Using UML for Modelling the Static Part of a Software Process[1]

Xavier Franch[1], Josep M. Ribó[2]

[1] Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
franch@lsi.upc.es
[2] Universitat de Lleida
P. Víctor Siurana 1, 25003 Lleida (Catalunya, Spain)
josepma@eup.udl.es

Abstract. We study in this paper the use of UML as a tool for modelling the process of software construction. As a case study, we deal with the process of building a library of software components. UML is used in order to define the static part of the process, i.e., the elements that take part on it and their structural relationships. We think that our approach supports some interesting properties in the field of software process modelling (e.g.: modularity; expressivity in model construction; sound formal basis; and flexibility in model enactment). Besides showing the adequacy of UML for modelling the static part, the paper outlines also some drawbacks concerning the description of the dynamic behaviour of the process using only UML, and some possible solutions to them.

1 Introduction

A model for a software development process (i.e., a *software process model* [5], SPM for short) is a description of this process expressed in some *process modelling language* (PML). The process can be viewed as the cooperation of many *tasks* (e.g.: requirements elicitation, component testing) that use and develop some *documents* (e.g.: specification, test plan) with the help of some *tools* (e.g.: CASE-tools, debuggers) and using some resources (e.g.: data bases, computer networks). Tasks involve many *agents* (e.g.: people, hardware media) which play specific *roles* (e.g.: programmer, manager) and which coordinate through some *communication* media (e.g.: e-mail, fax).

Hence, the definition of a SPM must state all the elements just mentioned, and also the way in which this model must be executed (*enacted*). This idea leads to the notion of *static* and *dynamic* parts of a model. The static part is given by means of a conceptual model that defines the elements that take part in the SPM. On the other hand, the dynamic part consists of a description of the way in which the model is enacted (e.g.: ordering of tasks). The systematic description of both parts not only helps in

[1] This work has been partially supported by the spanish project TIC97-1158, from the CICYT program.

understanding software development, but also allows the construction of systems for supporting automation of the process up to an acceptable level.

This topic has drawn a special attention within the scientific community and, as a result, several PMLs have been developed (see [6] and [5] for a survey). Currently a second generation of PMLs is coming into existence ([22], [23]) trying to fix some common drawbacks of most of former approaches: difficulty to express complex processes and to understand the resulting model; non-visual models or too naïve visual ones; difficulty to formalize the many facets of the process; use of one single language paradigm; etc. However, even taking this last generation into account, it is the case that most of them fail in a central issue, the use of standard, widespread notations and tools. It is a fact that this is one of the main reasons for which none of existing approaches has been widely adopted by the software engineering community.

We are developing a PML called PROMENADE (**PRO**cess-oriented **M**odelling and **ENA**ctment of software **DE**velopments) intended to be a part of this second generation of languages. PROMENADE, among other features (e.g. modularity, expressivity, formality and flexibility), aims at taking advantage of standard languages and tools in software engineering. One of such emerging languages is UML [20], which has acquired a great deal of interest in the last few years and which is becoming a standard *de facto* in software engineering (both in industry and in academia). This fact, together with its adequacy for modelling the structural elements of a software process, makes UML playing an important part in PROMENADE to describe the static part of a SPM.

In this article we propose, as a case study, the modelling of the construction process of a library of components in PROMENADE. We have chosen this particular process because it plays a central role in our research project, *ComProLab* (a Component Programming Laboratory, see [8]). We will focus on the description of the static part of such a process, made with UML.

In ComProLab, a *component* is defined by means of a *specification*, which includes two parts: a *functional specification*, stating how does the component behave, and a *non-functional specification*, that declares additional requirements referred to some operational *non-functional attributes* (as efficiency and reliability); these attributes are defined in *property modules*, which are imported in non-functional specifications. Once the specification is complete, an *implementation* may be built for this component, which is required to fulfill the properties stated in both parts of the specification. Implementations include a description of their *non-functional behaviour*, which determines the values that the operational attributes take in this implementation, possibly stating some additional constrains over implementations of the imported components. Fig. 1 shows the whole picture.

The rest of the paper is organised as follows. Section 2 presents both the PROMENADE metamodel and reference model, which acts as the basis on which any other PROMENADE model will be expressed. Sections 3 and 4 show the most relevant aspects of the static part of the modelled process (documents and tasks). Section 5 sketches some limitations of UML in order to model the dynamic part of a software process and outlines some possible solutions. Finally, section 6 provides the conclusions and some related work.

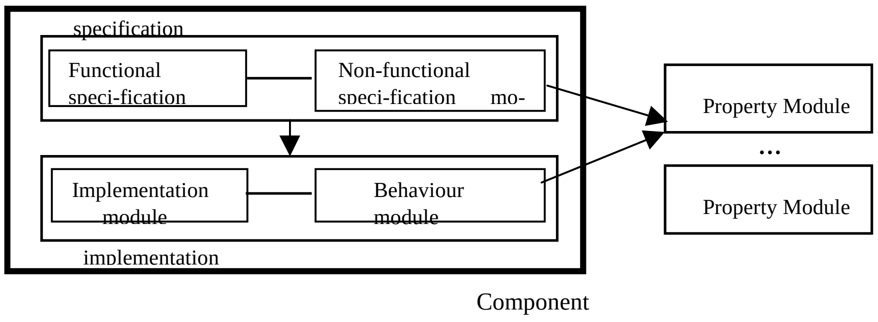


Fig. 1. Organization of a component in the ComProLab approach

2 The Metamodel and the Reference Model

The PROMENADE metamodel is built as an extension of the UML metamodel [18], adding a metaelement for each one of the core elements that take part in any PROMENADE model (e.g., it adds a *MetaTask* metaelement for tasks, a *MetaDocument* metaelement for documents, a *SPMetamod* metaelement for the model itself, etc.). The model elements (e.g., the class *SpecifyComponent* for the task of specifying a component) are seen as instances of the metaelements (e.g., *MetaTask*) whose (meta)features (metaattributes, metaoperations, etc.) have been given a value. Therefore the process of building a model in PROMENADE will consist mainly in creating instances for the metamodel classes and give values to its metafeatures.

The PROMENADE approach to process modelling defines a universal or *reference model* (an instance of the PROMENADE metamodel) that constitutes the basis on which any other process model will be built (as done for instance in [3]). Hence, this *reference model* is the common and extensible kernel shared by all processes modelled in PROMENADE. It is responsible for defining the core elements that will be a part of any model described in PROMENADE (e.g., the classes *Task*, *Document* and *Model* are defined by the reference model). The features associated to these elements are the ones needed to characterize any instance of them. Notice the difference between the metaelement features (which are used to characterize model classes) and the element features (which are used to characterize model class instances). Sometimes we may refer to the former group as *class features* and to the latter one as *class instance features.*

2.1 The Metamodel

The PROMENADE metamodel extends the UML one with the following elements (see figures 2 and 3):

- The classes *MetaDocument, MetaTask, MetaRole,* as *Class* subclasses (being *Class* defined in the UML metamodel). These three classes are the ones whose instances really characterize particular SPM.

- The class *SPMetamod* (which stands for *Software Process metaModel*), as *Model* subclass (again *Model* refers to the UML metamodel element).
- The classes *Precedence* and *Trigger*, that are used in the specification of the dynamic behaviour of process models.

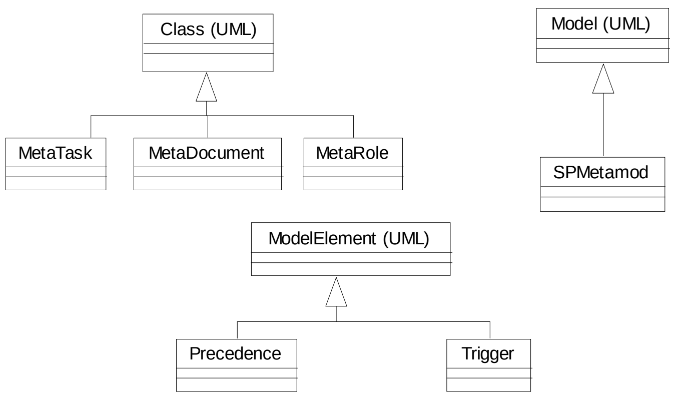


Fig 2. Elements and generalizations of the PROMENADE metamodel

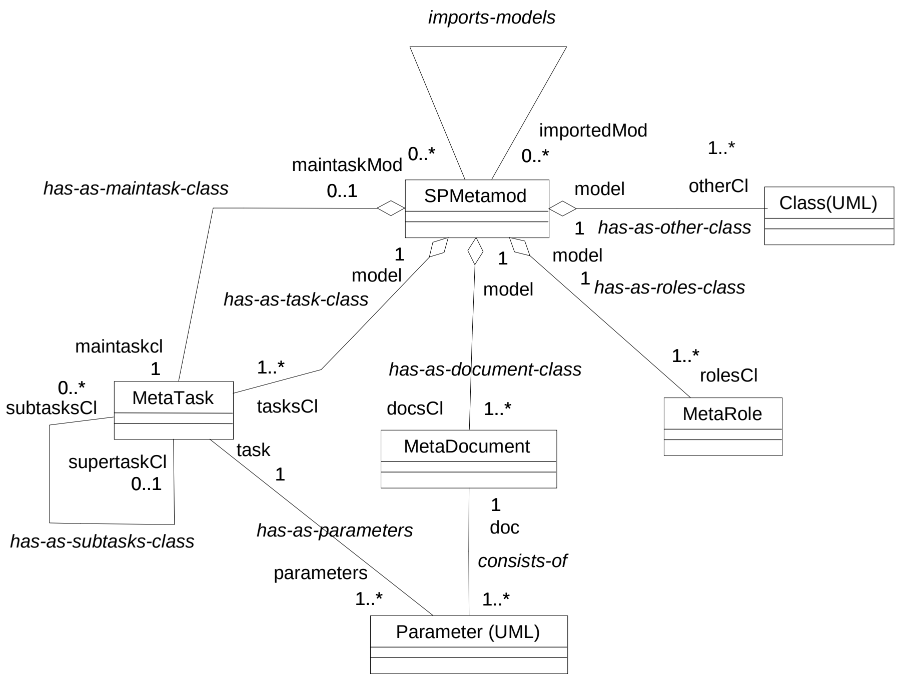


Fig 3. Associations of the PROMENADE metamodel

2.2 The Reference Model

The reference model is built upon three kinds of information, which yield to several complementary UML class diagrams. First, the individual information of the classes themselves, including constraints. Second, a class hierarchy which integrates all the documents by means of generalization. Last, other association relationships between classes (including aggregations).

Classes, class members and generalization hierarchy. In our O.O. approach, a generalization hierarchy of classes is the natural way to represent the many concepts involved in process modelling. Classes are characterized by many attributes and support many methods. Valid value attributes are stated through class invariants, while methods are specified through pre and post conditions.

Table 1 summarizes these classes. As heirs of a *Type* superclass, all of them share a few common attributes, such as *identifier*. We show in the table a few relevant attributes and methods. We do not refer either to structural methods or attributes that are directly related with the associations defined in the following section.

Table 1. Predefined classes

Class	Description	Some instances	Attributes	Some methods
Document	Any container of information involved in the software development process	A specification; a test plan; an e-mail	Link to contents; relevant dates; version; status	Document updating with or without new version creation; document edition
Communication	Any document used for people communication; can be stored in a computer or not	A fax; an e-mail; human voice	Link (if any) to contents; transmission date; status	*Send* and *Read*
Task	Any action performed during the software process	Specification; component testing; error-reporting	Precondition; status; success condition; deadline (if any)	Changes of task status
Agent	Any entity playing an active part in the software process	Myself; my workstation; a compiler	Profile; location; humans skills	Just structural ones
Tool	Any agent implemented through a software tool	A compiler; a navigator	Root directory for the tool; binary file location	Just structural ones
Resource	Any help to be used during software process	An online tutorial on Java; a Web site	Platform requirements; location	*Access*
Role	Any part to be played during the software process	Programmer; manager	Tasks for which the role is responsible	Just structural ones

These classes are put together in a natural way by defining some generalization relationships between them (see fig. 4). This default hierarchy may be extended by adding new classes in a classical manner, and this allows the creation of concrete models given different criteria as we do next for the component library case study.

Some predicates appear in the static part. One the one hand, *invariants*; that is, consistency predicates bound to classes that express constraints concerning attributes and relationships between them that should be kept at any time during the enaction of the model. On the other hand, *pre* and *post conditions* associated to class methods

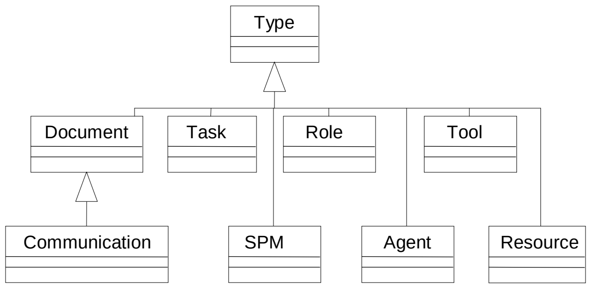


Fig. 4. Default generalization hierarchy in the PROMENADE reference model

which will be enforced respectively at the starting and ending point of such methods. We express these elements in UML as stereotyped constraints (with the predefined stereotypes *invariant, precondition* and postcondition, respectively) put in UML notes. We express the constraints in OCL as we consider it to be a natural, convenient and close to standard language to express constraints.

Association relationships. It is clear that the classes presented above must be related beyond the generalization relationship, and this is done by means of the UML association relationships (*associations* for short). Using associations, we can state which documents are manipulated by which tools, which tasks use which resources, etc. One significant kind of such associations are the *aggregation relationships* which relate a class with its components (whole-part relationship), as usual.

Fig. 5 shows the associations bound to the default classes. We highlight the existence of an association (in fact, a UML *aggregation*) from tasks to tasks to catch the concept of task decomposition.

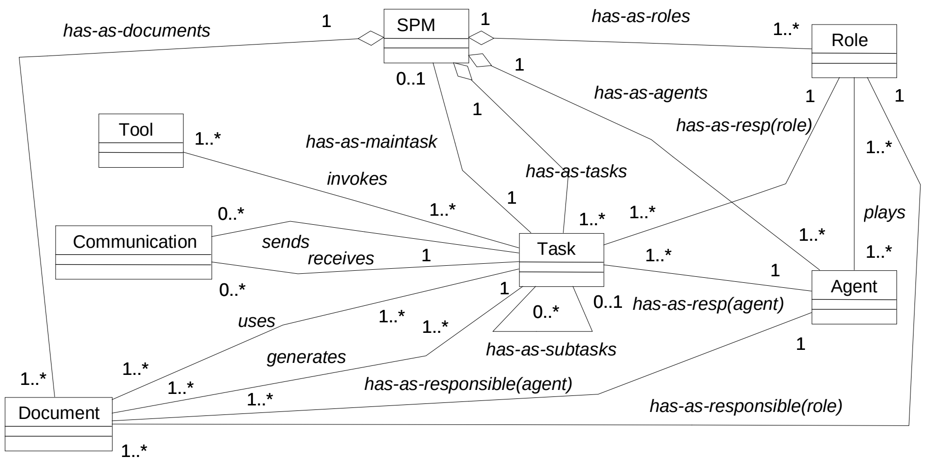


Fig. 5. Association relationships between classes

3 Static Part: Documents

Sections 3 and 4 are devoted to the presentations of some excerpts of a particular process, the construction of a library of components. In particular they present the most important aspects of the static part of this process (documents and tasks).

The main documents that take part in the library building process are shown in table 2, along with some relevant attributes and methods, and linked together through generalization relationships in fig. 6.

Table 2. Document classes

Class	Description	Attributes, methods
Component	A package of software provided with functional and non-functional specifications, code and non-functional behaviour. It may be reused and customized. It must be stored in a library.	stored flag; tested flag ; structural methods
Library	A collection of elements that may be reused when building software. Two kinds of libraries are considered: component libraries and property module libraries.	directory of contents; statistics; store, retrieve
SpecDoc	The specification part of a component. It is subdivided in functional and non-functional specification documents.	structural methods
ImplDoc	The implementation part of a component. It is compounded of a code document and a behaviour one.	structural methods
FSpec	The component functional specification document. It may be of several kinds: informal, formal, etc.	component signature; specification itself; tested flag
NFSpec	The component non.-functional specification document. It defines a list of non-functional (NF) requirements.	NF-requirements list; structural methods
Behaviour	The behaviour of the implementation with respect to the component NF-requirements.	behaviour description; structural methods
CodeDoc	The code of a component implementation.	code file; tested flag; compile, link
TestPlan	The list of tests that should be applied on a formal functional specification or an implementation.	list of tests; structural methods
EvalDoc	The result of the application of a test plan.	evaluation; structural methods
PropertyModule	A document containing a list of NF properties.	structural methods

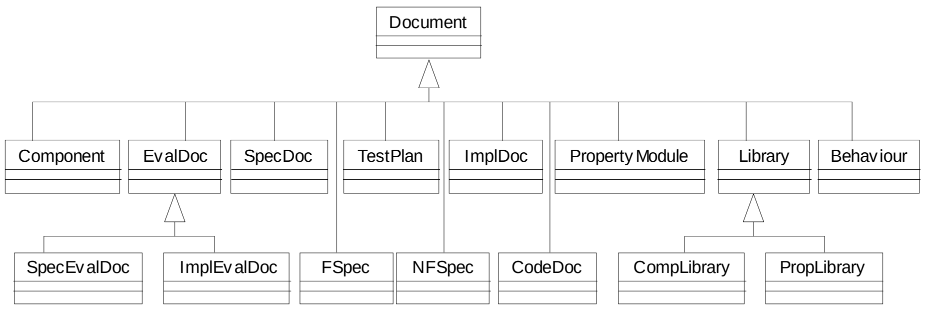


Fig 6. Generalization relationships for the component library model

The whole-part relationships between documents are shown by means of the aggregation relationships presented in figure 7. As an example, we depict in figure 8 the UML definition of one of these *Document* classes: *FSpec*. Notice the inclusion of the document invariants expressed in OCL.

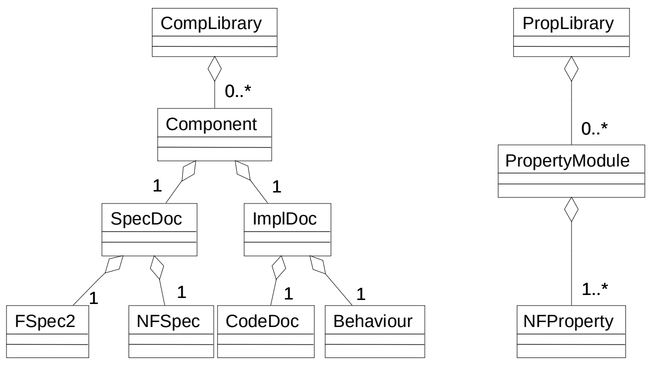
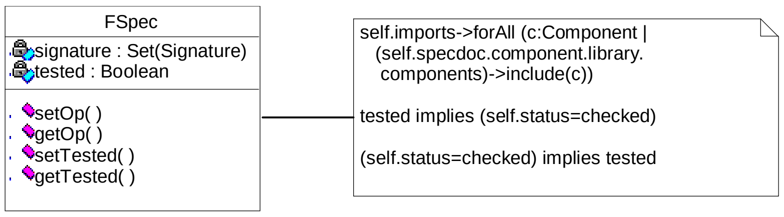


Fig 7. Aggregation relationships concerning documents



FSpec
signature : Set(Signature)
tested : Boolean
setOp( )
getOp( )
setTested( )
getTested( )
self.imports->forAll (c:Component |
(self.specdoc.component.library.
components)->include(c))
tested implies (self.status=checked)
(self.status=checked) implies tested

Fig 8. The UML description of *Fspec* document

Some important association relationships between documents are shown in figure 9. Among others, note that *SpecEvalDoc* and *ImplEvalDoc* are defined as association classes within the association relationships between *SpecDoc* and *SpecTestPlan*, on the one hand; and between *ImplDoc* and *ImplTestPlan*, on the other hand. Therefore, a document of the class *ImplEvalDoc* will contain the results of testing a specific component implementation by means of a given implementation test plan document. Notice also that the class documents *NFSpec*, *Fspec* and *ImplDoc* need to import components and that a specification document may be implemented by means of different implementation documents belonging to different components. Finally, a *SpecDoc* may be implemented by means of several *ImplDoc* documents belonging to different components.

Clearly, one crucial aspect concerning documents is the thorough modelling of the specific document classes that appear in table 2 which will lead to the extension of the hierarchy by defining new classes aiming at modelling (describing) finer aspects of the process. Let's focus, for instance, in the specification side of components.

There are several ways in which a component may be functionally specified. Each one of them generates a functional specification document (*FSpec*) with some specific features (i.e., attributes). Since we want to allow the coexistence of different specification techniques for different components, it becomes necessary to define a subhierarchy of functional specification documents in order to state precisely which kind of document will be generated by each functional specification choice.

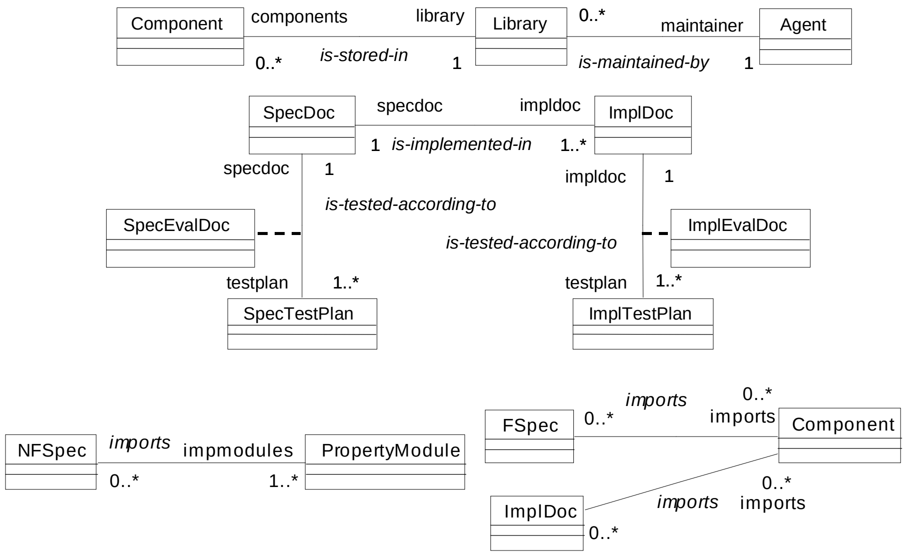


Fig 9. Association relationships concerning documents

Our model for the component library case identifies four general types of such specifications, which range from informal functional specifications to model-oriented ones (see fig. 10). But this is just a starting point; these classes should be further decomposed to introduce more concrete specification methods (e.g., initial or loose algebraic specifications) down to classes bound to concrete specification languages (e.g., *Z* or *VDM* model-oriented specifications; Larch and OBJ algebraic ones).

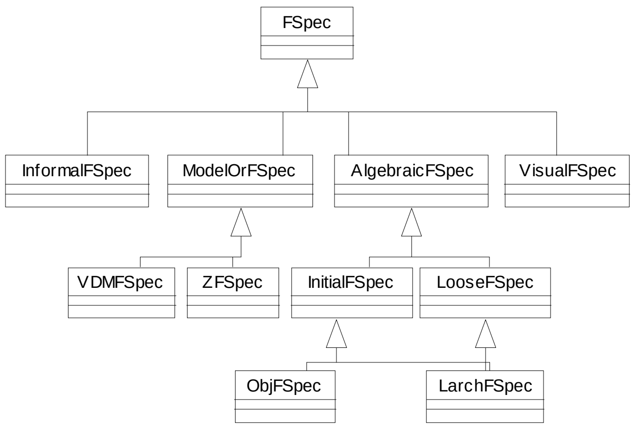


Fig 10. Functional specification document subhierarchy: an excerpt

Two aspects are intimately related with the coexistence of several functional specifications in the library of components: the definition of a different task refinement to specify functionally a component for each specification approach (see section 5), and the existence of a different *test plan* document class for each functional specification approach.

Non-functional specification documents are in charge of defining the non-functional requirements that should be fulfilled by any implementation of that component. As in the functional case, many notations can be used here, and the resulting hierarchy is similar to the one of fig. 10. Our usual choice is the NoFun language [7] defined as part of the ComProLab project, although other approaches can be followed [14,16].

4 Static Part: Tasks

From the PROMENADE point of view, two key concepts are relevant in order to describe tasks involved in process modelling: *task description* and *task decomposition*. The first one focuses on the statement of the dynamic behaviour of the task (mainly stating different kinds of precedence relationships between subtasks), and it is outlined in section 5. Concerning the second one, tasks may be decomposed along two dimensions that turn out to be orthogonal: task decomposition *by refinement* and *by aggregation*. This section focuses in these two aspects, which are part of the static model.

4.1 Task decomposition by aggregation

Tasks in PROMENADE may be of two kinds according to the complexity of their behaviour: *atomic* tasks and *composite* ones. Atomic tasks cannot be decomposed in other tasks. Their enactment is performed by means of a call to an external tool or in a manual way. On the other hand, composite tasks may be further decomposed into *subtask classes*, which may be in turn atomic or composite. Therefore, a composite task enactment may involve the enactment of several *subtasks* in some suitable order (determined in the dynamic part of the model).

Some of these aggregation relationships concerning task classes are shown in figure 11. This figure shows the task classes that are involved in the upper levels of the construction of the component library, which should be completed by tasks concerning lower levels. Specifically, some subtasks of *BuildComponent* and *SpecifyComponent* are shown. On the other hand, table 3 shows the parameters (documents) used and generated by some of these tasks along with a brief description. Notice in this table that in addition to the usual parameter modes (input, output and input/output), a feedback mode is introduced to represent the flow of information when a task ends in failure.

Table 3. Task classes

Task	Parameters	Description
BuildComponent	b: **i/o** Library, c: **out** Component	Construction of a component
SpecifyComponent	c: **out** Component, sd: **in** SpecDoc	Specification of a component (both functional and non-functional)
ImplementComponent	c: **i/o** Component, id: **out** ImplDoc, evd: **in fbk** ImplEvalDoc, b: **in** Library	Implementation of a component.
GenerateTestPlans	sd: **in** SpecDoc, stp: **out** seq(ImplTestPlan)	Generation of plans for testing a component.
TestComponent	id: **in** ImplDoc, evd: **out fbk** ImplEvalDoc, stp: **in** seq(ImplTestPlan)	Test of a component according to its test plans.
Store	c: **in** Component, b: **i/o** Library	Storing a developed component into a library.
FSpecifyComp	c: **i/o** Component fsd: **out** FSpec evd: **in fbk** SpecEvalDoc b: **in** Library	Functional specification of a component (different task refinements may be considered.
ModelOrFSpecify	c: **i/o** Component fsd: **out** ModelOrFSpec evd: **in fbk** SpecEvalDoc b: **in** Library	One of the task refinements of the functional specification of a component.

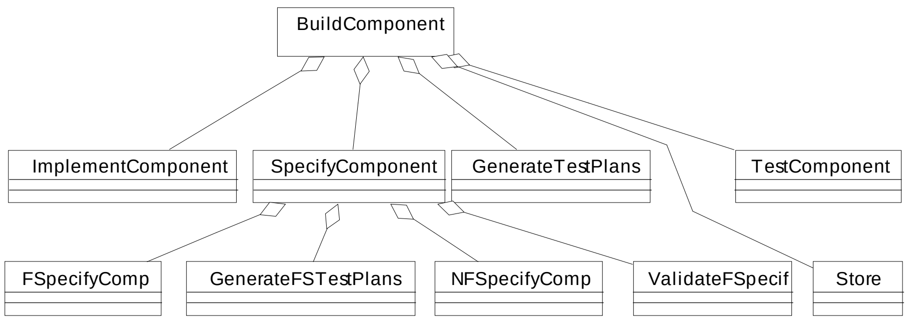


Fig 11. Aggregation relationships between tasks

4.2 Task decomposition by refinement

The behaviour of a composite task is encapsulated in PROMENADE by means of *task refinements*. Intuitively, a task refinement is a concrete way to perform a task. A bit more formally, a task refinement of a composite task class *T* is a task class that expresses one specific way in which *T* may be decomposed into subtasks and the precedence relationships that should be kept among them at enactment time. Since, in general, it is possible to think of several ways to perform a task, it makes sense to define several task refinements for a specific composite task.

Task refinements are modelled in PROMENADE by means of generalization relationships between a task class and the set of task classes that refine it. In this way, the subclasses of a task class *T* represent its possible refinements. Notice that task refinements and aggregations are two complementary mechanisms which will appear tightly intertwined in class diagrams: a task can be refined in many ways, and for each task refinement a particular decomposition into subtasks may exist, and those subtasks can be in turn refined in different ways, and so on.

Some examples of how a task class may be refined are given in figure 12. There are two ways to carry out the functional specification of a component (*FSpecify-Comp*): *model-oriented functional specification* (by which a formal model is associated to a component and its operations are specified according to that model) and *informal functional specification* (which describes in natural language the component and its operations). Each one of these functional specification methods is considered to be a new task refinement in PROMENADE and it is represented by means of a generalization relationship.

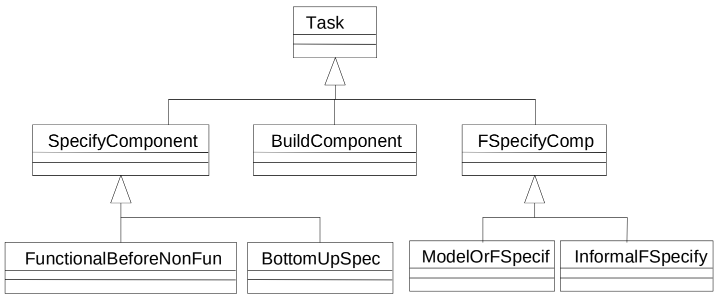


Fig 12. Generalization relationships between tasks (task refinements)

Usually, the selection of a particular task refinement enforces the use of some specific documents. This enforcement is established in the parameter definition of the refinement. For example, the selection of the *ModelOrFSpecify* task refinement (for the task *FSpecifyComp*) must result in the automatic selection of a *ModelOrFSpec* as functional specification document and a *ModelOrSpecTestPlan* as specification test plan and this is reflected in the *ModelOrFSpecify* parameters (see table 3). In general, a task refinement for task *A* may use as parameters some subclasses of the parameter classes of *A*.

The behaviour of a task refinement is described by means of a specific control flow which is depicted in a PROMENADE activity diagram (see section 5). Other task refinements may come up in the activity diagram that describes a specific task refinement for a given task class *T* (including other *T*'s refinements).

Task refinements help in achieving *expressivity* (since various different behaviours may be given to a single activity; hierarchies of task refinements are also allowed), *modularity* (task refinements are encapsulated into tasks; therefore it is easy to add /remove task refinements to/from a model), and *model flexibility* (since we can decide at enactment time, instead of modelling time, which specific task refinement is to be enacted; this allows the enactment of incomplete models).

5 Drawbacks Concerning the Dynamic Part Description with UML

While the static part of a SPM (its structural view) may be well described using class and object UML diagrams, this does not seem to be the case of its dynamic part. The dynamic part of a SPM deals with the description of the control flow of the modelled process. This control flow is usually split into composite and atomic activities (tasks) whose behaviour should be described somehow. Among the diagrams provided by UML to cope with the behavioural view of the system (sequential diagrams, collaboration diagrams, statechart diagrams and activity diagrams), activity diagrams have been reported to be the most suitable ones to describe SPM [1, 15], but it has also been pointed out that they suffer from several limitations, remarkably:

- *Lack of expressivity*. UML activity diagrams are a sort of event diagrams which are useful to express event-driven (*reactive*) control allowing *swimlanes* (partition of activities according to their responsible), *branching* (conditional flows), *forking* (control split into several independent flows) and *joining* (independent flow controls unified). However, the semantics of transitions in activity diagrams cannot express *proactive* control; this kind of control is important because it allows the enactment of tasks according to some predetermined precedence rules (e.g. task *A* should finish before the end of task *B*) rather than reacting to the rising of certain events (basically, task finalization). This latter fact is quite an important restriction since one of the features that have been recognised to be important for a second generation PML is its ability to combine both proactive and reactive control paradigms [22], in order to get less prescriptive and more expressive process models.
- *Activity properties cannot be shown in activity diagrams*. For instance, activity deadlines and duration, roles related to an activity (who should be informed about it...), used resources and tools... Needless to say that expressing all these aspects is crucial in both software process modelling and workflow management.

PROMENADE extends the expressivity of UML activity diagrams by providing (a) proactive control (provided by means of several kinds of precedence relationships) which may be combined in the same diagram with the usual event-driven control, and (b) visual access to activity properties (which may be depicted in the activity diagram). The description of the PROMENADE dynamic model is beyond the scope of this article. We refer the interested readers to [9] and [10].

6 Conclusions and Related Work

This paper has presented a case study in the field of software process modelling using UML for modelling the static part this process. Specifically, we have presented a model for building a library of software components involving the specification of non-functional requirements and the statement of the behaviour of a particular implementation with respect to those non-functional requirements.

The way to construct a software process model in our approach is based on the extension of a reference model which describes the hierarchy with the most important concepts related to process modelling, their structure, behaviour, constraints and the association relationships between them (task responsible, task attributes and resulting documents...). This reference model is described using several class and object UML diagrams. Any other specific model will be considered as an extension of the reference one (extending the generalization default hierarchy with new classes and new associations between them). Thus, we have defined the model for constructing a library of components by defining new documents (component, library, specification document, functional specification document...), new tasks (specify functionally a component, implement it...) and new associations. All these elements concern the static part of the model and have been described in UML (and the constraints in OCL) which has proved to be a suitable and powerful notation to deal with such structural part, and a very important improvement to our former approach using OOZE [9] (an object-oriented dialect of the specification language Z).

This has not been the case with the dynamic (behavioural) part. Activity diagrams seem to be the UML diagram that fit better into the control flow description that is needed in software process and workflow modelling. Unfortunately they seem not to be expressive enough to deal with all the elements required in such environments. We have suggested the highlights of some extensions for UML activity diagrams which have been included in PROMENADE, our process modelling language: the depiction of the required task properties (responsible, generated documents...) in the diagrams and the addition of proactive declarative control (by the definition of several kinds of precedence relationships) which do not enforce a strict execution order triggered by some events (like the ending of a task) but just the declarative enumeration of the essential precedence requirements to be kept during enactment. Our experiences seem to confirm that the event-driven transitions combined with proactive control (implemented by means of several types of precedence relationships with a declarative policy) provide a much more flexible, realistic and expressive process modelling. As we have pointed out, the combination of both controls is encouraged for second generation PMLs [22].

We are not aware of almost any other PML using UML to describe the static part of the SPM. For the sake of giving some examples of well-known PMLs: APEL uses OMT-like diagrams [4]; E3 defines its own notation [11]; MERLIN describes the structural aspects with extended entity-relationships and statecharts [19]; APPL/A and JIL use textual constructs strongly based on Ada (which has been extended with some additional features like relations between objects) [21, 22]. The Rational Software Corporation *et al*. have developed a UML extension for objectory process for software engineering [17]. Essentially, it extends some metamodel classes by means of stereotypes. Neither structure nor behaviour are given to those *stereotyped* classes; no integrity constraints are defined; and no means to improve the UML features in order to deal with the dynamic process are provided. Therefore, this proposal seems not to be adequate to meet the requirements of SPM.

[12] presents an approach which describes with UML the dynamic part of the model using class diagrams with stereotyped associations for showing the control and data flow. The metamodel is defined by attaching stereotypes to model elements. In our opinion, other UML diagrams (for instance, activity diagrams) are better suited for the description of the dynamic part of a model. On the other hand, stereotypes and the other UML extension mechanisms suffer from several limitations in order to define a metamodel (being expressivity and comprehensibility two of them).

Task refinements, defined as different implementations of a task that may coexist in a single model, also appear in [13]. In this approach, a task definition consists of a task interface and, potentially, various task bodies (in the form of workflow definitions) attached to it. At enactment time a decision is taken for each task about which task body is to be enacted. PROMENADE broadens this feature by allowing the definition of task refinement hierarchies along with the substitution of a task by any of its offsprings, either at modelling time or at enactment time and by providing a powerful way to describe a task behaviour as have been outlined in section 4.

Last, we are not aware of any approach intended to improve the capacity of UML in order to model the behaviour of a software process. But there are several proposals in the related field of business processes. [1], for example, also states that UML diagrams are not sufficient for business process modelling. The authors propose to integrate a well-known process modelling formalism (EPC, Event-driven process chains) with UML diagrams. [15] proposes the use of the *stereotype* mechanism of UML to extend activity diagrams in the context of business process modelling. The new diagrams can express the required activity properties (computer support to the activity, duration...). In both cases, no new control paradigm is provided.

References

1. Allweyer, T; Loos, P: Process Orientation in UML through Integration of Event-Driven Process Chains. Proceedings of UML 98' Workshop, Ecole Superioeure des Sciences Appliquées pour l'Ingénieur-Mulhouse Université de Haute-Alsace (1998), 183-193
2. Bandinelli, S.; Fuggeta, A.; Ghezzi, C.; Lavazza, L.: SPADE: An Environemnt for Software Process Analysis, Design and Enactment. In [6] (1994), 223-247
3. Conradi, R.; Larsen, J.; Minh, N.N.; Munch, B.P.; Westby, P.H.: Integrated Product and Process Management in EPOS. Journal of Integrated CAE, special issue on Integrated Product and Process Modeling (1995)
4. Dami, S.; Estublier, J.; Amiour, M.: APEL: a Graphical Yet Executable Formalism for Process Modeling. E. di Nitto and A. Fuggetta (eds.), Kluwer Academic Publishers (1998)
5. Derniame, J.-C.; Kaba, B.A.; Wastell, D. (eds.): Software Process: Principles, Methodology and Technology. Lecture Notes in Computer Science (LNCS), Vol. 1500. Springer-Verlag, Berlin Heidelberg New York (1999)
6. Finkelstein, A.; Kramer, J.; Nuseibeh, B. (eds.): Software Process Modelling and Technology. Advanced Software Development Series, Vol. 3. John Wiley & Sons Inc., New York Chichester Toronto Brisbane Singapore (1994)

7. Franch, X.: Systematic Formulation of Non-Funcional Requirements of Software. Proceedings 3rd International Conference on Requirements Engineering (ICRE), Colorado Springs (USA), IEEE Computer Society Press, Los Alamitos (1998), 174-181.
8. Franch, X.; Botella, P.; Burgués, X.; Ribó, J.M.: ComProLab: A Component Programming Laboratory. Proceedings 9th Software Engineering and Knowledge Engineering Conference (SEKE), Knowledge Systems Institute, Skokie (1997), 397-406
9. Franch, X.; Ribó, J.M.: A Structured Approach to Software Process Modelling. Proceedings 24th EUROMICRO Conference, IEEE Computer Society Press, Los Alamitos Washington Brussels Tokyo (1998), 753-762
10. Franch, X.; Ribó, J.M.: PROMENADE: A Modular Approach to Software Process Modelling and Enaction. Research Report LSI-99-13-R, Dept. LSI, UPC (1999)
11. Jaccheri, M.L.; Picco, G.P.; Lago, P.: Eliciting Software Process Models with the E3 Language. ACM Transactions on Software Engineering and Methodology (1999)
12. Jäger D., Schleicher A., Westfechtel B.: Object-Oriented Software Process Modeling. To appear in the proceedings of the 7th European Software Engineering Conference (ESEC), Toulouse, September 1999.
13. Joeris G., Herzog O.: Towards a Flexible and High-Level Modeling and Enacting of Processes. Proceedings of the 11th. Conference on Advanced Information System Engineering (CAISE), LNCS 1626, pp. 88-102, 1999.
14. Landes, D.; Studer, R.: The Treatment of Non-Funcional Requirements in MIKE. Proceedings 5th European Software Engineering Conference (ESEC), Barcelona (Catalunya, Spain). Lecture Notes in Computer Science, Vol. 989. Springer-Verlag (1995)
15. McLeod, G: Extending UML for Entreprise and Business Process Modeling. Proceedings UML 98' Workshop, Ecole Superioeure des Sciences Appliquées pour l'Ingénieur-Mulhouse Université de Haute-Alsace (1998), 195-204
16. Mylopoulos, J.; Chung, L.; Nixon, B.A.: Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. IEEE Transactions on Software Engineering, Vol. 18, N. 6 (1992), 483-497
17. Rational Software Corporation: UML extension for Objectory Process for Software Engineering. http://www.rational.com/uml
18. Rational Software Corporation *et al*.: UML Semantics. http://www.rational.com/uml
19. Reimar, W.; Schaefer, W.: Towards a Dedicated Object-Oriented Software Process Modelling Language. Workshop on Modeling Software Process and Artifacts, held at 11th ECOOP, Jyvaskyta (Finland) (1997).
20. Rumbaugh, J.; Jacobson, I.; Booch, G.: The UML Reference Manual. Addison Wesley (1999)
21. Sutton S.M.; Heimbigner D.; Osterweil L.J.: APPL/A: A Language for Software Process Programming. ACM Transactions on Software Engineering and Methodology. Vol 4. N. 3, July 1995. 221-286.
22. Sutton, S.M.; Osterweil, L.J.: The Design of a Next-Generation Process Language. Proceedings of ESEC/FSE '97, Lecture Notes in Computer Science, Vol. 1301, M. Jazayeri and H. Schaure (eds.). Springer-Verlag, Berlin Heidelberg New York (1997), 142-158
23. Warboys, B.C.; Balasubramaniam, D. *et al*: Instances and Connectors: Issues for a Second Generation Process Language. Proceedings of the 6th European Workshop in Software Process Technology, LNCS 1487, V. Gruhn (ed.). Springer-Verlag (1998)