SPEMArti: Towards detailed artifact specification

Marcos Silva, Kleinner Oliveira, Toacy Oliveira, Ricardo Bastos
Informatics Faculty
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
\{mtsilva,ksoliveira,toacy,bastos\}@pucrs.br

Abstract

In the context of Software Engineering, Software Process is a set of activities and its associated results. The Object Management Group (OMG) created the Software Process Engineering Metamodel (SPEM) in order to better represent and design software process. However, the artifacts are built as an auto-sufficient monolithic piece of information. The challenge is to build software artifacts that can better be controlled and managed. In this paper we present ongoing work in SPEM extension which represents information as high granularity pieces of data. Nevertheless, redefining and extending the SPEM metamodel gives us a flexible way to split the software process metamodel artifacts. Moreover, we can manage data in an easy way, avoiding some of the redundancy and organizing the artifacts as a composite element. Finally, this work presents a different perspective to artifacts using a rudimentary example.

1. Introduction

The concepts and definitions about software process are not a consensus between authors. In conformity to [11], a software process is a guide to increase the software quality and determines how it can be developed.

As expected, softwares are complex and difficulty to develop and test, sometimes they present unexpected and undesired behaviours, culminating, very often, in disasters [2]. Fuggetta, in 2000, defined software process as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product.

The Object Management Group (OMG) defines the Software Process Engineering Metamodel (SPEM) [6] which consists of a conceptual software process metamodel and a default language nomenclature. Needs for metamodel improvement allow efforts from IBM to develop the Software and Systems Process Engineering Metamodel (SPEM v2) [5] based on the Unified Method Architecture (UMA). SPEM v2 is a partial software process metamodel for software process authoring and still not complete, it is not helpful for process enactment and there are not definitions for artifacts internal structure.

The SPEM extends Unified Modeling Language (UML) capabilities and was defined by Meta Object Facility (MOF) [4]. UML [7] is a unified language for modeling object oriented systems and has becoming the standard representation for application architecture at the requirements, design, analyses and deployment of the software development process activities [12]. Hence, usages of UML allow an integrated representation of the system with models which are related to each other [13].

The OMG [4] defines two different strategies to extend UML. The first one is a lightweight built-in extension with the use of profiles mechanism. According to MDA [3], the model elements at the platform independent model (PIM) layer are marked with stereotypes which transform the PIM in a platform specific model (PSM). The second strategy is called a heavyweight extensibility mechanism as defined by the specification of MOF [4]. In this strategy the goal is to extends the UML adding new metaclasses and other metaclasses.

Our approach uses the second strategy, adding new metaclasses and extending the SPEM metamodel to build an advanced concept of logically and structured artifacts. The first strategy is not sufficient to produce specific artifacts structures since we need to define each artifact structure at the model layer.

Accordingly, the paper is structured in sections: section 2, gives some challenges and the objectives of our approach, section 3, presents a quick overview of the related works; section 4, presents an evaluation of the ongoing work; section 5, summarizes some possible directions; section 6, draws some conclusions.
2. Objectives

Nowadays, software process comprehend artifacts as monolithic pieces of information. But, on nearer view artifacts have internal logical structures and information representation either and without entire artifact comprehension we can not always realize redundancy or, sometimes, we do not know it exists. Since there are different artifacts, addressing this problem requires knowledge about the artifact structure since each artifact has its individual and sometimes different structure. Unfortunatelly, SPEM v2 also does not include artifact structure perspective.

In this sense, some challenges arise, such as: (1) How should the artifact be organized, structured, and filled in? (2) How to manage artifacts due its versions, its authors and its content, avoiding redundancy and rework? We present ongoing work which address the enumerated problems and suggesting one SPEM extension. Our objectives are:

1. To apply effective Software Process authoring with detailed artifact structure, at a high quality level (in terms of correctness, reusability, flexibility) for building real process documentation, helping the authors to define artifacts;
2. To create a common metamodel to handle different activities of process definitions including artifacts details, classification;
3. To specify software process artifact enactment with versioning, maturity and fill in control, permitting a more easier information management;
4. To reuse as much as possible existing information and logical artifact structure avoiding redundancy and lost of data;
5. To capture the important knowledge of artifact constructions, removing the responsibilities of building the artifact from roles defined at the software process enactment and giving to roles defined at the software process definitions and authoring.

3. Related Works

In this section we present related works about software documentation and classification. The subsection contents are based on respective cited authors.

3.1. INFO

INFO [14] presents an easy way for documenting and commenting source-code. Due to authors, without good and clear documentation, we need to analyze full source-code as the unique way to understand it. Although, sometimes source-codes have good documentation and comprehensive comments but they do not follow company standards. In the worst case, the source-code documentation may be incomplete.

At the programmer’s perspective, revising full source-code to check correctness of the comments is a repetitive and mechanical task. In such case, documentation remains incomplete and little explanation. Since the programmer mainly idea is to write source-code, he is not really worried about documenting it.

On the authors’ perspective separating comments from source-code should solve the problem. However, this separation introduces a new problem: both of them keeps out-of-date and unsynchronized. One unique up-to-date well-formed artifact with good comments is difficult to produce. Twice would be almost impossible.

The authors’ solution (called INFO) is to use a hypertext facility which allows documentation annotations in a non-intrusive way. The source-code comments and documentation are persisted in a text file and bound for tagged information at the original source document. As an example, a C++ source-code fragment annotated with an info tag should be as follows:

\[\texttt{int \text{ parse}(\text{ Node } \&); \}; //\text{ info kw=parse}\]

3.2. Hipikat

Talking about open-source projects, Hipikat [15] tries to address the newcomers’ project entrance with a different approach. The authors’ approach is divided at two parts helping the newcomer become productive more quickly. First, the project documentation is allocated as a group memory with a schema. Second, the group memory is presented to the newcomer with the more relevant documents.

The open-source projects vary in different contexts (like domain and team) and according to authors such projects typically produce four electronic artifacts: source version control and repository (like CVS and SVN), issue-tracking system (such as bugzilla), communication channels (mailing lists and newsgroups as an example) and online documentation. Thus, Hipikat has three distinct functions: Identification; Selection; and Update.

3.3. 4 - Level Documentation Process Maturity

Due to [16] documentation consists of 60%-75% from the software cost. Within the industrial environment was detected that about 20% of maintenance problems occurred because the lack of good documentation or wrong document classification.
The authors’ approach identifies a way to improve the process and defines four maturity levels: ad-hoc, inconsistent, defined and controlled. Each level is described as following fields: Name, Succinct Description, Keywords, Key Process Areas, Key Practices, Key Indicators, Key Challenges and Key Significance.

More information about maturity levels can be viewed at [16] and are not discussed here.

### 3.4. Document Classification

According to [10], every software development project is unique. Thus, different kinds of artifacts are produced, different amount of documentation are consumed and different documentation methods are used. Nevertheless, it is important to know how to control and manage every artifact in a uniform way.

This control has various kinds of conventions and followings such as identification, storage, approval and modification of artifacts. Due to the author, other approaches focus on quality and want to keep the artifact in the correct form during the project life-cycle. However, they do not define how to classify the document neither how to organize the information.

The author suggests an approach to classify documents, providing identification about the kind of document information. Thus, nomenclature defines: Software Description, Utilization Documents, Development Plans, Quality Control Documents and Administrative Documents.

More information about the document classification can be viewed at [10] and are not discussed here.

### 3.5. Modelling Document Structure

Akpotsui et. al. in 1992 consider a document as a logical structure, made up of elements such as title, section, paragraph, which are composed into a structure that represents the document structure and organization. Due to [1], each element has a type and the relationships between them defines a generic document model structure. Each document has a specific structure as an instance from the generic structure.

Since the document is considered a logical structure, they can be viewed into a tree structure. According to [1], each node has a type; the generic structure is a set of types constructed with a set of basic types called $\beta$ and a set of constructors called $C$:

- $\beta = \{\text{String, Picture, Graphic}\}$
- $C = \{\text{Ordered Group, Unorderer Group, List, Choice, Identity}\}$

More information about the document tree structure and the document DTD can be viewed at [1].

### 4. The SPEMArti

In the context of software process metamodels, SPEM is a good open-source solution and constitutes of seven metamodel packages [5]. The SPEM v2 is still not complete and there is a gap between the artifact and its fill in. Figure 1 presents the SPEM with focus on WorkProduct. The metamodel has limitations defining process workproduct: it does not admit uses of the workproduct as a structured, organized and logical element.

![Figure 1. SPEM resume on Workproduct](image)

The SPEM v2 specification defines three compliance points [5]: Complete, Process with Behavior and Content and Method Content. According to our needs we decide to use the SPEM Process with Behavior and Content Compliance Point with focus on the modeling aspects of SPEM.

#### 4.1. The problem of the artifact organization

In a software process, each artifact has its own organization and structure. As an example, source-code is totally different from vision document. As SPEM defines every artifact as a simple and monolithic structure, there aren’t concerns about the artifacts standards. In that way, the software process will be defined with known artifacts such as use case and vision document. Otherwise it is very complicated to realize that during the software process enactment there is redundancy or rework, since we cannot know who (or what) will enact in the roles described in the software process.

The software quality standards usually determine what need to be documented during the software process enactment [10]. It is not sufficient, according to our approach,
because we do not know the artifact organization neither its structure. According to [1], there is one generic structure which can be instantiated as a specific structure. As an example, once use case has different structure from vision document, they should be two different specific structures instantiated from the same generic one.

Based on this idea, we extend the SPEM. Figure 2 depicts the additional elements at the metamodel.

![Figure 2. WorkProduct Metamodel of the SPEMArti](image)

Artifact, ContentElement and WorkProduct are metamodel elements from SPEM v2 [5]. Extending the concept of the artifact and using the logical structured idea, we can create a new artifact called MyArtifact. MyArtifact extends all concepts of the original Artifact and is a StructuredArtifact that can be sliced in pieces of information having one logical and defined structure. Thus, the StructuredArtifact metamodel element extends ContentableElement, responsible for every element which needs contents. Concluding, we have obtained an artifact which has structured content.

This extension addresses the question about artifact structure and allows us to use the structured content to create the structured artifacts generic metamodel.

4.2. Using artifacts metamodel

Once defined the extension, we need to create a generic metamodel structure for artifacts. At this paper, we presents only the document artifact structure (constituted of chapters and sections), which is composed with organized (with maturity levels, version and classification) and logical structured.

The metamodel is depicted in Figure 3 and it is a reducted version from the complete metamodel. The NamedElement and PackageableElement metamodel elements merge from SPEM, the ContentableElement, ClassifiableElement and MaturityLevelElement are the mainly metamodel elements since they are key extension points.

The ContentableElement is an extensible and versionable metamodel element. The PackageableElement extension is important allowing packageable versionable elements. Once the ContentableElement extends VersionableElement, every instance can be packageable and versionable. According to [1], the logical structure of the document can be viewed as a tree, with data types and its constructors as nodes. Our approach uses the ContentableElement as a logical tree structure based in a composite element, the StructuredArtifact. In this sense, we mapped the information on atomic pieces of data and we typed them as leaves.

According to the metamodel, AuthoredElement, DescriptionedElement and TitledElement are elements which permit the use of author, description and title as data types. The DocumentContentableElement has all the document specific concepts, such as the kind of allowed information, and it is an extension point to other types of document contents. Thus, DocumentContentableElement has a relationship with InformationPieceElement which can be an extension point, admitting special types of data. For an example, the metamodel depicted at Figure 3 presents GraphicElement, PictureElement and StringElement.

As an extension of DocumentContentableElement we have Chapter and Section and they can be used to separate the document in two groups. It is necessary to comprehend that the document does not need to use all structure but it has a generic logical structure and organization modeled at the metamodel.

Based on [10] approach, our metamodel also permits use of document classification using ClassifiableElement key extension point. Each class of information, Software Description as an example, can be an instance of Information-Class.

The last key extension point is MaturityLevelElement. [16] defines four levels of maturity classification for documents produced by software process. Using that metamodel element, we can instantiate maturity levels as cited by [16], controlling the maturity of the information.

4.3. Instantiating the metamodel

Until now, the SPEM extension was defined, the artifacts were extended and we have a metamodel which represents artifact organization and specific document logical structure. Since we have modeled the artifact versioning, author and content elements with the classification and maturity levels concepts, we address the challenges 1 and 2.

The metamodel structure constraints necessary metamodel elements, creating a model instance with information about how to fill in an artifact using the InformationPieceElement. The artifact management is made from versioning (we can know artifacts version), classification (we
can classify the artifacts and its pieces due to its information content type), maturity (each artifact can be in different level of maturity) and author (we can know who produce the parts of the artifact). Thus, once the information piece of the artifact instance will be shared with others artifacts, the information piece dependency will automatically be filled out.

As a rudimentary example we presents two RUP artifacts instances, the Functional Requirement and Vision Document [9]. According to the four levels of abstraction of MDA have the SPEMArti metamodel at level M3, the the RUP artifact model at level M2, the RUP artifact at level M1 and the Requirement and Vision Document at level M0. Figure 4 shows the SPEMArti instantiation.

While the functional requirements of the software are filled in, the section Functional Requirements of the Vision Document is automatically filled in, avoiding redundancy and rework.

**5. Some directions**

The next steps are:

- A further refinement of the Artifact and WorkProduct extention;
- To define other artifacts metamodel until we complete the generic artifact structure;
- To create tools to support the approach;
- To execute the validation of the final metamodel.

Based on the chosen SPEM Compliance Point we can extends others metamodel elements creating our metamodel package which should merge with SPEM MethodContent. This solution can make the workproduct extension more flexible, preserving the original SPEM metamodel. Once created, the metamodel package can be imported, merged or extended (making easy to practioners and others researchers to use our approach).

The metamodel presented here addresses versioning and organization about artifacts, but only the document logical structure. It is important to reach others artifacts, such as source-code, creating one really generic structure wich can be used at the entire software process. One generic structure can address part of the artifacts redundancy and management, defining pieces of information without copy and paste procedures or rework, consenting queries the retrieve different view of information.

The Eclipse Process Framework (EPF) Composer [8] is an IBM’s open-source framework for authoring the software process. At this moment, EPF Composer uses the
Finally, we observe that our metamodel is in initial stage and there is very little experience to determine the worth of current approaches. Moreover, is absolutely necessary the evolution of the metamodel and validate it in industrial reality.

6. Conclusions

Without logically structured neither organizationed ones artifacts it is impossible to author one software process which allows detailed artifacts. Once the artifacts details aren’t defined they can be anything and can be created somehow. The artifacts domain has lots of challenges, such as such as, how to organize, structure, and fill in artifacts. In order to briefly answer these questions (this work does not claim completeness) our approach addresses the artifacts organization and document logical structure.

This work has not been validated yet and it is one of the most important points to be explored. Although, create a metamodel which represents the artifacts of the software process with composed information has a limitations: we do not know how to control the information production, since most of the software processes use roles act activities which produce the entire artifact.

References