An Information Systems Design Product Theory for the Class of Integrated Requirements and Release Management Systems

Timo Käkölä & Mervi Koivulahti-Ojala
University of Jyväskylä
40014 Jyväskylä
Finland
{timokk, meelheko}@jyu.fi

Jani Liimatainen
Accenture
00101 Helsinki
Finland
jani.liimatainen@accenture.com

Abstract: High-tech companies conducting product development need to collect and analyze requirements effectively, plan and implement releases, and allocate requirements to appropriate releases. Requirements and release management is complicated because development activities typically are scattered across multiple sites, involve multiple partners in different countries, leverage various development methods and tools, and are realized through various organizational arrangements such as release projects in organizations structured around products and permanent release teams in organizations responsible for the long-term development and maintenance of strategic software and hardware assets. Flexible, scalable, and secure groupware-based support for the activities provides substantial payoffs. Yet, the extant literature provides little theoretical guidance for designing and using requirements and release management systems in multi-site, multi-partner environments. This article develops the meta-requirements and a meta-design of an Information Systems Design Product Theory for the class of Requirements and Release Management Systems based on a case study in a global company and a literature review. The theory is scalable to meet the needs of global companies but simple enough so small and medium sized companies can also leverage it to implement requirements and release management solutions.

Keywords: Global software product development, Information systems design theory, Knowledge management, Release management, Requirements management, Software process improvement

1. INTRODUCTION

To succeed in the global markets of software-intensive products, high-tech companies need to shorten the cycle time of new product development while improving product quality and service delivery and maintaining or reducing the total resources required (Brown and Eisenhardt 1995; Meyer and Selinger 1998). This concern can be dealt through (1) internal strategies such as global software development, where development resources are distributed globally to reap cost benefits, leverage specialized competencies, and address specific needs of geographically-defined markets (Carmel and Agarwal 2001; Herbsleb and Moitra 2001; Ramasubbu et al. 2005), and software product line engineering and management, that is, the strategic acquisition, creation, and reuse of software assets (Käkölä and Dueñas 2006; Pohl et al. 2005; Van der Linden et al. 2007) or (2) external strategies such as acquiring commercial off-the-shelf components and outsourcing software development, maintenance, and related services to best-in-class service providers (Carmel and Agarwal 2002; Käkölä 2008).

All the strategies require companies to effectively collect, analyze, and utilize requirements (Adelson and Soloway 1985; Grynberg and Goldin 2003; Halbleib 2004; Hrones et al. 1992; Regnell et al. 2001; Salo and Käkölä 2005). This is particularly true during the earliest phases of product development in which different stakeholders need to integrate their knowledge into product concepts that direct the internal personnel and the service providers during the downstream phases of product development (Akao 1990; Burchill and Fine 1997; Griffin and Hauser 1992). A well-defined product concept is necessary to establish a viable product line
architecture that can be shared across the products within the product line to enable strategic reuse. Well-defined requirements, architectural interfaces, and product architectures are prerequisites for assigning appropriately scoped projects to internal units and service providers for implementation (Carmel and Agarwal 2002; Käkölä 2008).

The achievement of such integration is complicated by several factors (Curtis et al. 1988). Numerous requirements ranging from abstract wishes to detailed technical solution proposals are created continuously. Development activities are scattered across many sites and partners in different countries, limiting possibilities for setting up face-to-face meetings (Hrones et al. 1992). Organizational changes, differences in organizational cultures, and divergent perceptions about the prospective product’s mission may make it difficult to reach an agreement about the product definition (Ciborra 1996).

A commonly enacted software product line governance model and a strategic product line roadmapping process should be instituted to ensure the organization is ready for multi-site development (Van der Linden et al. 2007; Wesselius 2006). All sites should use compatible processes, methodologies, tools, and terminology as much as possible to enact the governance model (Herbsleb and Moitra 2001). Each product roadmap outlines the respective product line at a given point in time, explicates (from the market viewpoint) the major common and variable features of all foreseeable products of the product line, and schedules the deliveries of the products to markets (Pohl et al. 2005, Ch. 2, Ch. 9). For every product, a release plan should be made that allocates the features to scheduled product releases and responsible development organizations and documents the allocations.

Release planning must be conducted carefully and systematically by the stakeholders responsible for the requirements and the product strategy and by the internal and external stakeholders responsible for implementing the requirements in releases and the resulting release plans must be communicated clearly and in time to the stakeholders (Käkölä and Taalas 2008). Otherwise, it is difficult for the providers to commit resources for scheduling and synchronization of their production activities to meet the requirements specified in the release plans. For example, the scopes of software releases cannot be measured in terms of functional size (Forselius and Käkölä 2009; ISO 2006) if the requirements are not linked to the releases implementing them because functional size measurement is solely based on requirements.

A critical component of the governance model is that all requirements are (1) captured in a repository to ensure they are neither missed nor overlooked and (2) subjected to effective filtering in order to prevent information overload (Käkölä and Taalas 2008; Salo and Käkölä 2005; Van De Ven 1986). The remaining requirements are then refined, specified, estimated in terms of cost and resource implications, prioritized, and allocated to product releases and development units.

Flexible, scalable, and secure communication, coordination, and collaboration systems are needed to support the enactment of the governance model. Little theory-based guidance is available to help companies design and use such systems to achieve the goals of cycle time reduction and improved product quality, service delivery, and overall effectiveness.

Design theories, unlike other theories, support the achievement of goals (Hevner et al. 2004; Markus et al. 2002; Van Aken 2004; Walls et al. 1992, 2004; Walsh and Dickey 2003). Walls, Widmeyer, and El Sawy (1992, p. 37) argue that the information systems (IS) “field has now matured to the point where there is a need for theory development based on paradigms endemic to the area itself” and call for information system design theories to fulfill that need. An IS design theory is “a prescriptive theory based on theoretical underpinnings which says how a design process can be carried out in a way which is both effective and feasible” (ibid, p. 37). It prescribes both the design product and process aspects of a class of IS, that is, what are (1) the value propositions to be fulfilled by implementing an instance of the class, (2) meta-requirements describing the problem(s) to be solved by the class, (3) the meta-design prescribing the solution for the problem(s), and (4) applicable kernel theories from social and natural sciences for
understanding and/or solving the problem(s) shared across all products within the class, and how the products should be built (Walls et al. 1992, 2004).

Salo and Käkölä (2005) found that groupware-based requirements management systems (RMS) need to be designed and used to redesign and enact the earliest phases of product development effectively in multi-site, cross-functional organizations. They developed an IS design theory for the class of RMS in order to (1) facilitate endogenous theory development in the context of RMS research, (2) to help RMS designers build successful RM systems for creating, prioritizing, refining, storing, and managing requirements, and (3) to guide organizations in evaluating and deploying RMS. However, the benefits afforded by RMS were hampered if the RMS instances prescribed by the IS design theory were not integrated with systems used in the downstream phases in order to provide transparent end-to-end support throughout the product development lifecycle (Salo and Käkölä 2005). For example, customer representatives responsible for entering requirements could not use RMS instances to follow-up if and when the requirements would be implemented, lowering their interests in entering the requirements. The scope of the IS design theory should thus be broadened to design systems that support the lifecycle more comprehensively.

This research focuses on integrating requirements management with release management that is concerned with the identification, packaging, and delivery of product’s elements (ISO 2005). Releases can be realized through various organizational arrangements such as release projects in organizations structured around products (Rautiainen et al. 2002) and permanent release teams in organizations responsible for the long-term development and maintenance of strategic software and hardware assets. An illustrative scenario of release management practices for software product businesses is presented next. Each product identified during product line roadmapping is developed incrementally in release projects that follow the release plan and typically last from a few months to a year. Each release project is executed in a number of iterative cycles in which new features are added and product quality is improved so every cycle yields a tested and stable product version. During each cycle, feedback is collected from key stakeholders and used to plan and execute the next cycle(s). In addition to traditional project management activities, release management determines how many cycles and internal releases are needed (for testing purposes) in a release project, refines the requirements identified during product line roadmapping, allocates the requirements to the most appropriate cycles, and schedules the cycles. It thus ensures that internal and external releases meet the (specified and managed subset of) requirements identified and agreed upon in the front end of product development.

Based on our extensive industrial experience and review of academic literature, we hypothesize that the theoretical validity and practical relevance of the IS design theory for the class of RMS can be enhanced most effectively by extending the theory to provide integrated support for requirements and release management. The extant literature provides little guidance for designing and deploying integrated Requirements and Release Management Systems (RRMS). This article develops the design product theory (i.e., the product aspects of the IS design theory) for the class of RRMS (c.f., Walsh and Dickey 2003). It addresses the following research question:

• What are the necessary and sufficient properties for the class of RRMS in a multi-site and multi-partner environment?

The main contributions of the article are the meta-requirements of the design product theory and a meta-design that partially meets the meta-requirements. They are crystallized and validated based on (1) a case study in a global organization that deploys a RRMS instance organization wide for effective requirements and release management and (2) a literature review in the areas of requirements management, release management, and process integration.

The design product theory for the class of RRMS can be useful and generic only, if the two key concepts requirement management and release management and the scope of the theory are clearly defined. In this article, the two concepts refer to generic requirements and release related
actions, information entities, and roles, which can be adopted throughout the multi-site and multi-partner organization regardless of (1) the organizational design, (2) the product characteristics, and (3) the selected software and/or hardware development methods. As a result, the theory is independent of issues such as: how product lines and platforms are organized, which types of customers exist, and how much the efforts and times needed to develop different types of products vary. We have determined the scope of the theory by analyzing the RRMS instance in the case organization. The instance has been successfully used for years without any major design changes while the case organization has instituted numerous major organizational changes. The RRMS design invariance has been possible because the organization has scoped the design effectively by (1) determining the generic requirements and release related actions, information entities, and roles that always need to be supported by the RRMS instance and (2) interfacing the generic design to varying (A) project management practices and systems deployed to plan and monitor project resources and costs, (B) release planning practices deploying different heuristics, methods, and systems on a case by case basis to plan one or more releases based on only a limited set of instances of information entities (e.g., some features and a limited number of releases), and (C) product portfolio management practices (where the portfolio of products is agreed). In sum, requirements and release management processes and enabling RRMS instances, respectively, need to be interfaced with project management, release planning, and product portfolio management processes and enabling systems. For example, RRMS instances need to provide middle managers responsible for requirements and release management processes with good overviews of all the requirements, features, and releases. During release planning a subset of all the features and releases is planned using various heuristics, methods, and systems. The RRMS instances serve as sources of feature and release information. Release planning usually requires information about other issues (e.g., available resources) from other sources too. The resulting release plans are then made available through the instances, so all the teams working on the releases involved in a release plan can see the release schedule and the middle management can monitor the development efforts.

The design product theory has been created partially based on the experiences in the global case organization to ensure the meta-design is flexible and scalable, that is, the RRMS instances following the meta-design can handle large volumes of information entities and their relationships (provided that the necessary hardware resources are available) and enable diverse organizational designs, development methods, and types of products (including both hardware and software). However, we have made every effort to simplify the meta-design so even small and medium sized organizations can leverage it to implement RRMS solutions. The design process aspects of the IS design theory for the class of RRMS are not elaborated because our industrial experiences indicate that, at least in the context of the class of RRMS, it is most useful to prescribe the design product and let the designers adopt the development methods most suitable for implementing the design product in their socio-technical contexts.

2. DESCRIPTION OF THE CASE ORGANIZATION AND THE RESEARCH METHOD

A literature review was performed to develop preliminary meta-requirements and meta-design elements before the case study started. Later on, it was complemented by a review of commercially available requirements management and release management systems. The review was essential to reduce bias induced by the single case study and ensure generalizability of the meta-requirements and the meta-design to maximum possible extent (Yin 2003). Potential meta-design elements that according to the review were peculiar to the organization or its RRMS instance (hereafter, the RRMS) were eliminated. For example, the RRMS consisted of numerous information entities but many of them were peculiar to it because they reflected the same underlying concepts in different abstraction levels to facilitate the technical implementation of the RRMS.
The RRMS-enabled requirements and release management process had been institutionalized across the organization by the time the study was started. Business units ran product lines in which product programs produced product releases under the guidance of product roadmaps and release plans for customers in specific market segments. Product programs deployed software and hardware platform releases developed by internal platform units, inter-organizational consortiums, and external providers. Platform releases integrated hardware and/or software component releases that were developed internally or by partners or purchased off-the-shelf from external providers. Requirements were collected from markets, service providers, and other internal and external sources. Requirements triage was then conducted to eliminate requirements that did not warrant further actions. Acceptable requirements were allocated to appropriate units and component providers using the RRMS and iteratively refined into increasingly detailed product, platform, and/or component features that could be scheduled, implemented, and released by individual development teams. The process typically involved complex negotiations between stakeholders.

Product lines and platform units produced a diverse set of products that were in different stages of product life cycles, targeted different markets, and had different component vendors. They could thus vary their RRMS-enabled requirements and release management processes within boundaries agreed-upon at the organizational level. For example, product, software platform, and software component releases could be planned months in advance whereas features, releases, and release dependencies of hardware platforms could be planned even one or two years in advance depending on how accurately the providers could estimate their release plans and schedules for new hardware components. Product programs that needed to develop and deliver new products quickly could utilize the RRMS (1) to know which critical new features the platforms were planning to release and when and (2) to link their requirements to the features.

The RRMS was a proprietary Lotus Domino based application developed in the organization. An organization-wide Lotus Domino infrastructure had been institutionalized before the development started and the organization was competent in developing and deploying Domino applications. The RRMS had been productized, that is, one RRMS design and repository was used. Different organizational units had been closely involved in designing the RRMS from the beginning and become committed to using and further developing it. They considered the RRMS highly malleable to changing business needs partly because the organization controlled it and business units did not need to negotiate with external vendors when changes were needed.

Requirements, features, and releases were the key information entities to be managed throughout their lifecycles using the RRMS. Most importantly, the RRMS was used to continuously manage dependencies within and between the information entities and enable traceability between the entities and all the organizational units responsible for creating and updating the entities during their lifecycles. Requirement, feature, and release managers were responsible for managing the respective entities. Product line managers coordinated product programs and associated platforms. For example, product line requirement managers received requirements from sources such as product marketing. If the requirements could not be addressed within the product programs because they belonged to the scope of the product platforms, product line requirement managers allocated them to appropriate platforms and later on followed their progress using the RRMS. Release managers in the various levels used the RRMS to scope and schedule releases and to create and analyze dependencies between releases. Line managers used the RRMS to ensure the availability of appropriate resources when needed. The RRMS also enabled release teams to search and locate reusable assets quickly, substantially increasing perceived productivity.

However, while using the RRMS for achieving these objectives, some problems prevailed. Orchestration of complex parallel development programs, involving multiple internal and external development units across multiple sites, was challenging and coordination breakdowns sometimes occurred. For example, product releases depended on platform releases, which, in turn, could depend on other platform releases and/or component releases. Component providers ran their own
businesses and product platforms were not necessarily their most important customers. Component providers sometimes had to change their commitments to meet emerging business needs, resulting in issues such as delayed releases. The RRMS was critical for managing these dependencies and recovering from breakdowns. For example, if a release was unexpectedly delayed, the release manager updated the release schedule and the information about the delay was immediately available to all stakeholders. The schedules and scopes of the releases dependent on the delayed release could then be revised as necessary.

Indeed, the ability of the RRMS to support efficient routines and the recovery from complex breakdowns was a major reason for the successful organization-wide institutionalization of the RRMS-enabled requirements and release management process. After institutionalization, the use of the RRMS has been relatively stable with respect to measures such as the number of active users and the number of documents created. For example, personnel of all product programs have entered into the RRMS the requirements for the platforms the programs deploy. During a three month period of 2008, the RRMS was used in read-only mode by more than 6000 people and updated by more than 2000 people within the organization. In addition, around one hundred people used the RRMS in external service providers’ sites. In the end of 2008, it contained about 22000 active requirements, more than 50000 active features, and 23000 active releases. The documents that were no longer active had been automatically archived.

The RRMS was critically important for the top and middle management because it was the primary organization-wide information system containing real-time information about all the releases and associated requirements and responsible stakeholders. While there were many reasons contributing to the success of the RRMS within the organization, it is crucial, from the viewpoint of creating an IS design theory for the class of RRMS, that the RRMS enabled real-time transparency and management of product development within the entire organization. Other significant factors contributing to the success of the RRMS were: it imposed the appropriate amount of control on the people using it, its information model included mostly textual descriptions of information and minimized redundancy of information between the RRMS and related information systems, and it was easy for users to add information to it. In addition, since the RRMS focused on release management, it did not replace higher level product portfolio management and product line roadmapping systems, which require advanced algorithmic models (e.g., what if-analyses, cost and effort estimation, optimization of inter-dependent releases) and visualization techniques. But all the results (e.g., feature priorities, their allocations to releases, and release schedules) from using such systems had to be stored in the RRMS because (1) they guided the planning and implementation of releases and (2) the middle management based its product development decision making largely on the information available in the RRMS.

Software and hardware development processes and supporting information systems were not significantly affected either because they were beyond the scope of the RRMS. Software development efforts increasingly leveraged agile development practices. The RRMS thus could not impose unnecessarily stringent control mechanisms on them. Only the inputs to and the deliverables of management and software and hardware development processes and systems were dealt with by the RRMS. For example, if requirements in the RRMS needed specific product or organizational models to make them understandable to executives, managers of service providers, or other critical stakeholders, the models were crafted in appropriate modeling environments as necessary and hyperlinked or, sometimes, imported to the RRMS. Unified Modeling Language (UML) was deployed (Koivulahti-Ojala and Käkölä, 2010) but thorough UML modeling was conducted only when necessary because the organization managed tens of thousands of requirements and features through the RRMS. The priorities, schedules, and other information stored in the RRMS could be used to prioritize modeling efforts. The models were created, modified, and managed throughout their lifecycles using the modeling environments instead of the RRMS. The analysis of RRMS-enabled processes and interfacing information systems has thus helped us to scope the design
product theory for RRMS appropriately. More details concerning the organization and the RRMS and related information systems are beyond the scope of the paper.

Two of this paper’s authors, a doctoral student and a master’s student in information systems research, worked full time in the organization during a 6 month study period. The RRMS had become increasingly complex over the years when its designers had tried to meet the ongoing influx of new requirements. While its functionality had been documented well, the organization was keen to further develop it. A current state analysis of the RRMS was thus deemed necessary to better understand its limitations and possibilities. Major design changes were not realized during the study. Authors had access to all relevant information and could interact with all people who had been involved with the RRMS design. They observed the use of the RRMS, analyzed documentation, discussed informally with various stakeholders, and conducted six semi-structured interviews with middle-level managers who had been involved with the design and use of the RRMS for process improvement. After interviews were completed, interviewees were provided with interview transcripts and summaries. Interviewees reviewed the meta-requirements and proposed new ones that were added to the original set if all interviewees considered the proposed meta-requirements critical for the class of RRMS. The proposed meta-requirements and meta-design were used in the organization for further development of the RRMS. The authors have retained access to the organization and periodically observed the successful co-evolution of the RRMS and the organization until the time of completing this article.

Walls, Widmeyer, and El Sawy (1992, 2004) argue that kernel theories from natural and/or social sciences need to be identified and used to derive and govern meta-requirements of IS design theories. Interestingly, the people the authors interviewed or otherwise interacted with could not specify academic papers or theories influential during the RRMS design. They were experts with long organizational tenures and relied on theories-in-use (Argyris and Schon 1995) developed primarily through social interactions (c.f., (Perry et al. 1994)), experiences from earlier projects, and agile development practices instead of academic papers, kernel theories, or design theories. The authors thus became increasingly intrigued with how to build such a simple but scalable and effective IS design theory for the class of RRMS based on the case study that IS designers and managers in other organizations would be willing to use the theory in addition to trial and error mechanisms and long-reinforced theories-in-use. Kernel theories were determined to be out of the scope of the design product theory because no empirical evidence could be found about their usefulness in the context of the RRMS design.

3. META-REQUIREMENTS OF THE INFORMATION SYSTEMS DESIGN PRODUCT THEORY FOR THE CLASS OF RRMS

This section presents the meta-requirements for the design product theory for the class of RRMS. They are introduced by revising a framework of Salo and Käkölä (2005). The framework considered meta-requirements in relation to three categories of services that requirements management systems (RMS) have to offer: (1) communication, (2) control and (3) change. Communication refers to the ability of RMS to disseminate requirements information within an organization, including information about the rationale for RM and its relationships to the external environment. Support for control ensures that requirements are dealt with in accordance with approved principles and procedures. Support for change is needed because products, technologies and customers change and RMS must remain amenable to adjustments at all levels of RM activity.

The three categories are valid for the class of RRMS but two new ones are needed: (4) Platform-based product development and (5) Process integration. Platform is the physical implementation of the baseline entity that contains the common business requirements for all the derivative products the platform has been designed to support (c.f., (Bosch 2002; McGrath 2001; Meyer and Lopez 1995; Meyer and Selinger 1998; Pohl et al. 2005)). Market-driven product development
occurs on top of the platform. End-to-end process integration is necessary to ensure all requirement and release managers have all the high quality information they need available at the right level of detail when they need it. Process integration also helps ensure that all the release level information is available in a repository so the managers can analyze it and take decisions by means of other information systems (which are beyond the scope of the design product theory presented in this article) that aggregate the requirement, feature, and release level information and link it with product portfolios and roadmaps, release plans, and other information related to strategic business management. Table 1 classifies all meta-requirements.

(Insert “Table 1. A framework for categorizing the meta-requirements of the design product theory for the class of RRMS.” about here)

3.1 Meta-Requirements in Support of Communication

Prioritization and valuation of requirements and the allocation of requirements into releases

Requirements must be allocated into releases using requirement prioritization and valuation methods that enable the most crucial requirements to be implemented and released first (Carlshamre 2002; Ruhe and Saliu 2005). The methods are typically based on trade-off analysis between the economic values and implementation costs and resource constraints associated with the requirements (Carlshamre 2002, p. 140; Gilb 1998; Wesselius 2006). Moreover, all stakeholders do not have the same relative importance and each stakeholder may value each requirement very differently (Greer and Ruhe 2004; Momoh and Ruhe 2006; Ruhe and Saliu 2005).

According to interviews, customer involvement in valuation, prioritization and selection adds value to these processes: “Requirements can be prioritized to releases by communicating with customers and agreeing on what functionalities are wanted and on what timetable.” RRMS instances must provide the prioritization and valuation methods with the necessary requirement, feature, and release information and store the resulting valuations, priorities, and allocations.

Traceability

The purpose of requirement management is to achieve complete traceability from customers via the organizational departments to delivery (Grynberg and Goldin 2003, p.69). Traceability improves risk assessment, project scheduling, and change control (Halbleib 2004, p.12). All interviewees agreed that traceability from requirements elicitation to product delivery is critical for RRMS success. Interviewees also suggested other needs for traceability (e.g., linking components, errors, and use cases). But it is expensive to collect and manage traceability information (Kotonya and Sommerville 1998, p.129). Based on the analysis of the case organization, only the following traceability meta-requirements must always be implemented by RRMS instances:

- Ability to trace forward from requirement sources to requirements, from requirements to subsequent features and corresponding design elements and designs (Davis 1992), and from features to future releases defined by release plans.
- Ability to trace backward from releases to the features packaged in the releases and from the features to the requirements implemented by the features (Davis 1992).
- Ability to trace from requirements directly to design entities and backward (Kotonya and Sommerville 1998).
- Ability to trace requirements dependencies (Forsgren and Daugulis 1998; Kotonya and Sommerville 1998) and release dependencies.

Single capture of information

RRMS instances must be the single capture points for requirements, features, and releases, ensuring that there is no redundant and inconsistent requirement, feature, and release information
in the organization and that all requirements are handled appropriately in a single effective and transparent process reducing the risks of missing or forgetting requirements. RRMS instances should thus be (1) easy to use for occasional users in order to ensure that they enter the information, (2) interfaced to related systems such as requirements and architecture modeling and defect management environments, and (3) interfaced to partners’ systems to ensure that partners can create, update, and review information as necessary.

3.2 Meta-Requirements in Support of Control

Content ownership and accountability of experts
Requirements and release management activities should have appropriate owners who establish and reinforce appropriate norms for them (Grynberg and Goldin 2003, p.70). Content and process ownership can be enhanced by assigning roles with clearly defined responsibilities to persons. For example, a set of requirements can be allocated to a requirement manager while a particular release can be assigned to a release manager. The role definitions and assignments improve accountability, enable evaluations of people with respect to their role expectations (e.g., release managers may be evaluated based on the quality of releases they are responsible for), and can ensure that all agreed upon deliverables are delivered in time and meet the defined quality criteria. Role based management also facilitates organizational (e.g., people may continue their work in their old roles in a new organization) and individual level changes (e.g., a new person takes responsibility of a role) (Käkölä and Koota 1999; Käkölä and Taalas 2008). RRMS instances must thus meet this meta-requirement to facilitate personnel evaluation, process execution and improvement, and quality control.

Management and coordination
RRMS instances must enable the coordination of flows of requirements and the allocation of requirements to releases through managerial activities and decisions. For example, decisions need to be taken concerning the acceptance of particular requirements to the development process and the maturity levels of the requirements. To allocate requirements to specific releases and implementation teams and track their progress, RRMS instances must enable the assignment of different statuses to requirements. Interviewees commented that: “It should be possible to see from the tool who is responsible for certain parts of the process, who makes decisions concerning those parts, what the timetables are, and what kind of documentation should be available.”

Creating and sharing of metrics information
Measurement is an essential part of process management (Bundschuh and Dekkers 2008; Daskalantonakis 1992; Forselius and Käkölä 2009; Jones 2008; Käkölä and Koota 1999). Defined and measurable objectives are needed to evaluate the current status and develop the process. Metrics information enables comparison of process effectiveness across organizational units and similar releases over time (e.g., product line management). RRMS instances must provide a balanced set of process quality metrics (e.g., within each organizational unit) such as (1) the ratio of releases delivered in accordance with the planned release schedule to all delivered releases, (2) the ratio of releases delivered in accordance with the planned release scope (i.e., all requirements planned and assigned to the release have been realized) to all delivered releases, (3) the ratio of releases delivered in accordance with the planned work effort to all delivered releases, and (4) the ratio of cancelled releases to all releases.

Access rights and information security
Access rights and information security policies are crucial in high-technology companies. Products and platforms are typically designed and implemented in complex networks or consortia of companies where competitors are involved. RRMS instances must help ensure that partners do not access each other’s sensitive information. For example, multiple partners can co-operatively build a platform and use a RRMS instance to share information about it. At the same time, they
may build competing products on top of the platform and the RRMS instance must not leak any confidential information related to partners’ products and objectives.

### 3.3 Meta-Requirements in Support of Change

**Version management of requirement documents**

Versions of individual requirements and requirements specifications need to be controlled (Wiegers 2003, p.268). Change management and document version management processes must be in place to create and maintain requirement baselines. Requirement document version management is related to the general workflow management. However, one interviewee emphasized that it is most useful in requirement development phase, not in later phases when the documents are relatively stable. Change management and version management processes should be aligned, agreed upon, and enabled by the RRMS instances: “If the change is large, a new requirement can be created or the old one can be changed via the change management process. In practice, we could use version management for small revisions. But if the changes are too large, change management needs to be used.”

**Release re-planning**

When software development is market-driven (Regnell et al. 2001), release planning and requirement prioritization are parts of strategic product line roadmapping (Carlshamre 2002; Pohl et al. 2005). Especially in product programs the stakeholders need to continuously create and share knowledge to deal with uncertainty and turbulent market conditions. Release re-planning is needed when, for example, the product strategy is changed. Release re-planning may be related to planning the scope, role and timing of every product release specified in the product roadmap or to release management (i.e., re-planning the length of development, the scope of features, and the number of iteration cycles involved in a release) when, for example, several key developers leave unexpectedly (Rautiainen et al. 2002). Release managers may need to re-plan releases with the stakeholders on a weekly basis.

The RRMS instances prescribed by the RRMS design product theory must thus enable the relevant stakeholders to be involved in release re-planning at the right time and at low cost and provide the stakeholders with necessary requirement, feature, and release information for re-planning both individual releases and all releases involved in a roadmap. Release managers can then use appropriate release planning heuristics, methods, and systems to plan and re-plan the releases so the stakeholder priorities are best satisfied while the utilization of resources available to release development is maximized (Momoh and Ruhe 2006). To facilitate stakeholder involvement and reduce the burden of re-planning, the release planning systems should generate alternative release plans when the stakeholder priorities, available resources, and/or requirement dependencies change; let the release managers rank the plans and take decisions together with the stakeholders; and help store the resulting release information in the RRMS instances.

**Change management and impact analysis**

A clearly defined change management process is needed to estimate impacts of possible changes and to control the changes made to the requirements and releases (Kääkölä and Taalas 2008). Change management and impact analysis enable organizations to be aware of the influences of requirement changes on software and hardware components, test cases, resources, and market situation. Change management and impact analysis are tricky when RRMS-enabled requirements and release management processes have not been institutionalized across the geographically distributed sites, projects, and partners involved, because no one has adequate visibility into the detailed activities of the projects. However, the institutionalization of these processes may dramatically improve the situation, because the RRMS instances must (1) trace requirements to designs and development teams responsible for them, (2) trace requirements to product releases delivering them, (3) trace requirement dependencies and release dependencies, and (4) clarify to
all stakeholders designing a system in detail and in real-time which deliverables for that system are coming from which releases and when. For example, when the change of a major requirement for an important component is considered, a RRMS instance makes it relatively easy to see not only how much time and money the implementation of a new design would take based on the features affected, but also which other releases depend on those features. Relevant stakeholders can then be contacted to better understand the impacts and decide about the feasibility of the requirements change.

**Defining and maintaining the requirements baseline**

RRMS instances must support requirements baselining, that is, the freezing of current agreed upon requirements. When the baseline decision has been made, subsequent requirements are treated as change requests and compared to the baseline. If the requests are accepted, they will enter product development through the change management process (Käkölä and Taalas 2008). The requirements in a baseline that has been incorporated into a delivered release should not be changed in subsequent releases. After all, the delivered release can not be changed later on. Only a new release can be established to replace the delivered one as necessary. Changing the requirements associated with a release after the delivery of the release would yield inconsistency and jeopardize traceability. New requirements should thus be created for the subsequent releases and linked to the baselined requirements that have already been delivered.

3.4 Meta-Requirements for Platform Development

**Creation and reuse of reusable assets**

Platforms are strategic organizational assets designed to be reusable and afford common features and predefined variation mechanisms through which mass-customized products can be created quickly and cost effectively (Pohl *et al.* 2005). Platforms consist of assets such as requirements, design elements, components, and user interfaces. RRMS instances should provide a comprehensive information model to describe and document the assets adequately. They should not only document platform releases at the time of creation but also help link the releases to all associated requirements, features and releases. They should provide advanced search functionalities to help developers and other stakeholders to easily deploy the requirement, feature, and release information in order to search, retrieve, and leverage compatible assets in novel and possibly unforeseen ways. These issues are clarified by two interviewees:

“Requirements management involves the identification and management of baselines and products [for strategic reuse]. For example, we could see from the [RRMS] tool that this [requirements] baseline is good for our purposes and we just have to change it from there and there in order to have a right configuration for our needs.”

“One of the main purposes of the RRMS tool is to support reuse of both requirements and assets (modules, components)...Different knowledge search capabilities are essential.”

3.5 Meta-Requirements in Support of Process Integration

**Process transparency**

RRMS instances must help make integrated requirements and release management processes transparent, that is, the stakeholders involved should be able to understand the results of their RRMS-mediated actions (e.g., which product and platform releases would be affected and how, if a particular release was delayed by a month) and create knowledge for dealing with unexpected errors or coordination breakdowns as quickly and proactively as possible before expensive disruptions in routines and flaws in deliverables occur (Käkölä and Koota 1999; Käkölä and Taalas 2008). One interviewee stated: “Process transparency is especially important in decision making situations. Another important situation is when someone cannot implement, for example, a
requirement in a given timetable.”

Most developers in the case organization could access (in read-only mode) all documents in the centralized RRMS database, critically improving transparency. For example, release managers in platform units used the RRMS instance to follow the development and testing of the components their releases depended on and took corrective actions proactively, if the components were likely to be delayed. Transparency was also facilitated by using the RRMS instance to standardize the most critical information entities and the terms and forms used in them.

Providing high quality information

Organizations have to be able to base their requirements and release decisions on high quality (i.e., accurate, specific, relevant, reliable, timely, and accessible) information (O’Reilly 1982). Transparent RRMS-mediated processes and committed, skillful people are crucial to ensure high quality. RRMS instances should also help users to identify which pieces of information are the most critical in each phase of the process, for example, by sending reminders and highlighting the required fields of respective information entities in the different phases.

Providing information at the right and consistent level of detail

Appropriate and consistent granularity of information facilitates decision making and eliminates extraneous activities required to decompose or summarize information (Aubert 2003; Salo and Kääkölä 2005). The right level of detail depends on the situation. For example, highly mature requirements and release management processes can leverage much more detailed (quantitative) information than immature processes. RRMS instances must incorporate and represent requirement- and release-related information in consistent granularity levels (e.g., system requirements, subsystem requirements, and functional requirements) that are useful for different process contexts and roles. For example, RRMS instances may be used to generate requirements specifications for product releases. Each requirement in the generated specification should be testable by means of a small number of tests. If some requirements are not testable, they are probably represented on higher granularity levels than the testable ones and there is no consistent level of detail in the specification.

4. A META-DESIGN OF THE INFORMATION SYSTEMS DESIGN PRODUCT THEORY FOR THE CLASS OF RRMS

This section first outlines a generic meta-design for the class of RRMS based on the analyses of interview transcripts, the RRMS instance in the case organization, and the literature review. It covers most meta-requirements specified in the previous section. The section concludes by explaining the linkages between the meta-requirements and the meta-design to validate the meta-design and to justify its scope.

4.1 Information Model for Integrated Requirement Management and Release Management Process

To design an effective requirement management and release management process, the information model for the process must be defined. We have synthesized a simple but scalable model based on the literature review and a detailed examination of the RRMS instance. The experts of the case organization have reviewed and accepted the model. It consists of four entities used in the integrated process and links between and within the entities to enable traceability, hierarchical composition (i.e., each entity can consist of any number of the same type of entities), and appropriate granularity of information (Figure 1): Customer Requirement, Requirement, Feature and Release.

(Insert “Figure 1. Information model of the meta-design of the design product theory for the
Customer Requirement is used to model requirements from the external environment. Internal and external requirements are separated to meet the meta-requirements related to platform-based product development and information security. For example, customer requirements are business critical and can provide competitive advantage by enabling organizations to focus on implementing the differentiating and high-value adding requirements. Access rights for them and for internal requirements have to be defined and enacted differently.

Requirement is used for internal requirements developed by product creation processes within an organization or a network of companies collaborating to create a joint product and/or platform. Requirement can thus be used for platform requirements related to the platform offering. In the platform context, Customer Requirements can be used as the basis for developing derivative products on top of the platform. Separation of Customer Requirement and Requirement also facilitates change management. Requirements can be changed only through negotiations with their originators. Negotiations with external requirement suppliers are often more challenging than with suppliers of internal requirements, especially when the external suppliers contribute to funding the development efforts.

Features denote intended behaviors or properties of software-intensive systems. They are usually created and managed as hierarchical feature structures solving the problems Requirements identify (Zhang et al. 2006). The solutions may reflect, for example, business processes, organizational structures, or product architectures. Feature is the largest entity in the information model containing the technical specification, workflow planning, and implementation. By using Feature as a basis of implementation and technical specification, Requirements become more manageable and there are clear traceability links to the origins of Features and to implementation phases, that is, specifications, responsible teams, and realized pieces of software code.

Product roadmaps and the associated release plans often trigger the development of new releases and provide them with high-level requirements. For every new release, a release manager is typically assigned and a Release instance is created to plan, implement, and document the resulting release. If a previous release exists, its feature set may serve as the starting point. New functional and non-functional requirements to be delivered in the release are identified and refined and Requirement and Feature instances are linked to Release instances as necessary. RRMS instances can be used to locate, reuse, and modify existing Requirements and Features whenever possible, implement new Features as necessary, and group the implemented Features into the Release instance documenting the release. Releases can also be organized hierarchically into, for example, platform and product releases.

Customer Requirements are linked to one or more Requirements, which, in turn, are linked to one or more Features. Highest level Requirements are typically large scale system-level definitions of business problems. Dividing them into sub-problems (i.e., lower level requirements) which are linked to Features enables more accurate project cost, schedule, and effort estimation and better workflow management. Features describe implementable partial solutions to the business problems. The highest level releases are comprehensive, valuable solutions consisting of Releases and Features whereas lower level releases can, for example, deal with components.

### 4.2 Generic Structures of Entities

This section introduces generic structures, classes, and attributes of the entities presented in the information model. According to the design product theory, RRMS instances prescribed by the theory should include at least these entities, structures, classes, and attributes to be effective.
**Requirement and Customer Requirement**

Table 2 presents the generic structure of Requirement and Customer Requirement by revising and elaborating on the work of Salo & Käkölä (2005). Next, each class within the structure is presented.

(Insert “Table 2. Generic structure of Requirement and Customer Requirement.” about here)

*Description* class describes the intent of and justification for a Requirement and a Customer Requirement. *Version* attribute indicates the version number of the document. *Name* and *ID* attributes are used for identification and traceability.

*Origin* class describes where the Requirement comes from and when. For Customer Requirements the sources are external organizations. For Requirements, sources are Customer Requirements and internal organizational units.

*Categorization* class describes the parts (i.e., *platform*, *product*, and *responsible person*) of the product and the development organization managing the Requirement or Customer Requirement. Platform works as the base architecture for derivative products. Requirements and Customer Requirements will be linked to appropriate platforms and final products via Features and Releases. Responsible persons update Requirements and Customer Requirements as necessary.

*Analysis* class is used to probe the implications of the Requirement. *Priority* and *customer value* can be handled as one attribute, but organizations needing sophisticated valuation processes should divide them in two. *Customer need* attribute is used to describe the detailed business case, which the Requirement or Customer Requirement is trying to solve. If there is a firm deadline by which the Requirement needs to be implemented and released for use of customer(s) (e.g., in their products), the deadline must be made explicit through *deadline* attribute. The *total cost* and *required work effort* need to be estimated (Bundschuh and Dekkers 2008; Jones 2008) to determine whether the Requirement is feasible from economic, personnel, and schedule viewpoints. *Risks* associated with the (Customer) Requirement need to be assessed. *Status* attribute models the requirements life-cycle. Examples of requirement statuses include: New – Categorized – Analyzed – For Review – Approved / Rejected / Postponed (Salo and Käkölä 2005).

*Workflow* class describes what should be done next to Requirement or Customer Requirement and by whom. Customer Requirements and Requirements are allocated, respectively, to Requirements and Features and responsible persons are assigned.

*History* class is used to provide information about all prior changes and editors of requirements documents (Salo and Käkölä 2005). It enables traceability and the development of organizational memory that is especially useful when routines break down unexpectedly and the reasons for the breakdowns must be found and eliminated to continue the effective execution of routines (Käkölä and Koota 1999).

**Feature**

Each class within the Feature-entity (Table 3) is presented in this section.

(Insert “Table 3. Generic structure of Feature.” about here)

*Description* class tells the intent of and justification for the Feature. *Origin class* indicates the author, date of creation, and Requirements, if any, from which the Feature is allocated.

*Categorization* class links Features to products and/or product platforms and identifies the person having the feature responsibility. Because Feature is an entity for managing detailed implementation, it has a *traceability links* attribute containing links to technical specifications, documentations and code. Features tend to have complex dependencies (Zhang et al. 2006). For
example, a Feature may be incorporated into a Release only if its parent Feature is also included because only the two Features together fulfill a particular Requirement. A dependency link between these two Features thus provides valuable information for decision makers.

**Analysis** class contains most attributes that Requirement and Customer Requirement have, with the exception of *customer value*-attribute used to decide whether Requirements or Customer Requirements should be implemented or not. The *required work effort* needs to be estimated to assess implementation costs and help teams in their work allocation and scheduling. It also useful to determine *Realized work effort* when the feature is ready for release because estimation practices can be systematically improved by comparing original work effort estimates to actually realized work efforts.

**Workflow** class consists of detailed *task descriptions* together with traceability links to provide the guidelines for implementation work and to enable organizational learning through, for example, post-mortem analysis (i.e., what was planned vs. realized). Before starting the work, Features are assigned to responsible teams or persons. **History** class is used to provide information about all prior changes and editors of feature documents.

**Release**

Classes within the Release-entity (Table 4) that need elaboration are explained in this section.

(Insert “Table 4. Generic structure of Release.” about here)

**Description** class describes what the Release is about. For example, the Release may fix some specific quality problems of the previous Release without providing new functionality. In **Origin** class, *source features* and *source releases* attributes indicate which Features and Releases are included in a Release. In **Categorization** class, a Release is related to specific product *platforms* and/or *products* and has a *responsible person*.

**Analysis** describes the lifecycle of a Release through *status* attribute. Statuses include: Planned – Ready to be Released (i.e., all Features belonging to the Release have been implemented, tested, and found to be stable and all lower level Releases the Release depends on have been released) – Released. It should be noted that a Release can also be canceled but only if no other Release is dependent on it. Dependent Releases must thus be canceled or made independent from the Release to be canceled. The allocations of Features to a canceled Release must be removed in respective Feature-entities.

For every source Feature, the *required work effort* should have been estimated when the source Features were analyzed. When a Release has been delivered, it is useful to assess and document (1) the work effort that was necessary to realize each Feature, (2) the total work effort realized to implement the Release, and (3) the reasons for any major discrepancies between estimated and realized efforts. Impact analysis practices and the accuracy of release schedules can then be improved in the future.

Because Releases constitute manageable and releasable entities, the only **Workflow** related attributes tell the *planned* and *actual release dates* and provide links to all Releases depending on this Release. The release managers of *dependent releases* can thus be notified, for example, when the Release has been delivered or when it will be unexpectedly delayed. This information together with the information stored in *realized work effort* attribute and **History** class is adequate for organizational learning and performance monitoring.
4.3 Validating and Scoping the Meta-Design by Analyzing how it Meets the Meta-Requirements

This section analyzes how the meta-design satisfies the meta-requirements because “a design artifact is complete and effective when it satisfies the requirements and constraints of the problem it was meant to solve” (Hevner et al. 2004, p. 85).

Prioritization and valuation of requirements and the allocation of requirements into releases

Prioritization and valuation of requirements is enabled by the entities Requirement and Customer Requirement. Their attributes *priority* and *customer value* are used to store and access the prioritization and valuation information in the RRMS instances. The prioritization and valuation methods are not included in the meta-design for two reasons. First, the literature provides hardly any methods that are generalizable and scalable to meet the needs of complex industrial environments where multiple interdependent releases of interdependent products and platforms are planned simultaneously (Lehtola and Kauppinen 2006). Second, the product programs of the case organization used different prioritization methods and tools because the programs differed in size, duration, and product maturity. Organizations must decide which prioritization and valuation methods they wish to use. The meta-design ensures that RRMS instances can provide the methods with most if not all the necessary information and store and share the results organization-wide.

Allocation of requirements into releases is enabled transitively through features, that is, requirements and customer requirements are allocated to features, which are linked to releases. Releases provide implemented functionality and are thus linked to features directly.

Traceability

The information model enables bi-directional traceability between entities through *Origin* and *Workflow* classes. In Customer Requirement and Requirement, *source* attribute is used for backward traceability and *allocation to features* attribute enables forward traceability to features. *Source requirement* and *assignment to release* attributes of Feature enable, respectively, backward and forward traceability from features. *Traceability links* attribute enables the traceability from Features to implementation specific documentation and software code.

Single capture of information

Based on the analysis of the RRMS instance in the case organization, the information model is comprehensive enough so the RRMS instances prescribed by the meta-design can be the single capture points for requirements, features, and releases in organizations.

Content ownership and accountability of experts

Content ownership and accountability are determined through the *responsible person* attribute of *Categorization* class. For example, each release has to specify who is responsible for planning, which features are released in which release. The meta-design does not detail the metrics that could be used for measuring performance. However, it can be used as a basis for sophisticated measurement systems.

Management and coordination

The meta-design supports management and coordination across multiple, interdependent product, platform, and component releases, for example, by explicating the schedules imposed on various entities, the products and organizational units the entities are related to, and the workflows the entities are subjected to.

Creating and sharing of metrics information

The meta-design affords a balanced set of process quality metrics. Releases contain information about planned and actual release dates, making it easy to measure (e.g., within an organizational unit) issues such as what the ratio of releases delivered in accordance with the planned release schedule to all delivered releases is. Status information is readily available, making it easy to see,
for example, what the ratio of cancelled releases to all releases is. Detailed work effort information can be collected, making it possible to determine, for example, what the ratio of releases delivered in accordance with the estimated work effort to all delivered releases is. It is also possible to measure the volatility of release scopes because for each release the associated Release instance documents (primarily through the source feature attribute) the evolution of the feature set associated with the release from the time the release is planned to the time the release is delivered or cancelled.

**Version management**

*Description* and *History* classes enable the version management of requirements, releases, and other entities by, respectively, numbering versions and showing the actors involved with each version and the actions taken.

**Release re-planning**

The individuals responsible for particular features and releases have to re-plan the releases when some unexpected (coordination) breakdowns occur (e.g., Features belonging to a Release cannot be released because their implementations are unexpectedly delayed; a platform cannot be released because it depends on a component Release that has been canceled; a competitor releases a competitive product unexpectedly and fast response is necessary). Bidirectional traceability links between Features and Releases (stored in *assignment to release* and *source features* attributes) and between Releases (stored in *source releases* and *dependent releases* attributes) facilitate the implementation of the meta-requirement.

When there are numerous interdependencies between releases, between features, and between features and releases, the appropriate data stored in a RRMS instance can be transferred into a release re-planning and optimization system (c.f., (Carlshamre 2002; Ruhe and Salii 2005)) for analysis and creation of a new release plan. Prescribing the features of such systems is beyond the scope of the design product theory for the class of RRMS presented in this article because the systems are algorithmically complex, enable cost, effort and schedule estimation based on historical data (Forselius and Käkölä 2009), and operate on a higher level of analysis than the RRMS instances where strategic and operational decisions (e.g., about the common features within and across the product lines) are taken based on information in the RRMS instance and other systems. Future research is needed to study whether it is beneficial and feasible to extend the IS design product theory for the class of RRMS so it covers such classes of systems.

**Change management and impact analysis**

Change management is facilitated by the *History* class in all entities. Change requests can be considered as normal (Customer) Requirements, analyzed, linked to the respective existing Requirements in the RRMS instance that are within the scope of the change, and implemented and released by following the integrated requirements and release management process. Impact analysis is enabled by *Categorization* and *Analysis* classes. *Platform* and *product* attributes show the organizational entities affected by each Requirement and Release. *Customer value* and *required work effort* attributes are used to decide the feasibility of implementing a (Customer) Requirement.

**Creation and reuse of reusable assets**

The hierarchical composition of Requirement, Feature, and Release entities provided by the information model enables the comprehensive documentation of product and platform releases and all associated assets. This information together with bidirectional traceability links between the entities help organizations analyze their asset base and establish, find and use reusable assets.
5. CONCLUSIONS AND FUTURE RESEARCH

This article has focused on the RRMS-enabled multi-site and platform-based product development. It has synthesized the meta-requirements and a meta-design of a comprehensive information systems design product theory for the class of RRMS to help practitioners in both small and large software and software-intensive organizations implement and evaluate scalable RRMS solutions. For example, organizations can use the theory (1) to ensure the key roles (e.g., requirement manager and release manager) are established and adequately staffed in all release efforts and made responsible for the information entities specified by the meta-design of the theory, (2) to develop requirements management and release management tools and/or select and acquire commercial off-the-shelf tools, and (3) to integrate the tools into RRMS instances that meet the meta-requirements and support the information model specified in the meta-design. The validity of the theory has been enhanced by using methods such as the analysis of the RRMS instance in a case organization and by explicating the meta-requirements met by the meta-design. The meta-design is scalable because its most essential elements have been abstracted from the RRMS instance that, at the time of writing this paper, has thousands of users and manages tens of thousands of documents in the case organization alone.

Due to space limitations, the design product effectiveness hypotheses of the theory (clarifying the expected organizational benefits from using a RRMS instance (i.e., the design product) derived from the class of RRMS) are beyond the scope of this article. The hypotheses are needed for the empirical validation and possible revision of the theory in future research. The deployment of RRMS instances can be hypothesized: to reduce resources needed in product development (e.g., through strategic reuse of product platforms and components); shorten time-to-market (e.g., through reuse and by ensuring right information is available at the right time for the right people); improve customer satisfaction (e.g., by ensuring that requirements are transformed efficiently to product features); and improve process and product quality (e.g., by improving the synchronization of work across multiple sites, projects, and partners; minimizing the number of errors during development; and easing up error tracking). Future research is necessary to assess, extend, and elaborate these design product effectiveness hypotheses.

Future research is also necessary to devise extensions to the design product theory such as improved RRMS support for (1) strategic product line roadmapping and release planning processes (Pohl et al. 2005; Rautiainen et al. 2002) that take a long-term view and thus steer release management and (2) finding and reusing implementation level assets that meet the needs of releases. The first extension would require future research concerning how Release entities can be used to enable general managers, product managers, and release managers to understand even better which individual product and platform releases are linked with which release plans and product roadmaps and why they are linked. The second extension would require at least the inclusion of a Component entity in the information model of the meta-design. Our preliminary industrial experiences show that Component is useful especially if it describes how and when Component instances have been tested.

The single case study methodology may not provide a sound basis for generalization (Yin 2003). Therefore, new case studies and action research projects are necessary to make the design product theory more credible for IS designers and researchers. Design science research leveraging the methods of action research (Hevner et al. 2004) helps examine the applicability of the theory by finding out to what extent organizations that want to acquire or design and implement RRMS systems can utilize the theory for those purposes. The theory can then be revised as necessary.
ACKNOWLEDGEMENTS

The authors wish to thank Kirk Kandt from Jet Propulsion Laboratory, Rick Kazman from University of Hawaii, and the three anonymous reviewers for their excellent comments on the earlier versions of this paper. The authors also gratefully acknowledge the support of the numerous employees of the case organization. Dr. Timo Käkölä is grateful to the Academy of Finland, whose generous grant facilitated the writing of this paper while he served as the Senior Research Scholar of the Academy of Finland in Claremont Graduate University, California, USA.

REFERENCES


<table>
<thead>
<tr>
<th>Communication</th>
<th>Control</th>
<th>Change</th>
<th>Platform development</th>
<th>Process Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Prioritization and valuation of requirements and the allocation of requirements into releases</td>
<td>• Content ownership and accountability</td>
<td>• Version management of requirement documents</td>
<td>• Creation and reuse of reusable assets</td>
<td>• Process transparency</td>
</tr>
<tr>
<td>• Traceability</td>
<td>• Management and coordination</td>
<td>• Release re-planning</td>
<td>• Providing high quality information</td>
<td>• Providing information at the right and consistent level of detail</td>
</tr>
<tr>
<td>• Single capture of information</td>
<td>• Creating and sharing of metrics information</td>
<td>• Change management and impact analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Access rights and information security</td>
<td>• Defining and maintaining the requirements baseline</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. A framework for categorizing the meta-requirements of the design product theory for the class of RRMS.

<table>
<thead>
<tr>
<th>Class</th>
<th>Question</th>
<th>Attributes</th>
</tr>
</thead>
</table>
| **Description** | What is the requirement about? | Name  
ID  
Description  
Rationale  
Version |
| **Origin** | Where does the requirement come from? | Author  
Source  
Date of creation |
| **Categorization** | What parts of the product and the development organization is the requirement related to? | Platform  
Product  
Responsible Person |
| **Analysis** | What are the implications of the requirement? | Status  
Priority  
Customer need  
Deadline for the customer need  
Customer value  
Risks  
Required work effort  
Total cost |
| **Workflow** | What should be done to this requirement next? By whom? | Allocation to Requirements/Features  
Assignment to Requirement/Feature responsible |
| **History** | What has been done to the requirement? When? | Information about all prior edits, editors, and changes |

Table 2. Generic structure of Requirement and Customer Requirement.
<table>
<thead>
<tr>
<th>Class</th>
<th>Question</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>What is the feature about?</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rationale</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Version</td>
</tr>
<tr>
<td><strong>Origin</strong></td>
<td>Where does the feature come from?</td>
<td>Author</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source Requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date of creation</td>
</tr>
<tr>
<td><strong>Categorization</strong></td>
<td>What parts of the product and the development organization is the feature related to?</td>
<td>Platform</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Responsible Person</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Traceability links (e.g., documentation, code)</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>What are the implications of the feature?</td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priority</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Customer need</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Risks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required work effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Realized work effort</td>
</tr>
<tr>
<td><strong>Workflow</strong></td>
<td>What should be done to this feature next? By whom?</td>
<td>Task description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assignment to teams/persons</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assignment to Release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date when Feature is ready for Release</td>
</tr>
<tr>
<td><strong>History</strong></td>
<td>What has been done to the feature? When?</td>
<td>Information about all prior edits, editors, and changes</td>
</tr>
</tbody>
</table>

Table 3. Generic structure of Feature.

<table>
<thead>
<tr>
<th>Class</th>
<th>Question</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>What is the release about?</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Version</td>
</tr>
<tr>
<td><strong>Origin</strong></td>
<td>Where does the release come from?</td>
<td>Author</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source Features</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source Releases</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Date of creation</td>
</tr>
<tr>
<td><strong>Categorization</strong></td>
<td>What parts of the product and the development organization is the release related to?</td>
<td>Platform</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Responsible Person</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>What are the implications of the release?</td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priority</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required work effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Realized work effort</td>
</tr>
<tr>
<td><strong>Workflow</strong></td>
<td>What should be done to this release next? By whom?</td>
<td>Planned release date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Actual release date</td>
</tr>
<tr>
<td></td>
<td>What should be done to dependent releases?</td>
<td>Dependent Releases</td>
</tr>
<tr>
<td><strong>History</strong></td>
<td>What has been done to the release? When?</td>
<td>Information about all prior edits, editors, and changes</td>
</tr>
</tbody>
</table>

Table 4. Generic structure of Release.
Figure 1. Information model of the meta-design of the design product theory for the class of RRMS. Note: The links that go back to themselves in each entity represent parent-child relationships.