey is that CCU improvements show a significant relationship with software product success. When also taking into account that 16% of product software development personnel is involved in CCU related processes, CCU proves to be a fruitful area for research and evaluation. Part of our future work will be to explore a so-called organizational maturity indicator, where we attempt to find several measures (company size, product age, lines of code, etc) to establish at what level the CCU processes of a software vendor should be performing, based on the indicator. In the future we also hope to do the survey internationally. Currently the survey is still active (in Dutch) /citeotherKCUSite. Also, we are currently using the survey for one organization that has 46 product lines that all use different methods to support the CCU processes.

The fact that many software vendors underperform in crucial CCU processes and the many suggested products by vendors that are currently unavailable on the market, suggest that CCU is a developing process area of software engineering. When looking at software products currently available to support CCU processes it becomes clear that especially in the area of knowledge interaction between customers and vendors there is a lack of support tools. As a result of this we have started working on the Pheme [114] prototype communication infrastructure for knowledge delivery. This infrastructure consists of small server applications that are installed on different nodes (for instance customers, vendors, and release servers) that share different knowledge packages, such as updates, product information, product feedback, etc. We hope to implement the tool at a number of the respondents of the survey.

## 4.7 The Survey and the CCU Relationship Tables

In this section we provide the survey questions. Furthermore, the relationships between yes/no questions are provided, including their Pearson value and their probability.

| **Question** | **Question Type** | **Section** |
|---|---|---|
| Please provide the name of your organization | String | Company Profile |
| How many people are currently working within your organization? | Numeral | Company Profile |
| What is the name of your company? | String | Company Profile |
| Please specify the number of people working in your organisation. | Numeral | Company Profile |
| Please provide your products name. | String | Product Profile |
| Please provide a short description of your product. The answer to this question might be presented to other submitters of the survey | String | Product Profile |
| How many active users are there to your product currently? | Numeral | Product Profile |
| Please specify the number of lines of code that make up all components of your product in KLOC. | Numeral | Product Profile |
| Please specify the languages or development technologies used to develop your product. | Numeral | Product Profile |
| Please specify the number of active licenses there are to your product. | Numeral | Product Profile |
| Please specify the industry type for your product. | Numeral | Product Profile |
| How much time (in FTEs again) is spent on the processes described in this survey? | Numeral | Product Profile |
| In how many natural languages is your product available? | Numeral | Product Profile |
| How many developers are currently working on the product in FTEs? | Numeral | Product Profile |
| How many years have passed since the first line of code was written for your product? | Numeral | Product Profile |

Table 4.7: Survey Part 1

| **Question** | **Question Type** | **Section** |
|---|---|---|
| How often do you publish a major release of your product? ( Daily, Weekly, Monthly, Every 3-6 months, Every 6-12 months, Every 1 to 3 years, Every 3-5 years, 5+ years, We haven't had a minor release yet, Not applicable ) | Option list | Release Frequency |
| How often do you publish a minor release of your product? ( Daily, Weekly, Monthly, Every 3-6 months, Every 6-12 months, Every 1 to 3 years, Every 3-5 years, 5+ years, We haven't had a major release yet, Not applicable ) | Option list | Release Frequency |
| How often do you publish a bugfix release of your product? ( Daily, Weekly, Monthly, Every 3-6 months, Every 6-12 months, Every 1 to 3 years, Every 3-5 years, 5+ years, We haven't had a bugfix release yet, Not applicable ) | Option list | Release Frequency |
| Which of your updates are evaluated by pilot customers before actual releasing the update? ( Major releases, Minor releases, Bugfix releases, Content releases ) | Option list | Release Frequency |
| Please also state how many pilot customers on average test the product before (any type of) release. | Numeral | Release Frequency |
| Does your company use a formal release planning that states dates or durations for the next major | Yes/No | Release Planning |
| Is the release planning published in such a way that all employees who are involved with the product can see it (whenever)? | Yes/No | Release Planning |
| Is there a formal publishing policy with regards to this document? | Yes/No | Release Planning |
| Are releases planned according to customer demands? | Yes/No | Release Planning |
| Is there a formalized release procedure or scenario that states exactly what happens on the day of a release? | Yes/No | Release Planning |
| Releases are stored: ( in a versioned repository, on a network drive, on CDs/DVDs, at customers ) | Option list | Release Planning |
| The latest (published) release can be downloaded or requested by: ( All release personnel, All development employees, All employees, All partners, All customers, All sales personnel ) | Option list | Release Planning |
| All tools created by your organization that support the CCU process (release, delivery, deployment of your product) are managed explicitly as if they were commercial products. | Yes/No | Release Planning |
| Are external relationships managed explicitly such as MyProduct requires MSSQL? | Yes/No | Release Planning |
| All commercial tools that are used to support the CCU process (release, delivery, deployment of your product) are managed explicitly as if they were commercial products. | Yes/No | Release Planning |
| Does your organisation use COTS components that are shipped with the product? | Yes/No | Release Planning |
| Are these COTS components stored in a repository in accordance with the releases of your product? | Yes/No | Release Planning |
| We inform our customers of new releases through: ( e-mail, our website, a paper newsletter, domain specific channels (conferences),through the product, phone, We push the update to the customer automatically ) | Option list | Release Planning |



Table 4.8: Survey Part 2

| **Question** | **Question Type** | **Section** |
|---|---|---|
| Customers inform us of bugs by: ( An on-line bug system, e-mail, phone, fax, the product itself automatically sends an error message to us ) | Option list | Knowledge Delivery |
| We inform our customers about the product on at least a: ( daily basis, weekly basis, monthly basis, 3 monthly basis, yearly basis, never ) | Option list | Knowledge Delivery |
| Our products can be delivered using the following methods: ( our website, CD-ROM, floppy, secure phoneline or internet connection, e-mail, DVD,USB Stick ) | Option list | Knowledge Delivery |
| Our products are delivered using the following mechanisms: ( manual pull, automatic pull, manual push, automatic push ) | Option list | Knowledge Delivery |
| Our product's update facility can be used to download updates from ANY location | Yes/No | Knowledge Delivery |
| Is your product delivered to customers on ( a "get-what-you-paid-for" basis, by sending the full set of components to customers where a license (file) later deactivates the unpurchased modules? ) | Option list | Knowledge Delivery |
| What types of payment do you use? ( Pay per usage, Pay per user(name), Pay per time unit, Pay per floating user, Pay for services, Lump sum ) | Option list | Licensing |
| Does the license file contain information on: ( The customer name and address?, The amount of users?, The modules that have been purchased?, The customers' server or CPU id?, The customers' contract id? ) | Option list | Licensing |
| Can the license be renewed or downloaded by the customer without intervention from you? | Yes/No | Licensing |
| Do your licenses expire? | Yes/No | Licensing |
| Can you provide temporary licenses? | Yes/No | Licensing |
| Can the customer manage the license explicitly? | Yes/No | Licensing |
| Are licenses generated from contracts automatically? | Yes/No | Licensing |
| What tools do you use for the following processes? Please type p̈roprietaryïf you have developed the tool yourself. | String | CCU Tool Support |
| What tools do you personnally believe are lacking in the industry for support of the CCU process? | String | CCU Tool Support |
| How many percent of your deployments are unsuccessful the first time? | Numeral | CCU generic |
| What can be done to reduce this number? | String | CCU generic |

Table 4.9: Survey Part 3

| **Question** | **Question Type** | **Section** |
|---|---|---|
| In what release formats is your product released? ( rpm, exe (wise install), Portage, zip/rar archive, exe (installshield), msi, msi WIX, exe (PowerUpdate), APT-GET, Proprietary installer format, source bundle ) | Option list | Deployment |
| Is it possible to check the configuration of components for completeness regularly? | Yes/No | Deployment |
| Can updates cope with customisations made by customers or third-parties? | Yes/No | Deployment |
| Is the configuration of a customer checked before deployment to see whether the product can safely be deployed? | Yes/No | Deployment |
| Can updates be done at runtime? | Yes/No | Deployment |
| Is the update process semi-automatic? | Yes/No | Deployment |
| Is it possible to uninstall your product without too many manual steps? | Yes/No | Deployment |
| Is it possible to rollback from an update? | Yes/No | Deployment |
| Can your product be deployed in a DTAP environment? | Yes/No | Deployment |
| Are you aware of each customer's configuration with regards to customisations the customer has built into your product? | Yes/No | Deployment |
| Does your product use usage reports? | Yes/No | Usage and Activation |
| If errors are encountered are they automatically resolved? | Yes/No | Usage and Activation |
| Does your product do automatic feedback reports in case of errors? | Yes/No | Usage and Activation |
| Are you aware of each customer's configuration with regards to her/his operating system and hardware configuration? | Yes/No | Usage and Activation |
| Are you aware of each customer's configuration with regards to your product? | Yes/No | Usage and Activation |

Table 4.10: Survey Part 4

| **Question** | **Question Type** | **Section** |
|---|---|---|
| What process discussed in this survey will you invest in in the near future? | String | Generic |
| Please indicate how your release, delivery, and deployment processes have developed over the past two years. ( Not at all, Small improvements (new configuration management system), Large improvements (completely automatic license generation from contracts), Complete process redesign ) | Option list | Generic |
| Please indicate how these improvements (if any) have influenced your organization. ( Lower costs, Better product quality, Shorter release cycles, Less deployment errors, Shorter bug discovery times, Better product stability, More knowledge about customers ) | Option list | Generic |
| When compared to competitors ( lagging behind, at a comparable level, much better ) | Option list | Generic |
| Please prioritize the main reasons why you wish to improve or have improved the release, delivery, deployment, or activation and usage process. ( To serve more customers, To shorten release times, To shorten the time before bugs are found, To serve customers more cost effectively, To decrease deployment and update failures, To improve a customer's product experience, To provide a more flexible licensing and pricing model. ) | Option list | Generic |
| Please indicate the development of your product over the last two years (when possible). ( The product is highly successful compared to two years ago, The product is fairly successful compared to two years ago, The product has been neither successful nor unsuccessful. , The product is less successful than two years ago, The product is a lot less successful than two years ago. ) | Option list | Generic |
| Please indicate how this development over the last two years has been influenced by process improvements made to the release, delivery, deployment, and activation and usage process. ( The product's development was highly influenced by the improvements, The product's development was fairly influenced by the improvements, The product's development was not at all influenced by the improvements. ) | String | Generic |
| Please provide the company's correspondence address. | String | Contact Data |
| Please provide your e-mail address for future correspondence. | String | Contact Data |
| Please provide your name. | String | Contact Data |

Table 4.11: Survey Part 5

| QID | Question | Pearson value | Probability |
|---|---|---|---|
| 125 | Does your product use an update tool that updates the customer configuration? | Related to | |
| 102 | Can your product update during runtime? | 4.6423 | 0.0312 |
| 103 | Can your update tool deal with customizations, extensions, and customer specific solutions? | 4.6875 | 0.0304 |
| 24 | Does your organization use a formalized release scenario that describes step-by-step what happens when a product is released? | 4.5068 | 0.0338 |
| 19 | Does your organization use a formalized release planning that states dates for the next major, minor, and bugfix releases? | 14.4317 | 0.0001 |
| 58 | Do you regularly provide temporary licenses? | 13.7601 | 0.0002 |
| | | | |
| 19 | Does your organization use a formalized release planning that states dates for the next major, minor, and bugfix releases? | Related to | |
| 24 | Does your organization use a formalized release scenario that describes step-by-step what happens when a product is released? | 6.9163 | 0.0085 |
| 32 | Are relationships between external components and your product explicitly managed, such as “ourproduct requires MySql”? | 7.9383 | 0.0048 |
| 102 | Can your product update during runtime? | 6.1945 | 0.0128 |
| | | | |
| 57 | Do your licenses expire? | Related to | |
| 58 | Do you regularly provide temporary licenses? | 16.1141 | 0.0001 |
| | | | |
| 59 | Can the enduser specify under which license he/she will use the application before start-up? | Related to | |
| 100 | Is it possible to undo an update? | 6.5526 | 0.0105 |
| | | | |
| 106 | Is it possible to check a customer’s configuration for completeness regularly? | Related to | |
| 103 | Can your update tool deal with customizations, extensions, and customer specific solutions? | 9.1168 | 0.0025 |
| 89 | Does your product include COTS? | 7.0652 | 0.0079 |
| 90 | Are these COTS stored in a repository, to maintain version compatibility? | 10.2774 | 0.0013 |
| | | | |

Table 4.12: Relationships Between Yes or No Questions (part 1)

| QID | Question | Pearson value | Probability |
|---|---|---|---|
| 100 | Is it possible to undo an update? | Related to | |
| 103 | Can your update tool deal with customizations, extensions, and customer specific solutions? | 5.8681 | 0.0154 |
| 108 | Can the product be deployed in a DTAP (development, test, acceptance, production) environment? | 6.6069 | 0.0102 |
| | | | |
| 112 | Does your product regularly report the usage of your customers by usage reports? | Related to | |
| 57 | Do your licenses expire? | 9.6038 | 0.0019 |
| 111 | Does your product send automatic error reports if a problem occurs on the customer site? | 3.9422 | 0.0471 |
| 106 | Is it possible to check a customer's configuration for completeness regularly? | 5.8311 | 0.0157 |
| | | | |
| 32 | Are relationships between external components and your product explicitly managed, such as "ourproduct requires MySql"? | Related to | |
| 112 | Does your product regularly report the usage of your customers by usage reports? | 4.1983 | 0.0405 |
| 57 | Do your licenses expire? | 6.6359 | 0.0100 |
| 106 | Is it possible to check a customer's configuration for completeness regularly? | 4.5352 | 0.0332 |
| 83 | All commercially purchased development support tools are managed explicitly, including which version of the product has been used to supply a development task? | 11.5870 | 0.0007 |

Table 4.13: Relationships Between Yes or No Questions (part 2)

| QID | Question | Pearson value | Probability |
|---|---|---|---|
| | | | |
| 39 | Can your product settings be changed such that updates and products can be downloaded from ANY location? | Related to | |
| 103 | Can your update tool deal with customizations, extensions, and customer specific solutions? | 6.9543 | 0.0084 |
| | | | |
| 28 | All development and support tools built by your company are managed as if they were purchased externally. | Related to | |
| 32 | Are relationships between external components and your product explicitly managed, such as "ourproduct requires MySql"? | 6.0092 | 0.0142 |
| 113 | Are you aware of all customer specific solutions that have been built for your product by third parties and customers? | 4.5003 | 0.0339 |
| 83 | All commercially purchased development support tools are managed explicitly, including which version of the product has been used to supply a development task? | 19.9523 | 0.0000 |
| | | | |
| 24 | Does your organization use a formalized release scenario that describes step-by-step what happens when a product is released? | Related to | |
| 28 | All development and support tools built by your company are managed as if they were purchased externally. | 8.1318 | 0.0043 |
| 102 | Can your product update during runtime? | 6.1830 | 0.0129 |
| 83 | All commercially purchased development support tools are managed explicitly, including which version of the product has been used to supply a development task? | 14.2998 | 0.0002 |
| 108 | Can the product be deployed in a DTAP (development, test, acceptance, production) environment? | 7.7168 | 0.0055 |
| 89 | Does your product include COTS? | 4.5867 | 0.0322 |

Table 4.14: Relationships Between Yes or No Questions (part 3)

# Part III

# Tool Support for CCU

# CHAPTER 5

# A Process Framework and Typology for Software Product Updaters

*Product software is constantly evolving through extensions, maintenance, changing requirements, changes in configuration settings, and changing licensing information. Managing evolution of released and deployed product software is a complex and often underestimated problem that has been the cause of many difficulties for both software vendors and customers. This chapter presents a framework and typology to characterize techniques that support product software update methods. The framework is based on a detailed process model of software updating. Finally, this chapter assesses and surveys a variety of existing techniques against the characterization framework and lists unsolved problems related to software product updaters.*[1]

## 5.1 Product Updating

Managing evolving software is a complex task for software distributors and vendors. Moreover, maintaining a large software system, such as a business ERP application, can be particularly difficult and time consuming. The tasks of adding new features, adding support for new hardware devices and platforms, system tuning, and defect fixing all become exceedingly difficult as a system ages and grows.

One particular area of software evolution that requires more research, is the evolution of released and installed applications. To deal with the evolution of released software, distributors and vendors currently have the choice of either buying an (expensive) general product updating tool or building proprietary tools. After a thorough analysis, to be presented in this chapter, we conclude that both approaches unfortunately have significant problems since existing software update tools usually do

[1]This work was originally published in the proceedings of the 9th European Conference on Software Maintenance and Reengineering, entitled "A Process Framework and Typology for Software Product Updaters" in 2005 [147]. The work is co-authored with Sjaak Brinkkemper.

not provide all the required functionalities and the effort and risk of building product update tools "in house" is often underestimated.

The contribution of this chapter is threefold. First, a framework is provided that embodies the software update process and the uncovered areas of deployed software evolution. Secondly, a typology is provided to classify software product updaters. Finally, the framework is used to compare current techniques and technology, and to indicate what areas still need to be covered.

Updating software can be seen as moving from one configuration to another by addition, removal, replacement, or reconfiguration of software functionality. A physical software update contains the applicable functionality and configuration alterations. By this definition, changing a license or some configuration setting can also be seen as part of the software update process. A software updater automates the process steps involved with software updates. To discuss the concepts and technologies of this chapter, we introduce the notion of software product updaters.

The remainder of this chapter is organized as follows. Section 5.2 describes what the software update process looks like. The update process is modelled and explained. We also provide a typology for updaters and finally evaluate current software updaters in relationship to the framework. Section 5.3 further defines the steps of delivery and deployment and uses the detailed descriptions to evaluate the same updaters against the detailed descriptions. Finally, we discuss the presented framework and our future work in Section 5.4.

## 5.2 The Product Software Updating Process

### 5.2.1 Update Process Framework

This Section describes the software product update process framework and a detailed description is given of the steps of which the update process consists. The update process model has two participants: the customer and the vendor. The framework, shown in Figure 5.1, is based on customer states and vendor-customer interaction, and has been derived from other update model descriptions and the evaluated tools. The customer blocks are states in this diagram, with the bold lined states being final states. The "Uninformed Customer" state is the start point for the process. Solid arrows are state transitions, which can be activated by both the vendor and by the customer. The dotted arrows show interaction between the vendor and the customer. Once the vendor offers the customer the ability to update a product of that vendor the update process is initiated. The following list describes the process steps:

**Advertise Update -** An update will first be made available in some release repository. When a vendor wishes to provide updates to its customers, the customers first need to be informed through the available communication channels.

**Receive Information -** Customers inform themselves about updates from a vendor through commercial channels, such as web sites, mail, e-mail, and portals. Other channels are memory resident notifiers, such as the Windows Update Notifier, and memory resident processes that automatically start downloading an update once a customer accepts the update that is sent.

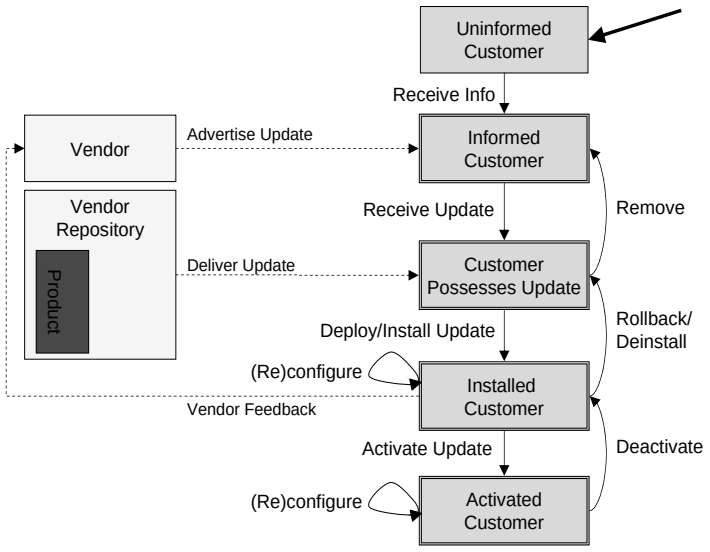



Figure 5.1: Update Process Framework

**Receive Update -** A customer can receive an update automatically and manually. Issues for receiving the update are security, authenticity of the update, and integrity checks. Another issue is the checking of pre-download dependency checks such as available disk space and the presence of dependent components.

**Remove Update -** The presence of the update data that has been downloaded during the Receive Update step, enables switching between configurations and redistribution of updates. For this reason the remove update is an explicit step in the update process framework.

**Deliver Update -** Once a customer has been informed of an update, the vendor wishes to transfer the update to the customer site by mail, e-mail, a website from which the customer can download (pull) the update, or a memory resident process that automatically receives and installs an update. Several issues, which are partly discussed in this chapter, arise when the transfer of an update occurs, such as security problems and the format in which the update is sent to the customer.

**Install/Deploy Update -** A customer installs an update wishing to gain functionality, improve performance, and fix problems. The deployment of updates is the most complex software update process step, and is explained in further detail in Section 5.3.

**Rollback/Deinstall Update -** When a customer wishes to go back to a previous configuration, an update must be rolled back or deinstalled. Deinstallation introduces requirements on the software architecture and its extensibility, such as

state transformations to the older configurations, and incremental updates instead of destructive updates.

**(Re)configure Update -** An update can be (re)configured before activation and after activation. These settings can often be changed at run-time or by editing some configuration file, such as the httpd.conf file for the Apache webserver.

**Vendor Feedback -** An opportunity that is often missed by software producers, but widely used by for instance Microsoft and Exact Software [141], is the use of vendor feedback after the deployment of an update or component. Feedback generated by the deployer of the update can be sent back to the vendor to be used for future testing and feedback on the deployment process.

**Activate Update -** After deployment the update must be activated so that the update can be used by the customer. The activation process step is threefold and consists of configuration, a license approval, and running the update. The configuration binds all unbound variabilities that have been introduced by the update. Licensing, if necessary, makes sure that the software update is used according to the vendor-customer contract.

**Deactivate Update -** Deactivation is required when a user does not want or is not allowed to use the update anymore. The most important part of deactivation is the return of a license key to the licensing system or deployment and distribution system. If a deployment and distribution system is present, the deactivation process can also signal the server so that future updates for the deactivated software are no longer sent to this workstation or workspace.

For our research we have evaluated the coverage of these process steps for a number of techniques currently used in the field or implemented by academia. The evaluation shows what parts of the process framework are still uncovered, how process steps have been implemented by the techniques, and what requirements are imposed by these implementations.

Each of the process steps has specific requirements and problem areas. Two process steps that are crucial for the framework, being delivery and deployment, are further explained in detail in Section 5.3. The release and derelease steps on the vendor side have not been included in this model. The reason for this is that in this chapter we do not focus on the processes that take place on the side of the software vendor. At present our focus lies on the implementation and framework of product update software and we are less interested in the development process of the software that is actually distributed.

### 5.2.2 A Typology for Product Updaters

In order to obtain more insight in the available product update technology we distinguish three types of product updaters. The typology is created because it creates more insight into the specific available technology and draws out the framework for evaluation of product software update techniques. The three types are distinguishable by looking at delivery and deployment methods and policies, and by looking at process coverage.

- **Package Deployment Tools -** During the evaluation of update tools many

package deployment tools (PDTs) were encountered. These deployment technologies are based on the concept of a package, and on a site repository that stores information representing the state of each installed package. A package is an archive that contains the files that constitute a system together with some meta-data describing the system. Examples of these package tools are Red Carpet, APT, Loki-Update, RPM-update, Nix [37], SWUP, and Portage. RPM, Portage, and Nix are the most advanced.

- **Generic Product Updaters -** Generic product updaters (GPUs) are updaters that attempt to be usable for any product. Two generic product updaters that are available commercially are InstallShield [58] and PowerUpdate [115].
- **Vendor Product Updaters -** Vendor product updaters (VPUs) specifically facilitate the update process of one product, such as Microsoft's Windows XP update, Exact Software's Product Updater [141], and Symantec's LiveUpdate.

The typology described above is largely inspired by Carzaniga's grouping [51] of deployment techniques and Ajmani's listing of update techniques [3]. One specific technology has not yet been included in the typology, being runtime updaters, which are further discussed in Section 5.4. This technology, however, can still be described using the updater typology.

### 5.2.3 Evaluation of Update Process Coverage

In Table 5.1 is displayed how the evaluated update techniques cover the process steps that make up the update process framework. The process coverage for update techniques shows different classes of updaters and enables identification of updaters. The process coverage also displays what areas certain techniques focus on and what process steps need more research from both academia and the industry.

• Means that a process is completely covered. ◦ Means that the process is only partially covered. Coverage has been evaluated based on a number of characteristics of each process step, but for the sake of brevity we cannot go into more detail. For instance, partial support for "send update" means that there are means to get the update to the customer, such as a release repository and communication channels. Full support for "send update" means that push technology is also available.

### 5.2.4 Discussion

When looking at the process coverage of the various techniques, there are clear distinctions between the types. One of those distinctions is that current package deployment tools do not support any form of vendor feedback. We will not discuss each type of updater.

The generic product updaters (GPUs) cover many of the process steps. Especially in the area of licensing and customer interaction the GPUs are strongly represented. Firstly, the GPUs have to be used by different parties, sometimes even using different platforms, and therefore need to provide as many different update scenarios as possible. Secondly, the GPUs in this evaluation are, with the exception of the Software

| Product Name | Type | Customer interaction | | | Transferral | | Deployment | | | | Licensing | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Receive Info | Advertise Update | Vendor Feedback | Receive Update | Send Update | Install Update | Roll-back | Re-move | Re-configure | Activate Update | Deacti-vate |
| PowerUpdate | GPU | ○ | ○ | ○ | ● | ○ | ○ | | ● | | | |
| InstallShield | GPU | ○ | ○ | ○ | ● | ○ | ○ | | ● | | ● | ● |
| Red Carpet | GPU | ● | ● | ● | ● | ● | ● | ○ | ● | ○ | | |
| Software Dock | GPU | ● | ● | | ● | ● | ● | ● | ● | | ● | ● |
| FileWave | GPU | | | ○ | ● | ○ | ○ | ○ | ○ | | ● | ● |
| APT | PDT | | | | ● | ○ | ● | | ● | | | |
| RPM-update | PDT | | | | ● | ○ | ● | | ● | | | |
| Nix | PDT | | | | ● | ○ | ● | ● | ● | ● | | |
| SWUP | PDT | | | | ● | ○ | ○ | | ● | | | |
| Portage | PDT | ○ | ○ | | ● | ○ | ● | ○ | ● | ● | | |
| Loki Update | VPU | ○ | ○ | | ● | ○ | ○ | | | | | |
| Exact PU | VPU | ○ | ○ | ○ | ● | ○ | ○ | | | | ○ | ○ |
| MS SUS | VPU | ● | ● | ○ | ● | ○ | ○ | | | | | |
| LiveUpdate | VPU | ● | ● | | ● | ○ | ○ | | | | ● | ● |

Legend: ● Full support; ○ Partial Support

GPU: General product updater; VPU: Vendor product updater; PDT: Package deployment tool

Table 5.1: Update technique Process Coverage

Dock [52], commercial tools, and therefore licensing and customer interaction are required. Finally, when compared to other updaters, the GPUs have most options for vendor feedback, which is a commercially attractive solution for getting feedback from customers.

The package deployment tools (PDTs) are tools specifically designed to deploy and install packages on (usually) open source based systems. These systems are often extended with external tools, which are ignored in this research. The tools therefore cover all standard process steps, but in the areas of customer interaction and licensing they are not sufficient. The reasons for this are part of the nature of package deployment. Firstly, issues such as vendor feedback are solved on another level, usually through bug reporting systems and developer communities. Secondly, licensing is not an issue, since most of the software available in the open source community is free.

Vendor product updaters (VPUs) are generally weaker in the areas of transferral and deployment, yet stronger in the areas of customer interaction and licensing. In the area of customer interaction the VPUs are strongly represented, because that is their ”bread and butter”. One clear distinction between VPUs and GPUs is that removal and rollback is not supported in most VPUs. Whereas GPUs assume that the deployed products will be removed, VPUs assume their products and updates will remain deployed forever, which is not surprising in the case of updates for a virus removal tool or security updates. VPUs are have restricted functionality, because they have been designed to perform these steps for one product and one way of vendor-customer interaction. We see that many of the methods used in VPUs are simplifications of the more complex software update models.

## 5.3 Delivery and Deployment

Two steps in the proposed process model form the core of our model, being delivery and deployment. In this Section the process steps of delivery and deployment are further explained. The updating techniques are then evaluated against the provided definitions.

### 5.3.1 Delivery

Delivery formats identify many characteristics of updaters. Some updaters, such as PDTs focus on the sole delivery of packages, whereas GPUs attempt to support the full myriad of delivery formats. Delivery formats affect the size of updates that are delivered to customers. The choice of delivery format therefore affects the total model of delivery, especially in an environment with limited resources.

New configurations can be delivered to customers in different ways. The configurations can be transferred in the following formats:

- **Packages of Components -** A package of components can be delivered to a customer. Usually these packages first need to be unpacked, before they can be installed and activated. Examples of techniques that use packages are RPM-update, APT, DeployMe, Red Carpet, Portage, and Nix.
- **Components -** A separate component consists of a batch of files.

- **Files -** The simplest form of transfer data are separate files. These files can be licenses, configuration settings, and binaries.
- **File deltas -** Differences between a customer site configuration and a vendor site configuration can be expressed as file deltas. File deltas can be transferred using efficient algorithms such as Rsync [117]. A file delta is a listing of differences between two file versions, with which any of the two versions can generate the other version. Sending just the difference between files is more efficient than sending the complete file.

Without some pre-processing at the customer site, each of these formats would place some restrictions on the final deployment environment. However, when correctly assembled before deployment these formats are interchangeable. For example, file deltas for a complete component can be used to generate the new component. The chosen delivery format(s) affect different factors, such as the size of updates and the deployment method, and together with the deployment issues and deployment policies uniquely identify an updater. Service packs are similar to component packages in our delivery formats.

### 5.3.2 Deployment

The process of installing updates introduces most complexity for software vendors. The software architecture of a system determines the extensibility of the system, whether the update can occur at runtime or not, and whether there are scripting tools available to perform certain tasks (such as *Make*). Finally, dependencies need to be checked during deployment, such as dependencies on the operating system, the presence of certain components, the compatibility between the update and the current customer configuration, and many others.

To deploy or install the delivered software, a choice for an appropriate deployment method needs to be made. Some of these methods are:

- **Overwrite -** The deployment method employed most often by software vendors is the method of overwriting the application files, license files, or configuration settings. The solution bases itself on the assumption that the deployed set of files or components does not change over time due to external forces. There is no way to rollback an overwrite, unless the customer is using a versioned file system. One example of an overwriting update method is the Windows Updater which will first unregister a dll, overwrite it with a newer version, and register it again. Another example is the Exact Software Product Updater, which compares all the versions of the locally available files to the available files on the release site. When there are differences, the product updater overwrites only the different files on the customer site.
- **Plug-in -** Plug-ins are often used to create extensible configurations. The method of using Plug-in architectures simply support the extensions of a configuration by addition and removal of unique Plug-ins. Other Plug-in architectures such as the one presented by Ajmani [2], can handle different versions of the same Plug-in as well.

- **Deinstall/Reinstall -** For many applications an update starts with the uninstallation of all previous installed versions of that application (Examples are: NullSoft Winamp, LavaSoft Ad-Aware, etc).

In the open source community applications are often delivered and deployed as source distributions. These source distributions first need to be compiled, which can be seen as a separate step in the deployment process. Well known systems that assist with source distributions are Maak [34] and RTools [53]. It should be noted that the three deployment methods mentioned above can just as well be applied to source distributions.

Other issues that deal with deployment are the ability of a technique to provide scripting, to do dependency analysis, to perform integrity checking, to deploy multiple versions of the same component, and to enable push technology. Each of these abilities puts specific requirements on the deployment and implementation architecture.

Scripting is used to perform post deployment configuration on an update. Such scripts can be used to execute, activate, configure, compile and build an update. Scripts can be shell scripts, which are often used by package deployment tools, but also a specifically designed language that registers or unregisters Plug-ins. In the framework presented in Figure 5.1 we did not yet introduce verification of an update, such as synchronization checks, signatures, and completeness checkers. In each of the three final states, a customer should be able to perform verification steps.

Dependency analysis is a much studied area of deployment [73] and aims to provide a complete and consistent set of components. To achieve this goal many problems need to be tackled, such as support for multiple versions of components, automatic resolution of dependencies, and explicit management of the dependencies. One specific ability of dependency checking that places extra requirements on the deployment architecture is the support for multiple versions of a component. Multiple version support is therefore part of the evaluation framework and is a technology that enables switching between configurations and having two components depend on different versions of another component. Finally, push technology puts extra requirements on the implementation of the messaging architecture of an updater. A customer needs to be able to receive updates automatically and the vendor needs to be aware of all the customer workspaces.

### 5.3.3 Evaluation of Delivery and Deployment

The evaluation of the following techniques includes more specific definitions of the delivery and deployment process steps than the evaluation done by Carzaniga et al. [21], because the definitions need to be made more explicit. The evaluation shows that updaters grouped by just the process coverage framework do not distinguish subtle yet important differences in delivery formats and deployment policies. These differences have been listed here, and provide a more detailed and defined evaluation framework. To obtain the detailed framework, we have focused on delivery and deployment. Delivery and deployment are more complex than the other process steps, because there are more alternatives to efficiently achieve the goals that are part of these process steps.

The evaluation in Table 5.2 includes a description of what formats of delivery are used by each updater. The evaluation also describes what deployment methods and architectures are supported by each updater. Finally, some issues that uniquely identify an update technique are evaluated. The criteria for evaluation are similar to those for Table 5.1.

| | | | Delivery Format | | | | Deployment Policy | | | Deployment Issues | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Package | Component | Files | File delta | Overwrite | Plug-in | De-Reinstall | Scripting | Dependency Analysis | Integrity Checking | Multiple Versions | Push | Environment Checking |
| | | Type | | | | | | | | | | | | | |
| PowerUpdate | GPU | | ● | ● | ● | ● | ● | | ● | ● | | ● | | | ● |
| InstallShield | GPU | | ● | ● | ● | ● | ● | | ● | ● | | | | | ● |
| Red Carpet | GPU | | ● | | ● | | ● | | ● | ● | ● | ● | | ○ | ○ |
| Software Dock | GPU | | ● | ● | | | ● | ● | ● | ● | ● | ○ | ○ | ● | ○ |
| FileWave | GPU | | | | ● | | ● | | | | | ● | | | ○ |
| APT | PDT | | ● | | | | | | ● | ● | ● | ● | | | |
| RPMupdate | PDT | | ● | | | | | | ● | ● | ● | ● | | | ○ |
| Nix | PDT | | ● | | | | ● | | ● | ● | ● | ● | ● | | ○ |
| SWUP | PDT | | ● | | | | | | ● | | ● | ● | | | |
| Portage | PDT | | ● | | | | | | | ● | ● | ● | ● | | ○ |
| Loki Update | VPU | | | | ● | | ● | | | | | ● | | | ○ |
| Exact PU | VPU | | | | ● | | ● | | | | | | | | ○ |
| Windows XP SUS | VPU | | | | ● | | ● | ● | | | ○ | ○ | | | ○ |
| LiveUpdate | VPU | | | | ● | | ● | ● | | | | ● | | | |

Legend: ● Full support; ○ Partial Support
GPU: General product updater; VPU: Vendor product updater;
PDT: Package deployment tool

Table 5.2: Update technique Business and Deployment Issues

From the evaluation of the updaters against the descriptions of delivery and deployment we deduce the following. To begin with, the generic product updaters (GPUs) support all different delivery formats. The two most advanced tools in this category, PowerUpdate and InstallShield, are the only tools able to deal with all formats of delivery. These are also the only tools that are able to send across file deltas,

instead of complete files. The GPUs are not well represented in the deployment feature area, because these features are specific to deployment environments, which the GPUs must ignore to reach larger markets. However, GPUs are quite able when it comes to commercially interesting push technology, especially when compared to the other updater categories. GPUs generally do not make use of Plug-in technology, which can be explained by the fact that Plug-ins are largely dependent on the Plug-in software architecture. GPUs generally supply the scripting feature since it is required to perform post installation configuration steps.

The package deployment tools (PDTs) support only package deployment and generally only support deinstallation and reinstallation to update a package. Scripting and dependency analysis are always present in package deployment systems, to enable post deployment configuration and completeness checking with other components. PDTs do not use push technology, which can be explained by the fact that (open source) users of these PDTs often do not want others to be in charge of their software. PDTs are strongly represented in the areas of dependency analysis and integrity checking. The dependency analysis is required for PDTs because packages have many dependency relationships with other packages. Automatic resolution of these dependencies therefore is a valuable feature. Integrity checking prevents instability and ensures authenticity.

Finally, the Vendor product updaters (VPUs) all depend on files as the primary format of transfer to the customer. These files generally overwrite the previous installation, except when these files are special Plug-ins, such as virus definitions for LiveUpdate or unregistered dlls for Microsoft SUS. The VPUs do not incorporate much dependency analysis, scripting, or integrity checking. Finally, the VPUs do not make use of push technology.

## 5.4 Discussion and Future Work

The aim of this chapter is to show that there is no product updater that provides all functionalities required by software vendors. On the other hand the development of VPUs is not an efficient solution, since each software vendor is implementing a subset of the process steps shown in our framework. It is surprising that no GPU has yet been adopted universally by the industry. One of the reasons for presenting the framework in Figure 5.1 and the typology is to redefine the requirements on and re-establish the need for such GPUs.

### 5.4.1 Typology

The types presented in the typology all have specific requirements and functionalities. To begin with GPUs are generally commercial tools focused on deploying software on Windows based systems, with the exception of PowerUpdate, which is now focusing on multi platform deployment.

Secondly, the discussed PDTs have some interesting characteristics. Nix, for instance, is a “stop the world” system, whereas Portage and RPM simply extend current functionality. Nix, however, stores components in isolation from each other in

a part of the file system called the store, where each component has a globally unique name that enables pointer scanning. The construction of component configurations and the resulting closures are described using Fix store expressions. Safe deployment is achieved by distributing these expressions, along with all components in the store referenced by them.

Another interesting PDT is Portage. Portage, as most other package management system, can resolve dependencies; but one feature that makes it different is the fact that it also supports conditional dependencies. By changing one configuration variable in a Portage configuration file it can disable optional support (and thus the need to depend on it) for particular features or libraries at compile time. In addition Portage enables multiple versions of packages installed simultaneously to satisfy the demands of other packages. The traditional approach to this problem has been to treat different versions of the same package as different packages with slightly different names, such as with RPM and APT.

Thirdly, there are advantages and disadvantages to VPUs. To begin with there are commercial advantages to VPUs. An important reason for software vendors to use VPUs instead of GPUs is that they themselves are responsible for the update processes of their products. For Norton Anti-Virus for example, Norton is completely responsible for security procedures, network management, and all other aspects having to do with product updating. Often VPUs are a cheap solution over GPUs, however, VPUs can only cover a small problem area compared to general product updaters and the complexity of the software updating process grows as requirements increase. When requirements are stated for the product updater to support different versions, customers, customizations, and licenses, it soon becomes apparent to the software vendor that specialized knowledge is required. The limited availability of such tools and the cost of implementing a GPU, have lead many software vendors to develop their own VPUs and essentially reinvent the wheel. Another disadvantage is that the updaters commonly perform destructive updates. Microsoft Software Update Services, for instance, overwrites dlls, without any rollback functionality.

A category of update technology that is not specified in this chapter is runtime updating, because run-time updating is not widely applied for software products yet. Much work has been done in the areas of runtime and dynamic updating [55]. Providing a service or system that is available 24 hours a day is a commercially attractive solution to many problems. These systems of course also evolve with time, thereby requiring some extensible mechanism. We shall not list these mechanisms here, but Ajmani has created a list of mechanisms and component frameworks [3]. There are two important factors to consider when looking at runtime updating, being continuity and state transfer [78] [12]. An interesting technique, designed by Ajmani and Liskov [2], attempts to support many different versions of one component at runtime, thereby enabling runtime extension. Runtime updaters, however, are generally focused around one technology, such as CORBA or J2EE, and do not focus on any other process modules than transferral and deployment. Simple versions of these technologies are often used in other product updaters, such as Microsoft SUS or LiveUpdate.

### 5.4.2 Delivery and Deployment

The discussed features of the deployment process introduce many questions about software updating techniques. To begin with, the file delta format and push technology is not (yet) strongly represented among the evaluated software updaters. The absence of the file delta format can be explained by the fact that bandwidth and disk space are cheap nowadays and therefore the time and money invested in such technology is not profitable. The fact that push technology is hardly available can be explained by the type of software evaluated. Most of the techniques mentioned in this chapter are product updaters and customers are more interested in having a working product than a product that is acutely and always up to date. Secondly, multiple versions are only supported by technologies from academia (software dock, Nix) and Portage. The complexity of dealing with multiple versions of the same component, which is crosscutting through a system, has not received sufficient attention. Finally, practically all tools perform some deployment environment checking, whether the tool checks for disk space, such as the Exact Product Updater, or provides an advanced customizable checking mechanism, such as the PowerUpdate and InstallShield GPUs.

### 5.4.3 Future Work

One requirement that has as of yet not been discussed is what Carzaniga et al. [21] refer to as site abstraction, the ability to abstract from the vendor-customer model and introduce one or more redistribution sites into the model. Carzaniga et al. already refer to a redistribution tool, the Interdock, in their model, yet no implementation has yet been created. An open research issue is to redefine such an architecture where (re)distribution of components, files, licenses, and configuration settings are modelled.

The aim of the issues listed in this chapter is to explicitly define software update problems experienced in the field. One striking conclusion that can be drawn from the evaluation is that re-configuration is generally unsupported for product updating. Another problem can be found in the fact that many of the requirements of software vendors for product updaters are not yet satisfied by GPUs.

The listed techniques can support the industry and can be inspirational for those designing their own technique. The presented material paves the way to build a generally applicable product updater. However, many of the problems mentioned in this chapter have already been solved by tools such as Nix and the Software Dock. Our plan is to reuse some of these techniques.

### 5.4.4 Related Work

Carzaniga et al. [21] described some of the techniques mentioned in this article, however, recent developments have lead to new insights and techniques. For the evaluation, a list of techniques focused on runtime updating from Ajmani [3] has been used. On the lower levels of component update architectures, Clegg [23] provides an evaluation of component update methods for implementers of run-time updating.

### 5.4.5 Conclusion

The contribution of this article is threefold. To begin with we present a framework that models the software update process and uncovers the areas of deployed software evolution that require more research. Also, we provide a typology that classifies software updaters. Finally, we use the framework to compare current update tools.

## 5.5 Short Description of Update Technologies Used

**PowerUpdate -** PowerUpdate is a commercial multiplatform software updating and delivery tool designed to maintain software applications. PowerUpdate can be integrated into integrated development environments and supports features such as environment analysis and cross platform deployment. PowerUpdate can also check integrity of products on the customer side.

**InstallShield -** InstallShield is PowerUpdate's largest competitor and differs from PowerUpdate in that it is only suitable for deployment on Microsoft based environments and cannot do integrity checking.

**Red Carpet -** Red Carpet is a software deployment tool for Linux. Red Carpet works through installation channels that can be used to communicate and deploy updates at customers. Red Carpet supports automatic dependency and conflict resolution. One important feature of Red Carpet is that they provide Ximian, which is basically a server that contains many different packages that can be deployed for free.

**Software Dock -** The Software Dock, a project that started at the University of Colorado, is a system of loosely coupled, cooperating, distributed components that are bound together by a wide area messaging and event system. The components include field docks for maintaining site specific configuration information by consumers, release docks for managing the configuration and release of software systems by producers, and a variety of agents for automating the deployment process.

**FileWave -** FileWave is quite similar to Red Carpet but provides less features. Mostly, FileWave focuses on deployment of applications on Mac OS X environments, though recently they have started to support Microsoft based environments as well.

**APT -** The Advanced Package Tool installs packages and manages dependencies automatically for Debian environments. APT has been implemented for Red Hat by Connectiva.

**RPMupdate -** RPM is the Red Hat Package Manager.

**Nix -** Nix is a system for software deployment developed by the Trace research group. It supports the creation and distribution of software packages, as well as the installation and subsequent management of these on target machines.

**SWUP -** Swup is short for "Software Updater" and can automatically update packages together with *cron*, independent of the package manager.

**Portage -** Portage is the package manager for Gentoo Linux. Portage has some slight advantages over the other package deployment tools, such as conditional dependencies.

**Loki Update -** The Loki Update Tool is a small tool written to support the most trivial tasks of updating, such as downloading and installing.

**Exact PU -** The Exact Software Product Updater provides the mechanisms for delivering packages and updates to the customer. When the product updater is run at the customer site, it needs to be provided with an installation location (CD ROM or the Web), a license file and a local installation that is updated.

**Microsoft SUS -** Microsoft Software Update Service is used for Microsoft Office Update and Windows Update to deliver service packs, bug fixes, and security updates to customers. The updater works mainly at runtime.

**LiveUpdate -** Symantec provides different types of protection systems for computers connected through a network. Symantec's Antivirus and Firewall software is widely used, and are updated through LiveUpdate. Our evaluation also includes the license tool LiveSubscription, because it covers a relevant part of the update process.

# CHAPTER 6

# Modelling Deployment using Feature Descriptions and State Models

*Products within a product family are composed of different component configurations where components have different variable features and a large amount of dependency relationships with each other. The deployment of such products can be error prone and highly complex if the dependencies between components and the possible features a component can supply, are not managed explicitly. This chapter presents a model driven deployment method that uses the knowledge available about components to ensure correct, complete, and consistent deployment of configurations of interrelated components. The method provided allows the user who has all required knowledge to perform analysis on the deployment before the deployment is performed, thus enabling exception and impact assessment before making any changes to the system. The method and model are discussed and presented to provide steps towards an alternative to current component deployment techniques.*[1]

## 6.1 Component Deployment Matters

The deployment of enterprise application software is a complex task. This complexity is caused by the enormous scale of the undertaking. An application will consist of many (software) components that depend on each other to function correctly. On top of that, these components will evolve over time to answer the changing needs and configurations of customers. As a consequence, deployment of these applications takes a significant amount of effort and is a time consuming and error-prone process.

Software components are units of independent production, acquisition, and deployment [108]. Software deployment can be seen as the process of copying,

[1] This work was originally published in the proceedings of the 3rd Working Conference on Component Deployment, entitled "Modelling Deployment using Feature Descriptions and State Models for Component-Based Software Product Families" in 2005 [142]. The work is co-authored with Sjaak Brinkkemper.

installing, adapting, and activating a software component [52]. Usually the only way to find out whether a deployment has been successful is by running the software component. This leads to frustrating and complex deployment processes for both the software vendor and the system manager. There are many reasons why components that have been deployed onto a system cannot be activated and run. The factors that increase complexity during the steps of building, copying, installing, adapting, and activating a component are numerous.

To begin with, there are relationships amongst components. Components can explicitly require or exclude a specific revision of a component. Some components allow for only one version of the component to be deployed onto one system, placing a restriction on the components that are to be deployed onto that system. If such relationships are not respected the deployments will result in missing components and inconsistent component sets. Secondly, deployments are also complex due to the fact that components can be instantiated in different shapes and forms, to provide variable functionality with the same component [65]. A component that supports variability can have different features that are offered to the user, which are bound and finalized at different times during the deployment of that component. The binding time of a component can be at different stages of the component deployment, such as build-time or run-time. Thirdly, the order in which components are deployed can determine whether the deployment process of a set of components is successful. Components require other components during the deployment process and these can be removed when the system has a limited set of resources. Also, when components exclude each other and different deployment orderings are possible, the possibility arises that one of these orders does not ensure correct deployment. The above holds especially for component based product families [17], where many different variants of one system are derived by combining components in different ways.

A components' lifecycle consists of different states, such as *source*, *built*, *deployed*, and *running*. Many parts of the process of a component going through these phases have been automated to do such things as Components off the Shelf (COTS) evaluation, automated builds, automatic distribution, automatic deployment, and automated testing. Current component lifecycle management systems, however, do not support different component (lifecycle) types, variability, component evolution, and are not feature driven. One of the main reasons for initiating this research as described in chapter 5, is that the current tools for component deployment do not take into account variability, different types of distribution (source, binary, packaged), and different binding times [29].

There are tools that can manage the lifecycle of components, such as Nix [37], the Software Dock [52], and Sofa [57]. These systems have downsides however. To begin with, Nix is a technology based on an open source environment that can guarantee consistency between components and allows for concurrent installations of components. The biggest downside of Nix is that it requires a system manager to "stop the world", i.e., to adjust all the components the system uses to include a component description and reinstall the system. The Software Dock can be used to deploy software using the XML based Deployable Software Description [50] for describing the software. The Software Dock does not support the complete lifecycle of a software component. It does, however, focus on the complete deployment process of software,

including such states as *activated*. Finally, SOFA is a CORBA based component model that uses the OMG Deployment and Configuration specification [89], and is also focused on the deployment of generic components. Sofa, Nix, and Software Dock assume simple lifecycles with four states, being *source*, *built*, *deployed*, and *running*. Of these three component tools, only Nix focuses on variable features provided by different instantiations of components, and only Nix discusses the opportunities for a transparent configuration environment [36]. Software component developers often use their own specific deployment tools or custom built checks to see whether the system on which the component is deployed satisfies all requirements for consistent and correct deployment [21]. The developer therefore must develop its own models and formalizations to ensure a correct component deployment.

The situation described above calls for a generic modeling technique that can handle the complex issues that are introduced by the use of variable components that can be instantiated in different versions and forms on one system. Such a system requires a central knowledge base that stores the variables that initialize the different varieties of component instances and a categorization of such knowledge. This functionality is not provided by any of the systems described above (see chapter 5). This chapter presents a modeling technique that can support the deployment of a component in different versions and variants, and still guarantee consistency and correctness. The modeling technique is based on a central storing of the restrictions and knowledge about component features and the system, thus allowing all components to use such information for correct build, release, testing, deployment, and activation of software. The deployment method presented here can be classified as goal oriented or model driven deployment [109].

The rest of this chapter is structured as follows. Section 6.2 describes the knowledge that deals with a software component and its different features, revisions, and states. Section 6.3 describes how the knowledge can be used to create an instantiation tree of component instantiations by using the provided algorithm, thus enabling the user to reason about deployment of software components. The algorithm is clarified with an example. Finally, we discuss the proposed methods and models in Section 6.5 and describe the conclusions reached throughout this research in Sections 6.6 and 6.6.

## 6.2 Component Descriptions

Currently used component models generally do not support consistent and complete deployment. Most models used by conventional technology [147] such as InstallShield and RPM-Update, focus on the artefacts that make up a component. These technologies perform some very general dependency resolution and only support the “Requires Always” relationship. These tools often have some scripting capabilities that can be used to check whether the right resources are available, if required components for the deployment processes are available, and to perform some pre- and post-installation checking of the artefacts. These qualities, however, are underemphasized.

The model proposed here is based on three viewpoints. To begin with, a component is not merely a set of artefacts. A component has a context that describes

the relationships to the components, hardware, and configuration information that affect the component upon and after deployment. A component also has internal variability, influencing that context, which is bound at different times. Secondly, if a component model supports variability, component features must be communicated to the user. This allows for the user to select these features at different stages of the deployment, changing the context as the component is built, activated, copied, and run. Thirdly, components are available in different revisions. When relationships amongst components can be specified with a specific version number, many deployment problems can be averted. Most deployment tools, including Nix [35] and RPM-Update, already have advanced versioning and dependency resolution mechanisms.

To summarize; there are four factors that make the deployment of a software component with internal variability complex. Each of these factors is handled by the presented model using specific modeling techniques and model extensions. These are as follows:

- **States -** Components can exist on a system in different incarnations simultaneously. These incarnations, such as a *source* incarnation or *installed* incarnation, have relationships to each other. States enable the modeling of the complete build and deployment process, by describing such relationships as "to build this component the source is required first". The introduction of states leads from Figure 6.1(a) to 6.1(b).

- **Revisions -** Components are generally available in different revisions. Different revisions have different states, thus leading to a separate set of states for each component revision. Such revisions are modeled in Figure 6.1(c).

- **Features -** A component can have variable internal functionality, depending on parameters that have been bound at several times during the deployment process. Such points in time are known as binding times. These features can be modeled using a feature description language. In the model provided, each revision of the component source leads to a new feature description, since the code and thus the variability options might have changed. In Figure 6.1(d) this is displayed by the addition of feature trees per revision.

- **Relationships -** Component states have explicit and implicit relationships to each other. An example:the *built* state of an e-mail client requires both an instance of a *source* state of the e-mail client (explicit) and an instance of a running build tool state with a C++ compiler feature (implicit). These relationships can be further classified into "Requires Always" and "Requires Once" dependencies. An example of a "Requires Always" is that a *running* state instance requires a library at all time during the existence of the instance. An example of a "Requires Once" relationship is when a build instantiation requires the compiler only during its instantiation.

To support the presented techniques a component description describes the component name, revisions, the revision's states, and the revision's feature diagram. This definition shows that a component has one or more revisions. Each revision consists of a set of component states, a feature diagram, and a number of feature

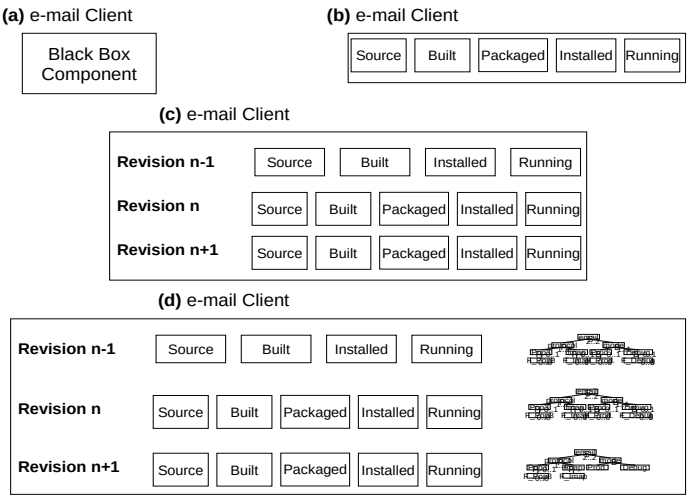



Figure 6.1: Expanding Model for Software Components

restrictions expressed by feature logic. The component states describe the shape or form in which a component can be present on a system. Examples include *built*, *activated*, and *running*. Component states can have relationships, such as *e-Mail client running always requires e-Mail client installed* or *e-Mail client built requires a running build tool once*. In the following sections these component states are described further. The feature diagram is used to describe the features a revision of a component supplies to the user and is defined using a feature description language (FDL) [67, 126]. The feature restrictions describe whether features exclude or require each other. Figure 6.1 does not show feature constraints. These constraints are, however, an important part of our model. Features and FDL will also be explained further in the following sections.

### 6.2.1 Component States and Instantiations

The introduction of component states has many reasons. To begin with, component states force a developer to manage component relationships, restrictions, and deployment environment from the moment the component is created. Component states allow for a more detailed specification of component requirements. Some tools, such as Nix and SOFA, already include implicit state models with the states *source*, *built*, *installed*, and finally *running*. Also, component states enable the component developer to specify and manage the process of how to create a component state instance.

A component state can generally be seen as a portable encapsulation format for a collection of artefacts, relationships to other component states, and a number of state

instantiations. A state instantiation is a list consisting of actions and requirements that upon fulfilment of all the requirements performs the list of actions to reach the requested component state (when fired). In the presented model component states belong to one revision. A revision of the component can thus have a set of component states in which it can reside. The component state definition describes the component state's name, its instantiation list, and its relationships to other component states. The component state has relationships to other component states including *Requires Always* and *excludes*. These requirements are actually expressed as a combination of a component state and provided feature(s). This allows for a component state to have a relationship with a component state with a specific feature, such as *excludes(eMailClientRunning, Pop3)* which can be interpreted as "this component state cannot exist on a system concurrently with the eMailClientRunning state instance that provides the Pop3 feature".

Once the developer is forced to consider component states many possibilities arise. To begin with, the processes of automated building, testing, and deployment can all be performed using the same component state model. Secondly, since the developer can describe any type of component [23] the component model described here can be used to manage different component types, such as Corba or Java components, using the same model. Thirdly, since component state instances are portable, component state instances can be distributed amongst different systems. The model allows for derivation of component dependencies, and can therefore be used to create complete packages of component state instances to be delivered to customers. Finally, a component state model allows developers to model and reason about component updates. Component states can have relationships with component states from other revisions, thus enabling modeling of complex patch or update processes. Such processes are, after all, nodes in the instantiation tree, which represents the full update process.

Previously some criticism was expressed toward a four component state model. The main reason for this is that evidence was found, during case studies at a number of software vendors, that there is a need for more states. The software vendor Planon [62], for instance, has a component state model with seven states, being *source*, *built*, *packaged*, *packaged with license*, *installed*, *activated* and *running*. Another software vendor applies six states, being the same ones as Planon but without the *packaged with license*. This software vendor builds plug-ins for Autocad for which the software vendor actually adjusts the component state model. The software vendor first unpacks Autocad from its installation package, then binds some variabilities, and then packages Autocad with their plug-in. In this real life example a component state has been added to the Autocad state model as well.

A component state description is only a description and does not have any effect on a system. To create a state instance on a system, a component instantiation is required. A component state instantiation consists of a list of actions and a list of component state instances that are required to execute the instantiation and create an instance. The component instantiation consists of a *Requires Once* list and an action list. The *Requires Once* list shows what component states and features are required once the instantiation is activated. The *built* instantiation will generally require a compiler and a component instance of *source*. The actions are specified as a tuple of *(precondition, action, post condition)*. These actions usually are operations on artefacts, such as copy

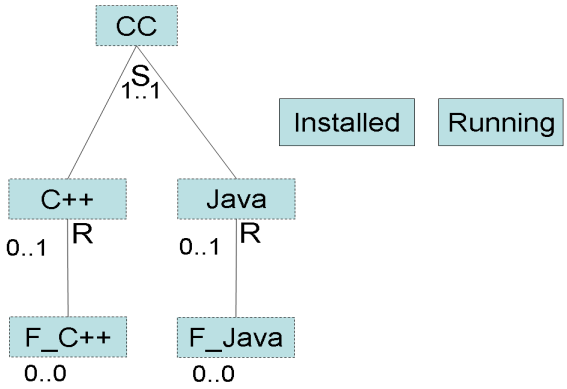



Figure 6.2: Compilation Component Example

or edit actions, but also scripts that modify the database of a product.

As mentioned earlier a component state can be instantiated to create a component instance on a system. The component instance is actually a simple data structure, containing a lists of artefacts created for this instance and a list of features this instance provides. It is well possible that a system contains multiple instances of a component state. An example is when a developer creates a debug build and a production build on his system. Another example is when a user concurrently starts two instances of an application.

To clarify the concepts of component state description, component state instantiation, and component state instance the following example is used. Figure 6.2 displays a compilation component, its feature tree, and the binding time of the feature. The feature tree can be interpreted as follows. The compilation component has one main feature, that is bound as soon as the component is instantiated, called “build”. The compilation component also has two features that mutually exclude each other (one-of). The next section provides more information on the feature descriptions at hand. When executed, the compilation component can build either C++ and Java code. The user binds this feature at run-time, i.e., when a developer wishes to compile his Java code, he will state at start-up time that the code to be compiled is written in Java. The R in figure 6.2 stands for *running* and corresponds to the *running* component state. It is necessary to remind the reader that the figure does not show anything about the state of the system. A system can contain just the knowledge about this component, but also multiple instances of this components’ states, such as two installed versions and one running.

For this example a system containing an installed version of the compilation component is used, implying that the component state *installed* has been instantiated on the system once. This component state instance can be used to create a *running* instance. The relationship for the build tool is “Compilation Component Revision 1: State Running Always Requires Compilation Component Revision 1: State Installed”. The fact that it is a *Requires Always* relationship can be derived from the fact that it is the state that requires another state, and not an instantiation that requires another state.

The *Requires Always* relationship describes that as long as a component state instance is present on a system, the required component must be present too.

The relationship described in Table 6.1 is common for all component revisions that have a *running* state, implying that the *installed* state instance cannot be removed as long as there is a *running* instance depending on it.

Once the presence of the *installed* state instance has been confirmed, we must check for feature bindings. In this case that means a choice must be made between Java or C++. Once the right language has been chosen the instantiation of the component state can be performed. As mentioned before, it is well possible to instantiate a state multiple times, for instance to do a parallel compilation of different source files. The aim of the algorithm described in Section 6.3 is to create an instantiation tree of component state instances, instantiations, and features. An instantiation tree for this component revision is quite simple, since no instances from other components are required and the component only has one revision. It will consist of two nodes, with the node "Compilation Component Revision 1: Running" depending on "Compilation Component Revision 1: Installed".

### 6.2.2 Feature Diagrams

Components often provide different features depending on variables that determine the final configuration of a component. Such variabilities can be bound at different times, such as build-time, package-time, or run-time [121], depending on different variables, such as user preferences, compiler flags, other components, operating system type, or hardware restrictions. Both the different features provided by a component and the binding time of these variabilities must be defined as part of the component description to be used during the process of deployment.

Using these features and binding times during the process of deployment in combination with the component state models provides many advantages. To begin with, it becomes possible to request a specific feature from a component. Also, to enable automatic binding of variabilities at specific times requires a developer to externalize the actions that bind these variabilities thus creating a transparent configuration environment [36], where all configuration options, binding information, and configuration settings are externalized and managed separate from the component artefacts.

To express variability we use the varied feature description language (VFDL). VFDL is a succinct, natural, and non-redundant language [15] that can be used to express features of components or products within a product family that contain any number of other components. The VFDL describes *and*, *or*, *mutex*, *xor*, and *requires* feature relationships. The *and* relationship is described by using a variation point that states that each of the features must be selected, by stating "S..S", where S equals the number of available features. An *xor* relationship can be described by introducing a variation point with two children stating "1..1", which means that one and only one feature can be selected. In case an *or* relationship must be represented a variation point is introduced stating "1..S", where S is the number of nodes and 1 means that at least one must be chosen. An *optional* relationship is described by adding a variation point is added stating "0..1" and using F_node is added that can either be chosen or be ignored.

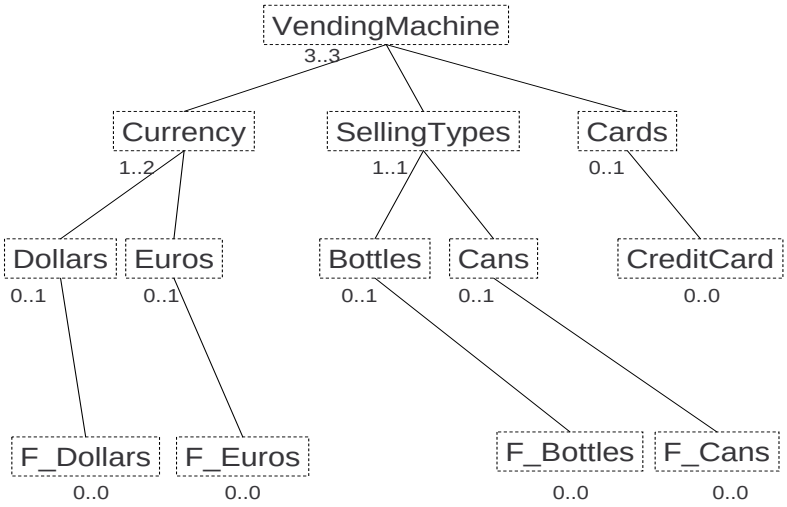



Figure 6.3: Varied Feature Description Language Example

If two features exclude each other, they share a top variation point (using "1..1"), and each feature is optional.

To explain this situation, we provide these relationships in an example, shown in Figure 6.3. The diagram shows a short feature description of a soda vending machine. The vending machine can sell either bottles or cans. The machine can accept one or more types of currency. The credit card feature is optional.

The advantages of using a feature description language to express variability are numerous. FDL allows us to describe complex composition relationships, such as *one-of*, *optional*, and *more-of*, for features. If we then annotate these features with component state requirements it enables the creation of large component compositions. This is best clarified with an example of an e-Mail client that can both support the IMAP and Pop3 protocols (see Figure 6.4 for its feature tree). The binding time of these features is at install time. This means that one or both of these protocols can be installed. If these features have Requires relationships with an IMAP and Pop3 component, it becomes possible to deploy (and build) only the minimal required set, which is useful for space restricted systems such as mobile phones. If a user chooses the IMAP protocol, only the IMAP component needs to be deployed onto his system.

Another advantage of using FDL to describe our feature model is the fact that there are many tools available to perform calculations and operations on the feature descriptions. More specifically, the techniques developed by van der Storm [123] allow for automatic composition of components using feature trees. Many of his techniques are reused here.

To satisfy the research goal of incorporating binding times as well, each relationship between two features, such as *one-of* and *more-of* is annotated with a

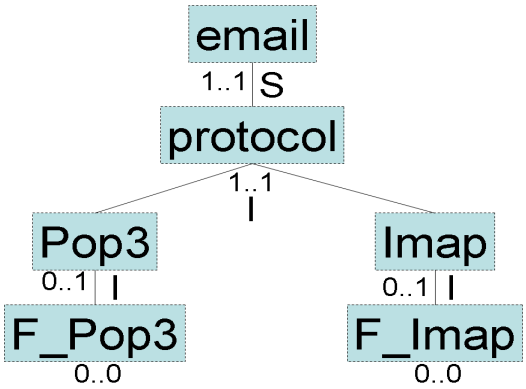



Figure 6.4: e-Mail Client Feature Tree including Binding Times

binding time. Binding times are directly related to component states in our model so each of these relationships is annotated with a pointer to a component state. Next to that, features have two lists of requirements attached to them, being *Requires Once* and *Requires Always*. Features can require component state instances and other features.

| State or Feature | Component State with Feature(s) |
|---|---|
| Requires Always | |
| ECR1: State Running | ECR1: State Installed |
| ECR2: State Running | ECR2: State Installed |
| CCR1: State Running | CCR1: State Installed |

Table 6.1: Requires Always Relationships

## 6.3 Instantiation Trees

To sum up the previous section the presented model consists of components. A component can be instantiated in different states. Furthermore, components can have different revisions. Component revisions have feature descriptions with binding times attached. Component states require other component states (with specific features) to be instantiated. Features can also require other component states with specific features. We will now use this knowledge to create instantiation trees.

### 6.3.1 Example 1: Instantiating the Pop3 Component

The aim of the following example is to clarify the workings of algorithms 1 and 2, to be presented in section 6.3.3. In Figure 6.5 five components are displayed. The components are an e-Mail client (EC), a Pop3 protocol implementation component (P3), an IMAP protocol implementation component (IM), a binary patch component

| Ins No. | Instantiation | Component State with Feature(s) |
|---|---|---|
| | Requires Once | |
| 1 | ECR1: Instantiation Built | ECR1: State Source |
| 1 | ECR1: Instantiation Install | ECR1: State Built |
| 1 | ECR1: Instantiation Built | CCR1: State Running with Java |
| 1 | ECR1: Feature Pop3 | P3R1: State Built |
| 1 | ECR1: Feature IMAP | imR1: State Built |
| 1 | ECR2: Instantiation Built | ECR2: State Source |
| 1 | ECR2: Instantiation Package | ECR2: State Built |
| 1 | ECR2: Instantiation Built | CCR1: State Running with Java |
| 1 | ECR2: Instantiation Install | ECR2: State Package |
| 1 | ECR2: Feature Pop3 | P3R1: State Built |
| 1 | ECR2: Feature IMAP | IMR1: State Built |
| 2 | ECR2: Feature Pop3 | P3R1: State Built |
| 2 | ECR2: Feature IMAP | IMR1: State Built |
| 2 | ECR2: Instantiation Built | ECR1: State Built |
| 2 | ECR2: Instantiation Built | PAR1: State Built |
| 1 | P3R1: Instantiation Built | P3R1: State Source |
| 1 | P3R1: Instantiation Built | CCR1: State Running with Java |
| 1 | IMR1: Instantiation Install | IMR1: State Source |
| 1 | IMR1: Instantiation Install | CCR1: State Running with C++ |

Table 6.2: Requires Once Relationships

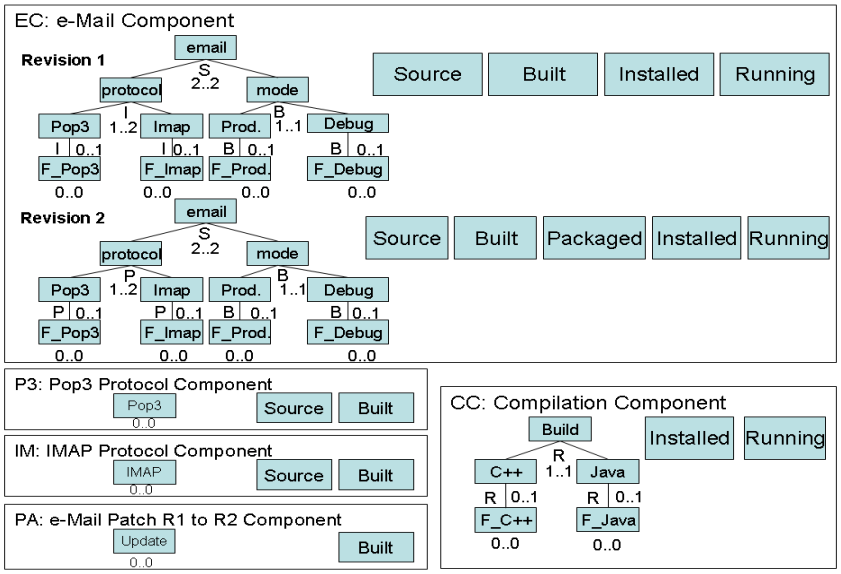



Figure 6.5: Component Definition Examples

for the e-Mail client (PA), and a compilation component (CC) that can compile Java and C++ source files. The e-Mail component is the focal point of our example and to instantiate the e-Mail client with certain features, all these other components are required. For the sake of brevity, a naming convention is used that individually identifies each component state, being CNRX: State, where CN stands for component name abbreviation, RX stands for revision number X, and State describes the state we are referring to. If we want to describe the second revision of the e-Mail client in its running state, it shall be referred to as "ECR2: Running".

To ease the reader into the creation of instantiation trees, a small example tree is created first as shown in Figure 6.6. The figure displays the instantiation tree for the first revision of the Pop3 component for the state *built*. The tree can be used to establish an order in which these states must be instantiated. The example allows for only one instantiation order, and requires that both the source code of the Pop3 component and the *installed* state of the compilation component are present.

The main function is called as follows: *createTree(P3B1, Built,* $\{\}$*)*. First the algorithm checks whether the requested feature or state is already present on the system, and if so, the node is returned as a bottom node with the alreadyExists flag set to true. If the feature or state is not present, the algorithm will check whether it is possible to provide the feature set requested in *requestedFeatures*. If these features do not result in an inconsistent or incomplete set the user is approached with questions. To begin with, the branches are created of component state instantiations this component directly depends on. Secondly, the branches are created of instantiations for this component state. A component state can have multiple instantiations, as we will later see in our larger example. The example shown, however, only has one instantiation branch

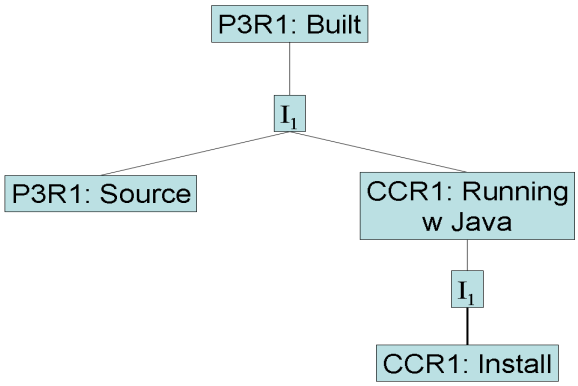



Figure 6.6: Instantiation Tree for Component State P3R1: Built

with one other dependency. The dependencies for this example are listed in Table 6.2. The first table is the Requires Always dependency table. The second table is the Requires Once dependency table. The instantiations and states are identified by their abbreviation and their revision. For the current example the components compilation component and Pop3 are abbreviated to CCR1 and P3R1, where the R1 stands for revision 1. The first entry in table 6.2 for instance reads "to get the e-Mail client in its running state, it must be installed at all times during its run". The example can now be used to tell us that before we can instantiate the *built* state of the Pop3 component, we must first instantiate the compilation component with feature Java.

1. First call createTree(P3R1, Built, {})

2. We see no features being requested, and no illegal feature set. Nu customer intervention is required at this point in time.

3. We look up what other components are always required (requiresAlways) and what components are required only once (requiresOnce). The lookup tables are tables 6.1 and 6.2. We find no "requiresAlways" and two "requiresOnce" for this instantiation, being "P3R1: Source" and "CCR1: Running". The first one is already present on the system.

4. "CCR1: Running" always requires an installed state instance "CCR1: Installed". This leads us to another state instance that is already present on the system: "CCR1: Installed".

The instantiation tree (as shown in figure 6.6) shows that we can now decide an "instantiation order". For this tree the order is simplistic, because there exist no options in build up time. In the next example we show how the alrogithm works for a larger set of components.

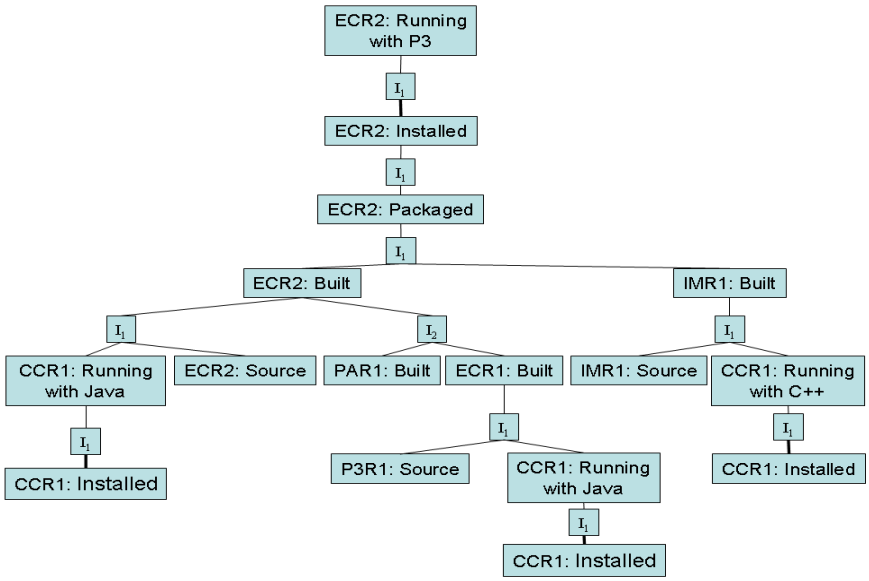



Figure 6.7: Instantiation Tree for Component State ECR2: Running With Pop3

### 6.3.2 Example 2: Instantiating the e-Mail Client

The following example is based on a system that contains two source state instances (revisions 1 and 2) of the e-Mail client, a source instance of the Pop3 and IMAP protocol implementation, an *installed* instance of the compilation component, and a *built* state instance of the update component. The component knowledge in Figure 6.5 will now be used to create an instantiation tree for the state *running* of the second revision of the e-Mail client component with the feature IMAP.

The example begins with the top node, being "ECR2: Running with IMAP". In the table for Requires Always dependencies is found that the *running* instance cannot exist without the *installed* instance of the second revision of the e-Mail client. The second node becomes the *installed* node. This node requires the *packaged* instance of the e-mail client. At this point the tree building has been straightforward. However, the IMAP feature inclusion now causes there to be two requirements at instantiation time, being the *built* instance of the e-Mail client and the *built* state instance of the IMAP protocol implementation. The *built* state of the second revision of the e-Mail client can be reached in two ways, being through the source of the second revision (ECR2: Source) or through the *built* state of the first revision in combination with the patch. The first instantiation thus depends on the compilation component with the Java feature and the source code of the second revision. The second instantiation depends on the patch and *built* state of the first revision of the e-Mail client component. The final instantiation tree can be found in figure 6.7.

### 6.3.3 Algorithms

One application of using feature descriptions and component state models is the creation of instantiation trees. An instantiation tree models a number of instantiation sequences to reach a certain state or feature. A configuration change can be reached by travelling the tree. This section describes algorithms 1 and 2, which creates an instantiation tree from a number of component descriptions and component state instances. We will now describe the two algorithms used to create instantiation trees, while binding the features of each component.

#### Algorithm 1

Algorithm 1 can be described as follows:

**Target:** Bind a correct and complete feature set for one component, while keeping in mind that different features are bound at different states.

**Steps:**

1. Use van der Storms technique to determine whether this is a legal and complete set of features. If it is complete and legal $\{\{\}\}$ is returned. If it is complete and illegal $\{\}$ is returned. If it is legal and incomplete the unbound features are returned.
2. If the list is empty, return a dead node.
3. If the list returns an empty list, do not do anything.
4. When the list is not empty, ask the user to complete the feature selection.
5. Return the node N.

**Remark:** The function attempts to satisfy as many features as possible. However, if the binding time of a conflicting feature is later than the state we are currently interested in, it is of no concern to the feature selection. After all, the newer state might never be instantiated.

**Functions used:**

- *featureSetConsistent(N.requestedFeatures, C.state)* - van der Storms [123] function to check if the feature selection is legal and complete. Returns nothing if not legal, an empty list if complete and legal, and a list of unbound features when incomplete and legal.

#### Algorithm 2

The second algorithm uses the first algorithm to create an instantiation tree. Instantiation trees consist of nodes that have a number of instantiation children. These instantiation children, describing different ways to instantiate the component state,

**Algorithm 1** function bindFeats(N, state)

```
new featureList unboundFeatures = featureSetConsistent(N.requiredFeatures, N, state)
while unboundFeatures != {{}} AND unboundFeatures != {} do
    newReqFeatures = Ask user
    N.requestedFeatures = N.requestedFeatures ∪ N.newReqFeatures
    featureList unboundFeatures = featureSetConsistent(N.requestedFeatures, N, state)
end while
if unboundFeatures == {} then
    N.dead = true
end if
return N
```

have dependencies on other nodes. A node without instantiation children is either a dead node or a state that is already present on the system. Algorithm 2 is described as follows:

**Target:** Create an instantiation tree that enables a user to evaluate changes to his or her configuration.

**Steps:**

1. Get all features that must be bound at this stage
2. Use algorithm 1 to find out if the requested feature selection is legal and complete. If not, interact with the user.
3. Check whether the component instance is already provided on the system. If so, return the node with the alreadyExists flag set to true.
4. Add all required component instances by requested features to the requiresAlways and requiresOnce lists.
5. Create child instantiation nodes and the instantiations on which they depend as children of these nodes.

**Remark:** Once finished the generated tree displays different instantiation "routes". These routes need to be evaluated before being sure that these routes are legal. This is caused by the fact that two choices in an instantiation might exclude each other.

**Functions used:**

- *returnFeatureBindings(component, component.state)* - Returns the features that must be bound to reach the given state and this state alone.
- *bindFeatures(component, N, state)* - Algorithm 2.
- *alreadyInstantiated(component, state, requiredFeatures)* - This function returns true if there is already an instance of a component state present on the system.

**Algorithm 2** function createTree(component, state, requestedFeatures)

```
new featureList stateReqFeatures = returnFeatureBindings(component, state)
new featureList requiresOnce = {}, requiresAlways = {}
new Node N
N.requestedFeatures = requestedFeatures
N.c = component
N = bindFeats(N, state)

if alreadyInstantiated(component, state, stateReqFeatures ∩ requestedFeatures) then
   N.alreadyExists = true
   Return N
end if

for all currentFeature ∈ (stateReqFeatures ∩ N.requestedFeatures) do
   requiresAlways = requiresAlways ∪ currentFeature.requiresAlways
   requiresOnce = requiresOnce ∪ currentFeature.requiresOnce
end for

x = 0, y = 0
for all i ∈ state.instantiationList do
   N.requiresAlways = N.requiresAlways ∪ i.requiresAlways
   N.requiresOnce = N.requiresOnce ∪ i.requiresOnce
   for all ra ∈ N.requiresAlways do
      N.I[y].achild[x] = createTree(ra.state, ra.requestedFeatureList ∪ N.requestedFeatures)
      x = x + 1
   end for
   for all ro ∈ N.requiresOnce do
      N.I[y].ochild[x] = createTree(ri.state, ro.requestedFeatureList ∪ N.requestedFeatures)
      x = x + 1
   end for
   y = y + 1
end for
return N
```

The tree will expand until an instantiation tree is created that shows for each node what component states must be instantiated first before that node can be created. There are some prerequisites, however. Some branches will end because no legal

feature selection has been bound. If there is no sequence available due to the fact that insufficient feature bindings have been specified, the user will need to add more features. Also, if the current system contains no first component instances a problem is encountered, simply because there is no knowledge available and the tree building cannot end. Another problem occurs when a component state diagram includes a circular dependency since that leads to an endless tree. There can be no circular dependencies.

A node N in the instantiation tree has a variable number of instantiation children, a requiredFeatures list, a requiredOnce list, and a requiredAlways list. The instantiation children have child nodes of the same type as N, but are typed either requiredOnce children or requiredAlways children. Please note that the AlreadyInstalled function returns data from the global state. Furthermore, the data that is currently provided in tables 6.1 and 6.2 is also part of the global state. Also, keep into account that at every node, a number of features are potentially bound. To clarify, in the instatiation tree we mark these nodes with their features. An example is the instantiation tree in figure 6.7 where "CCR1: Running with Java" indicates that the compilation component is running with the Java feature selection.

When deciding how a component state will be instantiated with the requested features, two steps need to be taken. First, the tree is created, including the binding of feature decisions. Secondly, a route is determined in the tree. This sequential order has some effects on the final instantiation route, however. Two partial routes, for instance, might exclude each other. The tree must be evaluated and travelled intelligently to find the most appropriate route, based on *requiredConcurrently* sets. There is a large advantage of dividing excludes amongst states instead of full components, since it imposes a minor restriction compared to full component exclusion. The example presented in section 6.3, displays that many components are only required once during the deployment process of others. This allows for removal (of the patch, for instance) after a certain state has been reached. To support this, the *requiredConcurrently* sets have been introduced as a property of the instantiation tree. These are sets of component state instances that must be present on a system concurrently during the deployment of a component state instance. When two component state instances are in a *requiredConcurrently* set they cannot exclude each other (or that partial route becomes unvisitable).

An example of a *requiredConcurrently* set is {"IMR1: Built", "ECR2: Built"} because both components need to reside in their state simultaneously to instantiate "ECR2: Packaged". When two instances exclude each other and do not share a *requiredConcurrently* set, smart removals can be used to decorate an instantiation sequence and still obtain a valid and consistent deployment of component state instances. The instantiation tree shown in figure 6.7 is used as an example. An exclude relationship could be "ECR2: Built" excludes "PAR1: Built". The models can be used to derive two different instantiation sequences. One can exclude the branch containing "PAR1: Built". Another solution can be presented by adding a remove of "PAR1: Built" in the sequence before the instantiation of "ECR2: Built".

## 6.4 Component Knowledge Management

The presented models manage meta-knowledge about components and their features. This meta-knowledge will always describe some kind of software or hardware on a system with a running configuration. The stored meta-knowledge can therefore be seen as a software knowledge base [80] that is used to support the processes of development, release, and deployment. Though the models currently only support component descriptions, our future will include the incorporation of consequences for the system configuration. The knowledge stored can then quickly be expanded to include knowledge about the artefacts that are affected by the deployment of a component or instantiation of a component state. Such knowledge will then enable validation of the deployed component information, through hashes, or the knowledge can be used to check whether a set of artefacts present in a configuration is the correct set for a component instance.

The sharing of knowledge between components introduces many possibilities. Due to the fact that users define their configuration settings, these can be reused for different components. This will allow, for instance, a user to type his e-mail account settings only once, and then reuse the account settings across different e-mail clients. All these different types of information require some kind of categorization. The actual structure in which the component knowledge must be stored still needs to be designed.

The implementation that is presented here only distinguishes one category of knowledge, being component knowledge. However, further categorization of knowledge items is possible into system layers, for instance, where the categories could be divided into a user, component, hardware, and network layer. The layer structure conveniently models the knowledge a component can be related to when instantiated.

The examples and descriptions of actions on knowledge provided above can be categorized.

- **Extraction -** One of the main questions is where the information should come from. Some of the dependency information can be automatically extracted [38], however, most of the information, such as the component states descriptions, simply need to be input by a developer or a party that is well accustomed to the component. One solution that is appealing, is where users of the models share information about components, which allows for reuse of the knowledge. Incompatibilities can also be shared in this case, since they can be sent back to the knowledge provider. Many scenarios are possible here, where information is spread through the BitTorrent [112] protocol or through RSS feeds.
- **Representation -** There have been many attempts to categorize and further define the data structure for a software knowledge base [70]. At present there is no proof as to whether the knowledge representation presented in this chapter is optimal. We consider proving this to be future work, in which different ways of storing knowledge about software components are compared.
- **Inference -** The presented models show the advantages of using the feature descriptions for component states. Many other advantages can be achieved once the component states and instances are linked to artefact knowledge that can be used to establish the validity of a configuration. Such information can also be

used to remove the right artefacts once an instantiation is removed from a system.

- **Application -** The knowledge allows for a myriad of applications most of which are part of our future work. The models can be used to update a software configuration and with the right communication protocols and procedures can even become a generic release, deployment, and update tool, that supports all features (instead of a subset) described in chapter 5. The envisioned deployment tool should also be able to perform automatic deployment within a multitude of scenarios, such as a software product that consists of client and server components that must be deployed automatically on different nodes in a network.

These operations are all applicable to the software knowledge base described earlier. In the case of the presented models, the software knowledge base consists of the component descriptions and the system description. In our future work we hope to add network knowledge to this, so that the software knowledge base can be used to distribute components to other users. In our view the software knowledge base is going to play a big role in the future of component based development, release, delivery, and deployment.

## 6.5 Discussion

The main advantage of the presented models, besides correct and consistent deployment, is the possibility of "what-if" questions. The presented models enable analysis on the deployment of a component *before* the deployment of a component state instance, its dependent features, and state instances. The what-if questions are answered using a number of properties of the instantiation trees. **Excludes relationships** are specific to one state instance instead of components, allowing for components that normally exclude each other to still reside on a system simultaneously. The **tree depth** and instantiation descriptions can be used to evaluate deployment effort. Finally, during the building of the tree, **users can be queried** about what options they have left open, to reduce both the number of possible configurations and to give the user insight into the instantiation order building process.

Another advantage of the instantiation trees is that the depth of the tree can be used to estimate the effort a deployment costs. The example instantiation tree in figure 6.7 has two instantiations below the "ECR2: Built" state instance. The branch on the left has less children thus indicating less steps to a final deployment. This tool must be used with care, however. When an instantiation sequence is shorter, that does not necessarily mean it takes less deployment effort. Another indirect advantage of composing these instantiation trees is that during the composition of such a tree, when unbound features are encountered these can be communicated back to the user. Then the user can bind the feature to see what the results are of that action. The user can now remove the feature if it is not to his liking, before actually executing the instantiation sequence.

There are clear links between the methods applied here and the practices of product lifecycle engineering. This research can be seen as a first step in creating a software product lifecycle management system that can facilitate and support the processes of development, release, delivery, and deployment. The following steps in

this process are a distribution architecture and a knowledge management framework. The closeness between software product management and product data management is further confirmed by Crnkovic et al. [60].

The main downside of the presented models and methods is that the data entered by the component developers is crucial for the correct functioning of the deployment algorithms, since "garbage in" results into "garbage out". As discussed in the previous section, however, there are many possibilities for adding information to the software knowledge base. To begin with, automatic feedback can be used to report back to a supplier of a component after the deployment of that component [148]. This feedback can then be used to test for excludes on external products that a software vendor can never discover independently.

Feature descriptions are rather misused here, since they are generally used to describe high-level application requirements and features [67]. The framework, however, uses feature descriptions to model the binding times of features, the requirements of components, and the relationships between the features. We firmly believe that feature descriptions form the solution to many of the complexities related to component configuration and deployment. Feature descriptions can be used to model binding times and show the relationships between features and other required instances. Feature logic and restrictions allow for complex relations to be modeled and simplified, thus enabling algorithms such as shown in algorithm 2. The final question that needs to be answered is whether a software knowledge base really improves the processes of release, delivery, and deployment. There are four facts that point in that direction.

- **Product data management** improves the release and delivery of other products [54]. Since software production processes share many similarities with other production processes [60], software release, delivery, and deployment can also be improved.
- Since the current trend in the software market is **mass customization**, much of the information gathered in the development stages of the product can be reused at later stages during implementation at the customer and customization phases.
- **Case studies** [148] [62] show that centrally storing knowledge leads to reduced delivery effort.
- The ability to present **"what-if" questions** to a local software knowledge base that is connected to multiple component sources can increase the reliability of the component deployment process. These questions enable a system manager to more explicitly predict what changes can be made to a system and what features can be provided within a certain configuration of components.

Some practical uses of the instantiation trees are explained here, using example 2. To begin with, different instantiation sequences can be derived using the instantiation tree. It is possible, for instance, to first satisfy the right subtree of the instantiation of "ECR2: Packaged" and then decide which of the two instantiations must be used for the left side. An example instantiation sequence for "ECR2: Running" thus consists of "CCR1: Running with Java (to build ECR2: built)", "ECR2: Built", "CCR1: Running with C++ (to build IMR1: built)", "ECR2: Packaged", "ECR2: Installed",

and finally "ECR2: Running w IM". In the following section is demonstrated that it is possible to perform some calculations using the properties and prerequisites for state instantiations.

The algorithms presented have been applied to two theoretical experiments of 10 and 32 components respectively, with approximately 7 features per component. When applied to larger configurations of components, we expect the instantiation trees to become quite large. This presents problems for the instantiation tree evaluation, since currently we do this by hand, using *requiredConcurrently* sets. In the future we hope to create an algorithm that presents the user with a number of routes, such that the user can choose the one he or she prefers. Furthermore, instantiation routes can be determined automatically by storing preferences ("shortest" or "smallest memory signature").

## 6.6 Conclusions and Future Work

Currently the models have been implemented in Prolog, however, to fully apply the models in an industrial setting, a new implementation technology must be chosen with the support of cross platform compilers. We are hoping to apply the tools in a practical situation in the context of a case study. To avoid reinventing the wheel and to standardize the models, the applicability and feasibility of the OMG specifications for reusable assets [91] and IT portfolio management [90] must be evaluated for the current models. The current algorithm blindly builds trees that can explode in complexity quite quickly. There are many opportunities for reuse and further research is required in that area to reduce the complexity of these instantiation trees. Also, the representation of the software component knowledge must be compared to other methods [50] to store and share software component knowledge.

This chapter establishes a relationship between component state models and feature descriptions enabling reasoning about the deployment of a component or component set without actually deploying the software. An algorithm is provided that can build instantiation instantiation trees to determine the deployment order of components. These trees can be used to answer "what-if" questions about the deployment of a component or set of components. The research has shown that both feature descriptions and a component state model are required to create a software knowledge base that stores information about components and their context. The knowledge used to achieve this, relies on information provided by developers and users of the components.

To maximize the usefulness of the models a tool is required that can facilitate the distribution of components and their state model. Such a tool should operate in a distributed environment so that components and their state models can be downloaded when a dependency needs resolving. A tool is also required to publish the component and its state model in different versions. For such a tool different publishing strategies can be implemented, such as "publish all states for that component except for source". In the next chapter we describe such a software knowledge delivery tool.

# CHAPTER 7

# Living on the Cutting Edge: Automating Continuous Customer Configuration Updating

*Product software vendors cannot continuously update their end-users' configurations. By not automating continuous updating, the costs to test, release, and update a software product remain exorbitantly high and time to market exorbitantly long. This chapter shows the feasibility and profitability of continuous customer configuration updating, with the help of two practical case studies and a technique to model and estimate time to market for a software product. Automating continuous updating enables vendors and customers to define flexible policies for release, delivery and deployment, and subsequently reduce time to market with minimum effort. Such flexibility enables customers to be in full control of how and when product knowledge and updates are delivered, tested, and deployed.* [1]

## 7.1 Introduction

Manufacturing product software is an expensive and non-trivial task for software vendors. Software vendors face challenges to divide their resources to develop, release, and deliver high quality software. Unfortunately, there always are more requirements and opportunities than a tenfold of developers could implement. Extreme competition forces software vendors to reduce time to market and to release new features as often as possible. As products and updates are changed and released more often, effort is saved by automating Customer Configuration Updating (CCU).

[1] A short version of this work has recently been accepted for publication in the Proceedings of the ERCIM Workshop on Software Evolution 2007 [149]. Also, a paper describing Pheme has been published at the conference on software maintenance in 2007 [140] for a tool demonstration. The work is co-authored with Sjaak Brinkkemper.

In earlier work CCU has been positioned as the release, delivery, deployment, and usage and activation processes of product software [143]. In this chapter we focus on the automation of these processes to reduce overhead per release. The release process is how and how frequent products and updates are made available to testers, pilot customers, and customers. The delivery process consists of the method and frequency of update and knowledge delivery from vendor to customer *and* from customer to vendor. The deployment process is how a system or customer configuration evolves between component configurations due to the installation of products and updates. Finally, the activation and usage process concerns license activation and knowledge creation on the end-user side.

The importance of CCU is often underestimated as systems nowadays are increasingly supplied to customers as an on-line service, thus requiring no more software delivery and deployment on the customer side. There are multiple sides to this service orientation trend, however. These days, services are deployed on home user systems as well. Simultaneously, an increase is seen in the use of product software on mobile devices, requiring different deployment mechanisms. Examples can even be found of services and products that are deployed on mobile and embedded devices, offered as services, and on customer PCs. One such example is TomTom, offering its product on embedded devices in the TomTom Go, as an online service with TomTom Maps, and deployed on your portable device with TomTom Navigator. Another example is developed by Google, offering the Google Mini embedded device, Google on-line websearch, and Google Desktop for the home PC. Interesting blends of software are becoming more common than traditional retail software products, increasing the need for smart CCU.

Automating steps in the CCU process contributes to product software vendors in four ways. First, software vendors serve a larger number of customers when less overhead is required per customer [143]. Secondly, due to more frequent integration builds and automated test runs developers see results of their contributions quicker and testers can test more recent versions of the software, improving the vendor's internal development process. Also, when deployment is automated developers and testers lose less time (re)deploying the application locally after different developers have contributed to the project. Finally, managing specific policies on both the customer and vendor side enables advanced release, delivery, and (re-)deployment scenarios. Vendors can flexibly define when a product must be released, whereas customers can decide to periodically check for new updates and not install them until formal approval has been given. If customers are involved in testing new features, they can work with newer versions than with traditional weekly or daily builds. Finally, customers are better protected against security holes because they can be patched quicker. Overall, by automating steps in the CCU process a software vendor can reduce its Time to Market and battle the demand for flexibility of its software development, delivery, deployment, and integration processes [14].

In section 7.2 we introduce the concept of continuous CCU and define the research approach and identify tools that potentially automate continuous CCU. One such tool, Pheme, is described in section 7.3 and specifically built to serve this purpose. In section 7.4 we show, by means of two case studies, that continuous CCU can be automated with relatively little effort. Finally, in section 7.7 we conclude that continuous CCU tools

provide more flexible processes at lower costs.

## 7.2 Continuous CCU

By automating steps in the CCU process as much as possible, CCU cost and effort can be reduced effectively and larger numbers of customers can be served [143]. Automating steps in the CCU process ultimately contributes to Continuous Customer Configuration Updating (C-CCU). C-CCU is defined as being able to continuously provide any stakeholder of a software product with any release of the software, at different levels of quality. This way developers, testers, and even end-users can always be fully up to date. Before an organization can properly set up C-CCU, however, policies must be defined for all processes in the software product life cycle. These policies are displayed in Figure 7.1, which models the C-CCU process for any type of product release, such as a new product, a major update, or a minor bug fix. In the software product lifecycle different policies define the method and freqency of the release, delivery, deployment, logging, feedback, and debug processes.

C-CCU must not restrict customers or vendors in any way, instead, it should be a mere facilitator for more responsive software product management. C-CCU enables a software vendor to become more responsive to changes in customer and market demands. It provides developers and testers with the most recent (working) version available. C-CCU is an organizational driver for continuous integration, continuous testing, and continuous quality control. As such, customers are provided with better guarantees of quality when code is released. As long as the software vendor and customer share knowledge and manage it explicitly, CCU becomes less error-prone and less time consuming. The current trend towards agile development only strengthens the belief that C-CCU is essential in the current market and that the return on investment into C-CCU automation is high.

### 7.2.1 Research Approach

The aim of this research is to find out whether automating C-CCU is feasible and profitable for software products. C-CCU automation feasibility is established by showing two cases in which we have automated C-CCU. Profitability is established by showing the low overhead for automating C-CCU. Two cases were selected with different development technology, working on different platforms. Then a tool selection was made to automate and implement the C-CCU process for these applications following an inventorization of C-CCU automation tools. Finally, some aspects of C-CCU have been automated for the two applications and implementation time was written down per feature. The C-CCU automation was considered successful when the following three criteria were met. First, updates to the software had to be seamless, without any manual intervention. Second, the development and delivery process had to be improved significantly. Finally, the investment had to be small.

Threats to validity of this study [133] are that the applications are not representative and that process changes are underestimated. Two applications were selected, Joomla[2]

[2]http://www.joomla.org/

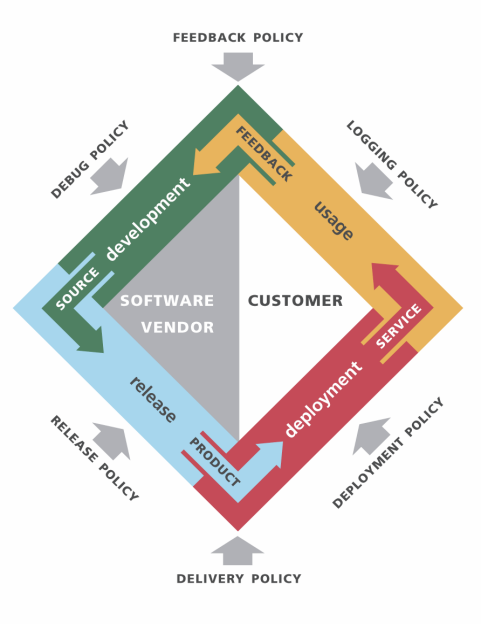



Figure 7.1: C-CCU and its Policy Context

and the Meta-Environment[3], both freely available as open source products. The applications are very different with regard to development methodology (monolithic vs component-based) and technology (PHP/MySql vs. Java, C and several domain specific languages). Finally, due to the similarity between the presented model and development planning models, such as presented in [71], we strongly believe that the presented model is a reliable method to estimate Time to Market.

### 7.2.2 Related Work and Tools

The tools evaluated for this research range from scientific open source prototypes to commercial products with a very high turnover. For a more extensive discussion on product update tools (such as package managers, knowledge delivery tools, etc.) we direct the reader to [147].

**Nix -** Nix [35] is a package management system that ensures safe and complete installation of packages for Unix based systems. Nix approaches software deployment as a memory management problem, concurrently storing different versions and variants of components (values) identified by unique hashes (pointers). Nix ensures atomic updates and rollbacks that guarantee that existing component dependencies never break. Such updates can be downloaded from channels in which updates for specific components are published, and are deployed as binary patches and as source

[3] http://www.meta-environment.org/

patches. Because Nix manages *all* dependencies of every component, it derives continuous releases from continuous integration. With regard to C-CCU Nix supports deployment policy management. However, writing Nix expressions that describe dependencies amongst components in their domain specific language, is not a trivial task. Furthermore, Nix is currently a scientific research tool that is restricted to Unix-like software environments.

**Sisyphus -** Sisyphus (http://sisyphus.sen.cwi.nl:8080/) is a component-based continuous integration and release tool. Component versions are built in an incremental fashion, i.e. only if there are affecting changes, sharing previous build results if possible. Accurate bills of materials (BOMs) are maintained in a database that allows derivation of release packages for every successful build [124, 125]. Passing the integration build is but the first QA milestone. Because every build corresponds to a (internal) release, testers can easily update their configuration on a regular basis. Formal releasing a product simply consists of labelling a particular build that satisfies the required quality properties. This allows the development organisation to setup different channels that have different frequencies of release.

**FLEXnet -** Macrovision's FLEXnet (http://www.macrovision.com) is a suite of release, delivery, and deployment products, such as InstallShield, InstallAnywhere, FLEXnet Connect, and AdminStudio. Macrovision's product suite provides tooling to create releases for any platform, install them on any platform, and let them be managed by a system administrator. Their tools provide licensing, copy protection, and patch delivery solutions. Macrovision's strength can be found in the fact that they manually support many of the process steps that are part of C-CCU.

**Software Dock -** The Software Dock [52] is a system of loosely coupled, cooperating, distributed components that are connected by a wide area messaging and event system. The components include field docks for maintaining site specific configuration information by consumers, release docks for managing the configuration and release of software systems by producers, and a variety of agents for automating the deployment process. The Software Dock is an early attempt to correctly, consistently, and automatically deliver and deploy software. The Software Dock does not focus on release management or continuous release practices and is lacking in the area of knowledge delivery from customer to software vendor.

Each of the tools and research projects discussed in this section is focused on one particular aspect of the customer configuration updating process (see Table 7.1. These properties have been determined by reading documentation and testing the tools with small cases, *A* means that the tool supports the process automatically, i.e., no user intervention is required and policies can be flexibly defined. *M* means that the tool requires manual steps from the user). Sisyphus supports continuous release, but has no feedback loop from the customer to the vendor. Nix is primarily geared towards deployment as is the Software Dock; both are lacking in the areas of release and customer feedback. MacroVision's FLEXnet supports many of the processes manually, but not automatically. The next section introduces the knowledge distribution framework Pheme, which has been developed to remedy this situation. In Greek mythology, Pheme was the personification of fame and renown, described as

"she who initiates and furthers communication"[4].

| | Nix | Sisyphus | FlexNet | SWDock | Pheme |
|---|---|---|---|---|---|
| **Release** | | A | M | M | A |
| **Delivery** | | | A | A | A |
| **Deployment** | M | | M | M | A |
| **Logging** | | | M | | A |
| **Feedback** | | | A | M | A |
| **Debug** | | M | | | |

Table 7.1: Manual (M) or Automated (A) Tool Support for Policies

## 7.3 The Pheme Delivery Hub

Pheme is an infrastructure that enables a software vendor to communicate about software products with end-users and enables system administrators to perform remote deployment, policy propagation, and policy adjustment. The infrastructure consists of a server tool (Pheme), a protocol between software product and Pheme, a protocol between Phemes, and a GUI. The Pheme server resides on each system that acquires and distributes software knowledge through subscribe/unsubscribe channels. The server can accept and distribute all types of knowledge, including policies concerning software knowledge delivery and deployment that describe behaviour of the Pheme tool. These policies can be manipulated securely and remotely.

Pheme enables software vendors to publish software knowledge in the form of licenses (for one end-user), software updates (for a group of end-users), software content, and software news (for another group of end-users). Pheme enables customers to send knowledge in the form of usage and error feedback. A system administrator can use Pheme to instruct other Phemes, change and distribute delivery and deployment policies, control all communication between end-users and vendor, and redistribute software (knowledge). Finally, an end-user can edit policies, execute deployment policies (such as remove/install/update a software product), determine when and how feedback will be sent to the vendor, and refresh all types of knowledge such as licenses.

The processes in Figure 7.1 are all covered by advanced tools and methods, such as release by Sisyphus and deployment by Nix. However, none of the tools fully cover deployment policy and delivery policy management for multiple participants in a software supply network (SSN) [151]. Pheme was created to enable these participants to explicitly manage and share knowledge about software components, and thus provide coverage for the release publication, feedback, and knowledge delivery process steps.

[4] http://en.wikipedia.org/wiki/Pheme/

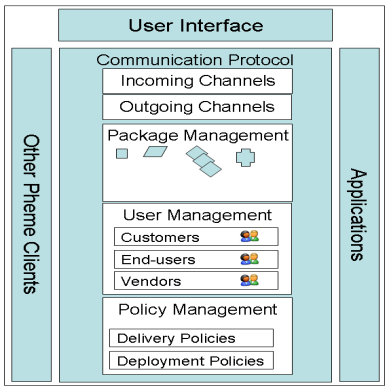



Figure 7.2: Pheme Architecture

### 7.3.1 Pheme Architecture

Pheme's architecture is modelled in Figure 7.2. The core components are policy management, package management, user management, and channel management. The policy management component enables the user of Pheme to define delivery policies (check for and download software updates on a weekly basis, for instance) and deployment policies (check every time when the product is shut down whether any new updates have been downloaded and deploy the newest one). The package management component supplies Pheme with knowledge package support, in the form of files, reports, facts about the product (version numbers, dependencies, etc.) and human readable product news. The user management component enables different types of users to be known to the Pheme instance. Such users can be local administrators, knowledge suppliers, and knowledge consumers. The user management component also enables the Pheme administrator to contact other Phemes and indirectly change policies of other Pheme instances. Finally, the channel management component manages different channels that can automatically and manually push or pull knowledge packages to and from other Phemes and URLs.

Pheme is interacted with through its user interface, through another Pheme, and through the software product itself using SOAP procedure calls. Software products use Pheme as a gateway to vendor release and feedback repositories. Typically such interactions include receiving updates, product news, and sending feedback and usage statistics.

Pheme handles knowledge packages as opaque artifacts, however, in some cases where software product knowledge is required, the package handling component can be used as a fact base. These facts are then used to store, for instance, a product configuration. Pheme then serves as a knowledge base from which both the vendor and the customer can get information. Furthermore, this enables a vendor to develop a specific fix for a specific group of customers. The suitability of the update can then be decided by Pheme.

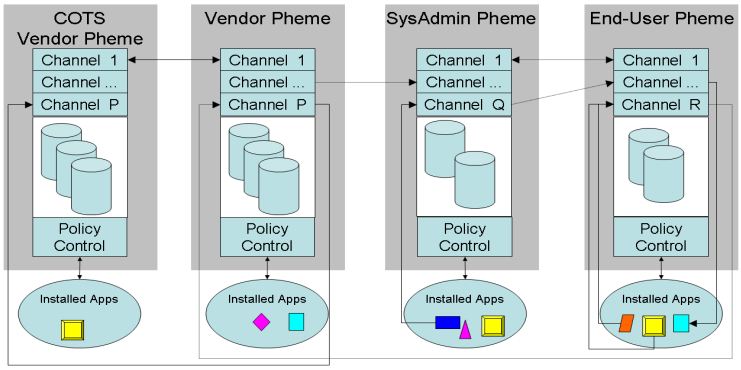



Figure 7.3: Example Environment for Pheme

An example setting is presented in Figure 7.3, where four different instances of Pheme on four different hosts are shown. A Components off the Shelf (COTS) vendor sells components to a vendor, who can distribute knowledge through its channels to the system administrator, such as the announcement of a new release. The system administrator can communicate with the end-user(s) of the product. Finally, end-users send information regarding the day-to-day use of the product back to the vendor. The Joomla case study, presented in the next section, takes place in a similar setting.

## 7.4 Case Study Results

### 7.4.1 Joomla

Joomla is a leading open source content management system that is widely adopted for its intuitive user interface and low barrier for acceptance. It has a large user base and a large team of developers. Furthermore, a large collection of components and user generated content are available. Joomla can be installed on any web server that supports PHP and MySQL. Traditionally, Joomla applications are updated by hand. Minor releases exclude datamodel changes, so these can overwrite current deployments. For major releases tutorials are released that explain the proper method for updating the datamodel and the source code. The manual process of updating is error-prone, since it is possible to overwrite source files from a Joomla configuration with newer versions. This leads to many Joomla deployments that are never updated, which increases the risk of hacks.

The main advantages for the Joomla product in automating C-CCU are found in effort savings. Each developer, tester, and customer will at some point have to deal with an evolving instance of the product because they want new features or wish to plug a security leak. Their time can be saved by not having to manually evolve the configuration of their Joomla instance. Furthermore, we speculate that the Joomla development team can automatically generate and test releases and publish them on-line, which will no longer require the manual effort of checking the release for completeness and such. Finally, Joomla customers can be sure that their CMS

contains the most recent security updates.

To automate C-CCU for Joomla the processes of release, delivery, and deployment are automated. In order to prepare a release, the source code is checked out from the Joomla SourceForge repository periodically. To deliver the source code to the customer the source bundle is sent to the customer system. A policy control system is set up on the customer system to automatically deploy any new versions that come in for Joomla. To automate deployment the system automatically calculates the differences between a new version and the currently deployed version. The changes are then made to the deployed data model and source code.

**Automating C-CCU for Joomla -** To automate C-CCU for Joomla the tools Pheme[5], SubVersion[6], DataDiff[7], and MySqlDiff[8] are used. In this case study a release, an administrator, and a customer system are used. The release system locally prepares the releases by simply bundling the source code, data model, and data into one compressed file. The administrator system is used to instruct the customer system with delivery and deployment policies, such as "check for Joomla bundles from the release system every hour" and "deploy a new Joomla version as soon as it arrives". The release system obtains the sources on a daily basis, by using an automated check-out script. Furthermore, the release system zips the contents of the new release, and places it in a Pheme release channel. The system administrator system also runs Pheme, to instruct the customer system with new policies. The customer system runs Joomla, Apache, and MySql for its daily operations. Pheme is used to check hourly for new updates from the release system and to run the appropriate commands when a new release bundle comes in. When a new release bundle comes in, the release is deployed as if it were a "fresh" deployment. The configuration script, *configuration.php*, is overwritten with the configuration script from the deployed version, with a change in the database name. The data model is then created, using *joomla.sql*. The difference between the data models is calculated using MySqlDiff and a data model update script is generated. The update process is a three part process where the database structure is first updated, the new data is added, and then the changed files are overwritten.

Software delivery and deployment, software knowledge delivery, and feedback delivery were automated for Joomla. Software delivery and deployment cost two full days. The delivery of knowledge cost 4 hours of development time, including the adjustments to Joomla. Finally, to implement logging also cost 4 hours including development time.

**Technical Challenges -** Our approach is not safe. The automatic data model update does not apply any knowledge from the development process, which can lead to data loss, empty columns, and other data update problems [106]. Furthermore, there is no assurance whether the downloaded bundle is of reliable quality. Also, since updates occur at runtime the system can become unstable. For such runtime updating state-safe techniques can provide solutions [127]. To complete the C-CCU process an extension was built to Joomla's error handling mechanism, such that errors and warnings were reported back to Pheme and stored for further analysis.

[5] http://www.cs.uu.nl/Pheme/
[6] http://subversion.tigris.org/
[7] http://freshmeat.net/projects/datadiff/
[8] http://www.mysqldiff.org/

**Process Challenges -** With C-CCU in place, a number of process changes are required from the Joomla team and its customers. To begin with, the release process needs to be redesigned. To guarantee that high quality code is released to customers, a number of criteria can be devised to delay releases, such as test failures. Joomla can also, instead of opting for continuous CCU, define intervals at which new versions will be released (such as a once weekly update). After all C-CCU is a concept that provides maximum flexibility to customers and developers, and should not introduce extra risks or effort. For a future deployment system, knowledge about "preferred update routes" must be specified and shared between the Joomla release team and the Joomla customer, such that quality guarantees can be provided for defined sequences of updates.

**Joomla's Software Supply Network -** To display news from Pheme in the Joomla backend a module has been created called ModPhemeNews. This module will only work with approved versions of Joomla resulting in the fact that the ModPhemeNews team must approve of any new version of Joomla before an update of the full Joomla package can take place on the customer side, which is a common way of working with add-ons. If C-CCU were to be automated in such a process, customers could possess new code quicker because this dependency can be specified and shared. The Joomla creators could, for instance, provide an early release for module and add-on builders. Furthermore, the ModPhemeNews creators could automatically download any new releases for Joomla immediately, automatically test the new combination of Joomla and ModPhemeNews, and notify customers of its approval of the Joomla update.

### 7.4.2 The Meta-Environment

The second case study concerns the Meta-Environment, a integrated development environment (IDE) for language development, source code analysis and transformation. It is a component-based application consisting of tools for parsing, transforming, pretty printing and analyzing programming language sources connected together using dedicated middleware. The Meta-Environment is an open framework that can easily be extended or customized with third-party components. For the Meta-Environment the introduction of C-CCU decreases development, release creation, and deployment effort which is valuable in a setting where the developers are all full-time researchers. Before introducing C-CCU, the Meta-Environment was built using a daily build system, whereas today the Sisyphus continuous integration system is used which builds complete systems on every change to the source control (Subversion). Every component contains a manifest listing its dependencies. The manifest is used by the build system to determine how a component should be built and which earlier build result can be reused to satisfy these dependencies. Before the introduction of Sisyphus, release amounted to manually changing the version numbers of all components and the dependency specification accordingly and creating a release package using the tool AutoBundle [32]. However, since such compositions did not directly follow from the build system and were not formally tracked (i.e., in a database of builds) many mistakes were made in this process. This turned out to be a serious impediment when one wants to do more frequent releases and continuous releases.

**Automating C-CCU for the Meta-Environment -** For the Meta-Environment the

release, delivery, and deployment processes were automated. The release process provides a new release as soon as one of the Meta-Environment components is changed. With regard to release, Sisyphus is used to automatically release the most recent version that fully passed an integration build. To automate delivery, Pheme has been installed on the end-user system to automatically download the latest binary distribution on an hourly basis and deploy it on a nightly basis. Pheme checks whether the Meta-Environment is running and, if it is not, the new release is installed. Since installations of the Meta-Environment are stateless old files are overwritten. The implementation of Sisyphus for the release process took a developer five full days. Furthermore, to automate delivery and deployment with Pheme cost a developer two days.

**Technical Challenges -** The Meta-Environment does not support any error recovery or reporting at runtime. To build such error recovery, the architecture needs to be redesigned in such a way that the component configuration remains robust and can send an error report at runtime before a crash occurs. A complication is that the Meta-Environment is a heterogeneous system; i.e. the components are implemented in different languages (C, Java, proprietary).

**Process Challenges -** The C-CCU process for the Meta-Environment is not perfect. To begin with the quality criteria (it must pass the integration build and a large number of unit tests) are weak. Sisyphus has the infrastructure to provide for stronger criteria. Some manual intervention is required in order to label successful builds with the accompanying quality attributes. By publishing such releases at specific URLs, customers can choose among different release channels, such as “cutting edge”, “daily”, “weekly”, “minor”, “major” etc. Furthermore, there is no facility for users to report problems back to the developers except through the regular media such as e-mail and bugzilla. However, this way users have to manually link their error report to the version they have installed. Subsequently developers have to read the email and manually reproduce the problem. One would wish for the automatic identification of the installed version whenever a problem occurs at the customer site.

**Meta-Environment’s Software Supply Network -** The Meta-Environment is a component composition, with several open source components coming from different development locations. When a Meta-Environment release is created, stable versions of components are explicitly labelled to be part of the next release. This method of releasing reduces risk, although it does force a customer to install components that are potentially outdated. In future releases of the Meta-Environment, automation of Sisyphus and strict quality checking enables releases with the most up-to-date third-party components and a decrease in integration effort. The components of the Meta-Environment themselves are not strongly coupled, meaning that in the future components could be updated independently.

To demonstrate that C-CCU automation actually reduces Time to Market the PDCT models are introduced. These models, based on traditional supply chain and development planning models, enable a software developers to find bottlenecks in their development and delivery processes. A weakness of the PDCT models is that all durations are averages based on earlier events. The actual values are dependent on a large number of variables, such as number of features implemented in a release, number of developers working on this task, number of bug reports that need to be solved, etc. We do, however, see the PDCT models as a powerful tool, to demonstrate how C-CCU

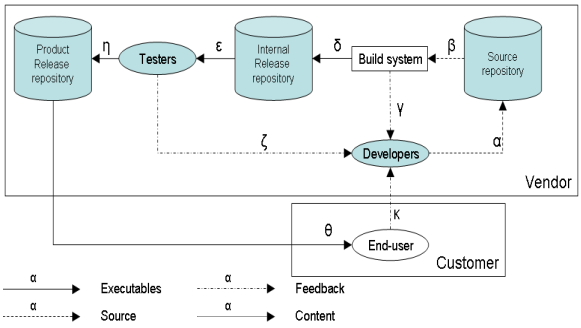



Figure 7.4: Product Development Cycle-Time Model

can reduce Time to Market and as a planning tool. Part of our future work is to evaluate the PDCT in additional real life situations.

## 7.5 Product Development Cycle

To prove that automating C-CCU in a development organization reduces Time to Market, a model is required that assists in estimating Time to Market of a software product. To estimate Time to Market for a software product the **Product Development Cycle-Time (PDCT) model** is introduced (see figure 7.4), in part inspired by Ford's dynamic model of product development processes [46]. A PDCT model displays the development cycle of a product release, that is the organizations involved, participants of the development process, repositories in which versions are stored, automatic testing and build systems, and the flows among participants, repositories, and systems. Such flows can be source code, executables, feedback, and product content. These flows are annotated with average durations in between transactions of source code, executables, feedback, and product content.

In Figure 7.4 a PDCT instance is shown of a software vendor that has a traditional product development cycle. A software developer changes code and commits changes to the source repository. The time between commits is named $\alpha$. Furthermore, a build system attempts to build the source code (every $\beta$) into an executable system. The build time ($\delta$) determines when the executables are done and uploaded into the internal release repository. Should the build fail, feedback is generated and sent to the developer. Testers decide when to download a newly built release ($\varepsilon$), when to approve it ($\eta$), and when not to approve it ($\zeta$). When approved the product release can be downloaded and installed by a customer ($\theta$). Finally, the customer will occasionally provide feedback and send this to the software vendor ($\kappa$).

The PDCT model in Figure 7.4 has three cycles, being the **build feedback cycle (BFC)**, the **test feedback cycle (TFC)**, and the **customer feedback cycle (CFC)** (see figure 7.5 ($V$ = Vendor)). The PDCT model provides a method to estimate the length of these cycles. The BFC is the addition of the development and commit time ($\alpha$), the

$$\text{BFC}(V) = \alpha + \beta + \gamma \tag{7.1}$$
$$\text{TFC}(V) = (A+1) \cdot \text{BFC}(V) - \gamma + \delta + \varepsilon + \zeta \tag{7.2}$$
$$\text{CFC}(V) = (B+1) \cdot \text{TFC}(V) - \zeta + \eta + \theta + \kappa \tag{7.3}$$
$$\text{TtM}(V) = (B+1) \cdot \text{TFC}(V) - \zeta + \eta + \theta = \text{CFC}(V) - \kappa \tag{7.4}$$

Figure 7.5: PDC Times for a Software Product

build frequency ($\beta$), and the time it takes to provide feedback ($\gamma$)(generally the build time, $\delta$). See equation 7.5.7.1.

The TFC is calculated using the BFC. However, before a release reaches the quality assurance department, the product will go multiple times through the BFC (when code is committed that does not pass the build), of which we call the average $A$. To calculate the length of the TFC, the BFC is multiplied with $A+1$, and the feedback time is subtracted. The feedback time must be subtracted once, because in the final cycle when the source code does pass the build, the feedback time is not relevant anymore. We furthermore add the build time ($\delta$), the test time ($\varepsilon$), and test feedback time ($\zeta$). This leads to Equation 7.5.7.1.

To calculate the CFC the TFC duration is required. Similar to the BFC, the TFC is multiplied with the average times the release does not pass the quality assurance team plus once, minus the test feedback duration. Furthermore the time to publish the release in the release repository ($\eta$), the time to deliver the release to the customer ($\theta$), and the time to publish feedback to the software vendor ($\kappa$) are added. This leads to Equation 7.5.7.3. Note that the **Time to Market (TtM)** for new features is included in the CFC, such that $\text{TtM}(V) = \text{CFC}(V) - \kappa$ for a vendor $V$ (Equation 7.4).

### 7.5.1 Joomla Case

The PDCT model is used to demonstrate that the market delivery time of Joomla can be reduced. The model in Figure 7.6 shows both the old (top) and the new situation (bottom) for Joomla. In the old situation major releases are published every three years, minor releases every three months, and bug fix releases are released approximately every 2 months. Upgrading without the DiffTools is a complex task. There are no evolution scripts for the database, and no guarantees can be given whether the update will damage a working website. A common way to update Joomla thus is to migrate all the data from one release to another. This method is tedious and results into Joomla configurations that are rarely, if ever, updated.

Due to the fact that Joomla is built using HTML and PHP code, there is no build cycle. The old TFC consists of the development and commit time ($\alpha$), the test time($\beta$), and the test feedback time ($\gamma$). Furthermore, the Time to Market consists of the sum of $A$ TFCs plus one, the release publish time ($\delta$), the delivery time ($\varepsilon$), minus the test feedback time ($\gamma$). The equations describing the old TFC time and the old TtM can be found in Figure 7.7 (Joomla = $J$).

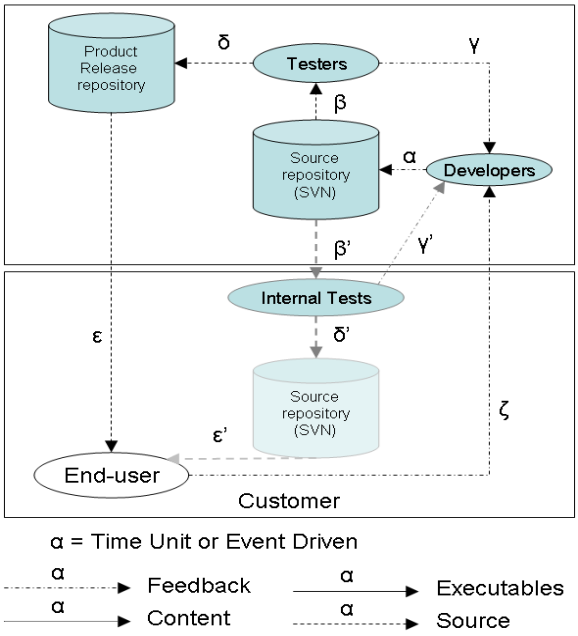



Figure 7.6: Joomla PDCT Model

$$\mathrm{TFC}(J) = \alpha + \beta + \gamma \tag{7.5}$$

$$\mathrm{TtM}(J) = (A+1) \cdot \mathrm{TFC}(J) - \gamma + \delta + \varepsilon \tag{7.6}$$

$$\mathrm{TtM}'(J) = \alpha + \beta' + \delta' + \varepsilon' \tag{7.7}$$

Figure 7.7: PDC Times for Joomla

In the new model tests run on the customer side to decide whether the new release will be installed or not. Furthermore, delivery of new releases $\delta'$ and deployment of these releases $\varepsilon'$ happens much more frequently. We claim that Time to Market (and thus the CFT time) in the new situation is shorter:

$$\mathrm{TtM}(J) > \mathrm{TtM}'(J)$$

A lot of quality guarantees are lost in this case because unapproved files are downloaded from the source repository. However, by automating C-CCU, customer feedback will be generated earlier, which increases product quality.

### 7.5.2 Meta-Environment Case

The Meta-Environment PDCT model exhibits some interesting properties. Since the Meta-Environment is a composition of components that are developed simultaneously by different developers, the BFC time has some optional components. The BFC for the component 1 in the source repository is the development and commit time $\alpha_i$, the build frequency $\beta_i$, and the build feedback time $\theta_i$. The BFC time becomes the longest of any of the separate component BFC times (see Equation 7.9.7.8 (Meta-Environment = $M$)). The TFC time is the BFC time multiplied by the number of times the product does not build plus one, minus the build feedback time (of the longest BFC time). Furthermore the time to build $\gamma$, the tester latency $\delta$, and the longest feedback time are added.

The Meta-Environment used to be built daily. Major releases of the Meta-Environment occur once every two years, minor upgrades every couple of months, and bug fixes were simply added to the source repository or included in daily releases prepared by the daily build system. For these reasons it is not uncommon for end-users to use the daily release over the latest production release. In the new situation the daily build system has been replaced by Sisyphus, which already reduces the BFC greatly. BFC duration is reduced because components that are developing slowly are not holding back those components that are developing quickly. Furthermore, end-users are always guaranteed to have the latest version of all components that have successfully been built together (see Section 7.2.2). Finally, due to the fact that Sisyphus continuously builds components instead of daily, feedback time is reduced.

By implementing an automatic update function, users of the Meta-Environment can be sure to use the latest version. Due to the fact that the Meta-Environment is constantly updated and does not use a version older than 48 hours in our implemented scenario, CFC times are reduced and TtM becomes smaller. By automating C-CCU for the Meta-Environment quantifiable results are yielded, in that Time to Market is significantly reduced. The quality assurance process of course has drastically changed. Whereas before users would have to wait months on end for a new well tested-release, the releases provided to customers now are only approved by Sisyphus against the criteria that it passes the integration build. The aim of this chapter, however, is to show that C-CCU automation can reduce Time to Market, not to compare quality assurance processes.

$$\mathrm{TtM}(M) > \mathrm{TtM}'(M)$$

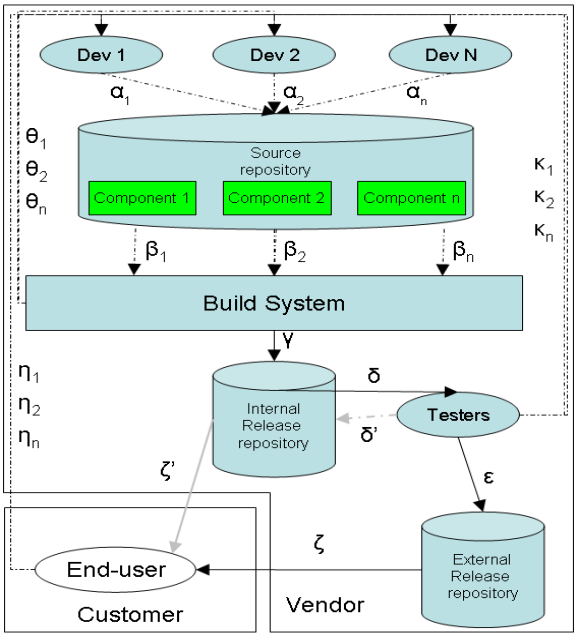



Figure 7.8: Meta-Environment Product Development Cycle-Time Model (simplified to three components)

$$\mathrm{BFC}(M) = \max_{i \in 1..n}((\alpha_i + \beta_i + \theta_i)) \tag{7.8}$$

$$\mathrm{TFC}(M) = (A+1) \cdot \mathrm{BFC}(M) - \max_{i \in 1..n}(\theta_i) + \gamma + \delta + \max_{j \in 1..n}(\kappa_j) \tag{7.9}$$

$$\mathrm{TtM}(M) = (B+1) \cdot \mathrm{TFC}(M) - \max_{i \in 1..n}(\kappa_i) + \varepsilon + \zeta \tag{7.10}$$

$$\mathrm{TtM}'(M) = (B'+1) \cdot \mathrm{TFC}(M) - \max_{i \in 1..n} + \varepsilon' + \zeta' \tag{7.11}$$

Figure 7.9: PDC Times for the Meta-Environment

## 7.6 Case Study Conclusions

The two cases put forward in this chapter demonstrate that Time to Market can be reduced.

An interesting aspect of the PDCT model is that it enables the modelling of delivery speed. While software vendors are trying to decrease Time to Market of new features they are held back by technical restrictions, but also by resource restrictions. One such example is that of a large ERP-Vendor with a customer feedback loop of three and a half months. To decrease delivery time, their development team starts two weeks early on developing a new release. By (re)starting the development cycle of a new release two weeks earlier and disregarding the fact that no customer feedback has come in yet, the development cycle is up and running during the time customers operate the previous release. In these two weeks the developers work on new features and remove bugs found earlier in quality assurance processes. In the following six weeks, when customer feedback starts coming in on the latest release, developers incorporate that feedback as well. This has shortened the release delivery time by two weeks, because now the processes of development and customer feedback partly overlap.

Another interesting case exemplified by the Meta-Environment, is the phenomenon of independently evolving components. As can be seen in Equation 7.8 the BFC time depends on the weakest link, the component that takes most time to develop. Clearly, this enables the prioritization of development activities and critical path analysis. This holds for the Meta-Environment as much as for any developer in a software supply network.

## 7.7 Conclusions and Future work

This chapter presents the concept of continuous customer configuration updating. Secondly, a new tool that facilitates delivery of software knowledge between vendors and customers is presented. To demonstrate the reduction in Time to Market two case studies are presented which show that the effort invested into automating C-

CCU is negligible compared to the reduction in Time to Market. Automating C-CCU reduces overhead from the release, delivery, deployment, and usage processes, enabling software developers to focus on their product instead of the supporting processes and tools. Furthermore, automating and implementing C-CCU in a software vendor's organization enables it to become more responsive to change [13].

The time to automate some of the C-CCU processes (three days for Joomla and seven days for the MetaEnvironment) is negligible compared to the time saved. The quick automation, however, could only be done with the availability of strong development support tools such as Sisyphus and Pheme. With the availability of such tools, software vendors can more easily adopt automatic CCU, since it enables software vendors to pace the heartbeat of software development and delivery [125]. Because C-CCU encourages releasing often, the quality assurance process must be designed to continuously test new releases. Furthermore, by delivering straight from the source repositories the cases only provide crude examples of what can be done by automating C-CCU. These automations would be used to provide recent releases to development, test, and integration personnel in practice.

Automation of C-CCU in such a manner that updates are safely and securely deployed (possibly at runtime) requires knowledge about product structure and architecture. The two presented case studies do not make use of product architecture information and can thus not guarantee safe and secure updates at run-time. A generic tool could be built that automates the update process for different types of software architectures, such as SOAs, plug-in systems, and monolithic systems. Furthermore, part of our future work is to build the Pheme prototype into a commercial product in cooperation with a number of industrial partners.

# Part IV

# Process Improvement

# CHAPTER 8

# Ten Misconceptions about Product Software Update Planning

*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 chapter presents Cost/Value functions for product software vendors to support their release package planning method. These Cost/Value functions are supported by ten misconceptions that vendors had to adjust during their lifetime, encountered in seven case studies of product software vendors. 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]

## 8.1 Introduction

Product software release planning has been characterized as a "wicked" [20] and "complex" [6] problem for which no perfect solution exists. One part of release planning, release package planning, is often underestimated due to its seemingly innocent and simple 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

[1]This work was originally published in the proceedings of the 1st International Workshop on Software Product Management, entitled "Ten Misconceptions about Product Software Update Planning explained using Update Cost/Value Functions" in 2006 [145]. The work is co-authored with Sjaak Brinkkemper.

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 [11]. However, from a number of case studies performed, as described in chapter 2, it is found that product software vendors generally do not sufficiently plan their releases.

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 8.1 displays a release snapshot from a recent case study [62], 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. 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 [100]. This model addresses minor and major releases. These releases are patched (with bug fix packages) until a release is phased out and closed down.

The objective of this chapter 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 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 [101], the method that supports the product software knowledge infrastructure [120], and other methods that support release planning [92]. Section 8.2 presents and describes the Cost/Value functions. Section 8.3 describes ten misconceptions encountered in seven case studies that support the Cost/Value functions as a valid release package planning method. In section 8.4 methods are described to

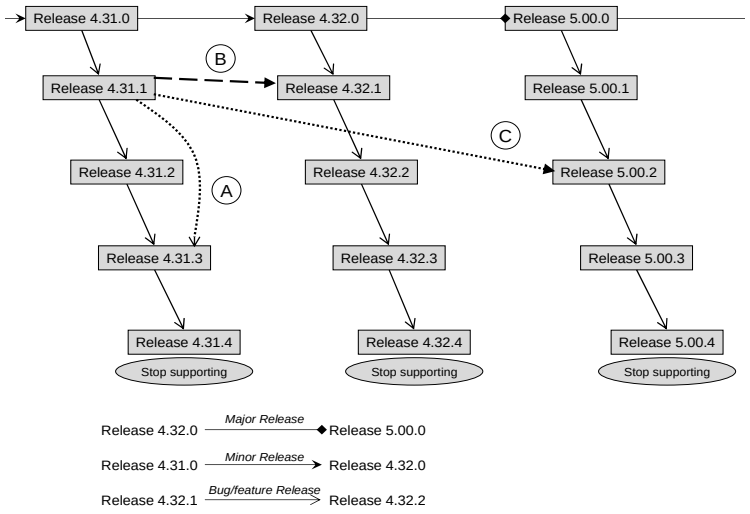



Figure 8.1: Typical Versioning Example

save either customer update costs or vendor side release costs. Finally, in section 8.5 we discuss the presented method and describe the conclusions.

## 8.2 Defining the Cost/Value functions

This section describes the Cost/Value functions for both the customer (see figure 8.2) when updating its software and the vendor (see figure 8.3) 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 the case studies performed at seven organisations described in chapter 2. Also, the customer functions are based on different research from Enterprise Resource Planning (ERP) application updates and migrations [85], and a recent case study we performed at a content management systems (CMS) vendor that also does updates and migrations for customers [63]. The Cost/Value functions are similar to the profit functions developed by the Research Triangle Institute [103]. 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.

$$Cval(update) = value(newFeatures) + value(removalOfWorkarounds) \tag{8.1}$$

$$Ccost(update) = \begin{cases} cost(downtime) & + & cost(training) & + \\ cost(updateEffort) & + & & \\ cost(lostFunc.) & + & cost(paymentToVendor) & \end{cases} \tag{8.2}$$

$$Cval(update) > Ccost(update) \tag{8.3}$$

Figure 8.2: Customer Cost/Value Functions

$$Vval(newUpdatePackage) = \begin{cases} newCusts & * & priceNewRelease & + \\ oldCusts & * & priceOfUpdate & + \\ costReduction(support) & & & \end{cases} \tag{8.4}$$

$$Vcost(newUpdatePackage) = \begin{cases} cost(development) & + \\ cost(updateCurrentCustomers) & + \\ cost(increasedSupport) & + \\ cost(marketing) & + \\ cost(deliveryToCustomers) & + \\ cost(packageCreation) & \end{cases} \tag{8.5}$$

$$Vval(update) > Vcost(update) \tag{8.6}$$

Figure 8.3: Vendor Cost/Value Functions

### 8.2.1 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* (8.1). 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* (8.2) 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* (8.3), 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 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?

To clarify the functions we provide an example of a customer using a web based CMS. The customer discovers a security flaw in its website and consequently in its CMS. The customer must now decide whether or not to get an update (which is freely available from the CMS vendor website) and apply it to the customer's configuration. The update contains extra functionality for which the customer has no use. The customer regards security highly, and therefore values the update at 10,000 euro. There are currently no workarounds in place. The value thus becomes *Cval = 10,000*. To install the update the website needs to be down for 10 minutes (at least) and the vendor estimates the cost of that to be 100 euro per minute. None of the new features require training and no features were removed. The deployment of the update needs to take place on Saturdays so that the business does not get disrupted, costing the customer another 2,000 euro. The cost of an update now becomes *Ccost* $= 10 * 100 + 0 + 2,000 + 0 + 0 = 3,000$. Because *Cval* exceeds *Ccost* the customer will proceed with the update.

### 8.2.2 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 current 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* (8.4) 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* (8.5) 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 [147], 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 artefacts, 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 *Vval* exceeds *Vcost* (8.6), 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 (8.3), 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 have largely different proportions 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. In that case the vendor must ask itself the question whether it was worth it to try and fix the bug in the first place.

To support the function description we provide an example of a CMS vendor. The vendor establishes that there is a security hole in the CMS. Immediately they create an update that fixes the hole and contains some extra functionalities. The vendor now needs to decide whether the update will be released immediately or not. The CMS vendor wishes to provide the update for free, and does not expect to gain any new customers with it. Furthermore, the support department will save an estimated 2,000 euro, because less security problems occur. The creation of the update package costs

the vendor 300 euro. Instructing and notifying all customers costs 500 euro. There will be a slight increase in support of 900 euro. All other costs can be considered 0. The value for the vendor thus becomes $Vval = 0*0+0*0+2000 = 2000$. The cost for the vendor is $Vcost = 0+0+900+500+0+300 = 1700$. Because *Vval* exceeds *Vcost* the vendor will choose to create and release the update.

## 8.3 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 [137]. The misconceptions are generally strategic misconceptions that beginning software vendors can 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 [62] [63]. The main focus of research was 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 chapter describing all seven cases [143]. The Cost/Value functions presented in this chapter support the lessons learnt.

**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 CMS product software vendor, where customers use versions from years back and never updated, due to the large number of customisations and complex update process involved. These customers, however, don't feel limited in their use of the product 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 the customers of a bookkeeping software vendor 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 had 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. One example 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 is a software vendor who called up a customer for a yearly check-up, and heard

that they had recently decided to buy a competitor's 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 pro-active 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 [152] 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 CMSs 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 regarding 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 will be 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.

## 8.4 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 artefacts 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 instance for Exact Software and for its 160,000 customers (described in chapter 3), the reduction in cost by introducing a combined software configuration management system and customer relationship system was huge. By combining these two systems Exact Software enables 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 a lot 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 [11].

### 8.4.1 Vendor Side Cost Reduction

To begin with a vendor must strive to **release often**, if not continuously [124]. 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. 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 the manual checks required before releasing.

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.

When a software vendor reviews its software releasing and updating, a number of things become clear, depending on the update method of choice. There are three different updating methods; destructive, migrating, and incremental. Destructive updates are generally changes that change the data model and files of an application in such a way that rollbacks are impossible. Migrations migrate data from one deployed release of the application to another release. Migrations are often time consuming and expensive, but guarantee low switching cost and downtime. Incremental updates are updates that simply increase functionality by stacking new features on top of old ones. This is commonly seen when software vendors have their own development platforms [152]. An interesting type of migrations is found in plug-in architectures, where a plug-in can be replaced in a short time frame. Furthermore, these three methods can be applied individually for the three types of updates described in this chapter: bug/feature updates, minor updates, and major updates. The decision to use one of these update methods, though often constrained by requirements such as uptime, should be business driven. Most software vendor make the decision early in their development process and do not show much willingness to change at a later stage, unless there are large amounts of customer problems and there is an opportunity for the business. This is striking, especially since a change in deployment strategy (e.g., from migration to incremental updates) can create new business opportunities. In table 8.1 different update methods are shown, together with their risk, investment, and volume of deployments the methods can handle.

| Costs / Method | Risks | Investment | Volume |
|---|---|---|---|
| Migrate | low | high | low |
| Destructive | high | low | high |
| Incremental | low | high | high |

Table 8.1: Update methods and costs

In figure 8.1 three different customer scenarios are displayed. The first one is a possible bug/feature update for the customer, labelled *A*. The second one is a possible minor update labelled *B*. The third one is a possible major update, labelled *C*. The update method for each of these updates determines both the risk and route that must be taken to update a certain customer. In the example of a vendor using incremental updates, example *A* is fairly basic, since these updates are incremental and can be applied freely on the configuration of the customer. When looking at scenario *B*, however, a choice needs to be made. If there are incremental updates available from release 4.31.1 to 4.32.1 there is no problem. It is possible, however, that a different approach needs to be taken because the incremental update has not been created yet. A vendor can then choose to build incremental updates only from any 4.31.X release to 4.32.0, and then apply the basic incremental updates created in the 4.32 branch.

The *C* scenario is not much different from scenario B in this case. Determining the update method and routes through the release branches are all part of release package planning.

Exact Software solved its updating quite elegantly. Their stable products, allowing downtime, can be updated with (destructive) file overwrites and data model alteration scripts. The data model alteration scripts are incremental, whereas for the files only the most recent version is published to customer. Whenever a customer connects to their service website for an update, the customer's files are compared against those of the vendor, and the new files and data model alteration scripts are downloaded. Once the download is finished, the data model of the customer is altered using the downloaded scripts. Immediately after this process the downloaded files are copied over the old deployed release. These destructive updates are possible due to the fact that this vendor simplified its release model, has produced a software product that can be down for a couple of minutes, and because it provided its customers with interfaces for customization in such manner that overwriting files does not cause the customizations to be overwritten.

Independent of the method of updating, vendors can save costs by **scoping the update range**. This scoping forces customers to stay up to date by increasing the costs of an update as a customer is further behind. Not only does this encourage customers to stay up to date but it also makes them more aware of the downsides of not updating frequently. Once a customer runs outside of the scope and wants to update, both the vendor and customer must realize that it might be cheaper for both parties to initiate a migration, instead of an update track.

### 8.4.2 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 regarding 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 newsletters, 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.

Regarding 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.

## 8.5 Discussion and Conclusions

This chapter 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 by product software vendors was that they are using all their resources developing their specific software solution, 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 [40] 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. Part of our future work is 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 will be 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 chapter 5 the lack of tools for software deployment is identified and possible solutions are presented. With respect to continuous software releasing the tool Sisyphus was built to support

product software vendors to automatically create software releases [124]. Work has recently 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.

# CHAPTER 9

# A Modelling Technique for Software Supply Networks

*One of the most significant paradigm shifts of software business is that individual organizations no longer compete as single entities but as complex dynamic supply networks of interrelated participants that provide blends of software design, development, implementation, publication and services. Understanding these intricate software supply networks is a difficult task for decision makers. This chapter outlines a modelling technique for representing and reasoning about software supply networks. We show, by way of worked case studies, how modelling software supply networks might allow managers to identify new business opportunities, visualize liability and responsibilities in a supply network, and how it can be used as a planning tool for product software distribution.* [1]

## 9.1 Software Businesses are Blends

Individual businesses no longer compete as single entities but as supply chains [72]. This holds for the software industry as well, where software products and services are no longer monolithical systems developed in-house, but consist of complex hardware and software system federations [48] produced and sold by different organizations. This development has led organizations to combine their business and components into complex Software Supply Network (SSN), from which they supply end-users with integrated products. As these SSNs grow more complex, it becomes harder for the participants of SSNs to make informed decisions on development strategy, responsibility, liability, and market placement [24, 49]. It also becomes harder to manage the risk associated with these decisions [152].

A SSN is a series of linked software, hardware, and service organizations cooperating to satisfy market demands. SSN management is different from physical

[1] This work was originally published in the proceedings of the 8th IFIP Working Conference on Virtual Enterprises, entitled "Providing Transparency in the Business of Software: A Modelling Technique for Software Supply Networks" in 2007 [151]. The work is co-authored with Anthony Finkelstein and Sjaak Brinkkemper.

goods supply chain management in two ways. First, software is malleable after release and delivery, giving rise to the need for extensive maintenance. Secondly, products with much lower quality levels are tolerated in SSNs compared to other types of products [8].

There is a clear difference between physical products and software products in that software products can be reproduced at relatively low cost. A result of the difference between conventional supply networks and SSNs is that literature on collaboration in supply networks [97] does not discuss maintenance and how it requires information about the supply chain. The same holds for other work on supply chain management, such as [74], which groups horizontal ties between firms (such as manufacturers and suppliers), but fails to recognize the importance of leveraging feedback in such networks, or Lambert and Cooper [72], who provide a conceptual framework for supply chain management without maintenance.

This chapter explores the new field of SSN research by presenting a method for modelling the complex relationships between participants in the supply networks of composite products and services. By conducting a case study of an organization that leverages the SSN we demonstrate that SSN models enable participants in SSNs to reason about business identification, product architecture design, risk identification, product placement planning, and business process redesign. Furthermore we demonstrate that modelling relations in SSNs is the first step in explicitly managing relationships with other participants in the SSN.

Value chains differ from SSNs in that value chains describe one product only, whereas SSNs specifically address networks of software systems that interact to provide software services. Attempts have already been made to model value chains surrounding large ERP configurations, by Messerschmitt and Szyperski [79]. Their model cannot represent relationships between, for instance, a Components off the Shelf (COTS) vendor and the application developer, making their models insufficient to describe a complete SSN.

In the following section SSN models and their accompanying software product context model are provided, by presenting the participants, the meta-model, a creation method, and a large example. In section 9.3 the case study Tribeka is presented. In sections 9.4 applications of SSN models are presented and discussed. In section 9.5 a description is provided on how organizations best ready themselves to participate in a SSN.

## 9.2 Software Supply Network Models

Models for SSNs consists of two parts, a software product context model and a supply network model. A software **product context model** describes the context in which a software service operates, and the software products, hardware products, and software services that are required to provide the software service. A **SSN model** displays all participants in a SSN, the connections between these participants, and the flows describing the type of product that flows down these connections. **SSN Participants** are any party that provides or requires flows from another participant in the network. The two models are related in that the product context model shows all products that

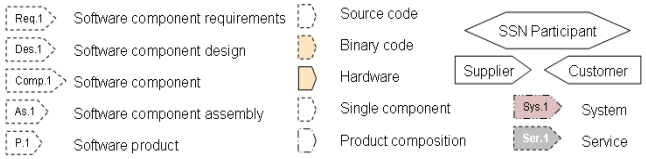



Figure 9.1: Software supply network model legend

are traded in their different forms in the SSN model.

### 9.2.1 Product Context Model

A **software service** is the provision of one or more functions by a system of interest to an end-user or to another software service. A **software system** is a combination of independent but interrelated software services, software components, and hardware components that provides one or more services. There are three types of entities in the software product context model, being (1) white-box services and their systems, (2) black box services, and (3) software and hardware components making up systems.

The main concept for product context diagrams is that of a software system, that consists of hardware and software components. When combined and activated these components provide a service. A software system requires at least one hardware component, at least one software component, and any number of services. A hardware component can support any amount of services and software components.

When a software component requires other software components (such as libraries and COTS) these are drawn under the software component. When a software component requires software services (such as databases and webservers), these are drawn under the software component as a service. The hardware on which the software components are running are drawn on the left left of systems. It is assumed that hardware components are complete, and thus do not have dependencies. Systems that provide services are drawn as containers containing the software components and services on which they depend. Please see the product context model in figure 9.3 for an example.

### 9.2.2 Supply Network Model

The supply network model connects participants in the SSN. Furthermore, these connections are annotated with flows, such as product requirements, product designs, software components, component assemblies, assembled products, assembled deployed systems (hardware and software), and services (provided by these systems). These artifacts all come from decoupling points for software products (see figure 9.2), which is the point at which demand and supply meet in SSNs.

When looking at the product software production pipeline seven decoupling points can be identified. First, a development organization can outsource the requirements engineering process and/or design process (*a,b*). Also, the developer can choose to

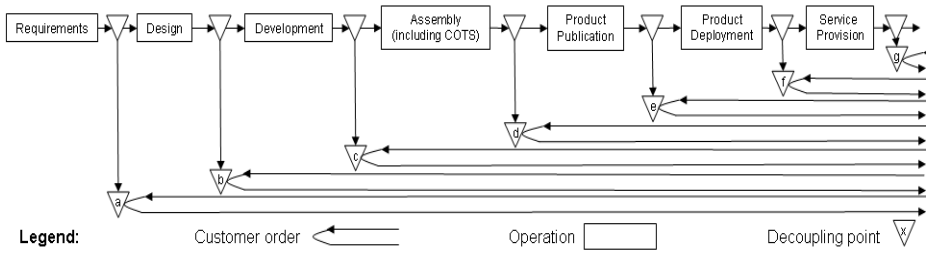



Figure 9.2: Software production decoupling points

release their source code, binaries, or assemblies of components (*c,d,e*) to another developing organization who uses these artifacts as a component to their product, or to a publisher who releases the product (common for games, where the vendor is rarely the developer). A software vendor can also choose to release the product itself, either as a package, or as a deployed system (*f*). Finally, a vendor can decide to offer their product to its customer in an application service provider model, where the vendor sells usage of its product instead of the product itself (*g*).

Flows are modelled as labels on the arcs between the participants and are distinguished by different colors and codings. The color indicates whether the artifacts are source artifact collections, compiled binary artifact collections, or complete systems and services. The codings are (in order of creation to usage) **Req** (requirements), **Des** (design), **Comp** (software component), **As** (software component assembly), **P** (software product), **Sys** (system, including hardware), **HW** (hardware), and finally **Ser** (services). It is not uncommon for software products going through iterations of the decoupling points before the product is delivered to a customer. It can be well imagined that a system designer creates a design, sells the design, and the software developer starts at the requirements phase again to see what can be added to the design. To indicate this, numbers are used in the codings after the abbreviation, such as **Des2.1**, which means that this is the second design for product 1. In the supply networks we only make this distinction when two generations of artifacts are produced by *different* participants.

### 9.2.3 SSN Model Creation Method

To help define the scope of a SSN model, the software product context model is created. The software product context model, which describes the systems that supply software services, must display all products and services that are specifically required for the service(s) of interest. Secondly, the participants must be determined. These participants are all parties that are involved with the products in the product context. The products in the product context will be presented as flows later. Finally, all relationships must be established between the participants. Whenever a product or service flows from one participant to another, they must be connected by an arc. Once

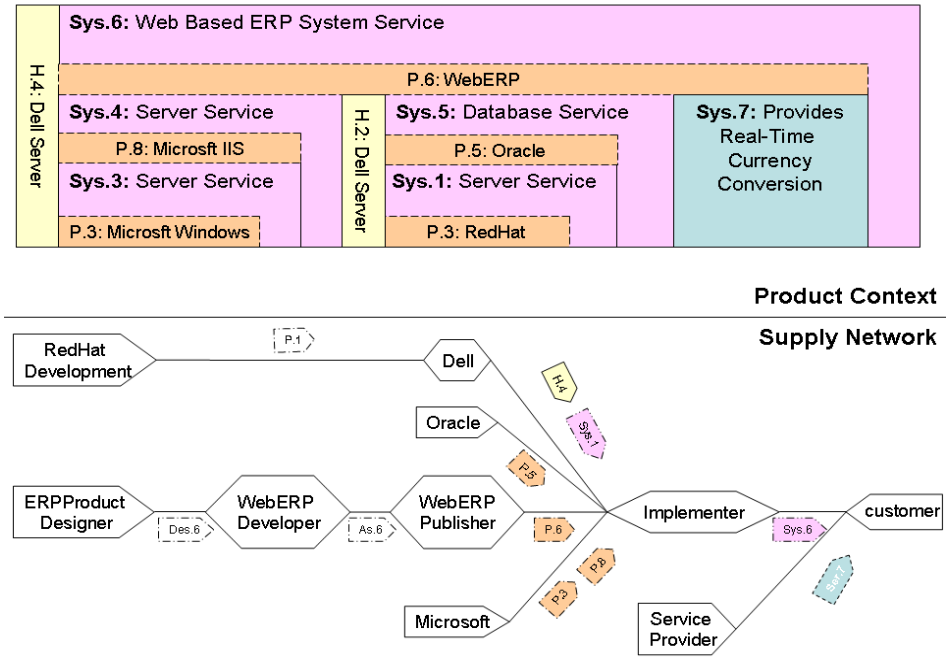



Figure 9.3: WebERP example models

the arc is drawn, it can be annotated with the type of commercial products and services that flow down this arc.

### 9.2.4 An Example: WebERP

In figure 9.3 the example models are presented for a customer requiring a Web Enterprise Resource Planning (ERP) service. To supply this service internally, the customer has decided to go with its personal service organization who implements a product WebERP on a newly purchased local database server and a local web server. The software product context model displays that to supply **Ser.6**, **P.6** is required. To run **P.6** a server is required that supplies WebERP through a web server application, in this case Microsoft IIS. A database server (**Sys.5**) is required as well that manages all the data for WebERP. Both servers, supplied by Dell, run a different operating system. Furthermore, to provide the WebERP service, a currency conversion web service is required. As products transition from source code to product to system, they generally retain the same number, such as for WebERP; **Des.6**, **As.6**, **P.6**, and **Sys.6** are all instances of the same (software) artifacts sold at their different decoupling points.

With all services and products laid out, the SSN model is created. The customer shall be supplying the service itself, so it only requires **Sys.6** from the implementer. The

implementer purchases two servers from Dell, one with RedHat installed, the Oracle database management system, WebERP, and Microsoft Windows and IIS for the web server. The implementer combines all components into a new system **Sys.6** that is delivered to the customer.

WebERP, the application, is designed by an external ERP product designer. Their design is the blueprint for WebERP, which is developed by WebERP developer and sent as open source to the WebERP publisher. The WebERP publisher compiles the components and turns the WebERP components into products, explaining the color change of the flows and the transition from component assembly to product. Finally, the service provider provides **Ser.7**, which is black box because the system that provides this system is (currently) not relevant.

## 9.3 A Case Study: Tribeka

We use a case study to demonstrate the SSN modelling technique. The company under study is Tribeka, an organization that attempts to break through the traditional product software retail supply chain, by delivering assemblies of components to retail outlets that can be burnt, packaged and turned into a finalized product *on-site*. Tribeka, founded in 1996, currently employs twenty-five people and has deployed its systems at large retail chains in the United Kingdom, such as WH Smith and HMV. Recently Tribeka has opened four high street outlets where it solely sells software created with Tribeka's SoftWide system.

The Tribeka SoftWide system consists of a server with a large storage memory, an internet connection, a number of CD and DVD burners, and a high quality cover printing facility for boxes, CDs, and DVDs. It is capable of creating between 50 and 100 different shrink-wrapped products per hour. The SoftWide system is not solely a hardware solution, since it is able to deliver the most up-to-date software onto the High Street. On a daily basis, software updates are sent to the SoftWide systems deployed in retail stores, including price information. The SoftWide system also stores the component assemblies in a coded manner, such that products are only produced when requested and authorized, using a proprietary auditable licensing system.

### 9.3.1 Tribeka Models

Two SSN models are presented in figure 9.4. At the top level of the figure the software product context model displays two systems, that provide the "computer use" and "entertainment" services. The entertainment system requires the software service "computer use" and the game product **P.3**. The system **Sys.2** requires a laptop and Microsoft Windows, before it can actually provide the "entertainment" software service.

Traditional software supply is depicted in the "before Tribeka" section of the figure. Here Microsoft is modelled as a software developer, who delivers its product to Dell. Dell, the hardware manufacturer, deploys the product **P.1** onto the laptop system and delivers **Sys.2** to its retailer, PC Store. PC Store sells the system to the customer, who also purchases a game **P.3** with it. **P.3** is designed by Game Designer and the

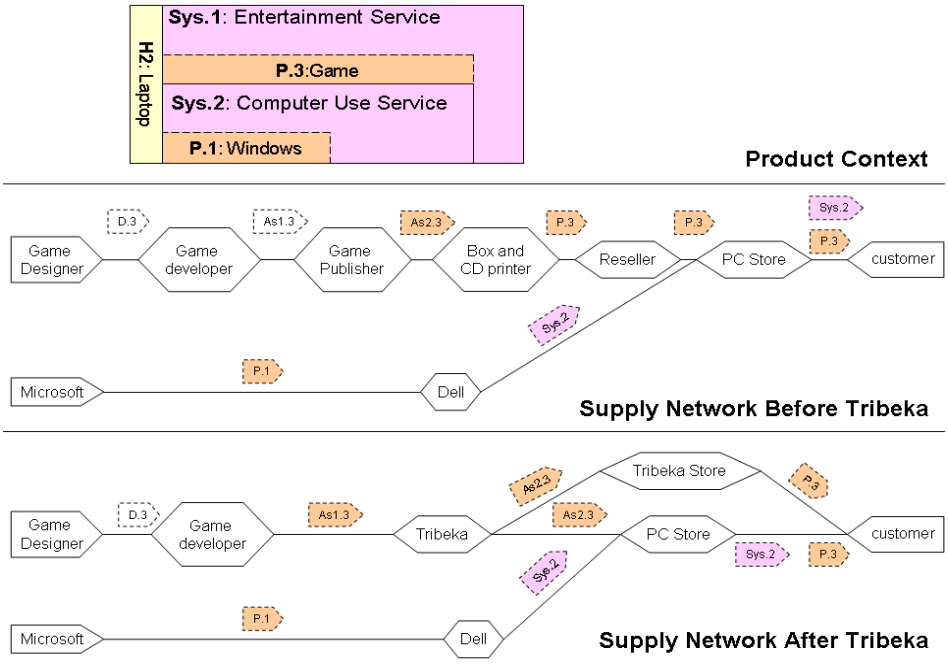



Figure 9.4: The Tribeka "before and after" supply network model

design is sold to the Game Developer, who actually implements the game. Once game development is finished a collection of source components (**As1.3**) is sent to the game publisher. These source components are then compiled, causing the **As2.3** to be shaded, and sent off to a printing facility. Finally, the game publisher sells the finished products **P.3** to a reseller. The reseller then sells the game to PC Store.

Tribeka takes over from the Game Publisher, the Reseller, and the printing facility, by directly publishing any product from a software developer to the High Street stores. The Game Developer now passes a compiled set of components directly to Tribeka. Tribeka sends the component assembly to PC Store directly, instead of to a printing facility and then reseller. The component assembly is then assembled into a product at the retail store, enabling the latest possible binding for physically sold software products.

### 9.3.2 Tribeka Relationships

The SSN model in figure 9.4 shows that Tribeka maintains intensive relationships with the game developer and with retailers. The presented Tribeka SSN model is slightly simplified because in many cases there will be a publisher in between the developer and Tribeka. As such, Tribeka has three types of participants in the SSN it deals with: retailers, game developers, and game publishers. Tribeka uses its SoftWide system to maintain relationships and transport data between these participants. Publishers and developers send their component assemblies to Tribeka, which are then uploaded into the SoftWide system, including price information, license codes, software artifacts, digital manuals, and images for box covers. These publishers are now able to see the status of their products, such as how many sales have been made and what types of licenses have been distributed. On the other side the retailers access the SoftWide system through their points of sale, which are used to sell and create software products from the component assemblies supplied by Tribeka. From Tribeka two lessons can be learnt about SSNs. To be successful in a SSN an organization must explicitly manage relations with the other participants. The second lesson is that an organization must observe opportunities to take part in different parts of the supply network, such as Tribeka's opening of retail stores that only use the SoftWide system.

## 9.4 SSN Model Applications and Usage

SSN models have five applications being business identification, product architecture design, risk identification, product placement planning, and business process redesign. The aim of SSN models is to clarify the blend that is software business. SSN models are thus used by policy makers, software architects, and entrepreneurs. Depending on the application, they must make the SSN model on a regular basis and observe changes, risks, and opportunities.

**Business Identification -** SSN models show the flows for each participant in the network. These flows can be used to determine the business type for that participant. When, for instance, a participant receives hardware and software components and has one system as output this is an *integrator* (Implementer in the WebERP example).

A participant that receives component assemblies and then publishes products is a *publisher* (WebERP publisher in the WebERP example). Another common example is a supplier that has no input but produces a software product (*software product developer*, RedHat in the example). We see that Tribeka (figure 9.4) functions as a *packager* and interestingly enough turns the PC Store into a *software product publisher*. A participant that has the same input as output is a *reseller*. Finally, according to these definitions and due to the absence of a hardware component input Dell is a *hardware producing integrator*.

**Product architecture design -** In deciding the type of software architecture a software developer must use, the SSN plays an important part. The software architecture decides how a product will depend on other products and services, and this will have far stretching consequences on the future of a software product. SSN models can thus assist in making architectural design decisions.

**Risk identification -** The SSN model uncovers, for instance, that a product cannot be used without the availability of some component or service. These dependencies on other organizations, though logical, can be disastrous for participants lower down the SSN. Such a dependency influences the total cost of ownership of the product, the possibility to internationalize, and even the future when such a dependency can no longer be fulfilled. This calls for diversification and architecting for product dependency variability [66]. The SSN model helps uncover such relations and dependencies. SSN models can be used by customer organizations to uncover whether they are in possession of certain products that are unsupported, or whether they are making use of a service that could easily be terminated. A common vulnerability, for instance, is a custom link between two products, built by a software implementation service provider, which stops working after an update for one of the products has been deployed. The SSN models can assist in finding and eliminating such weaknesses for all participants in the SSN.

**Product placement planning -** A vendor can use the SSN model to determine how to market its product, how to inform customers of product news and releases, and how customers will contact the vendor. The latter is especially important when looking at pay-per-usage feedback and error feedback as is shown in chapter 2. For example, when a bug report is sent from a customer to a participant in the SSN the participant must decide whether to solve this issue or to forward it to another party in the SSN. Software vendors can choose to sell their products as add-ons to other products, in combination with hardware (i.e., navigation systems), and as a service (on-line bookkeeping). Furthermore, software vendors can decide to sell the product through channels they own (their own site), through resellers, through service providers, etc.

**Business Process Redesign -** Participants of the supply chain must identify their business partners and establish different types of relations. SSN models can thus be used to design business information systems that take into account the participants of the SSN with which the business will have regular and even ad-hoc relations. Tribeka for example, manages explicitly its relationships with software developers, publishers, and retail outlets and has created different information systems and portals for them.

The SSN model reveals business opportunities and risks by making two types of changes. The first change is to alter the binding times and decoupling points for

products and services. Tribeka is a clear example of this, where it takes the role of the traditional reseller, but simply assembles the product at a later stage. This optimization allows resellers to replenish their stock dynamically, saving cost in the area of stock management, delivery, and deployment. The second type of change is seen when a change is made to the participants in the SSN, in the form of mergers, split-ups, developers buying new COTS, or customer organizations that become vendors of products or services themselves. An example of this is when Tribeka opened their own SoftWide stores.

### 9.4.1 Tribeka SSN Insights

When looking at **risk determination** the SSN model clarifies risk for the customer, Tribeka, and the PC Store. The customer provides for the deployment and integration of the hardware and software components itself, thus introducing a risk that deployment and service provision go awry. For Tribeka there is a risk that software stores start publishing and selling through on-line retail channels, which is why Tribeka has started their own on-line software store [116]. The SSN models the PC Store as the first contact point for customers, which implies that it must reduce risk by reselling products of a high quality. After all, the quality of the PC Store's product portfolio determines the amount of overhead from customer problems.

The game publisher uses the SSN for **product placement planning**. Tribeka provides another channel to reach customers at lower risk than conventional channels. After all, no mass distributing of products is required and the only cost that is necessary is the cost of packaging a product in the Tribeka format. The game developer, on the other hand, reduces its risk by placing its liability with its publisher.

Regarding **business process redesign** the SSN model shows that customers will want to buy the game with the laptop, indicating that it might be an idea to sell the complete system (**Sys.1**), instead of the laptop and game separately. In the model a similar situation can be found when Dell installs Microsoft's Windows onto the laptop and sells the complete system to the PC Store.

## 9.5 Ad-Hoc Software Supply Networks

The tendency to integrate components from different developers and manufacturers into new products and components by both customers and integrators has led to a phenomenon of quickly forming and dissolving of ad-hoc SSNs. Many organizations, however, are not specifically adjusted to manage relations within such ad-hoc networks. Simultaneously, software vendors and manufacturers are constantly approached by (new) business partners, such as manufacturers, resellers, and service providers, with bugs, feedback, requests for changes, and other communication about their software products.

An interesting phenomenon is encountered in open source software where endusers often solve problems themselves, by changing a product's source code or developing a workaround, before approaching the software vendors. This results in different

branches and fixes floating around not only in the original repository, but also on the web in several development forums and user groups.

A coalition between participants in a SSN is where participants rely on each other, yet do not have any of the skills required for collaborative unity [134], such as organizational measures, structured communication, and planned durability. Software organizations can profit from the many opportunities in these ad-hoc SSNs when properly prepared to engage (in order of intensity) in conversations, relations, partnerships, and even alliances with other participants. After all, these other vendors are willing to create a user and developer community around a (configuration of) software product(s), which will encourage use of the products and create new solutions and opportunities surrounding the product. The SSN modelling technique presented in this chapter assists a participant in understanding how these coalitions are formed. Secondly, a participant must build a community surrounding its product that unifies external and internal users, developers, implementers, and integrators of the product. Such a community can be built using ontologies, portals, customer meetings, and partner meetings. Especially portals, which can be used for the distribution and sharing of knowledge, development and bug finding tools, are an important factor to create a close network of participants willing to add value to the community, and thus increase the value of the SSN.

To transform a coalition to collaboration relations must be formalized by the facilitating organization. A clear distinction needs to be made between intensively and loosely coupled alliance partners. By classifying partners in such a way, participants can create different circles of trust with partners and users, which will clarify the different relationships within a SSN considerably for all participants in a network. A participant in the SSN must at all times be aware of opportunities to form coalitions, since each customer question could set in motion the cooperation between multiple participants. The ability to form strategic coalitions and collaborations within a SSN is further strengthened by Duyster et al. [39] who claim that to craft successful strategic technology partnerships, steps need to be undertaken to strategically position participants, to prepare alliance skills, to build a business community, and to do smart partner selection.

## 9.6 Conclusions and Future Work

SSN models provide a novel manner to perform strategic and risk evaluation in the business of software. The insights provided in section 9.4.1 have all been obtained using the SSN models for Tribeka. These insights can be realized through experience, but the ability to assess risks *avant la lettre* is a valuable contribution. This chapter presents a modelling method for SSNs to provide insight into SSNs, enabling participants to do risk assessment, strategic decision making, product placement planning, and liability determination. The method of a case study is used for this chapter because it is a proven method to introduce a novel research area, such as SSNs and product contexts.

The decoupling points are a concept taken from conventional product development and marketing planning. The combination of SSNs with these decoupling points

creates a powerful modelling tool that is generalizable to non-software products as well. To be able to do so one must define the decoupling points, the possible range of product decompositions, before creating the supply networks.

SSNs do seem to provide an environment for multiparty contracts [131], to simplify liability and problem resolution challenges. These multiparty contracts can assist customer organizations in protecting their service levels and also groups the participants in the SSN, stimulating collective problem resolution.

Currently we possess a collection of 30+ SSN models from start-ups and medium to large software enterprises. With these models we are hoping to further classify different business models for product software. Furthermore we are experimenting with different flows, such as content, money, and licenses.

# Part V

# Conclusion

# CHAPTER 10

# Conclusion

## 10.1 Conclusion

This research has led to two main conclusions. The first conclusion from this research is that there are now process descriptions for the Customer Configuration Updating (CCU) processes, whereas before these were lacking. The second conclusion is that explicit product software knowledge management improves product software development. The main contributions of this work are the CCU model and two CCU support tools. The CCU model enables evaluation of CCU processes and support tools. The two CCU support tools enable product software vendors to reduce time to market and increase the number of different configurations a product software vendor can provide to customers without a large increase in overhead.

## 10.2 Research Questions

**RQ1: What are the concepts and is the state of affairs of customer configuration updating for product software?**

The answer to this question is the CCU model that describes all four CCU processes in detail. Furthermore, capabilities have been attached to each process, to enable more detailed evaluations of product software update tools, product software vendors, and CCU support tools. The model is based on models from literature and nine case studies of product software vendors. The model has been used to create a survey for the product software industry, an evaluation method for product software vendors, and an evaluation method for product update and CCU support tools. The model can be used by practitioners to evaluate a product software vendor's processes, tools, and practices.

*SQ1.1: What is the state-of-the-art of customer configuration updating and who are the stakeholders?* This thesis provides evaluations of state-of-the-art techniques for CCU management and improvement. Furthermore, it notes and criticizes the lack of such techniques and process descriptions in (academic) literature. CCU definitions, if any, are immature and unclear. There are a number of reasons for this. A software vendor is always trying to add value to its product. CCU processes are often seen as

non-value add processes. Furthermore software engineering literature focuses more on the building of software than on the correct release, delivery, deployment, and usage and activation of a software product. To answer this question the boundaries of CCU research, the types of knowledge that can potentially be shared between a vendor and a customer, and the stakeholders of these processes needed to be established. Extensive literature study led to the creation of new product update process models, which evolved into the CCU model (see chapter 2). The CCU model and product updater evaluation model were used to establish weaknesses of CCU processes and support tools. This question was answered by conducting tool evaluations and by evolving process models from literature into the CCU model. The CCU model has been fundamental to this research, because it is used to evaluate both tools and practices and processes of product software vendors in case studies. The answer to this research question amplifies the need for more and better CCU process descriptions. Furthermore, practitioners can use this thesis to establish what the state-of-the-art of CCU currently is.

*SQ1.2: What is the state-of-the-practice of customer configuration updating for product software vendors?* CCU is an underdeveloped process that deserves more attention from product software vendors. The fact that CCU is underdeveloped is surprising considering the high rate of failures of deployments, the large amount of effort put into CCU, and the potential gains for product quality. When looking closer, the causes of this underdevelopment are clear. First and foremost, product software vendors do not have the process descriptions available to evaluate their CCU processes. Furthermore, there are no tools that support these processes in full. This has led to a diverse range of different tools being built by product software vendors, with varying success. The underdevelopment of CCU processes of product software vendors has many effects: vendors cannot serve the amount of customers they want, lose large amounts of effort towards migration and update projects, and cannot provide their products in wide ranges of configurations. The scientific answer has been, until now, unsatisfactory.

The answer to this question is provided in the form of case study and statistical survey evidence about the state-of-the-practice of product software companies. Nine product software vendors have been evaluated using the CCU model in extensive case studies. Furthermore, a survey has been conducted amongst Dutch product software vendors into their CCU processes. The answer to this question contributes to the research described in this thesis because it shows that many product software vendors lack sufficient process descriptions and tool support to optimize their CCU processes. Furthermore, the research to answer this question resulted in best practices which could later be reused for the CCU evaluation model.

*SQ1.3: What parts of the CCU processes are currently supported by product update tools?* Currently, there are different CCU support tools available, with varying degrees of support for the CCU processes (for a full description we direct the reader to chapters 5 and 7). One conclusion is that no one tool supports all parts of the CCU processes. The answer to this question is provided by establishing characteristics of different product update tools in case studies. The answer is fundamental to this

research because it proves once again that there is a need for tools that support more aspects of the CCU processes. The answer enables product software vendors to evaluate their own tools, and to develop new tools that support CCU processes.

**RQ2: Can customer configuration updating be improved by explicitly managing and sharing knowledge about software products?**

Yes. By sharing available product knowledge with customers, the processes of release, delivery, deployment, and usage and activation are improved. Furthermore, sharing of knowledge between customers and vendors increases the speed at which feedback, such as error reports and feature requests, arrives at the vendor, enabling a vendor to process feedback quicker, which improves software product development. The question has been answered by doing nine case studies of product software vendors and by building two CCU support tools that explicitly manage and share software knowledge. The answer to this question provides product software vendors with evidence that it is useful to invest in product software knowledge management.

*SQ2.1: What aspects of customer configuration updating can be improved by explicitly managing and sharing customer configuration updating knowledge and can these improvements be measured?* Improving the CCU processes is beneficial for the time to market of a product, success of a product, number of customers a product software vendor can handle, deployment success rate, and release frequency of a product. All except product success can be quantitatively measured. This question has been answered by conducting case studies, the survey, and tool implementations and evaluations. The answer to this question is fundamental to the research question whether product software development can be improved by explicitly managing and sharing knowledge about the software product. Product software vendors can use the answer to this question to establish which technique or technology must be implemented to improve a specific aspect of product software development.

*SQ2.2: Are product software vendors who explicitly manage customer configuration updating knowledge more successful?* Yes. When product software vendors have implemented larger numbers of CCU capabilities they are more successful. CCU enables them to publish releases more often, reduce bug discovery and fix times, handle larger amounts of customers, and provide them with a larger range of features and deployment platforms. To answer this question a modelling technique was developed to establish and calculate Time to Market of a software product. The modelling technique was then applied to example scenarios where new CCU prototype support tools were applied. This led to the conclusion that under certain conditions CCU improvements can shorten release cycles and thus reduce Time to Market. Furthermore, a survey was held amongst product software vendors in the Netherlands to see whether recent improvements to their CCU processes led to an increase in success for a software product. The answer to this question is beneficial to this research because it means that CCU research improves success of the product software industry. This conclusion also provides software developers with evidence

that CCU improvements are vital to a product software vendor.

*SQ2.3: What functionality is required from tools that manage and reuse customer configuration updating knowledge to support product software development and the customer configuration updating processes?* For this research two tools were built (see chapters 5 and 6), evaluated, and proven beneficial for product software vendors. These tools were created after finding that many CCU capabilities were not yet implemented in CCU support tools. These missing features were, after a potential gain analysis, implemented into prototype tools. These tools were evaluated by applying them in examples and real-life cases. The answer to this question is fundamental for the conclusion that product software knowledge sharing is a success factor for product software vendors.

### 10.2.1 Research Methods

Over the last four years we have conducted **mixed-method research** to evaluate the CCU model. The methods that have been applied are case studies, tool evaluations, prototype building, design research and two surveys. I recommend using the combination of many different research methods to improve product software management processes. The case studies were fit to establish new problem areas and map the CCU processes. The survey enabled further generalization of our conclusions about product software vendors and enabled a combined quantitative and qualitative analysis of CCU. The tool evaluations enabled a further understanding of CCU tool support and showed feature gaps that required new tools and solutions. Finally, the building of prototype tools and evaluation of these tools enabled a further demonstration of the contribution of these tools.

## 10.3 Chapter Conclusions

In chapter 2 CCU is defined in detail and nine product software vendors are evaluated using the presented CCU model. The main conclusions from this work are that product software vendors do not focus enough on CCU. They neglect to apply knowledge management techniques to improve CCU processes. Furthermore, product software vendors rarely use automatically generated customer feedback and automatic license generation. This is surprising when taking into account that 50-70% of revenue is coming from existing customers, who profit most from CCU improvements. The main contribution of the chapter is the CCU model that is used to evaluate the capabilities of product software vendors.

In chapter 3 one of the cases is described in more detail. The subject of study, Exact Software, provides an interesting example of how knowledge management is used to integrate customer relationship management, product data management, and software configuration management. The integration of these practices enables the software vendor to maintain a customer base of over 160.000 customers. The main contribution of the work is two-folded. The first contribution is that showing that integration of customer relationship management, product data management,

and software configuration management encourages knowledge sharing between customers and vendors, strengthening their relationship. The second contribution is the integration method and the demonstration of the use of a product data management system for a software product.

Chapter 4 describes the result from a survey amongst 74 Dutch product software vendors. The survey is based on the CCU model from chapter 2 and establishes, amongst other things, a relationship between the success of a product software vendor and any recent improvements that have been made to CCU. Furthermore, CCU practices are described in more detail. The survey results confirm conclusions drawn earlier in the case studies, such as the fact that product software vendors do not implement CCU sufficiently and that this is caused by lack of process descriptions, tools, and technologies.

Chapter 5 describes an evaluation model for CCU support tools. The presented model is used to evaluate fourteen CCU support tools and assesses their capabilities for each of the four CCU processes. A typology for product update tools is also given. The main conclusion of this work is that there are no CCU support tools that provide all capabilities required by product software vendors, even though many of these tools are specifically designed for that group. Furthermore, the work contributes in that it uncovers product update capabilities that are required by product software vendors, which are not implemented in current update tools.

Chapter 6 demonstrates a prototype tool that is used to ask a software knowledge base "what-if" questions. The tool proposes evolutionary steps for a customer configuration to get to a target configuration. The prototype tool makes it possible for product vendors, with the right knowledge available, to see whether deployment of a new variant, version, or feature of a component will lead to problems. The tool can calculate evolution steps in an acceptable amount of time. The main contribution of the chapter is showing that evolving a customer configuration is a knowledge management problem and not a complexity problem.

Chapter 7 concludes that the capabilities missing from the product updaters in chapter 5, can be built into product update tools such as Pheme and Sisyphus. These prototype tools are implemented in two example case study scenarios and further evaluated using the Product Development Cycle Time model. This model establishes that time to market can be reduced by spending relatively little effort on implementing Pheme and Sisyphus into a product software vendor's organization. The main contribution of this work is the presentation of the Pheme Knowledge Distribution Infrastructure for product software.

Chapter 8 claims that software vendors can plan their release packages more cost-efficiently by using Cost/Value functions. These functions provide criteria for when the value of a release package exceeds the cost of releasing one. Furthermore, CCU improvements are proposed to bring these costs down even further, as these changes will improve release frequency, product feedback, and finally product quality. The main contribution is a set of release criteria that assists product software vendors in their release package planning process.

Chapter 9 proposes a modelling method for software supply networks, in which different suppliers of software artefacts cooperate to deliver a product to an end-user. The presented software supply network models enable the user to analyze the business

of product software and the products that are built by these networks. Furthermore, the networks make a strong case for explicit software knowledge management, since the many different suppliers require more knowledge than just one end-user.

## 10.4 Future Work

With this thesis CCU research does not end. There are still five CCU-related areas and one general research area in which there are still open questions. To begin with, the plan is to build Pheme into a software product that is sold to product software vendors. The capabilities provided by Pheme, such as knowledge delivery across a software supply network make it a fertile ground for further experimentation in real life settings. One of the first steps in doing so is formalizing and testing the domain specific Pheme communication language. Furthermore, Pheme is an essential part of the Product Software Management Workbench [88].

Chapter 9 has only opened one of many doors. I expect software supply networks to be a novel research area with more product software vendors joining these complex business networks daily. Software supply networks seem to provide fertile ground, especially when looking at their relationships to a software product's architecture. This research enables further exploration of risk discovery methods and architectural decisions that might in the future prohibit a product software vendor's growth.

The exploration of release speed in Chapters 7 and 8 and its effect on product quality has led to many new research questions, such as "does an increase in release speed relate to product quality and product development speed"? Such questions touch different areas of interest, such as development methods, software quality, and product quality and must be adressed in the future to provide software producing organizations with more insight into their processes.

When looking at product development methods I would like to establish whether tools that can be used to protect the correctness of development and CCU, such as xLinkIt [83], actually deliver when applied in a practical setting. The use of such a tool for the assurance of software knowledge quality potentially provides a huge improvement in any product development process.

Finally, with respect to CCU many technologies provide their own standards, architectures, and technology to enable run-time deployment and updating. The work of Hicks [55] but also that of Ajmani [2] provide comprehensive models for runtime updating. The architectural constraints of such updates must further be explored, for instance to propose a set of basic architectural requirements for such tools.

The conducted research combined different research methods such as case studies and surveys to validate hypotheses about CCU. This multiple method multi-theory approach proved useful for a research area that is so diverse.

# Bibliography


[1] A. Abran, J.W. Moore, Pierre Bourque, R. Dupuis, and Leonard Tripp. *SWEBOK: The Software Engineering Body of Knowledge, last visited on 20-05-2007*. IEEE, 2004.

[2] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. Ph.D., MIT, September 2004. Also as Technical Report MIT-LCS-TR-1012.

[3] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 452–476, July 2006.

[4] the HASP tool site Aladdin.com. http://www.aladdin.com/. Last visited: 29-05-2007.

[5] Alain April, Jane Huffman Hayes, Alain Abran, and Reiner R. Dumke. Software maintenance maturity model (smmm): the software maintenance process model. In *Journal of Software Maintenance*, volume 17, pages 197–223, 2005.

[6] 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.

[7] Gerco Ballintijn. A case study of the release management of a health-care information system. In *proceedings of the 21st IEEE International Conference on Software Maintenance, Industrial Applications track*, pages 34–43, 2005.

[8] Lynne Baxter and John Simmons. The software supply chain for manufactured products: reassessing partnership sourcing. In *International Conference on Management of Engineering and Technology*, pages 468–477, 2001.

[9] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: a methodology to develop software product lines. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 122–131, New York, NY, USA, 1999. ACM Press.

[10] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile manifesto. http://agilemanifesto.org, Last visited: 21-05-2007.

[11] Kent Beck and Martin Fowler. *Planning Extreme Programming*. 160 pages, Addison-Wesley, 2001.

[12] Robert Bialek and Eric Jul. A framework for evolutionary, dynamically updatable, component-based systems. In *The 24th IEEE International Conference on Distributed Computing Systems Workshops*, pages 326–331, Hachioji, Tokyo, Japan, March 23-24 2004.

[13] Barry Boehm. Get ready for agile methods, with care. In *IEEE Computer*, volume 35, pages 64–69, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[14] Barry W. Boehm. The future of software processes. In *International Software Process Workshop*, pages 10–24, 2005.

[15] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. Semantics of feature diagrams. In Tomi Männistö and Jan Bosch, editors, *Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support)*, pages 139–148, Boston, 2004.

[16] Jan Bosch. Software product lines: organizational alternatives. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 91–100. IEEE, 2001.

[17] Jan Bosch and Mattias Högström. Product instantiation in software product lines: A case study. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering*, volume 2177, page 147. LNCS, 2001.

[18] Harry M. Brelsford. Connecting to customers. 240 pages, Microsoft Press, 2002.

[19] Sjaak Brinkkemper and Paul Klint. Intelligent software knowledge management and delivery. In *downloaded from http://www.cwi.nl/htbin/sen1/twiki/bin/view/Deliver/Publications on 27-4-2007*, 2003.

[20] Par Carlshamre. Release planning in market-driven software product development: Provoking an understanding. pages 961–965. Springer-Verlag, 2002.

[21] Antonio Carzaniga, Alfonso Fuggetta, Richard Hall, Andr van der Hoek, Dennis Heimbigner, and Alexander Wolf. A characterization framework for software deployment technologies. In *Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado*, 1998.

[22] Central Computer and Telecommunications Agency. ITIL Service Support. 200 pages, Stationery Office Books, 2003.

[23] Sam Clegg. Evolution in extensible component-based systems. In *Master's thesis, 36 pages, Imperial College London*, 2003.

[24] Ronni Colville and Patricia Adams. It service dependency mapping tools provide configuration view. In *Gartner Research News Analysis*. Gartner, 2005.

[25] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. In *ACM Comput. Surv.*, volume 30, pages 232–282. ACM Press, 1998.

[26] Exact Software corporate website. http://www.Exact.com. Last visited: 21-05-2007.

[27] Zylom Games corporate website. http://www.Zylom.com. Last visited: 21-05-2007.

[28] Michael A. Cusumano and Richard W. Selby. Microsoft secrets. 309 pages, Free Press, 1995.

[29] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming (invited paper). In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 2–19, London, UK, 1999. Springer-Verlag.

[30] Andrew Cotton Darren Saywell. Spreading the word: Practical guidelines for research dissemination strategies. 64 pages, WEDC, Loughborough University, UK, 1999.

[31] Thomas H. Davenport, Gilbert J. B. Probst, and Heinrich von Pierer. Best practices in siemens. In *Knowledge Management Case Book, Second Edition, 336 pages*. Hoboken NJ, USA, 2002.

[32] Merijn de Jonge. Build-level components. *IEEE Trans. Softw. Eng.*, 31(7):588–600, 2005.

[33] Eelco Dolstra. Integrating software construction and software deployment. In Bernhard Westfechtel and André van der Hoek, editors, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *LNCS*, pages 102–117, Portland, Oregon, USA, 2003. Springer-Verlag.

[34] Eelco Dolstra. Integrating software construction and software deployment. In Bernhard Westfechtel, editor, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, Oregon, USA, May 2003. Springer-Verlag.

[35] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In Lee Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.

[36] Eelco Dolstra, Gert Florijn, Merijn de Jonge, and Eelco Visser. Capturing timeline variability with transparent configuration environments. In Jan Bosch and Peter Knauber, editors, *Proceedings of the IEEE Workshop on Software Variability Management (SVM'03)*, pages 122–131, Portland, Oregon, 2003. IEEE.

[37] Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering*, pages 583–592. IEEE, 2004.

[38] John Dunagan, Roussi Roussev, Brad Daniels, Aaron Johnson, Chad Verbowski, and Yi-Min Wang. Towards a self-managing software patching process using black-box persistent-state manifests. In *Proceedings of the First International Conference on Autonomic Computing*, pages 106–113, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[39] Geert Duysters, Gerard Kok, and Maaike Vaandrager. Crafting successful strategic technology partnerships. pages 343–351, 1999.

[40] Christof Ebert, Reiner Dumke, Manfred Bundschuh, Andreas Schmietendorf, and Rainer Dumke. Best practices in software measurement. 300 pages, Springer, november 2004.

[41] Sebastian Elbaum and Madeline Diep. Profiling deployed software: Assessing strategies and testing opportunities. In *IEEE Trans. Softw. Eng.*, volume 31, pages 312–327, Piscataway, NJ, USA, 2005. IEEE.

[42] Barbara Farbey and Anthony Finkelstein. Software acquisition: A business strategy analysis. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 76–83, 2001.

[43] B. Flyvbjerg. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, 12(2):219, 2006.

[44] Organisation for Economic Co-operation and Development. 311 pages, information technology outlook 2002. 2002.

[45] Platform for Productsoftware website. http://www.productsoftware.nl. Last visited: 21-05-2007.

[46] David N. Ford and John. Sterman. Dynamic modeling of product development processes. In *System Dynamics Review*, pages 31–68. John Wiley and Sons, Ltd., 1998.

[47] LimeSurvey Website (formerly PHPSurveyor). http://www.limesurvey.org/. Last visited: 21-05-2007.

[48] Carlo Ghezzi and Gian Pietro Picco. An outlook on software engineering for modern distributed systems. In *Proceedings of the Monterey workshop on Radical Approaches to Software Engineering, Venice (Italy), October 8-12*, pages 10–17, 2002.

[49] Martin Grieger. Electronic marketplaces: A literature review and a call for supply chain management research. In *European Journal of Operational Research*, volume 144, nr.2, pages 280–294, 2003.

[50] Richard Hall, Dennis Heimbigner, and Alexander Wolf. Specifying the deployable software description format in xml. In *Technical Report CU-SERL-207-99, University of Colorado Software Engineering Research Laboratory, March*, 1999.

[51] Richard S. Hall, Dennis Heimbigner, and Alexander Wolf. Evaluating software deployment languages and schema. In *Proceedings of the 15th Conference on Software Maintenance*, pages 177–187, 1998.

[52] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proceedings of the International Conference on Software Engineering*, pages 174–183, 1999.

[53] Helen E. Harrison, Stephen P. Schaefer, and Terry S. Yoo. Rtools: Tools for software management in a distributed computing environment. In *Proceedings of the Summer 1988 USENIX Conference*, pages 85–93, 1988.

[54] Remco Willem Helms. Product data management as enabler for concurrent engineering, ph.d. dissertation. In *Eindhoven University of Technology press*, 2002.

[55] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[56] David M. Hilbert and David F. Redmiles. Extracting usability information from user interface events. In *ACM Comput. Surv.*, volume 32, pages 384–421, New York, NY, USA, 2000. ACM Press.

[57] Petr Hnetynka. Component model for unified deployment of distributed component-based software. In *Tech Report No 20044, Dep of SW Engineering, Charles University, Prague*, 2004.

[58] including FlexNet InstallShield tool website. http://www.installshield.com/. Last visited: 29-05-2007.

[59] Software Engineering Institute. CMMI SE/SW - Capability Maturity Model Integration. In *version 1.1 Pittsburgh*. Software Engineering Institute, Carnegie Mellon University, USA, 2002.

[60] Ulf Asklund Ivica Crnkovic and Annita Persson Dahlqvist. Implementing and integrating product data management and software configuration management. Artech House Publishers, 366 pages, 2003.

[61] Taqi Jaffri and Kuldeep Karnawat. Efficient delivery of software updates using advanced compression techniques. In *Proceedings of the 22nd International Conference on Software Maintenance*, pages 267–268, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

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

[63] Slinger 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-2006-012., 2006.

[64] Slinger Jansen, Gerco Ballintijn, and Sjaak Brinkkemper. Software release and deployment at exact: a case study report. In *technical report SEN-E0414*. CWI, 2004.

[65] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In *Proceedings of the Second International Conference on Software Product Lines*, pages 15–36, London, UK, 2002. Springer-Verlag.

[66] Michel Jaring and Jan Bosch. Architecting product diversification - formalizing variability dependencies in software product family engineering. In *Proceedings of the Fourth International Conference on Quality Software*, pages 154–161, Washington, DC, USA, 2004. IEEE Computer Society.

[67] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, Pittsburgh, PA, 1990.

[68] Dan J. Kim, Donald L Ferrin, and H Raghav Rao. A study of the effect of consumer trust on consumer expectations and satisfaction: the korean experience. In *Proceedings of the 5th international conference on Electronic commerce*, pages 310–315. ACM Press, 2003.

[69] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62, 1995.

[70] Paul Klint and Chris Verhoef. Enabling the creation of knowledge about software assets. In *Data Knowledge Engineering*, volume 41, pages 141–158. Elsevier Science Publishers B. V., 2002.

[71] Andrew Kusiak. *Engineering Design: Products, Processes, and Systems*. 427 pages, Academic Press, Inc., Orlando, FL, USA, 1999.

[72] Douglas M. Lambert and Martha C. Cooper. Issues in supply chain management. In *Journal of Industrial Martketing Management*, pages 65–83, 2002.

[73] M. Larsson and I. Crnkovic. Configuration management for component-based systems. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 159–172, 2001.

[74] S G Lazzarini, F R Chaddad, and M L Cook. Integrating supply chain and network analyses: the study of netchains. In *Journal on Chain and Network Science*, pages 7–22. Wageningen Academic, 2001.

[75] Software Asset Management and Compliance Tools. http://www.managesoft.com/. Last visited: 29-05-2007.

[76] Hong Mei, Lu Zhang, and Fuqing Yang. A software configuration management model for supporting component-based software development. In *SIGSOFT Softw. Eng. Notes*, volume 26, pages 53–58. ACM Press, 2001.

[77] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the 26th International Conference on Software Engineering*, pages 459–468. IEEE Computer Society, 2004.

[78] Vladimir Mencl, Zuzana Petrova, and Frantisek Plasil. Update description language. In *Canterbury University, Week of Doctoral Students WDS 99, Faculty of Mathematics and Physics, Jun*, 1999.

[79] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry (Chapter 6: Organization of the Software Value Chain)*. 424 pages, MIT Press, Cambridge, MA, USA, 2003.

[80] Bertrand Meyer. The software knowledge base. In *Proceedings of the 8th international conference on Software engineering*, pages 158–165. IEEE, 1985.

[81] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, 2000. ACM Press.

[82] Audris Mockus, Ping Zhang, and Paul Luo Li. Predictors of customer perceived software quality. In *Proceedings of the 27th international conference on Software engineering*, pages 225–233, New York, NY, USA, 2005. ACM Press.

[83] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: a consistency checking and smart link generation service. In *ACM Trans. Inter. Tech.*, volume 2, pages 151–185, New York, NY, USA, 2002. ACM Press.

[84] SDU State Publishers Netherlands. http://www.sdu.nl. Last visited: 21-05-2007.

[85] C S P Ng, G G Gable, and T Chan. An erp maintenance model. In *Proceedings of the 36th Hawaii International Conference on Systems Sciences*, pages 234–243, 2003.

[86] Frank Niessink, Viktor Clerc, Ton Tijdink, and Hans van Vliet. The it service capability maturity model. In *http://www.itservicecmm.org/, last visited 25-04-2007*, 2005.

[87] Frank Niessink and Hans van Vliet. Software maintenance from a service perspective. In *Journal of Software Maintenance: Research and Practice*, volume 12, pages 103–120, 2000.

[88] R Nieuwenhuis, I. van de Weerd, L. Bijlsma, S Brinkkemper, and J. Versendaal. The software product management workbench: An integrated environment for managing product releases in a distributed development context. In *Proceedings of the International Conference on Advanced Information Systems Engineering Forum*, pages 914–918, 2006.

[89] Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification. In *OMG document ptc03-07-08, please see htttp://www.omg.org*, 2003.

[90] Object Management Group. IT Portfolio Management Specification. In *OMG Final Adopted Specification (bei/04-06-07), please see htttp://www.omg.org, (last visited 27-05-2007)*, 2004.

[91] Object Management Group. Reusable Asset Specification. In *OMG Final Adopted Specification (V2.2), please see htttp://www.omg.org (last visited 27-05-2007)*, 2004.

[92] Johan Natt och Dag, Vincenzo Gervasi, Sjaak Brinkkemper, and Björn Regnell. A linguistic-engineering approach to large-scale requirements management. In *IEEE Software*, volume 22, pages 32–39, 2005.

[93] Home of the Spice ISO standard. http://www.isospice.com/. Last visited: 21-05-2007.

[94] Organisation and Information Research Group website. http://www.cs.uu.nl/groups/OI. Last visited: 21-05-2007.

[95] ISO International Standard Organization. (iso/iec 12207) standard for information technology—software lifecycle processes. New York, NY, 1998. 85 S.

[96] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[97] Juha Patosalmi. Collaborative decision-making in supply chain management. In *Seminar in Business Strategy*, 2003.

[98] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. The capability maturity model: Guidelines for improving the software process. In *SEI Series in Software Engineering.* Addison-Wesley Publishing Company, 1995.

[99] F. Plsil, D. Blek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE.

[100] Vaclav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. In *Computer*, volume 33, pages 66–71, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[101] Kristian Rautiainen, Casper Lassenius, Jarno Vahaniitty, Maaret Pyhajarvi, and Jari Vanhanen. A tentative framework for managing software product development in small companies. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.

[102] Karen Renaud and Richard Cooper. An error reporting and feedback component for component-based transaction processing systems. In *Proceedings of the 1999 User Interfaces to Data Intensive Systems*, page 141, Washington, DC, USA, 1999. IEEE.

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

[104] Pierre N. Robillard. The role of knowledge in software development. In *Commun. ACM*, volume 42, pages 87–92. ACM Press, 1999.

[105] Ken Schwaber and Mike Beedle. Development with scrum. 150 pages, Prentice-Hall, 2002.

[106] Dag Sjoberg. Quantifying schema evolution. In *Information and Software Technology*, volume 35, pages 35–44, 1993.

[107] University College London software systems engineering group website. http://sse.cs.ucl.ac.uk/. Last visited: 21-05-2007.

[108] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., 2002.

[109] Vanish Talwar, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. Approaches for service deployment. In *IEEE Internet Computing*, volume 9, pages 70–80, Piscataway, NJ, USA, 2005. IEEE Educational Activities Department.

[110] Walter F. Tichy. RCS — a system for version control. In *Software — Practice and Experience*, volume 15, pages 637–654, 1985.

[111] J. Tiihonen, T. Soininen, T. Mannisto, and R. Sulonen. State of the practice in product configuration – a survey of 10 cases in the finnish industry. In *Knowledge Intensive CAD, First Edition*. 352 pages, Chapman et Hall.

[112] BitTorrent tool website. http://www.bittorrent.com/. Last visited: 29-05-2007.

[113] MetaEnvironment tool website. http://www.meta-environment.org/. Last visited: 21-05-2007.

[114] Pheme tool website. http://www.cs.uu.nl/Pheme. Last visited: 30-08-2007.

[115] PowerUpdate tool website. http://www.powerupdate.com/. Last visited: 29-05-2007.

[116] SoftWide tool website. http://www.SoftWide.com. Last visited: 29-05-2007.

[117] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[118] V Vaishnavi and B Kuechler. Design research in information systems. AISWorldNet, http://www.isworld.org/Researchdesign/drisISworld.htm (last visited 27-05-2007), 2006.

[119] Inge van de Weerd, Sjaak Brinkkemper, Richard Nieuwenhuis, Johan Versendaal, and Lex Bijlsma. On the creation of a reference framework for software product management: Validation and tool support. In *Proceedings of the International Workshop on Software Product Management*, pages 3–12, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[120] Inge van de Weerd, Sjaak Brinkkemper, Richard Nieuwenhuis, Johan Versendaal, and Lex Bijlsma. Towards a reference framework for software product management. In *Proceedings of the 14th International Requirements Engineering Conference*, 2006.

[121] Andr van der Hoek. Design-time product line architectures for any-time variability. In *Sci. Comput. Program.*, volume 53, pages 285–304, Amsterdam, The Netherlands, The Netherlands, 2004. Elsevier North-Holland, Inc.

[122] Andre van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software release management. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 159–175. Springer–Verlag, 1997.

[123] Tijs van der Storm. Variability and component composition. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference*, LNCS, pages 86–100. Springer, 2004.

[124] Tijs van der Storm. Continuous release and upgrade of component-based software. In *Proceedings of the 12th International Workshop on Software Configuration Management*, pages 43–57, 2005.

[125] Tijs van der Storm. The Sisyphus continuous integration system. In *Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2007.

[126] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[127] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. An alternative to quiescence: Tranquility. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 73–82, Washington, DC, USA, 2006. IEEE Computer Society.

[128] Centrum voor Wiskunde en Informatica. http://www.cwi.nl. Last visited: 21-05-2007.

[129] TomTom Corporate Website. http://www.tomtom.com. Last visited: 21-05-2007.

[130] Microsoft's Open Source Deployment Tool WIX. http://wix.sourceforge.net/. Last visited: 29-05-2007.

[131] Lai Xu. A multi–party contract model. In *SIGecom Exchange*, volume 5, pages 13–23, New York, NY, USA, 2004. ACM Press.

[132] Lai Xu and Sjaak Brinkkemper. Concepts of product software: Paving the road for urgently needed research. In *First International Workshop on Philosophical Foundations of Information Systems Engineering*. LNCS, Springer-Verlag, 2005.

[133] Robert K Yin. Case study research - design and methods. 179 pages, SAGE Publications, 3rd ed., 2003.

[134] David Zager. Collaboration on the fly. In *Proceedings of the Academia/Industry Working Conference on Research Challenges*, page 65, Washington, DC, USA, 2000. IEEE Computer Society.

[135] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *SIGSOFT Softw. Eng. Notes*, volume 24, pages 253–267, New York, NY, USA, 1999. ACM Press.

# Publication List


[136] Sjaak Brinkkemper, Ivo van Soest, and Slinger Jansen. Modeling of product software businesses: Investigation into industry product and channel typologies. In *The Inter-Networked World: ISD Theory, Practice, and Education, proceedings of the Sixteenth International Conference on Information Systems Development (ISD 2007)*. Springer-verlag, 2007.

[137] Ilja Heitlager, Slinger Jansen, Sjaak Brinkkemper, and Remko 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.

[138] Slinger Jansen. Alleviating the release and deployment effort of product software by explicitly managing component knowledge. In *Proceedings of the Workshop on Development and Deployment of Product Software*, pages 21–30. US Education Service, 2005.

[139] Slinger Jansen. Improving the customer configuration update process by explicitly managing software knowledge. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 965–968, New York, NY, USA, 2006. ACM Press.

[140] Slinger Jansen. Pheme: An infrastructure to enable any type of communication between a software vendor and an end-user. In *International Conference on Software Maintenance 2007, tool demonstration*, 2007.

[141] Slinger Jansen, Gerco Ballintijn, Sjaak Brinkkemper, and Arco van Nieuwland. Integrated development and maintenance for the release, delivery, deployment, and customization of product software: a case study in mass-market erp software (chapter 3 of this thesis). In *Journal of Software Maintenance and Evolution: Research and Practice*, volume 18, pages 133–151. John Wiley & Sons, Ltd., 2006.

[142] Slinger Jansen and Sjaak Brinkkemper. Modelling deployment using feature descriptions and state models for component-based software product families (chapter 6 of this thesis). In *3rd International Working Conference on Component Deployment (CD 2005)*, LNCS. Springer–Verlag, 2005.

[143] Slinger Jansen and Sjaak 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 (chapter 2 of this thesis). In *proceedings of the International Conference on Software Maintenance (ICSM2006, Research track)*, September 2006.

[144] Slinger Jansen and Sjaak Brinkkemper. Evaluating the release, delivery, and deployment processes of eight large product software vendors applying the customer configuration update model. In *WISER '06: Proceedings of the 2006 international Workshop on interdisciplinary software engineering research*, pages 65–68, New York, NY, USA, 2006. ACM Press.

[145] Slinger Jansen and Sjaak Brinkkemper. Ten misconceptions about product software release management explained using update cost/value functions (chapter 8 of this thesis). In *First International Workshop on Software Product Management*. IEEE, 2006.

[146] Slinger Jansen and Sjaak Brinkkemper. A benchmark survey into the customer configuration processes and practices of product software vendors in the netherlands (chapter 4 of this thesis). In *submitted*, 2007.

[147] Slinger Jansen, Sjaak Brinkkemper, and Gerco Ballintijn. A process framework and typology for software product updaters (chapter 5 of this thesis). In *Ninth European Conference on Software Maintenance and Reengineering*, pages 265–274. IEEE, 2005.

[148] Slinger Jansen, Sjaak Brinkkemper, Gerco Ballintijn, and Arco van Nieuwland. Integrated development and maintenance of software products to support efficient updating of customer configurations: A case study in mass market erp software. In *Proceedings of the 21st International Conference on Software Maintenance*. IEEE, 2005.

[149] Slinger Jansen, Sjaak Brinkkemper, and Tijs van der Storm. Living on the cutting edge: Automating continuous customer configuration updating (chapter 7 of this thesis). In *Proceedings of the ERCIM Workshop on Software Evolution 2007*, 2007.

[150] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. Analyzing the business of software: A modelling technique for software supply networks. In *Caise Forum at the International Conference on Advanced Information Systems Engineering*, 2007.

[151] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. Providing transparency in the business of software: A modelling technique for software supply networks(chapter 9 of this thesis). In *Proceedings of the 8th IFIP Working Conference on Virtual Enterprises*, 2007.

[152] Slinger Jansen and Wilfried 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.

# Summary

Product software development is the activity of development, modification, reuse, re-engineering, maintenance, or any other activities that result in packaged configurations of software components or software-based services that are released for and traded in a specific market [132]. An increasingly important part of product software development is CCU. *CCU is the combination of the vendor side release process, the product or update delivery process, the customer side deployment process, and the activation process*. Product software vendors encounter particular problems when trying to improve these processes, because vendors have to deal with multiple revisions, variable features, different deployment environments and architectures, different customers, different distribution media, and dependencies on external products. Also, there are not many tools available that support the delivery and deployment of software product releases that are generic enough to accomplish these tasks for any product.

In figure 10.1 the product software development process is modelled. It models software development, release, deployment, and usage of a product. These processes are controlled by policies. The policies define how product software development and CCU processes are controlled. The debug policy describes what features are built into a product and how and when bugs are fixed. The release policy defines when a product or update is released by a product software vendor and to what customers. The delivery policy determines how products are delivered to the end-user, such as through the internet or on a DVD or USB stick. The deployment policy determines how and how often these updates and products are installed. While the product is being used, actions, errors, and exceptions may be logged and stored locally. The feedback policy then defines what and how often information is sent back to the vendor.

In 9 industrial case studies it was discovered that as much as 15% of the deployments and product updates of new products do not proceed as planned and require unplanned extra support from the software vendor. These organizations are held back in their growth, due to the fact that they cannot handle larger customer bases, since it would result into more configurations that require maintenance and updates. When software vendors attempt to improve CCU three things become apparent: (1) there are no adequate process descriptions for CCU, (2) there is a lack of tools to support CCU, (3) software vendors use a lot of time automating CCU tasks, even though these tasks are similar for all software vendors.

This thesis does not stand alone in its attempt to improve CCU for product software vendors. The work on evaluating product updaters is largely based on an earlier evaluation model provided by Carzaniga et al [21]. Furthermore, the entrepreneurial

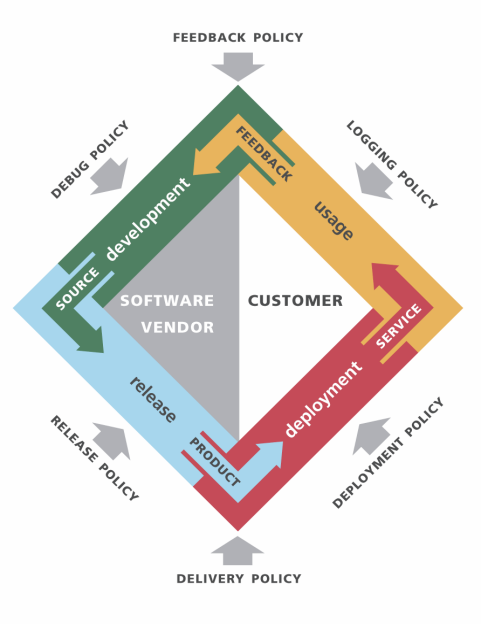



Figure 10.1: CCU and its Policy Context

aspects of this work were inspired by Xu and Brinkkemper's work on product software [132]. The work on software supply networks and business redesign were inspired by the works of Farbey and Finkelstein [42]. The work on deployment of components in multidimensional configurations could not have been carried out without the revelation of van der Storm that these configuration spaces can be reduced to binary decision trees [123]. The tool Pheme was largely inspired by the Software Dock [52] and can be considered a next generation of the Software Dock.

Even though some of these works have been successful in improving CCU, they approach the problem from a single direction. This thesis proposes a multidirectional approach, where best practices are combined with up to date reviews of CCU support tools, new tools and tool proposals, and finally a CCU process model. The contribution thus is threefold:

- Gives a detailed view of the state of the practice of CCU, i.e., release, delivery, deployment, and activation and usage in chapter 2. The case studies and tool evaluations tell the scientific community where the unsolved issues are, what practices are currently prevalent, and what practices are more successful (chapter 3) than others in the industry. A survey is used to test the hypotheses in these chapters and generalize the conclusions.
- Proposes a number of improvements for software release planning (chapter 8), software development [138], and business development within a software supply network (chapter 9), based on the results found in (1), to improve the art of CCU.

- Provides a tool ([140] and chapter 6) to improve deployment and configuration of components in large product lines. Furthermore, a tool is provided that improves customer-vendor relations by proposing an infrastructure for communication about software. These tools and fourteen others (chapter 5) are evaluated using the product updater evaluation method.

To ensure correct results from the case studies rigorous case study procedures were followed. The applicability of the CCU model was validated using a survey to find out whether applying the CCU model would result in correct and useful improvement areas and advice. The survey discovered the CCU capabilities of software vendors. The component deployment and configuration tool has been evaluated using different theoretical examples. The Pheme tool has proven useful in experimental setups for an open source content management tool and a scientific open source tool. Finally, the software supply network modeling method has been applied to a case study.

These three contributions allow software vendors to stop dabbling around small customer numbers, and to make the jump to larger customer bases, with only a small increase in manpower and effort. Simultaneously, when applied correctly, the presented principles for CCU enable product software vendors to shorten release times, by reduction of release and upgrade costs. This in turn facilitates quicker feedback cycles from customers and more agility for the product software vendor, enabling them to potentially reach higher quality levels for their products.

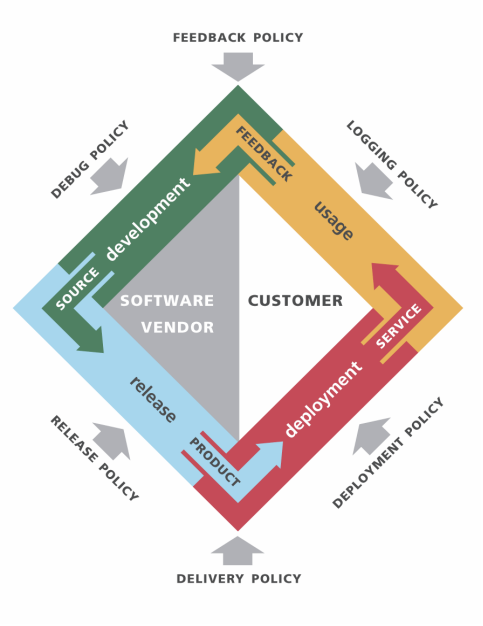
This thesis does not exhaustively solve problems in CCU. First and foremost, a workbench for product software management is required that integrates different tools and enables a software vendor to use an integrated toolset for software licensing, customer relationship management, development management, release planning, release creation, installer creation, product news publication, and user feedback processing. Secondly, knowledge about the CCU model needs to be disseminated to product software vendors through more surveys and publications.

# Nederlandse Samenvatting: Actualiseren van Klantconfiguraties via een Softwareleverantienetwerk

Het uitleveren van een softwareproduct is een complexe taak voor productsoftwarebedrijven. Het product kan bestaan uit verschillende onderdelen van verschillende leveranciers, heeft verschillende eigenschappen voor verschillende klanten, bestaat uit componenten die constant evolueren, en wordt geïnstalleerd op een systeem waarvan de specificaties à priori onbekend zijn. Dit proefschrift brengt de problemen in kaart die productsoftwarebedrijven tegenkomen bij het uitleveren, verschaft procesbeschrijvingen, en geeft gereedschap en modellen die productsoftwarebedrijven in staat stellen de uitleverprocessen te verbeteren.

Het ontwikkelen van productsoftware bestaat uit verschilende fasen: ontwerp, aanpassing, hergebruik, onderhoud, en elke andere activiteit die leidt tot een gebundelde configuratie van softwarecomponenten of software-gebaseerde diensten die verhandeld en uitgeleverd worden in een specifieke markt [132]. KlantConfiguratie Updaten (KCU), wat deel uitmaakt van het ontwikkelingsproces, is een steeds belangrijker wordend onderwerp, doordat productsoftwarebedrijven meer klanten met standaard producten willen bedienen. *KCU is de combinatie van de uitleverprocessen aan de verkoperkant, en de installatie-, gebruiks-, en activatieprocessen aan de kant van de klant*. KCU bevat dus de processen aan de kant van de softwareleverancier en de processen die de configuratie van de klant wijzigen waardoor deze nieuwe functionaliteit bemachtigt. Er zijn weinig generieke applicaties beschikbaar die softwarebedrijven kunnen ondersteunen bij al deze processen.

In figuur 10.2 is het ontwikkelingsproces van productsoftware gemodelleerd vanuit het KCU perspectief. De vier processen die hierbij een rol spelen zijn ontwikkeling (development), uitlevering, installatie (deployment), en gebruik (usage). Deze processen worden beïnvloed door sturing vanuit het productsoftwarebedrijf en vanuit de klant. Het debugbeleid (debug policy) bepaalt hoe ontwikkeld wordt en wanneer en hoe nieuwe functionaliteit geïmplementeerd en fouten verwijderd worden. Het uitleverbeleid bepaalt wanneer nieuwe versies van het product worden uitgebracht, in acht nemende dat er al versies geïnstalleerd kunnen zijn bij klanten. Het uitleverbeleid bepaalt hoe vaak software wordt verstuurd naar of opgehaald wordt door klanten en

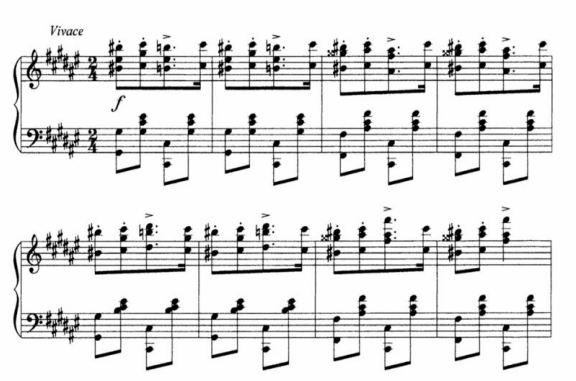



Figuur 10.2: KCU en Beleid (policies)

met welk medium, zoals via internet of via een USB stick. Het installatiebeleid bepaalt hoe en hoe vaak het product wordt geïnstalleerd en aangepast. Verder bepaalt het loggingbeleid hoe kennis en foutrapporten worden verzameld over het gebruik van de klant. Als laatste bepaalt het terugkoppelingbeleid (feedback) hoe en hoe vaak de kennis wordt teruggestuurd naar het productsoftwarebedrijf.

Een voorbeeld: een Amerikaanse componiste en dirigente begint vanaf volgende week een Europese tour met haar nieuwste werk voor orkest met koor (de instrumenten zijn de hardware, de instrumentalisten de gebruikers). De partituren (de software) zijn twee weken geleden naar de instrumentalisten en het koor verstuurd en iedereen kent de muziek. De dirigente besluit wat onhandige overgangen weg te werken (bugfixes) en nog wat speelse versieringen (functionaliteit) toe te voegen. Verder voegt zij ook nog een uitgebreide solo toe (een nieuwe softwarecomponent) die recentelijk door een collega van de dirigente is bedacht (een andere leverancier). De dirigente staat nu voor een lastige keuze. Stuurt ze de nieuwe partituur aan alle orkest- en koorleden via de e-mail of neemt ze deze vanavond mee in het vliegtuig (hoe levert ze de partituur *uit*)? Dirigeert ze de eerste keer nog de oude versie of de nieuwe versie (hoe installeert ze de software en wanneer)? Hoe berekent ze de opbrengst voor de collega dirigent? Per concert? Per opname?

De orkest- en koorleden (klanten) hebben evenzoveel problemen. Wat nu als de “onhandige” overgangen nodig waren om van instrument te wisselen (deels hardware, deels klanten)? En kan de solist wel zo snel spelen op haar viool? En wat doen ze




Figuur 10.3: Franz Liszt - Hungarian Rhapsody No. 2

met de partituur? Vervangen ze de oude door een hele nieuwe of brengen ze een paar wijzigingen aan? Ook kunnen er fouten in de partituur staan die de dirigente niet heeft opgemerkt. Vertellen de instrumentalisten dat tijdens de repetities, tijdens het concert (!), of al op voorhand via de telefoon?

Dit voorbeeld geeft maar een deel van alle problemen die een productsoftwarebedrijf tegenkomt. Een partituur zal bijvoorbeeld nooit zo vaak en snel veranderen als een stuk software, waar continu stukken aan worden gebouwd en herschreven. De hoofdgedachte van dit proefschrift is dan ook dat een softwareproduct zo vaak verandert dat het productsoftwarebedrijf een infrastructuur voor continue kennisuitwisseling moet inrichten (in het voorbeeld zou dat een vaste e-mail- en telefoonlijst kunnen zijn).

In dit proefschrift wordt aan de hand van negen industriële case studies aangetoond dat 15% van de installaties van aanpassingen en nieuwe producten niet werken als de bedoeling was en dus extra inspanning nodig hebben van het productsoftwarebedrijf. Deze organizaties worden dan ook beperkt in hun groei, aangezien de kosten per nieuwe klant te hoog zijn. Als productsoftwarebedrijven deze extra kosten willen verminderen door verbeteringen in de KCU processen door te voeren worden drie dingen duidelijk: (1) er zijn geen volledige omschrijvingen van de KCU processen. Daarnaast, (2) zijn er weinig software oplossingen beschikbaar die KCU kunnen ondersteunen. Als laatste (3) verliezen software bedrijven veel tijd bij het automatiseren van KCU processen, zelfs al zijn de processen voor het uitleveren gelijk voor ieder productsoftwarebedrijf.

Dit proefschrift staat niet alleen in zijn poging KCU voor productsoftwarebedrijven te verbeteren. Het hoofdstuk over productupdaters is gebaseerd op een evaluatie model van Carzaniga et al. [21]. De specifieke focus op productsoftwarebedrijven is geïnspireerd door Xu en Brinkkempers werk over productsoftware [132]. Het werk over softwarelevarantienetwerken en proces herontwerp is geïnspireerd door het onderzoek van Farbey en Finkelstein [42]. Het werk over installatie van componenten in multidimensionale configuraties had niet geschreven kunnen worden zonder de

methode van van der Storm waarmee deze configuratie ruimtes gereduceerd kunnen worden tot binaire beslissingsbomen [123]. De kennisuitwisselingsinfrastructuur Pheme is grotendeels gebaseerd op "the Software Dock" [52] en kan gezien worden als een volgende generatie van deze applicatie.

Hoewel deze werken succesvol zijn op het gebied van KCU, benaderen ze het probleem hoe KCU te verbeteren slechts vanuit één richting. Het onderzoek dat wordt gepresenteerd in dit proefschrift gebruikt een multi-directionele aanpak, waar best-practices vanuit product management worden gecombineerd met recente reviews van KCU applicaties, nieuwe applicaties, en als laatste een evaluatie model voor KCU processen. Dit proefschrift levert de volgende drie bijdragen:

- Het proefschrift geeft een gedetailleerd beeld hoe KCU op dit moment is geïmplementeerd bij productsoftwarebedrijven [143]. Dit beeld vertelt de wetenschappelijke wereld waar de onopgeloste problemen liggen, welke praktijken op dit moment het meest worden toegepast, en welke praktijken en processen succesvoller zijn dan andere [141]. Deze gegevens zijn verzameld door middel van case studies en een enquête.

- Het proefschrift geeft geïntegreerde procesmodellen voor elk proces binnen KCU. Deze modellen worden vervolgens toegepast op een aantal verschillende productsoftwarebedrijven door middel van case studies [143] en een benchmark enquête [146].

- Het proefschrift presenteert twee applicaties om KCU mee te verbeteren. De eerste is een applicatie [142] waarmee componenten uit grote productlijnen kunnen worden samengesteld en geïnstalleerd. De tweede applicatie [140] verbetert de relatie tussen een productsoftwarebedrijf en haar klanten door de invoering van een infrastructuur voor (continue) communicatie over software producten. Deze applicaties en veertien anderen worden geëvalueerd met de productupdater evaluatiemethode [147].

Om correcte resultaten te krijgen zijn voor de case studies vaste procedures gevolgd. De toepasbaarheid van het KCU model is gevalideerd met een enquête, door het voorstellen van KCU verbeteringen aan de respondenten van de enquête. De enquête geeft een momentopname hoe 74 Nederlandse productsoftware bedrijven hun software uitleveren. Het component installatie- en aanpassingsapplicatie is geëvalueerd met theoretische problemen en voorbeelden. De kennisuitwisselingsinfrastructuur Pheme is gevalideerd door het toe te passen op twee open source producten. Als laatste is de modelleermethode voor software leverantienetwerken toegepast op een case study.

De drie genoemde bijdragen stellen productsoftwarebedrijven in staat om, met een kleine toename in mankracht en inspanning, de sprong te maken naar grote klantenbestanden. Tegelijkertijd kunnen, mits correct toegepast, de KCU verbeteringsprincipes gebruikt worden om uitlevertijden te verkorten, door het verkleinen van uitlever- en aanpassingskosten. Dit stelt een bedrijf dan weer in staat om sneller terugkoppeling te krijgen op nieuwe uitgeleverde versies en maakt de productsoftware flexibeler, waarmee deze bedrijven potentieel hogere kwaliteitsniveaus halen voor hun producten.

Dit proefschrift lost niet alle KCU problemen op. Ten eerste is er behoefte aan een product management workbench die applicaties integreert die een productsoftwarebedrijf in staat stellen om een geïntegreerde applicatieset te gebruiken voor software licensing, customer relationship management, uitleverplanning, uitlevering, publicatie van nieuws over een product, en verwerking van (automatische) terugkoppeling van gebruikers. Als laatste moet de KCU kennis verder gedissemineerd worden via enquêtes en publicaties.

# Resume

Slinger Jansen obtained his Masters Degree in computer science at Leiden University in 2003. He is currently active as an assistant professor at Utrecht University at the Center for Organisation and Information. His research and educational activities focus on product software and entrepreneurship, with a strong focus on development methodology and product software management. He is active as a committee member for several software engineering and modelling conferences and publishes on an international level. He has previously worked at the Centrum voor Wiskunde en Informatica (CWI) and at University College London and maintains strong ties with those institutions.

# SIKS PhD Theses

98-1, Johan van den Akker (CWI),DEGAS - An Active Temporal Database of Autonomous Objects
98-2, Floris Wiesman (UM), Information Retrieval by Graphically Browsing Meta-Information
98-3, Ans Steuten (TUD),A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
98-4, Dennis Breuker (UM),Memory versus Search in Games
98-5, E.W.Oskamp (RUL),Computerondersteuning bij Straftoemeting
99-1, Mark Sloof (VU),Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
99-2, Rob Potharst (EUR),Classification using decision trees and neural nets
99-3, Don Beal (UM),The Nature of Minimax Search
99-4, Jacques Penders (UM),The practical Art of Moving Physical Objects
99-5, Aldo de Moor (KUB),Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
99-6, Niek J.E. Wijngaards (VU),Re-design of compositional systems
99-7, David Spelt (UT),Verification support for object database design
99-8, Jacques H.J. Lenting (UM),Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.
2000-1, Frank Niessink (VU),Perspectives on Improving Software Maintenance
2000-2, Koen Holtman (TUE),Prototyping of CMS Storage Management
2000-3, Carolien M.T. Metselaar (UVA),Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4, Geert de Haan (VU),ETAG: A Formal Model of Competence Knowledge for User Interface Design
2000-5, Ruud van der Pol (UM),Knowledge-based Query Formulation in Information Retrieval.
2000-6, Rogier van Eijk (UU),Programming Languages for Agent Communication
2000-7, Niels Peek (UU),Decision-theoretic Planning of Clinical Patient Management
2000-8, Veerle Coup (EUR), Sensitivity Analyis of Decision-Theoretic Networks
2000-9, Florian Waas (CWI),Principles of Probabilistic Query Optimization
2000-10, Niels Nes (CWI),Image Database Management System Design Considerations Algorithms and Architecture
2000-11, Jonas Karlsson (CWI),Scalable Distributed Data Structures for Database Management
2001-1, Silja Renooij (UU), Qualitative Approaches to Quantifying Probabilistic Networks
2001-2, Koen Hindriks (UU),Agent Programming Languages: Programming with Mental Models
2001-3, Maarten van Someren (UvA),Learning as problem solving
2001-4, Evgueni Smirnov (UM),Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5,Jacco van Ossenbruggen (VU),Processing Structured Hypermedia: A Matter of Style

2001-6, Martijn van Welie (VU),Task-based User Interface Design
2001-7, Bastiaan Schonhage (VU),Diva: Architectural Perspectives on Information Visualization
2001-8, Pascal van Eck (VU),A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9, Pieter Jan 't Hoen (RUL),Towards Distributed Development of Large Object-Oriented Models Views of Packages as Classes
2001-10, Maarten Sierhuis (UvA),Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11, Tom M. van Engers (VUA),Knowledge Management: The Role of Mental Models in Business Systems Design
2002-01, Nico Lassing (VU),Architecture-Level Modifiability Analysis
2002-02, Roelof van Zwol (UT),Modelling and searching web-based document collections
2002-03, Henk Ernst Blok (UT),Database Optimization Aspects for Information Retrieval
2002-04, Juan Roberto Castelo Valdueza (UU),The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05, Radu Serban (VU),The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06, Laurens Mommers (UL),Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07, Peter Boncz (CWI),Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08, Jaap Gordijn (VU),Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09, Willem-Jan van den Heuvel( KUB),Integrating Modern Business Applications with Objectified Legacy Systems
2002-10, Brian Sheppard (UM), Towards Perfect Play of Scrabble
2002-11, Wouter C.A. Wijngaards (VU), Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12, Albrecht Schmidt (Uva), Processing XML in Database Systems
2002-13, Hongjing Wu (TUE), A Reference Architecture for Adaptive Hypermedia Applications
2002-14, Wieke de Vries (UU), Agent Interaction: Abstract Approaches to Modelling Programming and Verifying Multi-Agent Systems
2002-15, Rik Eshuis (UT), Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16, Pieter van Langen (VU), The Anatomy of Design: Foundations Models and Applications
2002-17, Stefan Manegold (UVA), Understanding Modeling and Improving Main-Memory Database Performance
2003-01, Heiner Stuckenschmidt (VU), Ontology-Based Information Sharing in Weakly Structured Environments
2003-02, Jan Broersen (VU), Modal Action Logics for Reasoning About Reactive Systems
2003-03, Martijn Schuemie (TUD), Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04, Milan Petkovic (UT), Content-Based Video Retrieval Supported by Database Technology
2003-05, Jos Lehmann (UVA), Causation in Artificial Intelligence and Law - A modelling approach
2003-06, Boris van Schooten (UT), Development and specification of virtual environments
2003-07, Machiel Jansen (UvA), Formal Explorations of Knowledge Intensive Tasks

2003-08, Yongping Ran (UM), Repair Based Scheduling
2003-09, Rens Kortmann (UM), The resolution of visually guided behaviour
2003-10, Andreas Lincke (UvT), Electronic Business Negotiation: Some experimental studies on the interaction between medium innovation context and culture
2003-11, Simon Keizer (UT), Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12, Roeland Ordelman (UT), Dutch speech recognition in multimedia information retrieval
2003-13, Jeroen Donkers (UM), Nosce Hostem - Searching with Opponent Models
2003-14, Stijn Hoppenbrouwers (KUN), Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15, Mathijs de Weerdt (TUD), Plan Merging in Multi-Agent Systems
2003-16, Menzo Windhouwer (CWI), Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17, David Jansen (UT), Extensions of Statecharts with Probability Time
2003-18, Levente Kocsis (UM), Learning Search Decisions
2004-01, Virginia Dignum (UU), A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02, Lai Xu (UvT), Monitoring Multi-party Contracts for E-business
2004-03, Perry Groot (VU), A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04, Chris van Aart (UVA), Organizational Principles for Multi-Agent Architectures
2004-05, Viara Popova (EUR), Knowledge discovery and monotonicity
2004-06, Bart-Jan Hommes (TUD), The Evaluation of Business Process Modeling Techniques
2004-07, Elise Boltjes (UM), Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08, Joop Verbeek (UM), Politie en de Nieuwe Internationale Informatiemarkt Grensregionale politile gegevensuitwisseling en digitale expertise
2004-09, Martin Caminada (VU), For the Sake of the Argument; explorations into argument-based reasoning
2004-10, Suzanne Kabel (UVA), Knowledge-rich indexing of learning-objects
2004-11, Michel Klein (VU), Change Management for Distributed Ontologies
2004-12, The Duy Bui (UT), Creating emotions and facial expressions for embodied agents
2004-13, Wojciech Jamroga (UT), Using Multiple Models of Reality: On Agents who Know how to Play
2004-14, Paul Harrenstein (UU), Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15, Arno Knobbe (UU), Multi-Relational Data Mining
2004-16, Federico Divina (VU), Hybrid Genetic Relational Search for Inductive Learning
2004-17, Mark Winands (UM), Informed Search in Complex Games
2004-18, Vania Bessa Machado (UvA), Supporting the Construction of Qualitative Knowledge Models
2004-19, Thijs Westerveld (UT), Using generative probabilistic models for multimedia retrieval
2004-20, Madelon Evers (Nyenrode), Learning from Design: facilitating multidisciplinary design teams
2005-01, Floor Verdenius (UVA), Methodological Aspects of Designing Induction-Based Applications
2005-02, Erik van der Werf (UM), AI techniques for the game of Go
2005-03, Franc Grootjen (RUN), A Pragmatic Approach to the Conceptualisation of Language
2005-04, Nirvana Meratnia (UT), Towards Database Support for Moving Object data
2005-05, Gabriel Infante-Lopez (UVA), Two-Level Probabilistic Grammars for Natural

Language Parsing
2005-06, Pieter Spronck (UM), Adaptive Game AI
2005-07, Flavius Frasincar (TUE), Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08, Richard Vdovjak (TUE), A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09, Jeen Broekstra (VU), Storage
2005-10, Anders Bouwer (UVA), Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
2005-11, Elth Ogston (VU), Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
2005-12, Csaba Boer (EUR), Distributed Simulation in Industry
2005-13, Fred Hamburg (UL), Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14, Borys Omelayenko (VU), Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15, Tibor Bosse (VU), Analysis of the Dynamics of Cognitive Processes
2005-16, Joris Graaumans (UU), Usability of XML Query Languages
2005-17, Boris Shishkov (TUD), Software Specification Based on Re-usable Business Components
2005-18, Danielle Sent (UU), Test-selection strategies for probabilistic networks
2005-19, Michel van Dartel (UM), Situated Representation
2005-20, Cristina Coteanu (UL), Cyber Consumer Law State of the Art and Perspectives
2005-21, Wijnand Derks (UT), Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics
2006-01, Samuil Angelov (TUE), Foundations of B2B Electronic Contracting
2006-02, Cristina Chisalita (VU), Contextual issues in the design and use of information technology in organizations
2006-03, Noor Christoph (UVA), The role of metacognitive skills in learning to solve problems
2006-04, Marta Sabou (VU), Building Web Service Ontologies
2006-05, Cees Pierik (UU), Validation Techniques for Object-Oriented Proof Outlines
2006-06, Ziv Baida (VU), Software-aided Service Bundling - Intelligent Methods and Tools for Graphical Service Modeling
2006-07, Marko Smiljanic (UT), XML schema matching – balancing efficiency and effectiveness by means of clustering
2006-08, Eelco Herder (UT), Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-09, Mohamed Wahdan (UM), Automatic Formulation of the Auditor's Opinion
2006-10, Ronny Siebes (VU), Semantic Routing in Peer-to-Peer Systems
2006-11, Joeri van Ruth (UT), Flattening Queries over Nested Data Types
2006-12, Bert Bongers (VU), Interactivation - Towards an e-cology of people, our technological environment, and the arts
2006-13, Henk-Jan Lebbink (UU), Dialogue and Decision Games for Information Exchanging Agents
2006-14, Johan Hoorn (VU), Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change
2006-15, Rainer Malik (UU), CONAN: Text Mining in the Biomedical Domain
2006-16, Carsten Riggelsen (UU), Approximation Methods for Efficient Learning of Bayesian Networks
2006-17, Stacey Nagata (UU), User Assistance for Multitasking with Interruptions on a Mobile

Device
2006-18, Valentin Zhizhkun (UVA), Graph transformation for Natural Language Processing
2006-19, Birna van Riemsdijk (UU), Cognitive Agent Programming: A Semantic Approach
2006-20, Marina Velikova (UvT), Monotone models for prediction in data mining
2006-21, Bas van Gils (RUN), Aptness on the Web
2006-22, Paul de Vrieze (RUN), Fundaments of Adaptive Personalisation
2006-23, Ion Juvina (UU), Development of Cognitive Model for Navigating on the Web
2006-24, Laura Hollink (VU), Semantic Annotation for Retrieval of Visual Resources
2006-25, Madalina Drugan (UU), Conditional log-likelihood MDL and Evolutionary MCMC
2006-26, Vojkan Mihajlovic (UT), Score Region Algebra: A Flexible Framework for Structured Information Retrieval
2006-27, Stefano Bocconi (CWI), Vox Populi: generating video documentaries from semantically annotated media repositories
2006-28, Borkur Sigurbjornsson (UVA), Focused Information Access using XML Element Retrieval
2007-01, Kees Leune (UvT), Access Control and Service-Oriented Architectures
2007-02, Wouter Teepe (RUG), Reconciling Information Exchange and Confidentiality: A Formal Approach
2007-03, Peter Mika (VU), Social Networks and the Semantic Web
2007-04, Jurriaan van Diggelen (UU), Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach
2007-05, Bart Schermer (UL), Software Agents , Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
2007-06, Gilad Mishne (UVA), Applied Text Analytics for Blogs
2007-07, Natasa Jovanovic' (UT), To Whom It May Concern - Addressee Identification in Face-to-Face Meetings
2007-08, Mark Hoogendoorn (VU), Modeling of Change in Multi-Agent Organizations
2007-09, David Mobach (VU), Agent-Based Mediated Service Negotiation
2007-10, Huib Aldewereld (UU), Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
2007-11, Natalia Stash (TUE), Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System
2007-12, Marcel van Gerven (RUN), Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
2007-13, Rutger Rienks (UT), Meetings in Smart Environments; Implications of Progressing Technology
2007-14, Niek Bergboer (UM), Context-Based Image Analysis
2007-15, Joyca Lacroix (UM), NIM: a Situated Computational Memory Model
2007-16, Davide Grossi (UU), Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
2007-17, Theodore Charitos (UU), Reasoning with Dynamic Networks in Practice
2007-18, Bart Orriens (UvT), On the development an management of adaptive business collaborations
2007-19, David Levy (UM), Intimate relationships with artificial partners
2007-20, Slinger Jansen (UU), Customer Configuration Updating in a Software Supply Network
2007-21, Karianne Vermaas (UU), Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005