Concepts for Incremental Method Evolution: Empirical Exploration and Validation in Requirements Management

Inge van de Weerd, Sjaak Brinkkemper, and Johan Versendaal

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
{i.vandeweerd, s.brinkkemper, j.versendaal}@cs.uu.nl

Abstract. Product software companies are confronted with performance failures in their processes for which standard theories on situational method engineering need to be revisited. By developing a knowledge infrastructure, we support these companies with their method evolution by increasing the maturity of their processes incrementally. We first identify and formalize general method increments that are found in an exploratory case study. Then, we formalize common process needs, by developing a root-cause map for software product management and by identifying the root causes and process alternatives that are related to them. We validate the formalized method increments, and process needs by applying them to an extensive case study conducted at Infor Global Solutions. The results show that the formalized method increment types cover all increments that were found in the exploratory case study, and that the root-cause map is a useful technique to model the root causes encountered in product software companies.

Keywords: method engineering, meta-modeling, software process improvement, incremental method evolution, root cause analysis.

1 Introduction: Incremental Method Evolution

Many organizations are struggling with the evolution of their information systems development methods [6]. To control this, several software process improvement methods have been proposed (e.g. [8] [14]), which can be implemented in different ways and which are evolutionary in nature. In our research, we focus on such an evolutionary approach instead of a mere revolutionary approach for several reasons: a) it is a fundamental way to reduce risk on complex improvement projects [10]; and b) we observe in practice that this is the natural way for method evolution [26] [27].

This evolutionary approach has been subject of research in various scientific studies: methods have been developed to measure and to increase a company's maturity [8] [14]; studies have been carried out to find the best approach to instigate a process improvement [17] [22]; and research has been done on the key success factors that influence software process improvement [15]. However, in 2002, it was estimated that still 70% of software process improvement projects failed [21].

J. Krogstie, A.L. Opdahl, and G. Sindre (Eds.): CAiSE 2007, LNCS 4495, pp. 469–484, 2007.

In this work, we choose to take the existing research on software process improvement a step further. Our aim is to develop a knowledge infrastructure that supports product software (PS) companies that build off-the-shelf software products for a market [28] in the *incremental evolution* of their methods, by dealing with their process needs and guiding them to higher maturity levels. We keep the increments local (i.e. one process at a time is changed) and small (in comparison to existing incremental approaches with larger increments like CMM [14] and SPICE [8]).

In the next section, we describe our research approach, introduce process-deliverable diagrams for modeling methods, and describe the context of this research. In section 3, we define and formalize the process needs. In section 4, we validate the formalized method increments by carrying out a case study at Infor Global Solutions. Finally, in section 5, we describe our conclusions and future research.

2 Research Approach

Our aim is to support PS companies in their method evolution, by improving parts, or fragments, of their existing methods in an automated way. Method engineering [3] has been used successfully to engineer (parts of) methods for specific situations [1] [16]; to serve as an instrument in software process improvement [27]; and to use as an approach to manage evolutionary method development by integrating formal meta-models with an informal method rationale [19].

For scoping reasons we limit our research to the software product management domain of PS companies, covering requirements management, release planning, product roadmapping, and portfolio management. In industry, software product management is a clearly defined function, but in science research is fragmented [24].

2.1 Research Question and Methodology Outline

We define the following research question:

"How can product software companies improve their software product management methods in an evolutionary way, using method fragment increments?"

We address this question by applying method engineering theory. Incremental method engineering has been subject to research by e.g. [10] and [23]. However, a definition of method increment seems not to be available. Therefore, we define a *method increment* as: a method adaptation, in order to improve the overall performance of a method. Note that adaptation can mean insertion, editing or removal of method fragments.

Actual method increments in industry are explored in an explorative case study at a HRM software vendor (from now on: HRM case study), in order to derive a list of method increment *types* that occur during the evolution. By formalizing and generalizing the increments, we model incremental evolution of a product software company's processes. The formalized increments are then validated in an ERP case study. Using Root Cause Analysis (RCA, [18]) techniques we determine an initial set of root causes of process needs that PS companies may encounter in the software product management domain. RCA has been applied to process improvement and incident prevention in software and non-software industries; see for example [11].

With respect to the HRM case study we determine an initial set of root causes that may lead to process improvement alternatives. This set and our RCA application are also validated in the ERP case study.

2.2 Meta-modeling with Process-Deliverable Diagrams

For the analysis of method increments, we use process-deliverable diagrams (PDDs), a meta-modeling technique that is based on UML activity diagrams and UML class diagrams [25]. The resulting PDDs model the processes on the left-hand side and deliverables on the right-hand side (see Figure 1). Examples of PDDs can be found in Figure 5 and 6.

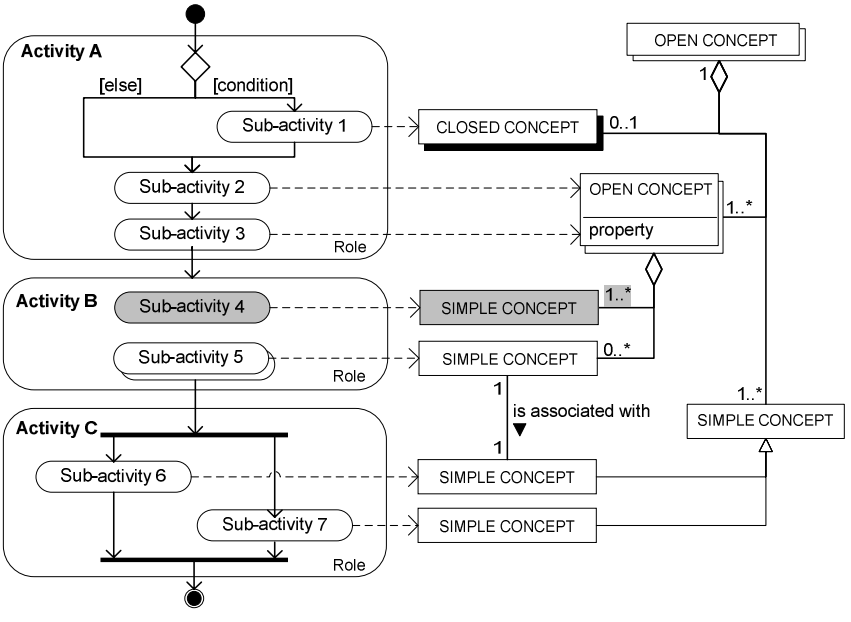

Fig. 1. Process-data diagram

We follow standard UML [13] conventions, but some minor adjustments have been made for modeling development processes. Firstly, deliverables can be **simple** or **compound**. Simple deliverables do not contain any sub deliverables and are visualized with a rectangle. Compound deliverables contain one or more sub deliverables. Compound deliverables can be **open**, visualized with an open shadow, to indicate that it contains sub deliverables. The sub deliverables can be shown in the same diagram, by using aggregation, or in another diagram (for example for space saving). **Closed** compound deliverables, visualized with a closed shadow, indicate that that sub deliverables exist, but are not relevant in this context. Similarly, open en closed activities are used in the diagram. The dotted arrows indicate which deliverables result from the activities. More details on this modeling technique can be found in [25] and [27].

The PDD, visualized in Figure 1, is called a **snapshot**, a model of the process as it was at a certain moment in time [27]. The evolution of a method over time exists of a number of these snapshots. By comparing snapshots, method increments can be analyzed. In Figure 1, we marked sub activity 4 and its corresponding concept. We use this notation to show the method increment of this snapshot compared to a snapshot earlier in time.

2.3 A Knowledge Infrastructure for Incremental Method Evolution

The context in which we want to support PS companies with the incremental evolution of their processes is described in [27], where we propose the Product Software Knowledge Infrastructure (PSKI, [27]). Several knowledge repositories for software development methods have been proposed and developed (e.g. the OPEN Process Framework [9]). However, the PSKI is not only a knowledge repository, but it also analyzes the process need of a company in order to deliver meaningful advice. In Figure 2, the PSKI is illustrated as well as the PS company that interacts with it. The PSKI contains a method base, in which method fragments, situational factors, maturity capabilities and assembly rules are stored.

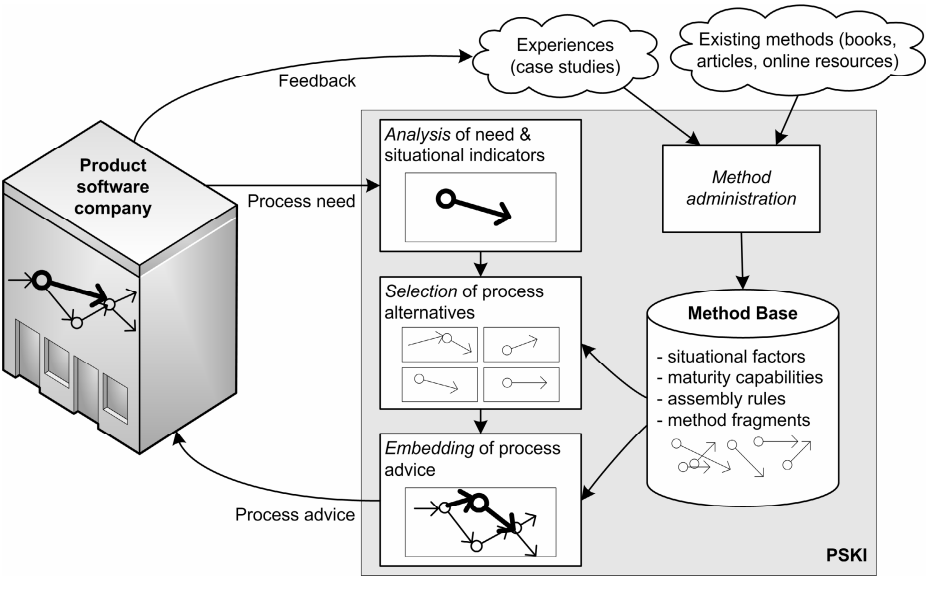

Fig. 2. Product Software Knowledge Infrastructure

Analysis of need and situational indicators
The first step is the analysis of the process need and situational indicators. The process need is analyzed using Root Cause Analysis, (RCA). Through RCA the root causes of a process need are determined using the following sequence, see also [11] and [18]: 1) which *process difficulties* actually occur; 2) what are the so-called *causal factors* of the difficulties; and 3) what are the actual *root causes* per causal factor,

using a root cause map designed for PS companies. We define a root cause as (one of) the underlying reasons of a process need, solving one or more causal factors, and relating to one or more actors, activities and deliverable concepts (referring to figure 1). Situational indicators contain information about the process and the company. Examples are company size, development platform and sector.

Selection of process alternatives
Once the root causes are known for a process need, directions for software process improvement can be sought taking into account situational factors. For this, the method base is used. Links between maturity capabilities and root causes are available in the method base in order to identify possible process improvement alternatives. Examples of maturity capabilities are listed in [26]. We define a process alternative as a method fragment of a particular maturity capacity that settles one or multiple root causes of the process need.

Embedding of process advice
The last step is embedding the process advice in the company's existing processes. A process advice, which contains a process description, templates and examples, is sent back to the company. The person responsible for process improvement at the company will then start the organizational deployment of the process advice. This roll-out process also includes the insertion of the increment in the existing processes.

3 Definition and Formalization

This section defines and formalizes method increments and the development problems that lead to these increments. The rationale for this formalization is twofold: First, we use it to analyze the method increments that we found in the HRM case study (see Section 4.4). Secondly, the formalization is used as a first step to develop a formal structure for the method base of the PSKI in which method fragments can be edited. Firstly, we define method evolution, snapshot and method increments. Then, based on the meta-meta model of PDDs we present a list of all possible increment types with some method fragment insertion rules. Thirdly, we analyze problems that lead to the method increments and develop a root cause map for software product management (RCM for SPM).

3.1 Definitions of Incremental Method Evolution

As the PDD technique is based on UML, we can utilize the available formalizations in the literature. There appears to be two kinds of formalizations: those based on the formal language **Z**, e.g. [5] and [20] and those using first order predicate logic, e.g. [2], of which we chose the latter due to its concise presentation.

We start the formalization with the assumption that there is some kind of universe of consistent methods, called $\mathbb{M}$. We assume furthermore, that these methods in $\mathbb{M}$ can be executed by project members, i.e. the method descriptions are available, complete, and consistent. The evolution of the method in a particular company can then be seen

as a series of methods $m_1, m_2, \ldots, m_n \in \mathbb{M}$. For reasoning about time we introduce the time dimension $\mathbb{T}$. The set of method fragments is called $\mathbb{F}$.

Definition 3.1. The mapping $\text{method}: \mathbb{T} \rightarrow \mathbb{M}$, where $m = \text{method}(t)$ means that the method $m \in \mathbb{M}$ is the valid method at time t.

The methods change in the course of time, and this allows us to define the notion of snapshot of a method.

Definition 3.2. A *method adaptation time* is a point of time where the method has been adapted. Let $\mathcal{T}$ be the set of method adaptation times, i.e. $\mathcal{T} = \{t_1, t_2, t_3, \ldots, t_n\}$ such that $\forall i\ \forall t$: $\text{method}(t_i) = \text{method}(t) \neq \text{method}(t_{i+1})$.

Definition 3.3. A *method snapshot* is a method $m \in \mathbb{M}$ that was valid at a particular time, i.e. $\exists t_i \in \mathcal{T}$; $m = \text{method}(t_i)$.

Definition 3.4. A *method evolution* is a set $\mathbb{S} \subseteq \mathbb{M}$ consisting of the method snapshots, i.e. $\mathbb{S} = \{\text{method}(t_i) | \ t_i \in \check{\mathbb{T}}\}$. So $\mathbb{S}$ is the set of methods that have been valid in the course of time.

We are now able to define method increments. As in common method engineering practices a method is seen as being composed of method fragments or method chunks [3] [16]. Such a method is consistently created using well-formedness rules of process composition and deliverable configuration. These rules are not elaborated here, as they can be found in [4].

Definition 3.5. The predicate $\text{contains}: \mathbb{F} \times \mathbb{S}$: $\text{contains}(f,s) \equiv$ fragment f is contained in snapshot s.

Then we can define an method increment as a method fragment that is part of $\text{method}(t_i)$ but not in $\text{method}(t_{i-1})$.

Definition 3.6. A *method increment* is a method fragment $f \in \mathbb{F}$ such that $\exists i$ $\text{contains}(f, \text{method}(t_i)) \wedge \neg\text{contains}(f, \text{method}(t_{i-1}))$

This means that the method increments are a collection of method fragments that have been introduced in the method during the method adaptations between t_i and t_{i-1}. In the following section we will then formalize the various types of increments

3.2 Formalization of Method Increments

In Figure 3 the meta-meta model of PDD is given, denoted in (again!) a UML Class diagram.

The meta-meta model is a simplified view of the full UML definition of Class diagrams and Activity diagrams [13] with special emphasis on the adaptations discussed in Section 2.2 and the definitions in 3.1. Figure 3 shows that a method consists of method fragments, that we distinguish as process fragments for the process part of a method and deliverable fragments similarly. Note that the creation of deliverables is modelled in the association *edits* between Activities and Concepts.

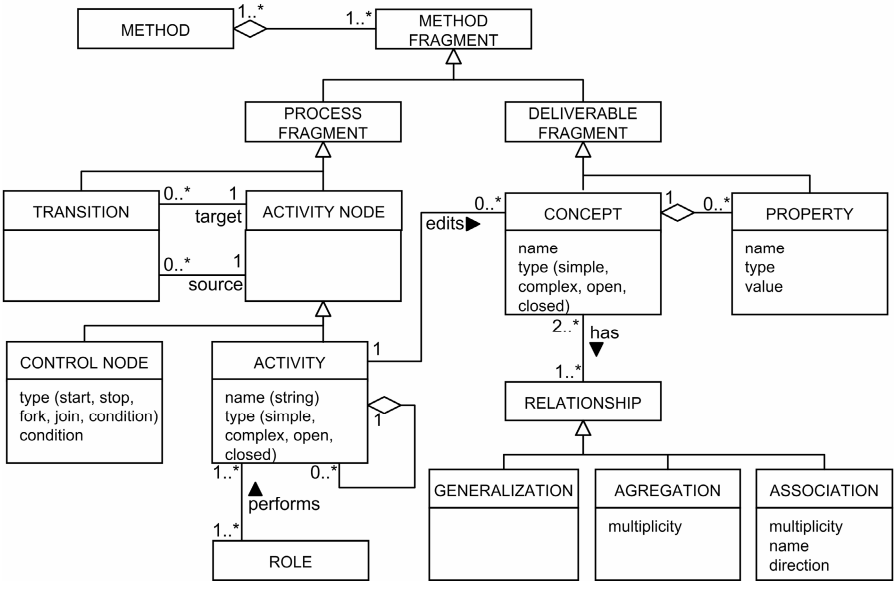

Fig. 3. Meta-meta model of PDD

The structure of the meta-meta-model and the earlier case studies [27] to method evolution revealed that 18 elementary increment types can be distinguished:

- *insertion* of a concept, property, relationship, activity node, transition, role
- *modification* of a concept, property, relationship, activity node, transition, role
- *deletion* of a concept, property, relationship, activity node, transition, role

The complete method increments from one snapshot to another can then be seen as a composition of elementary increment types.

The UML formalization of [2] postulates the existence of unary predicates for each class in a class diagram, e.g. concept(c) means that c is a concept in the model. However, in our research we require evolution of methods over the various snapshots, so we enhance these unary predicates to binary predicates with the method as an additional parameter. So concept(c,m) means that c is a concept in the method m. Method increments can now be defined as polymorphic mappings on the set of method fragments and methods.

Definition 3.7. The mapping insert: $\mathsf{F} \times \mathsf{M} \rightarrow \mathsf{M}$: insert($f$,$m_1$) = m_2 means that the method fragment f has been inserted in the method m_1 resulting into method m_2.

Definition 3.8. The mapping modify: $\mathsf{F} \times \mathsf{F} \times \mathsf{M} \rightarrow \mathsf{M}$: modify($f_1$,$f_2$,$m_1$) = m_2 means that the method fragment f_1 in the method m_1 has been modified to the fragment f_2 in method m_2.

Definition 3.9. The mapping delete: $\mathsf{F} \times \mathsf{M} \rightarrow \mathsf{M}$: delete($f$,$m_1$) = m_2 means that the method fragment f has been deleted from the method m_1 resulting into method m_2.

The rules for the elementary increments can then be formulated. For the sake of brevity we list the rules for the insertion of concepts and properties. Both rules are illustrated with an example that is taken from the increment example in Section 4.3.

Rule 3.1. Insertion of concepts:

$\mathsf{insert}(c,m_i) = m_{i+1} \Rightarrow \neg\mathsf{concept}(c,m_i) \wedge \mathsf{concept}(c,m_{i+1})$

Rule 3.1 states when a concept has been inserted into method mi to get method mi+1. So, for instance:

$\mathsf{insert}(\text{RELEASE TABLE},\mathsf{BaanIncr2}) = \mathsf{BaanIncr3} \Rightarrow \neg\mathsf{concept}(\text{RELEASE TABLE},\mathsf{BaanIncr2}) \wedge \mathsf{concept}(\text{RELEASE TABLE},\mathsf{BaanIncr3})$

This means that when the concept RELEASE TABLE is inserted into BaanIncr2 resulting into BaanIncr3, then RELEASE TABLE is not a concept present in BaanIncr2 and is present as concept in BaanIncr3.

Rule 3.2. Insertion of properties:

$\mathsf{insert}(p,m_i) = m_{i+1} \wedge \mathsf{property}(p,m_{i+1}) \Rightarrow [\forall c: \mathsf{concept}(c,m_i) \wedge \neg\mathsf{contains}(p,c)] \wedge [\exists 1 c: \mathsf{concept}(c,m_{i+1}) \wedge \mathsf{contains}(p,c)]$

Rule 3.2 tells that when property p is inserted into snapshot m_i resulting into snapshot m_{i+1}, then p is not a property of any concept in m_i and there is just one concept in m_{i+1} of which p is the property. So, for instance:

$\mathsf{insert}(\mathsf{topic},\mathsf{BaanIncr2}) = \mathsf{BaanIncr3} \wedge \mathsf{property}(\mathsf{topic},\mathsf{BaanIncr3}) \Rightarrow [\forall c: \mathsf{concept}(\text{REQUIREMENT},\mathsf{BaanIncr2}) \wedge \neg\mathsf{contains}(\mathsf{topic},\text{REQUIREMENT})] \wedge [\exists 1 c: \mathsf{concept}(\text{REQUIREMENT},\mathsf{BaanIncr3}) \wedge \mathsf{contains}(\mathsf{topic},\text{REQUIREMENT})]$

This means that when the property topic is inserted into snapshot BaanIncr2, resulting into BaanIncr3, then topic is not a property of any concept in BaanIncr2 and there is just one concept, namely REQUIREMENT, in BaanIncr3 of which is topic the property.

Analogously, rules for the other 16 elementary method increments can be formulated, while taking the method assembly rules in [4] into account. Based on our earlier work on method assembly this formalization is extremely straightforward and will support the construction of the PSKI currently under development.

3.3 Root Cause Analysis for Product Software

Based on the general Root Cause Map (RCM) [18], the reference framework for software product management (SPM) [24], and the HRM case study [27], we are able to construct an initial RCM for SPM, as is depicted in Figure 4.

During the interviews conducted in the HRM case study, two major process difficulties for requirements management were recognized:

A. Customers do not see that their required features and software improvement wishes are implemented in new releases.
B. The company finds its requirements gathering process for new features not productive.

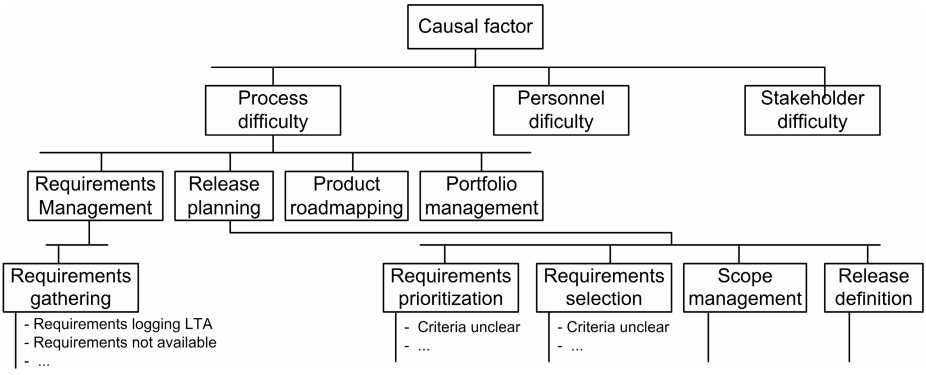

Fig. 4. The explorative case root-cause map

When we apply RCA to these process difficulties, we identify a number of causal factors: To communicate a suggestion for improvement, a customer can contact the sales representative; in some cases the sales representative replies that suggestions should be posted to the helpdesk; in other cases the sales representative forwards the suggestion to the helpdesk; and some suggestions are not logged at all.

As for the second process difficulty, when a new release is defined, the helpdesk, the development manager and the software engineers are consulted. Rather arbitrary, but fitting a defined planning schedule, the development of a new release is triggered. Consequently, we identify three causal factors:

C1. Customers have difficulty in making their wishes known
C2. Customer requirements are not registered effectively
C3. Scoping of releases is rather arbitrary

The following root causes can be identified (indicated are the corresponding causal factors):

R1. Requirement logging is less than adequate (LTA) (root cause for C1 & C2)
R2. Requirements are not available (root cause for C3)
R3. Criteria for requirements prioritization are unclear (root cause for C1 & C3)
R4. Criteria for requirements selection are unclear (root cause for C3)

In [27] a threefold solution for the two major process difficulties(A & B) is described:

S1. Introduction of a separate activity for receiving and logging new requirements;
S2. Introduction of a wish list (requirements database) with wishes (requirements) containing a priority attribute;
S3. Introduction of a separate activity for prioritizing wishes.

When we map this process on the PSKI, this threefold solution would be described in a process advice, containing process descriptions, templates and examples. Note that RCA was not the basis for the solution finding at the HRM case study. However, if we do take into account the RCA and the resulting root causes we find that solution S1 addresses R1 and R2, solution S2 addresses R2 and partly R3, solution S3 addresses R3. Note that R4 has not been properly addressed in the solution. We

conclude that in the HRM case study, RCA was a useful approach for finding process improvements alternatives. This will be further validated in Section 4.

4 ERP Case Study

We carried out a case study at Infor Global Solutions (specifically the former Baan company business unit), a vendor of ERP (Enterprise Resource Planning) software (see for example [12]). The goal of the ERP case study is to validate the increment types defined in Section 3.3 and the root-cause map in Section 3.4. In 1978, Baan was established as a book-keeping consulting company. Over the years, the company changed from a consultant company to a software developer for businesses. Baan was quoted on the Nasdaq stock exchange as an independent company from 1995 to 2000.

4.1 Case Study Design

Different sources are used to collect information. Firstly, several interviews have been conducted with six former employees of Baan. Two explorative 3-hour interviews were conducted with the Process Engineer of Baan. Based on this interview, the method evolution between 1997 and 2002 was modeled. This information was cross-checked by conducting 2-hour follow-up interviews with five other employees of Baan, consisting of two former (Senior) Product Managers, a Director ERP Development, a Manager ERP Product Ownership and a Software Engineering Process Group Manager for Baan Development. In these interviews, also the snapshots of 1994, 1996, 2003, 2004 and 2006 were identified and modeled.

Secondly, a document study was carried out. Documentation provided by the Process Engineer was used to complement and validate the results from the interviews. This documentation consisted of process descriptions, templates and examples of methods and work products used at Baan in the period 1997 until 2006. From the period before 1997 no documentation was available. We focused on the following case study questions, related to software product management:

- Which snapshots can you identify in the method evolution?
- Which methods were used per stage? Which activities can be distinguished?
- Which deliverables resulted from these methods?
- Which process difficulties arose in this stage? Why was an increment needed?

With the information gathered in the case study, we modeled 14 snapshots in PDDs, each representing a method that was used in a particular moment in time [26].

4.2 Method Snapshots

We analyzed 14 snapshots of the evolution of the software development process at Baan, with emphasis on product management activities. The time period that is covered in the ERP case study ranges from 1994 to 2006.

Note that, although some method increments entail the removal of a method fragment, we still describe them as increments, as described in Section 2.1. In the

Table 1. Overview of method increments at Baan

#	Increment	Date
0	Introduction requirements document	1994
1	Introduction design document	1996
2	Introduction version definition	1998, May
3	Introduction conceptual solution	1998, November
4	Introduction requirements database, division market and business requirements, and introduction of product families	1999, May
5	Introduction tracing sheet	1999, July
6	Introduction product definition	2000, March
7	Introduction customer commitment process	2000, April
8	Introduction enhancement request process	2000, May
9	Introduction roadmap process	2000, September
10	Introduction process metrics	2002, August
11	Removal of product families & customer commitment	2003, May
12	Introduction customer voting process	2004, November
13	Introduction master planning	2006, October

following section, one of these increments, namely the increment between snapshot 2 and 3, is further elaborated on. The other increments are described in [26].

4.3 Increment Example: Introduction of the Conceptual Solution

In Figure 5, increment # 2 of the ERP case study is visualized. Looking at the process-side of the diagram, we can distinguish one main activity, i.e. 'Requirements', and three sub-activities.

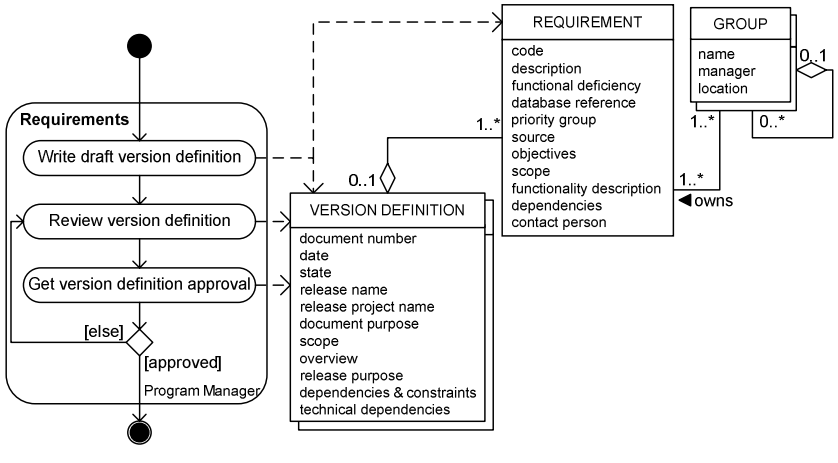

Fig. 5. Snapshot of increment #2

The first sub-activity, 'Write draft version definition', results in the concepts VERSION DEFINITION and REQUIREMENT. The latter is connected to VERSION DEFINITION by means of aggregation. Both have a number of attributes, and finally, a REQUIREMENT is owned by a GROUP, that has the responsibility for this REQUIREMENT. The next sub-activity is to review the VERSION DEFINITION. If the approval is obtained, the next activity can be started; otherwise the VERSION DEFINITION has to be reviewed again.

In Figure 6, increment #3 is visualized. In this snapshot, one extra activity is included. Note, however, that this activity is open, i.e. this activity contains further sub activities that are elaborated elsewhere. Due to space limitations, the elaboration on this activity is not included in this paper.

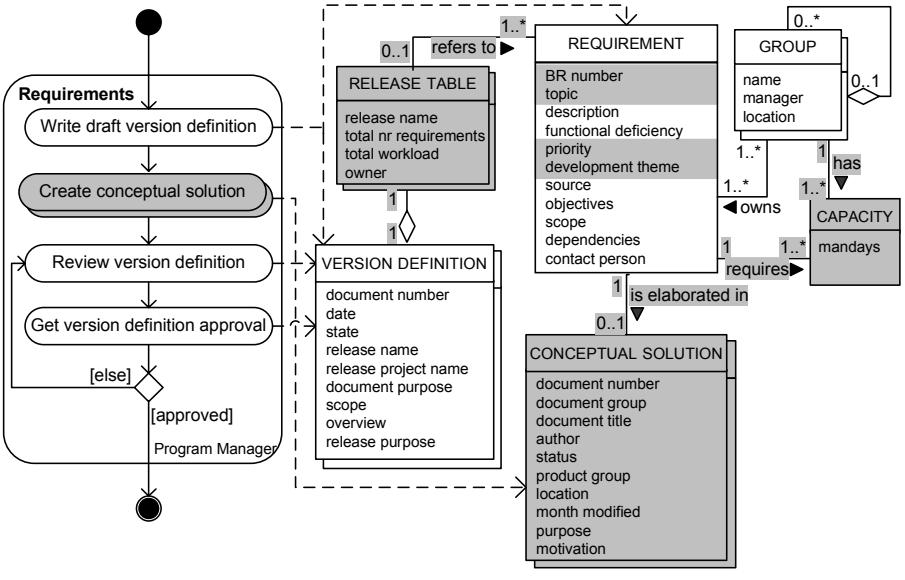

Fig. 6. Snapshot of increment #3

4.4 Root Cause Analysis of Method Increments

In increment 3 (Figure 6) we distinguish the following increment types, based on the formalization in Section 3.2:

I1. Insertion of an activity node, i.e. 'Create conceptual solution'
I2. Insertion of a concept, i.e. RELEASE TABLE, CONCEPTUAL SOLUTION and CAPACITY
I3. Insertion of a property, i.e. the properties added to REQUIREMENT
I4. Insertion of a relationship, i.e. the relationships connecting the introduced concepts to the existing concepts

Now we focus on RCA. The increments are included to solve one or more problems. Based on the interviews, several process needs were identified in the snapshot of increment #2. The most important ones were:

A. Development managers find it hard if not impossible to determine a VERSION DEFINITION that is feasible with available resources, and consequently makes sub-optimal scoping decisions. In detail: signals from the market, as well as from internal stakeholders, indicate that a new release should be developed. The development managers ask the program managers and architects to establish the version definition for the different software modules. The program managers and architects collect, with some difficulty, the features and requirement from different sources. They select a set of features to be developed according to their own opinions. The program managers and architects discuss the draft version definition with the development managers, and make changes to the selection of features.
B. Software engineers find it hard to read the VERSION DEFINITION in order to built what is requested, and consequently do not build the precise features that were intended to be build. In detail: in the version definition each new software product REQUIREMENT is elaborated by the program manager and/or product architect. They describe dependencies with other REQUIREMENTS in the text associated with a requirement. The software engineers read the (often badly written) requirements, interpret requirement texts, possibly asking their program managers and architects for explanations. Subsequently, the requirements are built in the software product.

We identify the following causal factors:

C1. Requirement collection is difficult
C2. Text elaborations of requirements have different authors
C3. Requirements dependency descriptions are unstructured
C4. Interpretation of requirement is ambiguous

If we apply the earlier constructed root cause map for product software to this particular increment we choose to extend it accordingly in order to address all identified causal factors (see Figure 7). Note that, although we had to extend the root cause map with the (bold) root causes, it fits the constructed structure as derived in Section 3 very well.
The root causes of the four identified causal factors are fourfold:

R1. Requirements are scattered throughout the company in different documents (root cause for C1 & C2)

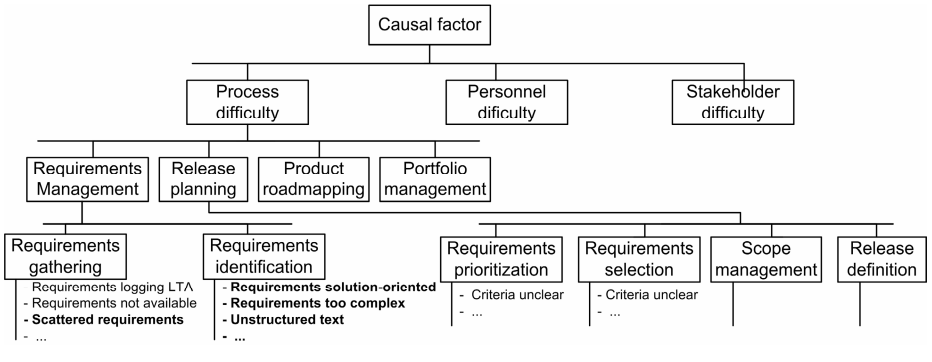

Fig. 7. Baan increment extended root cause map for product software

R2. Some requirements are written in a solution-oriented way (root cause for C2 & C4)
R3. Requirements are too complex (root cause for C4)
R4. Requirements are written in unstructured text (root cause for C3 & C4)

R2 and R3 led to the introduction of the CONCEPTUAL SOLUTION in increment #3. This document was used to write a solution on conceptual level for the particular REQUIREMENT. In this way, solution-oriented texts are also kept out the requirements themselves. R4 partly led to the decomposition of the VERSION DEFINITION and REQUIREMENTS. A RELEASE TABLE is used, in which information on the separate REQUIREMENTS is summarized. No (full) solution is implemented for R1 and R4. These are taken into account in the subsequent increment, which is described in [26].

We note that the RCM for SPM has been extended on the lowest level, but that higher levels were untouched, indicating that for two different companies, the RCM for SPM is a useful tool. We conclude that RCA can be used in software development improvements (as in [8]) and more specifically software product management. The root causes showed in the RCM for SPM provide means for the PSKI to determine process improvement alternatives.

4.5 Validity Threats

In exploratory research, three types of validity are important [29]. Firstly, construct validity concerns the validity of the research method. We satisfy this type of validity by using multiple sources of data (interviewees and documents) and by maintaining a chain of evidence. Furthermore, we had key informants review the draft case study report. Secondly, the external validity concerns the domain to which the results can be generalized. We carried out the case study in the software product management domain in PS companies. The same protocol is followed as in earlier case studies in PS companies. Finally, to guarantee the reliability of the case study, all information should be recorded. This is done by maintaining a case study database which contains all relevant information used in the case study. This case study database consists of interview notes, documentation and process-data diagrams of all modelled methods.

5 Conclusion

By presenting a formal approach to incremental process improvement, we provided PS companies with an instrument to improve their software product management methods in an evolutionary way. Firstly, we formalized the method increments that occur during method evolution. Doing this provided insight in the evolution process, which can be used when assembling a method advice. Secondly, we presented an approach for the structural analysis of process needs, by using root cause analysis. By applying this analysis in a case study, we found that this approach and the corresponding root cause map can be of great value in the support of incremental method evolution.

Currently, we are working on the realization of the PSKI. We aim to further integrate the root cause analysis approach in the PSKI in order to map root causes to maturity capabilities and method fragments. The formalization of method increments

is used to implement assembly rules. In the future, we plan to fill the method base with situational factors, method fragments and assembly rules. Finally, we plan to test the PSKI at PS companies of different sizes and in different sectors, in order to test the mapping between situational factors, maturity capabilities and method fragments.

References

1. Aydin, M.N., Harmsen, F.: Making a Method Work for a Project Situation in the Context of CMM. In: Proceedings of the 14th International Conference on Product Focused Software Process Improvement, Rovaniemi, Finland, pp. 158–171 (2002)
2. Berardi, D., Cali, A., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence 168, 70–118 (2005)
3. Brinkkemper, S.: Method Engineering: Engineering of Information Systems Development Methods and Tools. In: Information and Software Techn, vol. 38, pp. 275–280. Elsevier, Amsterdam (1996)
4. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-modelling Based Assembly Techniques for Situational Method Engineering. Information Systems 24(3), 209–228 (1999)
5. Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In: LNCS, vol. 2029, pp. 17 –31Springer, Heidelberg (2001)
6. Conradi, R., Fernström, C., Fuggetta, A.: A Conceptual Framework for Evolving Software Processes. In: ACM SIGSOFT Software Eng. Notes 18(4), 26–35 (1993)
7. Cronholm, S., Ågerfalk, P.J.: On the Concept of Method in Information Systems Development. In: Proceedings of the 22nd Information Systems Research Seminar in Scandinavia 1, 229–236 (1999)
8. El, E.K., Melo, W., Drouin, J.-N. (eds.): SPICE: The Theory and Practice of Software Process Improvement and Capability Determination. IEEE Computer Soc. Press, Los Alamitos (1997)
9. Henderson-Sellers, B.: Process Metamodelling and Process Construction: Examples Using the OPEN Process Framework (OPF). Annals of Software Eng. 14, 341–362 (2002)
10. Krzanik, L., Simila, J.: Is my Software Process Improvement Suitable for Incremental Deployment? 8th International Workshop on Software Technology and Engineering Practice (STEP'97) p. 76 (1997)
11. Leszak, M., Perry, D.E., Stoll, D.: A Case Study in Root Cause Defect Analysis, ICSE p. 428 (2000)
12. Natt och Dag, J., Gervasi, V., Brinkkemper, S., Regnell, B.: Speeding up Requirements Management in a Product Software Company: Linking Customer Wishes to Product Requirements through Linguistic Engineering. In: Proceedings of the 12th IEEE International Requirements Engineering Conference pp. 283–294 (2004)
13. Object Management Group: UML 2.0 Superstructure Specification. Technical Report ptc/04-10-02 (2004)
14. Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V.: Capability Maturity Model for Software (Version 1.1) (SEI/CMU-93-TR-24, ADA263403). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University (1993)
15. Rainer, A., Hall, T.: Key Success Factors for Implementing Software Process Improvement: a Maturity-Based Analysis. Journal of Systems and Software 62(2), 71–84 (2002)

16. Ralyté, J., Rolland, C.: An Assembly Process Model for Method Engineering. In: Advanced Information Systems Engineering. In: CAiSE 2001. LNCS, vol. 2068, pp. 267–283. Springer, Heidelberg (2001)
17. Richardson, I., Ryan, K.: Software Process Improvements in a Very Small Company. Software Quality Professional 3(2), 23–35 (2001)
18. Root Cause Analysis Handbook: A Guide to Effective Incident Investigation, ABS Group Consulting, Inc, Houston, TX (1999)
19. Rossi, M., Ramesh, B., Lyytinen, K., Tolvanen, J.-P.: Managing evolutionary method engineering by method rationale. Journal of the Association for Information Systems 5(9), 356–391 (2004)
20. Saeki, M.: Toward Formal Semantics of Meta Models. In: International Workshop on Model Engineering, Nice, France (2000)
21. SEI: Process maturity profile of the software community. Software Engineering Institute, Carnegie Mellon University (2002)
22. Stelzer, D., Mellis, W.: Success Factors of Organizational Change in Software Process Improvement. In: Software Process: Improvement and Practice, vol. 4(4), pp. 227–250. John Wiley & Sons, New York (1998)
23. Tolvanen, J.-P.: Incremental method engineering with modeling tools: theoretical principles and empirical evidence. Jyväskylä Studies in Computer Science, Economics and Statistics 47, University of Jyväskylä, PhD Dissertation thesis (1998)
24. Weerd, I., van de Brinkkemper, S., Nieuwenhuis, R., Versendaal, J., Bijlsma, L.: Towards a Reference Framework for Software Product Management. In: Proc. of the 14th International Requirements Engineering Conference, Minneapolis, Minnesota, USA pp. 312-315 (2006)
25. Weerd, I., van de Brinkkemper, S., Souer, J., Versendaal, J.: A Situational Implementation Method for Web-based Content Management System-applications. In: Software Process: Improvement and Practice. Vol. 11(5), pp. 521–538. John Wiley & Sons, New York (2006)
26. Weerd, I., van de Brinkkemper, S., Versendaal, J.: Incremental Method Evolution in Requirements Management: A Case Study at Baan 1994-2006. Institute of Computing and Information Sciences, Utrecht University. Technical report UU-CS-2006-057 (2006)
27. Weerd, I., van de Versendaal, J., Brinkkemper, S.: A Product Software Knowledge Infrastructure for Situational Capability Maturation: Vision and Case Studies in Product Management. In: Proceedings of the 12th Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'06), Luxembourg (2006)
28. Xu, L., Brinkkemper, S.: Concepts for Product Software. To appear in: European Journal of Information Systems (2007)
29. Yin, R.K.: Case study research: Design and methods (3rd edn.). Beverly Hills, CA: Sage Publishing (2003)