UNIVERSIDAD POLITECNICA DE VALENCIA

PROS
Centro de Investigación en Métodos de Producción de Software

Technical Report




A Software Architecture for Model Driven Method Engineering

Mario Cervera, Manoli Albert, Victoria Torres, Vicente Pelechano





Ref. #:	PROS-TR-2011-15				
Title:	A Software Architecture for Model Driven Method Engineering				
Author (s):	Mario Cervera, Manoli Albert, Victoria Torres, Vicente Pelechano				
Corresponding author (s):	Mario Cervera, mcervera@pros.upv.es Manoli Albert, malbert@pros.upv.es Victoria Torres, vtorres@pros.upv.es Vicente Pelechano, pele@pros.upv.es				
Document version number:	1.0	Final version:	Yes	Pages:	16
Release date:	July 2011				
Key words:	Software Architecture, Method Engineering, Eclipse, MOSKitt				

Chapter 4
A Software Architecture

CAME and metaCASE technology is still immature. Existing environments mostly represent incomplete prototypes that present important deficiencies [Niknafs08]. Furthermore, these tools are generally based on rigid architectures that hinder their adaptation to new contexts of use. In order to avoid this problem, software architectures for Method Engineering supporting tools should be defined according to a set of design guidelines. In this work the following are proposed:

- **Technology-independence**: the software architecture must be defined in a technology-independent fashion in order to decouple them from technological details. This approach increases the longevity of the architecture as its components do not become obsolete on account of technology changes.

- **Modularization**: the architecture must be defined in terms of loosely-coupled components. The main benefit of this approach is that tools implementing a modular architecture are composed of separate components, and thus they are easier to extend, modify and adapt to new requirements.

- **Separation of concerns**: the software architecture must separate components that deal with Method Engineering tasks from components that deal with ISD tasks. The former components make up the structure of the CAME part, which enables tasks such as method design. On the other hand, the latter components form the CASE part, which supports ISD tasks such as system specification.

Taking these guides into account, this chapter defines a modular software architecture that identifies the set of technology-independent components (and the relationships among them) that are required to support the methodological

framework presented in chapter 3. In addition, as a proof of concept of the proposal, a vertical prototype has been developed in the context of the MOSKitt platform. This prototype, called MOSKitt4ME, implements the proposed architecture and its main goal is to set the basis for the eventual development of a CAME environment that supports the design and implementation of methods, without presenting the deficiencies of current CAME and metaCASE technology.

This chapter is structured as follows: first, section 4.1 describes the requirements that the proposed architecture must address in order to provide complete support to the methodological framework. Then, section 4.2 presents the architecture in detail and also its implementation on the MOSKitt platform. Finally, section 4.3 concludes the chapter.

4.1. Architecture requirements

This section describes in detail the requirements that the proposed architecture must address in order to adequately support the methodological framework proposed in chapter 3. Specifically, this section is divided into two subsections, dealing respectively with the requirements of the CAME and CASE parts of the architecture.

4.1.1. Requirements for the CAME part

The CAME part of the architecture must include the required components to allow the method engineer to perform the method design and configuration phases of the methodological framework, and to invoke the CASE tool generation process that obtains the method implementation. Therefore, the following requirements have been identified:

Req. 1. A modeling tool for building method definitions

A modeling tool (a method editor) must be included in order to support the definition of software production methods based on a Method Engineering language such as the SPEM standard. Therefore, this tool allows the method engineer to perform the method design phase of the methodological framework.

As described in chapter 3, the method design can be performed from scratch or reusing conceptual fragments that are stored in a repository. Therefore, the modeling tool must also implement mechanisms that enable the integration of conceptual fragments into the method under construction. Furthermore, it must allow the method engineer to select parts of the method and create new conceptual fragments from these parts. This is done by means of a repository client (see req. 2).

It is also important to emphasize that the lack of a method editor is the major shortcoming of the metaCASE approach, since metaCASE tools generally focus on the method implementation. In general, metaCASE environments provide editors that enable the specification of the design notations that will be supported by the CASE tool under construction, but do not support the definition of software production methods that can be enacted in real software development projects.

Req. 2. A repository to store method fragments and mechanisms to access the repository

The method engineer must be able to reuse conceptual fragments during the method design. In addition, during the method configuration, he/she must be able to associate the tasks and products of the method with technical fragments that establish how these elements will be managed in the generated CASE tool. Therefore, mechanisms to connect the method editor and the repository containing these fragments must be provided. These mechanisms can be represented by a repository client. A repository client allows the method engineer to access the repository and search and select method fragments. For this purpose, the repository client must provide mechanisms for specifying the requirements of the fragments to retrieve. For instance, these requirements can be specified as queries that are formulated by giving values to the method fragment properties (i.e. type, origin, objective, etc.). Furthermore, the repository client must also allow the method engineer to store in the repository fragments that are created during the method design. These fragments can be later reused during the specification of other methods.

Req. 3. Mechanisms for the enactment of the Method Engineering process

The specification of software production methods is a task that must be adequately guided so that the method engineer can perform it properly. For this reason, a process that establishes the procedures and activities that must be followed during the method definition has to be defined. In order to support the execution of this process, a process engine can be included in the architecture. However, note that the inclusion of a process engine requires that the process is defined by means of an executable Process Modeling Language. Another possibility is to avoid the use of a process engine and define this process as a wizard or tutorial that textually guides the method engineer during the method definition.

Req. 4. A transformation engine

In order to automate the CASE tool generation process, a transformation engine is needed. The transformation engine is in charge of executing the model transformation that takes as input the model of the method (produced by means of the method editor) and obtains a CASE tool that supports it.

4.1.2. Requirements for the CASE part

The CASE part of the architecture must include the required components to allow the software engineer to perform the method enactment. Therefore, the following requirements have been identified:

Req. 5. Software tools that support the product part of the method

Software tools such as graphical editors, model transformations, etc. must be included in the generated CASE tool in order to support the creation and manipulation of the method products. These tools constitute the dynamic part of the CASE environments, since they depend on the method that has been specified. On the other hand, the static part corresponds to the tools that are always included in the CASE tools and, therefore, are independent of the specified method (see requirements 6 and 7).

Req. 6. Software tools that support the process part of the method

Tools such as a process engine must be included in the generated CASE tools in order to support the execution of the process part of the specified method. Thus, these tools provide a means for conducting the orchestration of the

different tools that allow the creation and manipulation of the method products (see req. 5). Specifically, these tools are a static part of the generated CASE tools, in the sense that they are independent of the specified method.

It is important to note that, the method must be specified in an executable language (such as the BPMN 2.0 standard [BPMN]) so that it can be executed in a process engine. In case the method is specified by means of a non-executable language (such as SPEM) a model transformation is required to transform the process model into an executable representation.

Req. 7. Project management mechanisms

The generated CASE tools must be endowed with a graphical user interface that allows software engineers to execute method instances (i.e. software development projects) by means of the tools that support the process part (see req. 6) and to invoke the tools that permit to create the method products (see req. 5). Like the tools that support the process part, the implementation of this graphical interface is independent of the specified method and, therefore, it is always included in the generated CASE tools.

4.2. The proposed architecture

This section describes the software architecture that is proposed in this work in order to meet the requirements presented in the previous section. Specifically, this section is divided into three subsections. First, section 4.2.1 defines the software architecture. Then, section 4.2.2 briefly presents some technological background that is needed in order to better understand how the proposed architecture has been implemented in the context of Eclipse (more specifically, on the MOSKitt platform). Finally, section 4.2.3 presents the implementation of the architecture, that is, the MOSKitt4ME prototype.

4.2.1. Conceptual definition

The proposed architecture (see Fig. 4.1) contains the set of loosely-coupled and technology-independent components that are required to support the methodological framework, i.e. to meet the requirements defined in section 4.1. These components are mainly divided into CAME components and

CASE components, and refer to the components that pertain respectively to the CAME and CASE parts of the architecture.

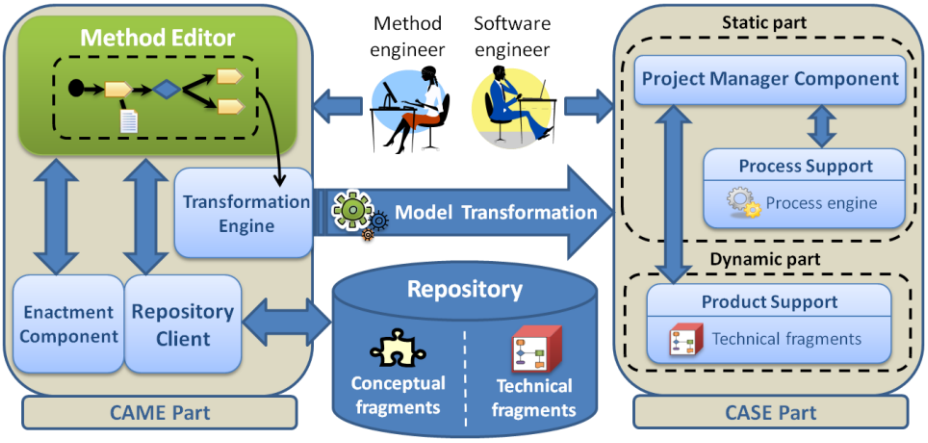

Fig. 4.1. Architecture components overview

CAME components

The CAME components make up the infrastructure of the CAME part of the architecture and are intended to meet from requirement 1 to requirement 4. Specifically, a *method editor* component (**req. 1**) has been included to allow the method engineer to perform the method design. During the construction of the method model, the method engineer can make use of the *repository* in order to reuse method fragments. For this purpose, the *repository client* (**req. 2**) is used. In general, the repository client allows the method engineer to connect to the repository, and select, reuse and store method fragments. Furthermore, the *enactment component* (**req. 3**) assists him/her during the whole method definition process. Finally, the resulting method model is fed into the *transformation engine* (**req. 4**) in order to obtain the method implementation (i.e. the CASE tool supporting the method). The method implementation is obtained by means of a model transformation that automates the generation process.

CASE components

The CASE components make up the infrastructure of the CASE part of the architecture and are intended to meet from requirement 5 to requirement 7. Specifically, the dynamic part (i.e. the components that are dependent on the

specified method) is composed of the *technical fragments* (**req. 5**). These components provide support to the product part of the method. On the other hand, the static part is composed of a *process engine* (**req. 6**), which provides support to the process part of the method, and the *project manager component* (**req. 7**), which embodies the graphical user interface that allows the software engineer to perform the method enactment.

4.2.2. Technological background

This subsection provides some technological background that is needed to facilitate the understanding of the prototype that has been developed in the context of Eclipse in order to implement the proposed architecture.

The Eclipse platform

Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. Specifically, there are two features of Eclipse that turn it into a very suitable platform to support Method Engineering approaches in the field of MDD:

- The Eclipse plugin-based architecture. Everything in Eclipse is a plugin but its runtime kernel. This means that Eclipse employs plugins to provide all of its functionality. This architecture allows developers to easily build Eclipse-based applications upon the Rich Client Platform (RCP)[1]. The RCP is, roughly speaking, the minimal set of plugins required to build an Eclipse application. This approach facilitates the development of the prototype, since the different components of the architecture can be developed as separate plugins that are easy to integrate into the same platform.

- The modeling technologies and tools. Within the Eclipse community a wide range of projects aim at providing as Eclipse plugins new tools and technologies for the support of different tasks. Specifically, one of these projects is the Eclipse Modeling Project [EMP] which focuses on model-based development technologies. This project contributes to

[1] Rich Client Platform , http://www.eclipse.org/home/categories/rcp.php

facilitate the development of the prototype, since it provides effective solutions for applying MDD techniques.

Below, the most significant Eclipse technologies that have been used in the development of the prototype are described.

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [EMF] is a modeling framework and code generation facility for building tools based on a structured data model. From a meta-model specification (called the "Ecore model") described in XMI, EMF provides a generator that produces a tree-based editor, together with the set of Java classes that implement the meta-model and allow the user to create models that conform to the meta-model. Therefore, EMF has been used as the underlying technology for the construction of the method models, which are stored in XMI format and conform to the SPEM Ecore model (i.e. a SPEM meta-model implementation for Eclipse).

Eclipse Process Framework Project

The Eclipse Process Framework (EPF) [EPF] aims to provide an extensible framework and exemplary tools for software process engineering. Specifically, one of these tools is the EPF Composer editor, which is an Eclipse-based editor that supports the construction of SPEM models in XMI format (based on EMF). Therefore, this tool has been used as the *method editor* component of the architecture.

Plug-in Development Environment

The Plug-in Development Environment (PDE) [PDE] provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins and Eclipse-based applications. Therefore, the functionality provided within the PDE has been used for facilitating the construction of the CASE tools that are generated from the method specifications. Specifically, the developed prototype makes use of the Product Configuration Files. These textual files contain all the required information (list of plugins, paths of images, etc.) to automatically build Eclipse applications from them. Hence, the model transformation that obtains the CASE tool support is in fact a M2T transformation that generates a product configuration file through which the final tool is obtained.

Xpand

Xpand [Xpand] is a statically-typed template language for implementing M2T transformations. Xpand was originally developed as part of the openArchitectureWare (oAW) project[2] before it became a component under Eclipse. Specifically, it is the language that has been used for implementing the M2T transformation that obtains product configuration files from method specifications.

4.2.3. MOSKitt4ME: An Eclipse-based CAME environment

In order to evaluate the proposed architecture, a vertical prototype, called MOSKitt4ME, has been developed in the context of Eclipse, more specifically, on the MOSKitt platform [MOSKitt]. In particular, this subsection details how the different components of the architecture have been implemented in MOSKitt.

Method editor

The method editor is the software component that supports the creation of method models. In particular, the methodological framework proposes the use of the SPEM standard as the Method Engineering language to carry out this task. Therefore, MOSKitt4ME must provide a method editor that enables the creation of SPEM models. For this purpose, the EPF Composer editor [EPF] has been integrated in MOSKitt. Fig 4.2[3] shows a snapshot of this editor.

Repository client

The repository client component must allow the method engineer (1) to connect to the repository, to (2) search and select method fragments for their use during the method design and configuration phases, and (3) to store newly created fragments. For this purpose, a repository client has been implemented as an Eclipse view. This view shows in a tree-based fashion the content of the repository it is connected to and provides searching capabilities based on fragment properties. In order to illustrate this idea, Fig. 4.3 and Fig. 4.4 show

---

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

[3] Also available at https://users.dsic.upv.es/~vtorres/moskitt4me/

this Eclipse view connected to the Method Base and Asset Base repositories respectively.

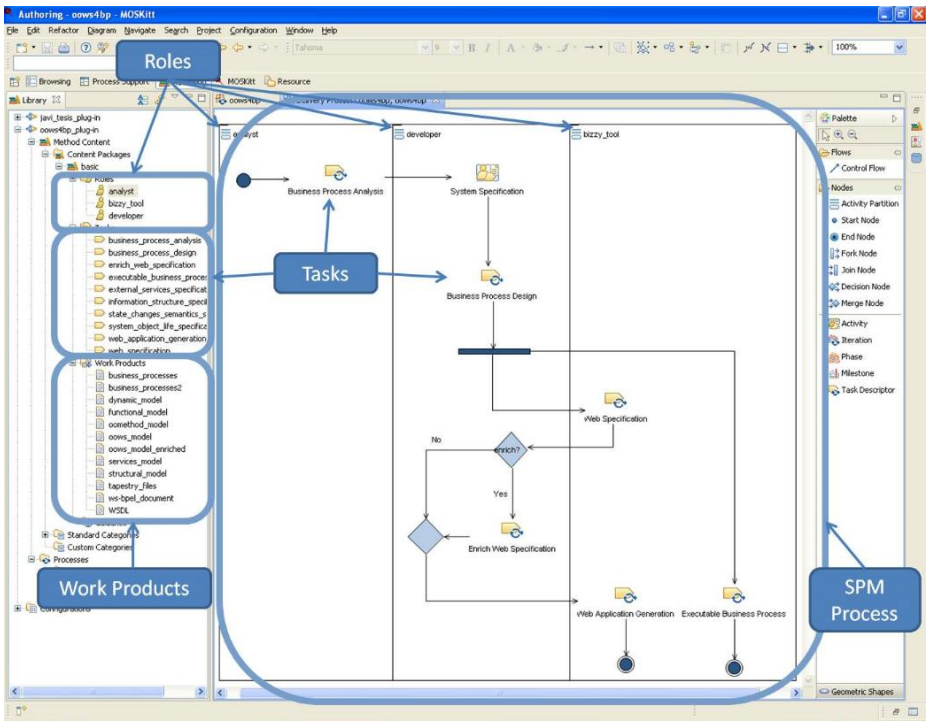

Fig. 4.2. EPF Composer editor in MOSKitt

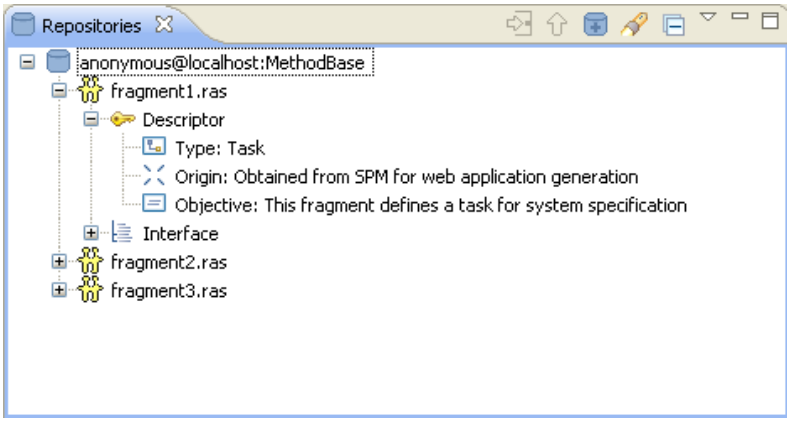

Fig. 4.3. Repository client (Method Base)

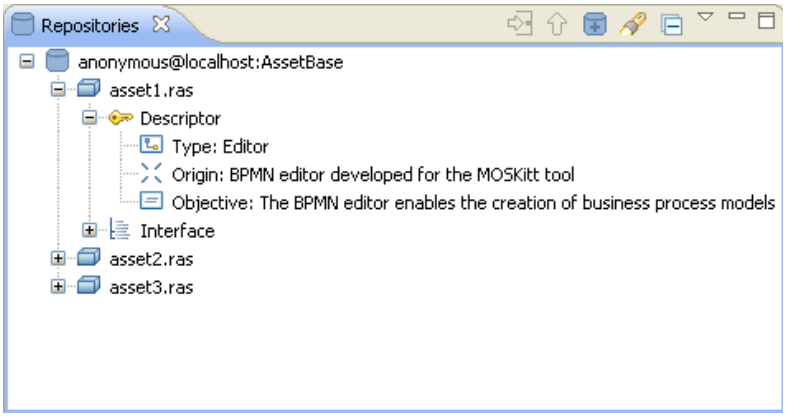

Fig. 4.4. Repository client (Asset Base)

Enactment component

A process engine has not been integrated into the prototype to guide method engineers during the method definition. Instead, two eclipse cheatsheets have been defined to assist during the method design and configuration phases of the methodological framework.

Transformation engine

In order to support the execution of the model transformation that generates the CASE tool support from method models, Xpand has been installed in the prototype. The Xpand plugins implement, among other things, the transformation engine that supports the execution of Xpand transformations.

Specifically, the model transformation has been implemented in the prototype as a M2T transformation that takes as input a SPEM model and obtains a product configuration file through which a MOSKitt reconfiguration supporting the method is obtained. As an example, two Xpand rules of the transformation are shown in Fig. 4.5. In these rules the list of features[4] of the product configuration file is generated. The first rule is invoked for each instance of the SPEM class *ContentElement* (i.e. tasks and products). This rule invokes the second rule, which produces the output. The second rule accesses the property "FeatureID" of the content elements. This property is created

[4] A feature is a group of Eclipse plugins

during the technical fragment association and contains the identifier of the feature packaged in the fragment.

```
«DEFINE contentElement FOR uma::ContentElement-»
«EXPAND asset FOREACH this.assets.typeSelect(uma::ReusableAsset)-»
«ENDDEFINE»

«DEFINE asset FOR uma::ReusableAsset-»
«FOREACH this.methodElementProperty AS property-»
«IF property.name == "FeatureID"-»
<feature id="«property.value»" version="0.0.0"/>
«ENDIF-»
«ENDFOREACH-»
«ENDDEFINE»
```

Fig 4.5. Excerpt of the M2T transformation

Technical fragments

Technical fragments are editors, transformations, etc. that provide support to the product part of the method in the generated CASE tools. These fragments are stored in the Asset Base repository as reusable assets that contain the Eclipse plugins that implement the encapsulated tool and the feature that groups these plugins (see Fig. 4.6). In order to install these plugins in the CASE tools, the M2T transformation must include in the product configuration file the features encapsulated in the fragments. This is done in the rules shown in Fig. 4.5.

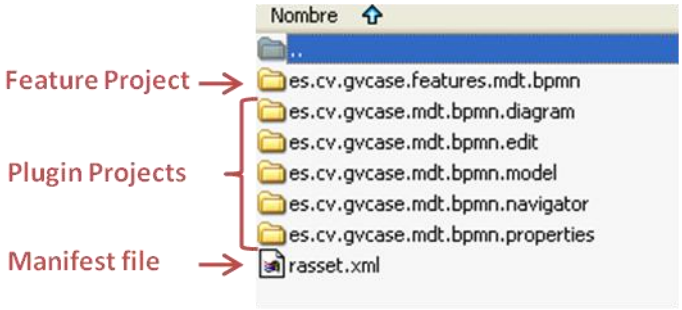

Fig. 4.6. Technical fragment

Process Engine

The process engine is the component in charge of the execution of method instances, that is, it gives support to the process part of the method in the

generated CASE tools. Up to now, the process engine has been implemented in MOSKitt4ME as a light-weight process engine that keeps the state of the running method instances. As future work, the integration of the Activiti engine [Activiti] into MOSKitt4ME is being planned. The use of Activiti will require the definition of a model transformation to map SPEM models into BPMN 2.0 models that can be executed by the engine.

Project Manager Component

The Project Manager Component endows the generated CASE tools with a graphical user interface composed of a set of Eclipse views (see Fig. 4.7[5]). Each of these views provides a specific functionality but their common goal is to facilitate the user participation in a specific project. The details of these views are the following:

- *Product Explorer*: This view shows the set of products that are handled (consumed, modified and/or produced) by the ongoing and finished tasks of the process. This view can be filtered by roles so that users belonging to a specific role have only access to the products they are in charge of. Then, from each product, the user can open the associated editor to visualize or edit its content.
- *Process*: This view shows the tasks that can be executed within the current state of the project. The execution of the tasks can be performed automatically (by launching the transformation associated to the task as a technical fragment) or manually by the software engineer (by means of the software tool associated to the output product of the task). Similarly to the Product Explorer, this view can be filtered by role, showing just the tasks in which the role is involved in.
- *Guides*: This view shows the list of guides associated to the task selected in the Process view. The objective of these guides is to assist the user during the execution of such task, providing some insights on how the associated products should be manipulated. These guides correspond to technical fragments that were associated to tasks during the method configuration phase.

---

[5] Also available at https://users.dsic.upv.es/~vtorres/moskitt4me/

- *Product Dependencies*: This view shows the dependencies that exist between the products that are handled in the project. So, it allows users to identify which products cannot be created or manipulated because of a dependent product has not yet been finished. In addition, these dependencies are organized by roles. This organization gives to the user the knowledge of who is responsible of those products he/she is interested in.

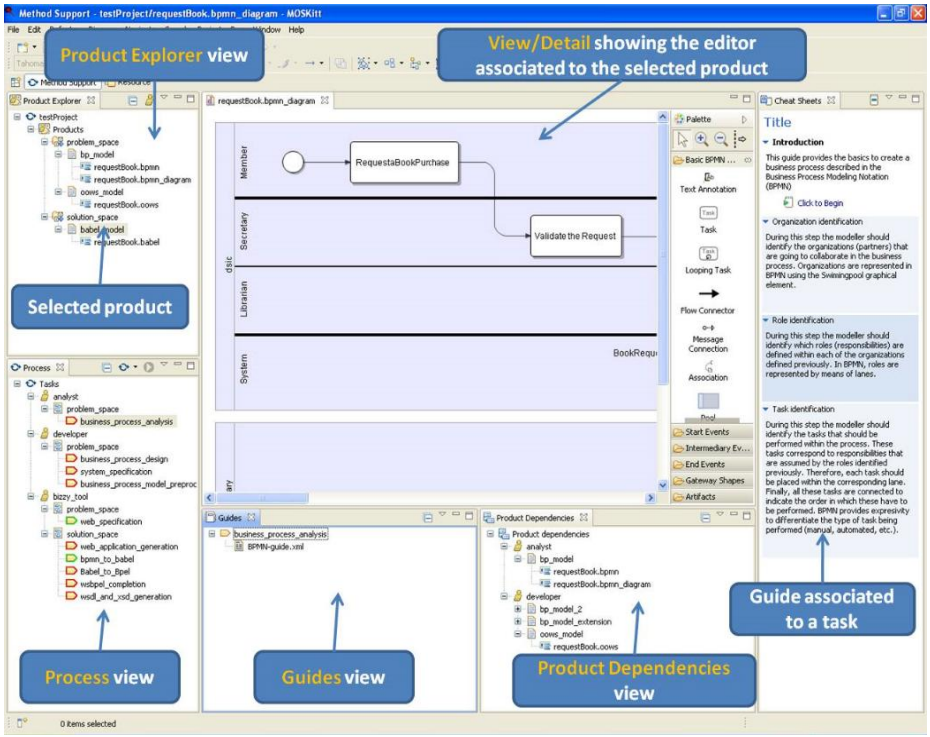

Fig. 4.7. Project Manager Component

4.3. Conclusions

Developing software systems is a highly complex endeavor and CAME and metaCASE environments are no exception. A solution that properly handles this complexity is software architecting. One of the main benefits of a software architecture is that it provides an abstraction of the system that establish how it must be structured and, thus, allow developers to focus only

on those elements that are significant. Therefore, in order to reduce the complexity that entails the development of tools that support Method Engineering, this chapter proposes a software architecture that establishes the series of components that are required to support the methodological framework presented in chapter 3.

Furthermore, a vertical prototype called MOSKitt4ME has been developed in the context of the MOSKitt platform as an implementation of the architecture. The development of this prototype has a threefold benefit. First, it helps to evaluate the proposed architecture. Secondly, it sets the basis for the eventual development of a complete CAME environment. Finally, stakeholders within the MOSKitt community can use the prototype and provide feedback that can be used for the refinement of the architecture and the methodological framework.

References

[Activiti] Activiti, http://www.activiti.org/

[BPMN] Business Process Model and Notation (BPMN). OMG Available Specification version 2.0. OMG Document Number: dtc/2010-06-05

[EMF] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[EMP] Eclipse Modeling Project, http://www.eclipse.org/modeling/

[EPF] Eclipse Process Framework Project (EPF), http://www.eclipse.org/epf/

[MOSKitt] MOSKitt, http://www.moskitt.org/

[Niknafs08] Niknafs, A., Ramsin, R.: Computer-Aided Method Engineering: An Analysis of Existing Environments. CAiSE, 525-540 (2008)

[PDE] Plug-in Development Environment, http://www.eclipse.org/pde/

[Xpand] Xpand, http://www.eclipse.org/modeling/m2t/?project=xpand