JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 4, May-June 05

Templates and Resources in Software Development Methodologies

Cesar Gonzalez-Perez, University of Technology, Sydney
Brian Henderson-Sellers, University of Technology, Sydney

Abstract

A great deal of effort is needed to construct software products in a predictable and repeatable manner. Having a precisely defined methodology in place can certainly help, especially if it includes the comprehensive specification of the process to be followed and the work products to be created. However, a convenient integration of these two aspects (process and work product) has not yet been performed. This paper presents a new approach to the definition of methodologies that supports the process and work product domains concurrently through the specification of discrete methodology elements. Some of these elements, called here templates, are designed to be instantiated during the use of the methodology in specific projects, while others, called resources, are intended to be used directly. Theoretical and practical implications of this division, especially regarding metamodelling and the use of powertypes, are explored. The proposed metamodelling approach is shown to facilitate the precise and complete specification of comprehensive methodologies, establishing the foundations for predictable and repeatable results from software development.

1 INTRODUCTION

The task of defining and describing a software development methodology must be approached with care, since ambiguities or omissions in its definition will certainly lead to vagueness in its enacted instances[1] and thus hinder its ultimate usability and usefulness. In order to achieve an acceptable degree of formality, precision and completeness, we must first understand *what* a methodology is. Although some authors identify methodology with process[2], we prefer to adhere to a much broader view and consider a software development methodology as *the specification of the process to follow as well as the work products to be generated, plus consideration of the people and tools involved, during a software development effort.* From this definition, a methodology

[1] An "enacted instance" refers to the process being used on a real project with real team members and real deadlines – as opposed to a methodology as defined in a handbook, applicable to many projects.
[2] The Catalysis approach offers "how to" guidelines plus techniques; see [5], p xx-xxi. Martin & Odell define *methodology* as a collection of methods, and *method* as a procedure; see [12], chapter 1.

Cite this article as follows: Gonzalez-Perez and Henderson-Sellers: "Templates and Resources in Software Development Methodologies", in *Journal of Object Technology*, vol. 4, no. 4, May–June 05, pp. 173-190 http://www.jot.fm/issues/issue_2005_05/article5

therefore comprises elements relevant to both the process domain and to the work product domain.

Overall, a methodology can be formally specified as a collection of interrelated methodology elements. Clearly, some of these elements must belong to the process domain, while others correspond to the work product domain. Enacting the methodology for a particular project means using the defined methodology elements[3] in specific ways. We will introduce (Section 3) the notion that some methodology elements (called "templates" here) are used by being instantiated from the methodology into project-specific elements, while others (named "resources") are used without instantiation, thus being directly applied to the project. This distinction is necessary to accommodate different types of methodology elements as detailed in the following sections.

The next section explains our approach to methodology definition based on the use of templates and resources, a necessary precursor for Section 3, which describes in detail how templates and resources work. In turn, this leads to some interesting metamodelling implications, which are discussed in Section 4. Section 5 then shows an architectural (i.e. static) description of an example metamodel including the necessary mechanisms to support templates and resources distributed across the process and work product domains. Finally, our conclusions are presented.

[3] Note that the argument presented here is independent of whether the methodology is to be constructed by the user from the methodology elements by means of method engineering (see [4] for an example) or whether the methodology is provided to the user as a single, pre-constructed entity by a methodology vendor.

2 DEFINING A METHODOLOGY

As we have already stated, the definition of methodology used here encompasses both a process domain and a work product domain. Also, a methodology is formally specified as a collection of elements that are distributed between the aforementioned domains. A point that is often missed is that the specification of the work products to be generated must be accompanied by the definition and description of the atomic modelling units that are to be used to construct such work products. Using a grammatical parallel, process elements can be viewed as verbs and work products as nouns. Because most "verbs" in a methodology are transitive, it is necessary to take into account the grammatical objects (noun-like) they act upon in order to obtain a complete and meaningful result. Therefore, the defined process (especially beyond a certain level of detail) must take into account the objects of its actions, i.e. the model units used to construct work products, in order for the methodology to attain a high degree of integration and cohesiveness. As an example, consider the following fragment of a process definition: "construct a class model". Any methodology containing such an indication must also describe what a class model is before any details regarding its construction can be offered. A shallow explanation for "class model" such as "the collection of classes and relationships that represent the structure of the system" is inadequate since (a) the model units "class" and "relationship" used in the explanation are not defined and (b) we know from practice that many additional kinds of model units may be necessary in order to complete a class model, such as interfaces, attributes, operations, roles etc. To make things worse, some of these kinds of model units (typically operations) are not to be added to the class model at this stage, but later, when a much richer and more expressive definition of the links between the process and work product domains is needed.

Interestingly, the often quoted term "object-oriented methodology" frequently addresses only the epistemological issue of using objects (and perhaps the very ontology[4] of software-intensive systems) that belong to the work product domain but, surprisingly, does not refer to the process domain. Paradoxically, object-oriented, process-focussed methodologies usually define the process elements but include little or nothing related to work products. This contradiction must serve as a call for attention toward the need for a complete and holistic approach to methodology definition, one that defines a methodology as a collection of method fragments (e.g. [4], [15]) or of interrelated methodology elements, distributed into stages, work units, techniques, actions, work products, model units, languages and notations [11]. For example, well accepted modelling languages such as UML [14] deal with modelling issues but neglect process, while widespread methodological frameworks such OPEN ([8]) or Extreme Programming ([3]) emphasize the process side and are less detailed when it comes to work products, in the sense that they usually have a pointer to an external modelling language package or product such as the UML. While modularity and decoupling issues are often used to

[4] We are using this term with its primary and most appropriate connotation, i.e. the study of being itself.

argue for such an imbalance, methodology definition could probably be considered as one of those wicked problems[5] for which the *what* and the *how* cannot be approached separately. We propose a revision of the traditional approach to one providing a comprehensive set of methodology elements that cover the whole spectrum of needs, potentially attaining the richness of UML on the modelling side and, at the same time, the power of OPEN on the process side, together with a neat integration of them both.

It is also appropriate at this point to note the approach taken by SPEM [13] in the sense that it is not sufficient for our purposes of an integrated approach to methodology. First of all, SPEM only addresses process issues, neglecting product and modelling needs. Although a WorkProduct class exists in SPEM, a complete methodology needs to describe the products used and created with finer granularity than this. Secondly, the connection between WorkProduct and process-related entities (such as WorkDefinition) is not expressive enough, since it relies on an input/output characterisation when real world applications need richer semantics, ideally including an extensible set of product-process interaction types and the capability to support constraints. Finally, SPEM does not distinguish between *what* must be done in a process and *when* it is done, encapsulating both issues in the same element, namely Activity. Using this approach, is not possible to define some work to be done without being forced to specify, as well, when (in the lifecycle) it must be done. In contrast, for example, OPEN uses the metaclass Activity for the *what* and the metaclass Stage for the *when*. For all these reasons, we must conclude that SPEM is not a suitable solution for the definition of methodologies.

Using our new, more holistic approach to methodology definition, we note that methodology elements are now the *only* component of a methodology specification; that is, no other information apart from them is needed to formally define and describe the methodology. Also, methodology elements are objects subject to the conventional rules for objects in an object-oriented environment: they possess identity, they may carry values (corresponding to attributes) and they may be linked to other methodology elements (as defined by associations). Being objects, methodology elements must be instances of some classes; this issue is discussed in Section 4. Figure 1 shows an example fragment of a methodology specification in the form of a UML object diagram. The idea of dealing with methodology elements as objects is interesting for several reasons. First of all, methodology elements exhibit typical object characteristics, as we have already mentioned; in addition, and from a method engineering point of view [4], as objects they are just as valid and useful as are objects representing automobile parts for a car manufacturer or functions and matrices for a mathematician. Finally, methodology elements are nicely managed by CASE tools as objects in a repository [16].

[5] Wicked problems are described in [6].

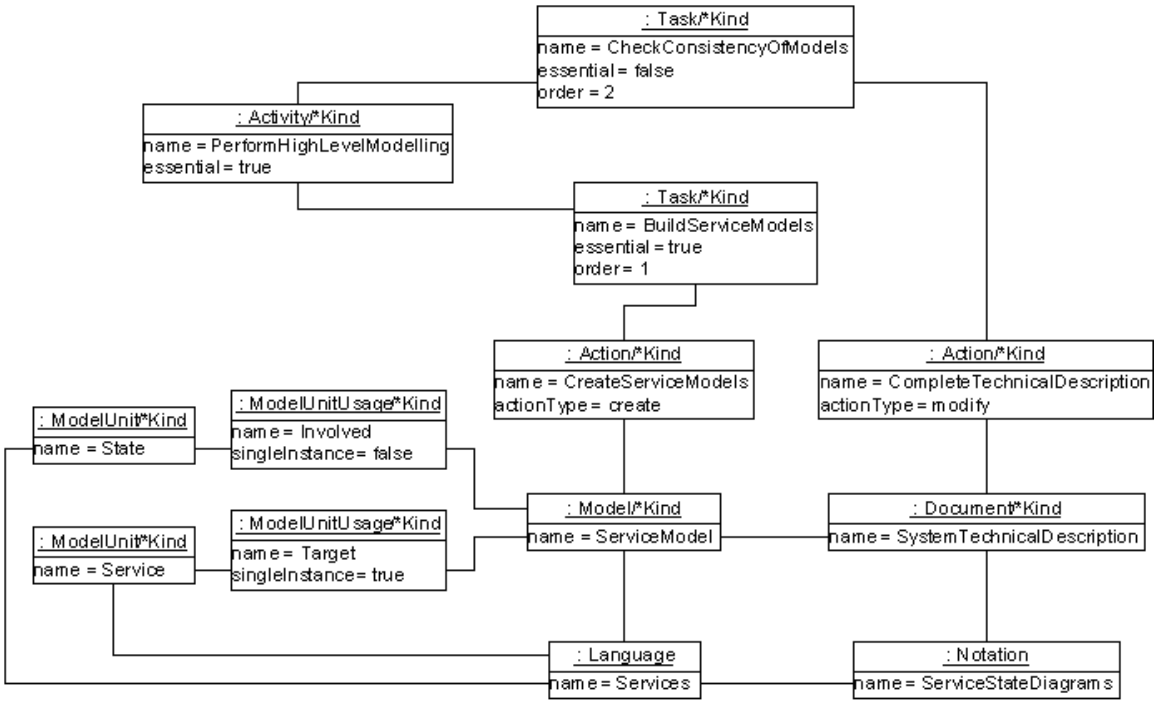


Figure 1. Sample fragment of a methodology. Methodology elements are shown as objects. Class names with a "/*" in them are actually powertype patterns and are explained in Section 4.

3 TEMPLATES AND RESOURCES

Let us now consider how a methodology is utilized. Usually, methodologies are applied to different projects, each of them being run by different teams and having different timeframes. As noted earlier, the action of applying a methodology to a specific project is called *enactment*. Enacting a methodology involves using the existing methodology elements to create *project elements* and, eventually, develop the targeted software system (Figure 2). Project elements, in turn, are elements that exhibit object characteristics at the project level; they may belong to the process or work product domains[6].

[6] Project elements are called "project entities" in [8]. We refer to the same thing, and will use "elements" henceforth.

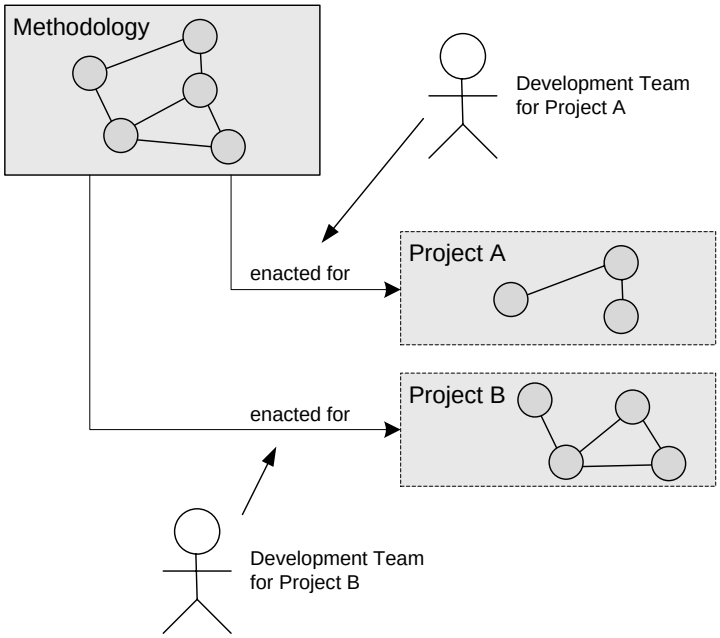


Figure 2. Enactment of a methodology for different projects. Methodology elements are shown schematically as circles inside the rectangle depicting the methodology. Project elements are shown schematically as circles within each project.

Some examples of process-related project elements are tasks and stages (performed by specific people by specific dates); some examples of work product-related project elements may include models (representing a particular view of the system to be built, and created by specific authors) and classes (representing specific concepts of the system's structure). Figure 3 shows an example fragment of a project being performed. Each project element is depicted as an (anonymous) object. The task being performed is that of building service models, commencing on 5/11/02 and with a stated termination date of 18/11/02. This task has performed an action consisting of creating a specific service model. The result of this action being performed is the service model with name "Service Model 12". It is version number 3 written by Terry and John. This model describes the service of printing a document, which includes two associated states: actually printing the document (here denoted as of type "busy") and that of showing an options window (denoted as of type "modal").

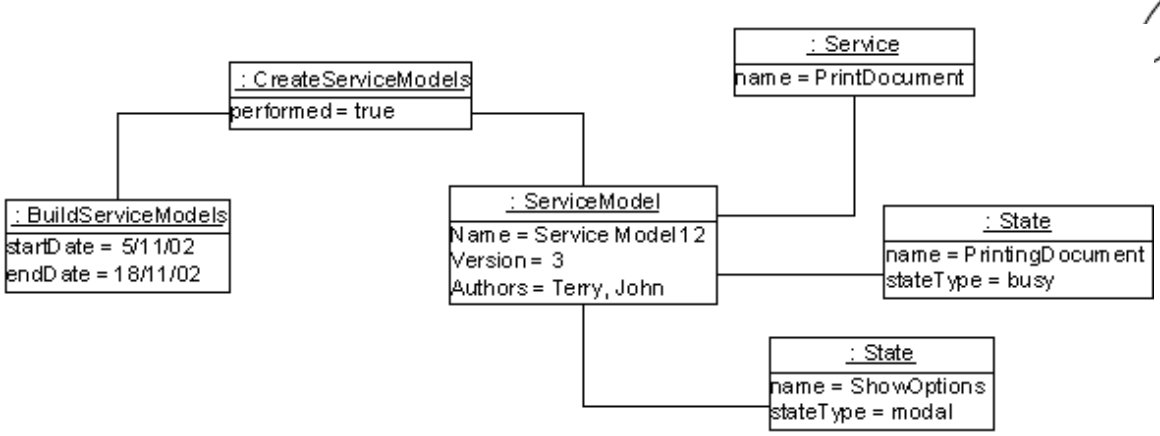


Figure 3. Sample fragment of a project being performed. Project elements are shown as objects.

Obviously, a strong connection exists between project elements and methodology elements. This relationship is often described as a conventional "instance of" dependency ([8], p 61, for example), but we believe that it is often more complex than that. It is true that project elements are created by instantiating some methodology elements, such as introducing a new attribute in the class model by instantiating the `Attribute` class in the methodology, or defining a new task to be performed by instantiating the `Task` class in the methodology. Figure 4 shows the same project elements as in Figure 3, but including the explicit connection to the methodology elements.

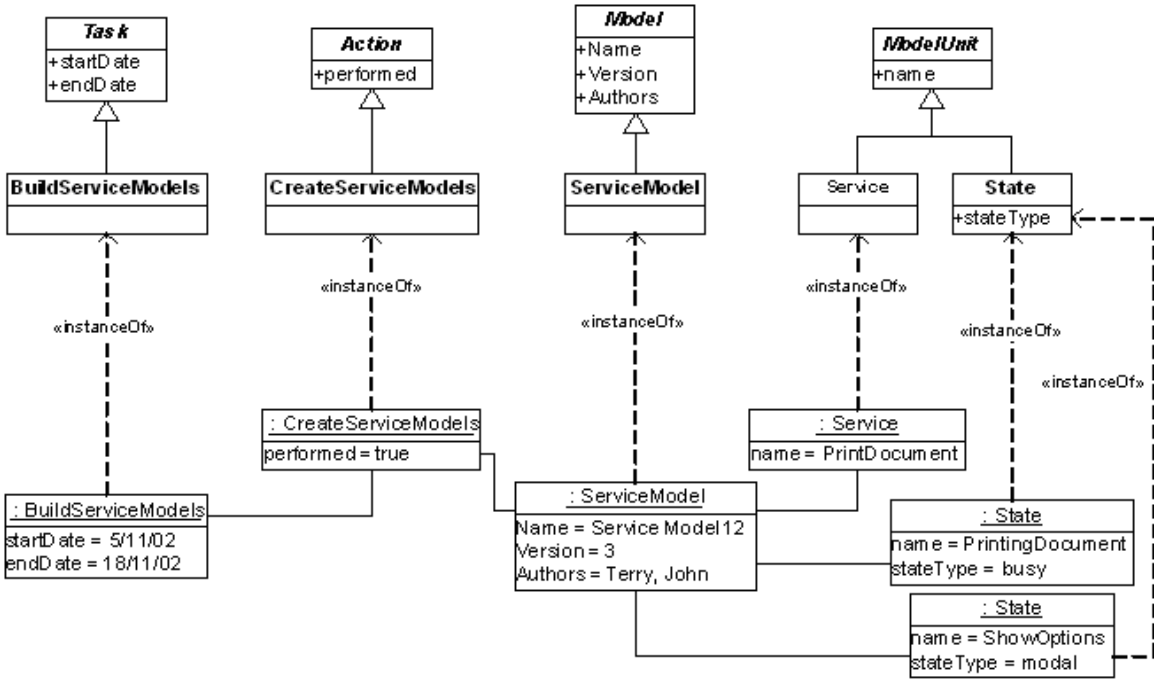


Figure 4. The same project elements as in Figure 3 are shown, but now their relationships to the methodology elements from which they are instantiated are also included. Metamodel elements such as Task and Action are also shown to help contextualize the latter. The generalizations between methodology elements and metamodel elements are explained in Section 4.

The very act of instantiating methodology elements to create new project elements needs some extra information that cannot be found in the methodology elements being instantiated, such as guidelines for the proper use of the methodology and the specification of notational artefacts to depict the aforementioned project elements. If we assume our already stated principle that the whole specification of a methodology must be done through methodology elements, such guidelines and notations *must* also be methodology elements but are *not* created by an instantiation mechanism. We must conclude, therefore, that some methodology elements are instantiated during enactment, while others (such as guidelines and notations) are not. We call the methodology elements that are *instantiated* into project elements *templates*, while those that are used directly without being instantiated are named *resources*. From conventional object-oriented wisdom, we can deduce that templates must be classes if they are intended to be instantiated; however, they also must be objects, since all methodology elements are objects. Therefore, *template methodology elements are simultaneously classes and objects* (see further discussion below and in [1]). Resource methodology elements, on the other hand, are simple objects, since they are not intended to be instantiated.

The dual facet of templates can be easily recognized through some examples (see Figure 5). Within the process domain, the concept of the "BuildServiceModels" task kind is represented as:

- An *object*, since it has identity (it is, after all, the "BuildServiceModels" task kind, as opposed to, say, the "DefineOperations" task kind) and it has attribute values (`name` = BuildServiceModels, `essential` = true).
- A *class*, since it has attributes (`startDate, endDate`) and associations (`creates`), serving as a template for actual tasks that create service models during the project.

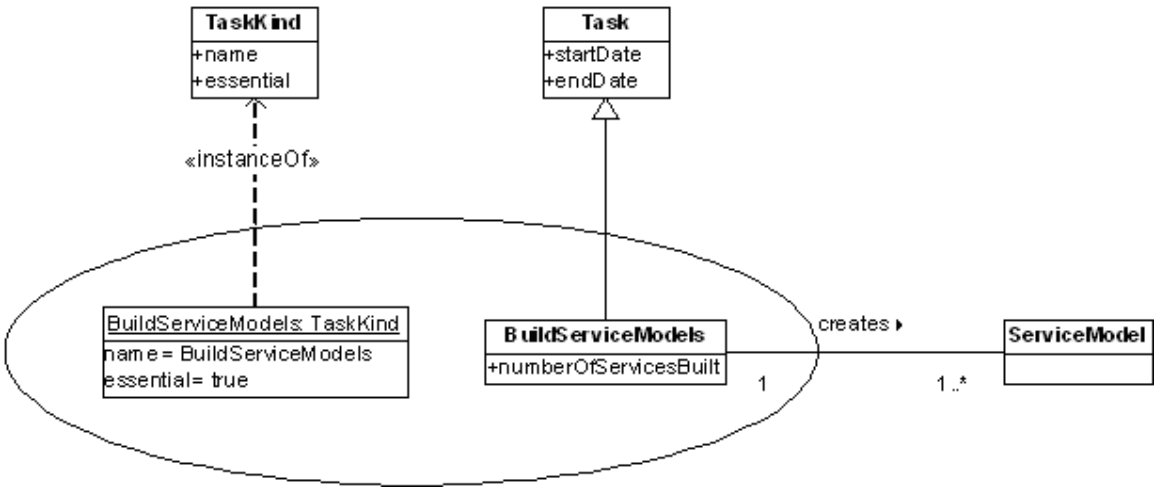
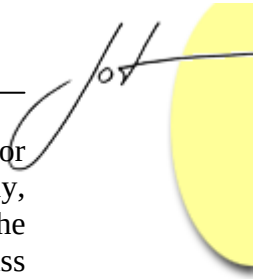


Figure 5. An example of the dual facet of templates. There is a "BuildServiceModels" object (inside the ellipse), which is an instance of TaskKind, and a BuildServiceModels class (also inside the ellipse), which is a subtype of the Task class.



The concept of "clabject", as introduced by Atkinson & Kühne in [2], is ideal for describing such dual-faceted entities; a *clabject* is an entity that can exhibit, concurrently, a type (or class) facet and an instance (or object) facet. For example, the `BuildServiceModels` template methodology element is a clabject, since it has a class facet (is instantiated into actual tasks that build service models during the project) and an object facet. While templates are clabjects, resources are not, as they can be described as simple objects, since they do not need the type facet. They exist at the methodology level, probably being linked to other methodology elements (both resources and templates, through their object facet) and are used during enactment as reference or guidance – but they are not instantiated.

4 METAMODELLING IMPLICATIONS

Describing methodologies in the context of an underpinning metamodel is a widespread practice that adds formality to the methodology definition and allows for its extension and adaptation. From this perspective, methodology elements are usually viewed as instances of their respective metamodel elements; for example, OPEN defines "Develop iteration plan" (a task at the methodology level) as an instance of `Task`, a metamodel element (see [8], p 264).

Although we agree that methodology elements must be defined as instances of some metamodel elements, we must make an interesting point here. Continuing with our example, the `Task` metamodel element in OPEN is instantiated during process construction into "instances of `Task`", i.e. kinds of tasks ready to be enacted. However, from an intuitive point of view, the *tasks* are those performed by actual people during the project, not the abstract definition at the methodology level. We therefore suggest using the name `Task` for project elements and use instead `TaskKind` for the methodology element to avoid confusion[7]. Following this assumption, "Develop iteration plan" and "Keep client informed" are not tasks, but *task kinds*. Every single enactment of one of these task kinds, with actual people and dates, is a *task*. Both concepts `Task` and `TaskKind` exist at the metamodel level; task-related methodology elements are instances of `TaskKind` and, simultaneously, subtypes of `Task`. An actual task at the project level is an instance of one specific subtype of `Task`. We must note, however, that only template methodology elements are subject to this condition; as noted earlier, resources are simple objects obtained through conventional instantiation of metamodel elements (Figure 6).

The generalization of this example to the whole methodology leads to the notion of "powertype-based metamodelling" [10]. From this perspective, template methodology elements are instances of metamodel classes named with a "kind" suffix (such as `TaskKind` or `ModelKind`), to indicate that they represent specific kinds of things. For example, the `DefineOperations` methodology element is an instance of `TaskKind` representing an abstraction of every single task that defines operations. Simultaneously, `DefineOperations` is a subtype of `Task`, since all tasks defining operations are, by

[7] In this context, *tasks* as defined by OPEN would be better named as *task kinds*. The same conversion must be applied to most metamodel elements.

definition, tasks. `Task` and `TaskKind` compose a *powertype pattern* at the metamodel level ([9], section 3.2). Powertype patterns are *pairs of classes in which one of them (the powertype) partitions the other (the partitioned type) by having the instances of the former be subtypes of the latter*. Note that powertypes, by definition, cross the *traditional* levels of a metamodelling hierarchy. In our example (see Figure 6), `TaskKind` is a powertype and `Task` is the associated partitioned type. When using UML to depict powertype patterns, two separate classes (for the powertype and the partitioned type) are sometimes used. However, it is often convenient to use a single class to represent the whole powertype pattern; in such cases, the class can be named as `<name>/*Kind`, where `<name>` corresponds to the partitioned type's name. For example, the `Task/TaskKind` powertype pattern would be depicted as a single class labelled `Task/*Kind`.

Finally, and since every methodology element must be derived from some metamodel element, we must enhance conventional metamodelling approaches in order to support clabjects. A clabject can be defined as an "instance" of a powertype pattern if we agree to (a) extend the customary meaning of the "instance of" relationship and (b) deal with powertype patterns as single entities when convenient. Assuming this, the object facet of a clabject is a conventional instance of the powertype class in the powertype pattern, while the class facet of the clabject is a subtype of the partitioned type class in the powertype pattern.

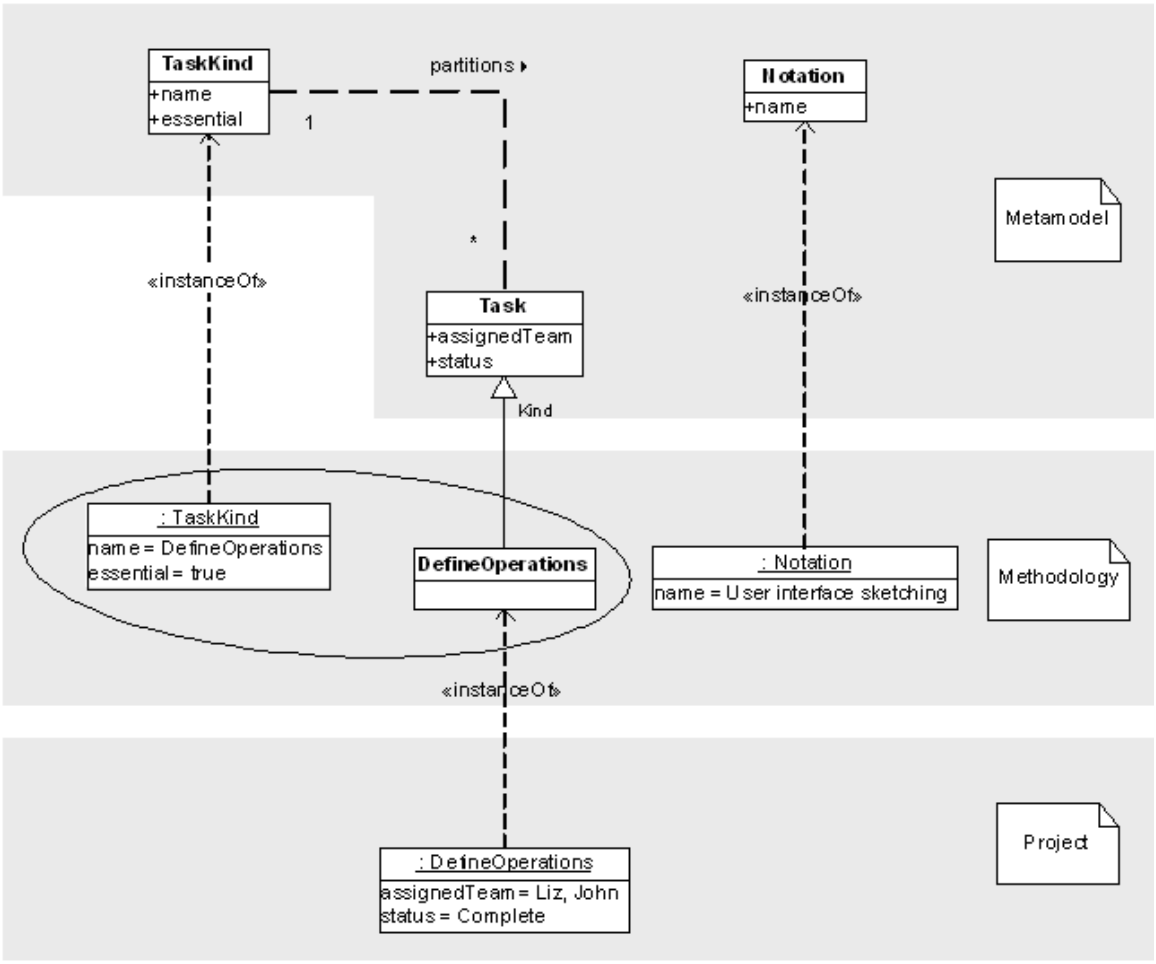


Figure 6. Example of relationships between metamodel, methodology and project levels. The "Define operations" template methodology element is an instance of TaskKind in the metamodel, and also a subtype of Task. Specific tasks defining operations performed at the project level are instances of such a subtype of Task. The ellipse at the methodology level represents the fact that the included class and object are two facets of the same clabject. The "User interface sketching" resource methodology element is an instance of Notation in the metamodel. (After [9]).

5 AN EXAMPLE METAMODEL

Taking the previous sections as an expression of what a comprehensive metamodel should offer, we would like to outline a specific example as a partial "validation" of the theoretical discussion above. From our perspective, a metamodel must allow the method engineer to exert some control over the project elements, as well as on the methodology elements. Our example metamodel includes a `MethodologyElement` class, which acts as an abstract type for all methodology elements, and a `ProjectElement` class, of which project elements would be indirect instances. Since every project element is

derived from a certain methodology element, these two classes are arranged into a powertype pattern, as shown in Figure 7. Templates and resources are modelled as abstract subclasses of `MethodologyElement`. In addition, `UserAttribute` and `UserAssociation` classes are provided to allow the method engineer to add attributes and associations to the class facet of template methodology elements. Conventional instantiation mechanisms used to generate methodology elements from metamodel classes do not support the manipulation of attributes, associations or classes at all at the methodology level, so the `UserAttribute` and `UserAssociation` classes are necessary at the metamodel level.

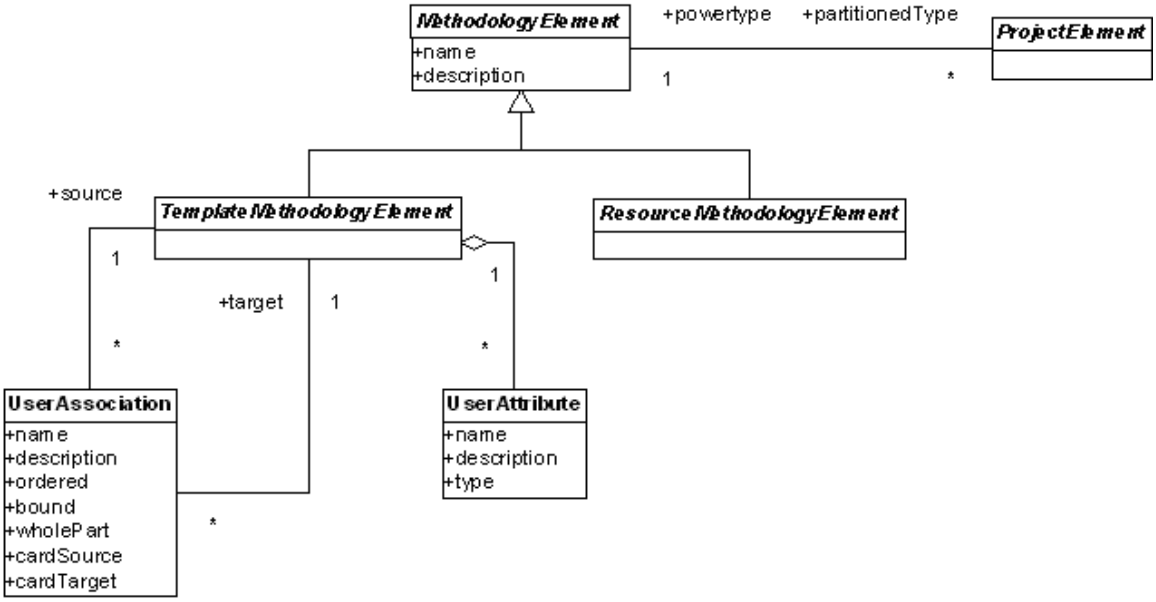


Figure 7. Very high-level view of the example metamodel. Methodology elements and project elements compose a topmost powertype pattern, shaping the linkages to be kept between metamodel, methodology and project. User attributes and user associations allow for customization of the class facet of template methodology elements.

From the described framework, we can derive more concrete classes. Specific kinds of template methodology elements are introduced, accompanied by their respective partitioned type classes (Figure 8). Classes used to model resources are also introduced.

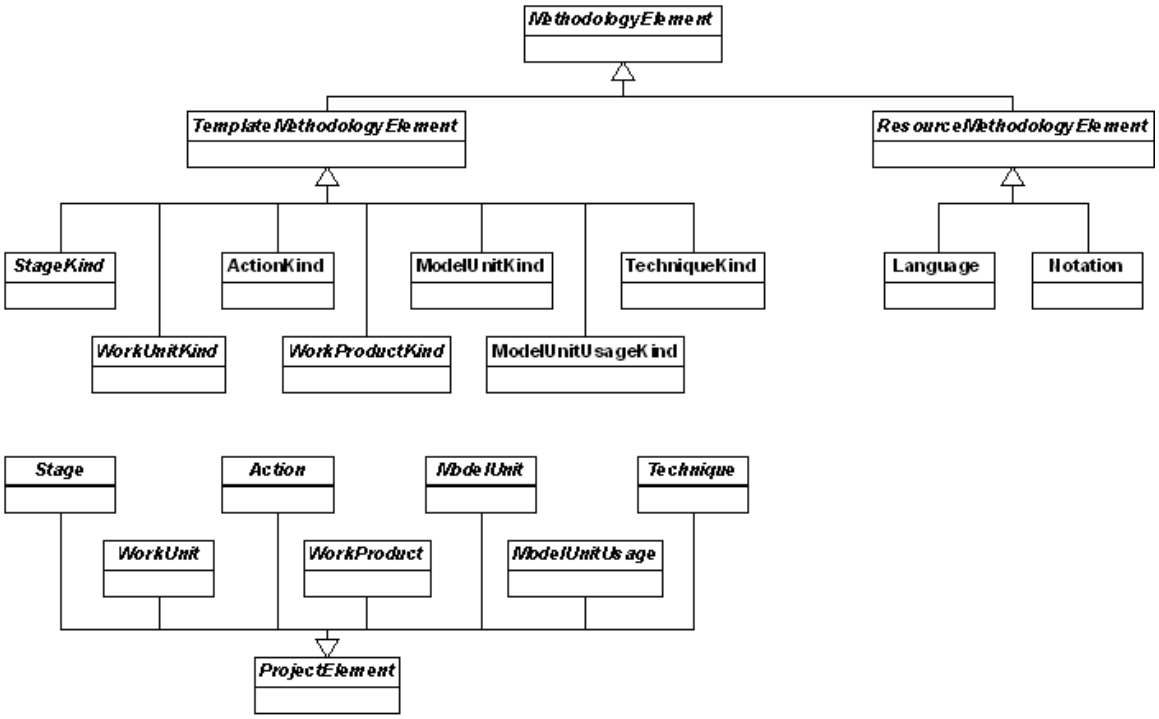


Figure 8. The example metamodel at an intermediate abstraction level. The UserAttribute and UserAssociation classes have been removed for clarity, as well as the powertype association between MethodologyElement and ProjectElement. Powertype associations also exist (but are omitted here for clarity) between StageKind and Stage, WorkUnitKind and WorkUnit, etc.

Finally, some additional classes must be introduced to provide support for every single type of methodology element. Associations between classes must also be incorporated. Figure 9 shows a detailed view of the resulting structure.

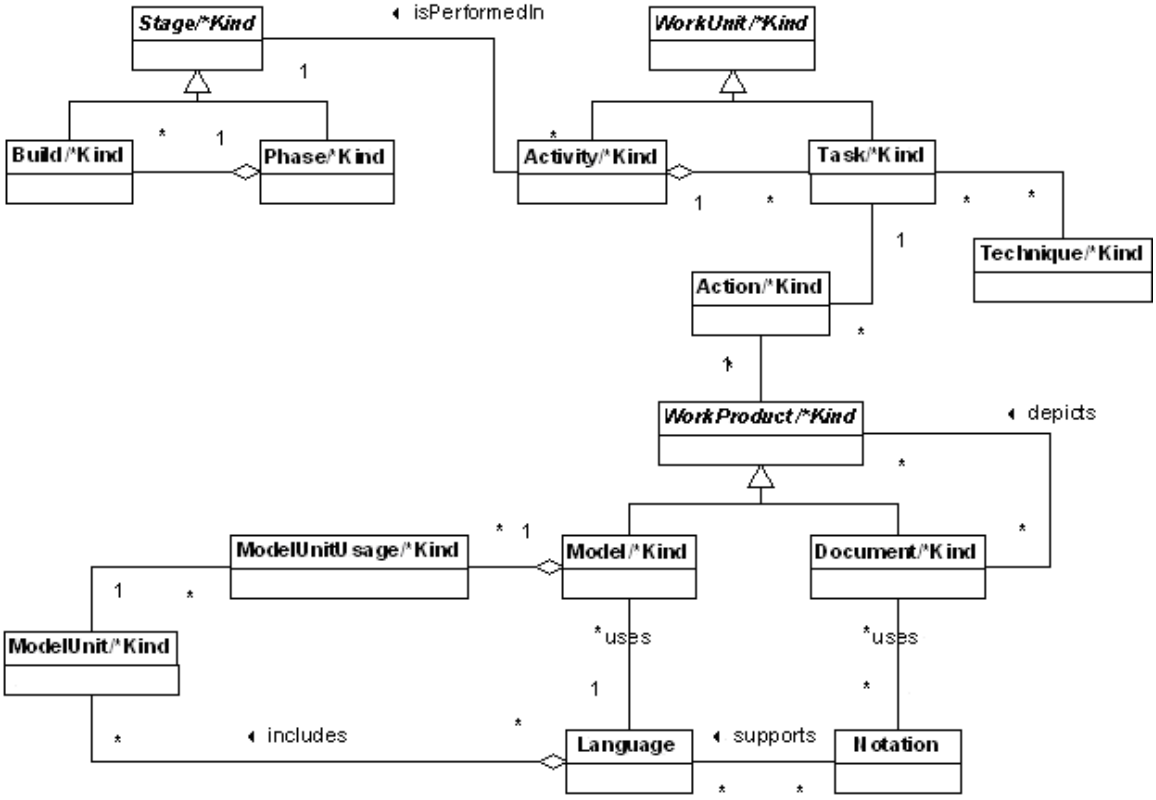
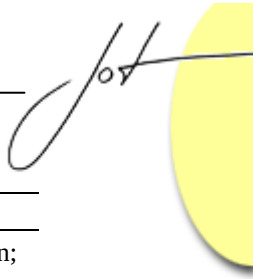


Figure 9. Low-level view of the example metamodel. Only instantiable classes (and their direct supertypes) are shown. Associations between classes establish a highly abstract structure for any methodology derived from the metamodel. (After [9]).

In summary, Table 1 shows a brief description of all the classes in the exemplar metamodel.

Class name	Description	Example instances
Action	A specific act or usage of a given work product by a given task.	Modifying the detailed class model when Designing class details for class “Invoice”
ActionKind	A specific kind of action.	Class Model can be modified by Design Class Details
Activity	A cohesive yet heterogeneous collection of tasks that achieves a set of related goals.	Specifying the requirements; Doing low-level modelling for feature set number 12
ActivityKind	A specific kind of activity.	Requirements Specification; Technological Design
Build	A scheduled part of a phase leading to an increment towards the final system.	Construction build number 132
BuildKind	A specific kind of build.	Construction Build



Class name	Description	Example instances
Document	A durable depiction of some of the problem's or system's properties.	System requirements description; Class description of class "Invoice"
DocumentKind	A specific kind of document.	Deployment Procedure; Class Description
Language	A set of interrelated model unit kinds, which can be used to construct certain model kinds.	User Interaction Language; Class Structure Language
MethodologyElement	An entity that exists at the methodology level, either a template methodology element or a resource methodology element.	Design Class Details (a task kind)
Model	A mental representation of the problem to solve or the system to build.	Domain class model; Persistence model for persistence cluster "Invoices"
ModelKind	A specific kind of model.	Class Model; Persistence Model
ModelUnit	An atomic unit used to compose models during a project.	Class "Invoice"; Attribute "Amount" of class "Invoice"
ModelUnitKind	A specific kind of model unit.	Class; Attribute; Operation
ModelUnitUsage	A specific usage of a given model unit on a given model.	Class "Invoice" is depicted in the Domain class model
ModelUnitUsageKind	A specific usage of a given model unit kind on a given model kind.	Operation is modelled in a Collaboration Model; Class is involved in a Class Model
Notation	A set of perceptible artefacts (usually graphical) plus usage rules, which can be used to depict specific model kinds.	User Interface Sketches; Class Diagrams
Phase	A usually long stage performed at a certain level of abstraction and focus.	Defining the system; Constructing the system
PhaseKind	A specific kind of phase.	System Definition; System Construction
ProjectElement	An entity that exists at the project level, either a stage, a work unit, an action, a work product, a model unit, a model unit usage or a technique.	Designing class details for class "Invoice" (a task)
ResourceMethodology Element	A methodology element designed to be used at the project level "as is", without being instantiated. It is either a language or a notations.	User Interaction Language
Stage	A managed interval of time, or a point in time, within a project.	Construction build number 132
StageKind	A specific kind of stage, either a BuildKind or a PhaseKind.	Construction Build; System Construction
Task	A single assigned job that creates or modifies one or more work products.	Defining system operations; Designing class details for class "Invoice"
TaskKind	A specific kind of task.	DefineOperations; Implement Exceptions

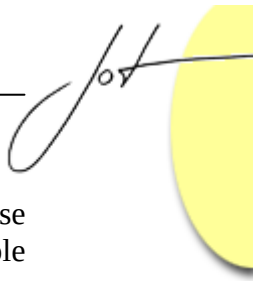
Class name	Description	Example instances
Technique	Usage of a specific way to perform a task.	Interviewing users on 5/11/02; Doing role modelling on 7/10/02
TechniqueKind	A specific kind of technique.	Interviewing; Role Modelling
TemplateMethodology Element	A methodology element designed to be instantiated to create project elements, either a stage kind, a work unit kind, an action kind, a work product kind, a model unit kind, a model unit usage kind or a technique kind.	Service Model (a model kind)
UserAssociation	An association between the class facets of specific template methodology elements.	Attribute “IsAbstract” of class “Class”; Attribute “Variations” of class “UseCase”
UserAttribute	An attribute of the class facet of a specific template methodology element.	Classes have Attributes; UseCases have UseCaseSteps
WorkProduct	A significant thing of value developed during a project.	Domain class model; Class description for class “Invoice”
WorkProductKind	A specific kind of work product, either a ModelKind or a DocumentKind.	Class Model; Deployment Procedure
WorkUnit	A functionally cohesive operation performed during a project.	Define operations for class “Invoice”
WorkUnitKind	A specific kind of work unit, either an ActivityKind or a TaskKind.	Requirements Specification; Define Operations

Table 1. Description of metamodel classes. The descriptions for most of the process-related classes are taken from [8].

Table 1 can be used as a reference for the classes in the example metamodel, summarizing succinctly the more detailed graphical depictions in Figure 7 to Figure 9. We can thus create metamodels specific to certain situations such as capability assessment, software development, computer-supported cooperative work (CSCW) or web development. Each metamodel can then be used to create methodologies useful to a particular organization or context. Such a methodology more closely describes, models and prescribes the steps and work products necessary to undertake the software development. It also clearly differentiates which project elements need to be instantiated and which can be used “as is” (templates cf. resources in the terminology used in this paper).

6 CONCLUSIONS

We have proposed a new approach to methodology definition, taking into account that both the process and work product domains must be described concurrently, and that both templates and resources must be supported at the metamodel level in order to accommodate the different types of methodology elements and also allow the method engineer to exert control on both methodology and project elements. Although some of these ideas have been dealt with in the literature for some time, no formalization of them



has been performed. We have presented in this paper a suitable formalization of these ideas. As validation of our approach to metamodelling, we have also defined an example metamodel that permits the precise specification of comprehensive methodologies.

ACKNOWLEDGEMENTS

We wish to thank the Australian Research Council for providing funding. This is Contribution number 04/05 of the Centre for Object Technology Applications and Research (COTAR).

REFERENCES

[1] Atkinson, C., 1998, Supporting and applying the UML conceptual framework, in *The Unified Modeling Language. «UML»'98: Beyond the Notation*, (eds. J. Bézivin and P.-A. Muller), LNCS 1618, Springer-Verlag, Berlin, 21-36

[2] Atkinson, C. and Kühne, T. 2000. Meta-Level Independent Modelling. *International Workshop on Model Engineering* at the *14th European Conference on Object-Oriented Programming 2000* (Sophia Antipolis and Cannes, France, 12-16 June 2000).

[3] Beck, K. 2000. *Extreme Programming Explained*. Addison-Wesley: Upper Saddle River, NJ, USA, 190pp.

[4] Brinkkemper, S. 1996. Method engineering: engineering of information systems development methods and tools, *Inf. Software Technol.*, **38(4)**, 275-280

[5] Brinkkemper, S., Saeki, M. and Harmsen, F. 1998. Assembly techniques for method engineering, *Procs. CAiSE 1998*, Springer-Verlag, 381-400.

[6] D'Souza, F. D. and Wills, A.C. 1999. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley: Upper Saddle River, NJ, USA, 785pp.

[7] DeGrace, P. and Stahl, L. H. 1990. *Wicked Problems, Righteous Solutions.* Yourdon Press: Upper Saddle River, NJ.

[8] Firesmith, D. and Henderson-Sellers, B. 2002. *The OPEN Process Framework — An Introduction*. Addison-Wesley: Harlow, UK, 330pp.

[9] González-Pérez, C.A. and Henderson-Sellers, B. 2003. A Powertype-based Metamodelling Framework. Submitted to *Software and Systems Modelling.*

[10] Henderson-Sellers, B. and González-Pérez, C.A. 2004. A Comparison of Four Process Metamodels and the Creation of a New Generic Standard. To appear in *Information and Software Technology*.

[11] Henderson-Sellers, B. 1995. Who needs an OO methodology anyway? *J. Obj.-Oriented Programming*, **8(6),** 6-8

[12] Martin, J. and Odell, J.J. 1996. *Object-Oriented Methods: Pragmatic Considerations*. Prentice-Hall: Englewood Cliffs, NJ.

[13] OMG. 2002. Software Process Engineering Metamodel Specification. OMG document formal/2002-11-14 [Online]. Available http://www.omg.org

[14] OMG. 2001. OMG Unified Modeling Language Specification, Version 1.4, September 2001, OMG document formal/01-09-68 through 80 (13 documents) [Online]. Available http://www.omg.org

About the authors

Cesar Gonzalez-Perez is a Post-doctoral Research Fellow at the Centre for Object Technology Applications and Research at University of Technology, Sydney (UTS), and has been developing and applying OO methodologies for over ten years to both research and commercial projects. He is the lead author of the OPEN/Metis methodology. E-Mail: cesargon@it.uts.edu.au

Brian Henderson-Sellers is Director of the Centre for Object Technology Applications and Research and Professor of Information Systems at University of Technology, Sydney (UTS). He is author of ten books on object technology and is well known for his work in OO methodologies (MOSES, COMMA and OPEN) and in OO metrics. He was recently awarded a DSc degree by the University of London for his work in object-oriented methodology. E-Mail: brian@it.uts.edu.au