

# Describing Fractal Processes with UML

Harald Störrle

Ludwig-Maximilians-Universität München  
stoerrle@informatik.uni-muenchen.de

**Abstract.** Component-based software has a self-similar structure on each level of abstraction, i.e. its structure is fractal. Traditional software processes, however, have a linear or iterated structure, and are thus not very well suited for component-based software development. Here, processes described by languages of patterns fit better.

To ensure a general understanding and easy applicability of processes patterns, I propose to (1) build on the well known description schemes for traditional product patterns and adapt them to the software process domain; and (2) use the description techniques and notions of the Unified Modeling Language (UML) is “*the lingua franca of the software engineering community*”. Some adaptations and extensions become necessary to both of these, however, and care has to be taken not to impede the goal of universal understandability.

## 1 Introduction

### 1.1 Motivation

Workflows concerning software processes have been described in a variety of formalisms and styles (see e.g. [17] for a survey). Today, some essential requirements are still not adequately addressed by many of them.

**Understandability** Practitioners explicitly ask for “*well-structured*” process models that are “*easy to understand and easy to apply*” (cf. requirements 2 and 3 in [19]). The vast majority of traditional processes like the “Unified Process” (UP, cf. [15]), however, are rather big sets of documents that take a lot of experience to understand and apply properly.

**Flexibility** Many developers feel that their creativity is stifled by too rigid a development process. In fact, some would argue, that this is exactly the purpose. Anyway, enforcing a process is sometimes a difficult managerial task, and there are circumstances where traditional (large scale) processes just aren’t appropriate, as is shown by the recently soaring interest in so called lightweight processes like “extreme programming”.

**Precision** On the other hand, a development process must be formal enough to allow for automatic enactment and formal analysis, at least in selected places. Ideally, this would be combined with the previous requirement to achieve a kind of controlled and graded formality of the process.

**Fractal structure** Finally, the componentware paradigm has manifest implications on the overall structure of the development process: if the product structure is fractal, the process structure better be fractal as well. Otherwise the mapping between process and product alone becomes a major challenge. Iterative process models, such as the UP have a crippling weakness here.

**Integrate aspects** Classical process models (e.g. V-Model, ISO 12207, UP) differentiate between different aspects (aka. submodels, core workflows). These, however, really are tightly interwoven. Separating them makes it easier to neglect the supportive workflows like risk management or quality assurance, as these do not contribute directly to the (amount of) output.

## 1.2 Benefits of approach

I believe, that process patterns using UML are a novel way for dealing with these issues that is superior to traditional approaches for the following reasons. First, consider the contribution of the UML. On the one hand, the UML is already “*the lingua franca of the software engineering community*” (cf. [25, p. v]), and it is very likely to spread even further. This means that almost every professional understands UML (or will have to do so pretty soon), and things will stay like this for some time to come. This results in network effects, rapidly increasing the number and quality of CASE-tools for, courses on, and experience with UML. By using the UML, the understandability of process models is increased dramatically. Some object that the UML is still not very expressive when it comes to modeling workflows, but with UML 1.4, this is can be solved without heavy weight-extensions.

On the other hand, UML is now rapidly being developed into a body of mathematically precise formalisms (cf. [23, 18, 26]), opening up the road to enactment and formal analysis, not only of process models, but also of the product models.

Now consider the contributions of process patterns. The notion of (design) pattern is widely accepted among practitioners, and has resoundingly proved its practical applicability. Patterns capture small, coherent, and self-sufficient pieces of knowledge, thus allowing to be applied in different contexts, and in particular, on different scales. Using (design) patterns is a form of reuse of (design) knowledge.

Process patterns are exactly like design patterns, except that they exist in the process domain. Actually, the term “design pattern” is a bit misleading: what classical design patterns like Model-View-Controller etc. really talk about is (only) the *product*, that is, the *result* of the design, not the *process* of designing it. So, I shall speak of result patterns vs. process patterns to make clear the difference.

The benefits of design patterns may be transferred to the process domain: by using the same pattern in different contexts, pattern-oriented process-descriptions are less redundant than traditional ones, and so a pattern language is easier to understand and apply than traditional processes. Also, each pattern may be formulated with its own degree of formality (i.e. strictness of applicability criteria), allowing as well for flexible, rigid, and mixed processes. Patterns

may be used both to guide the development process, and to document a development after-the-fact. Patching-in ad-hoc steps in a process is trivial for a pattern-oriented process.

Finally, patterns are scale-invariant, that is, they may be applied on any scale of granularity or abstraction, as long as their application criteria are satisfied. Using a language of process patterns this way, a component-oriented discipline of software construction is easily accommodated: building systems from components results in a hierarchical and self-similar (i.e.: fractal) product structure (cf. e.g. [4, 14, 10, 20]). This is probably *the* single most important advantage process patterns have over traditional processes.

### 1.3 Outline of approach

The approach pursued in this paper may be characterized as follows: first, I summarize UML's facilities for modeling process, and propose some small extensions. Note that this approach is based on the very recent UML 1.4, which contains some improvements over UML 1.3. Then the descriptive schema laid down in [12] (and very similarly in [6], the other of the two major design patterns books) is adapted to fit the new requirements of the process domain.

### 1.4 Related work

There is already some work done on pattern languages for software processes. The earliest reference that I know of is the work on "process chunks" by Rolland et al. (cf. [24]). Process chunks are 4-tuples consisting of situation, decision, action, and argument, formalizing design decisions taken during the requirements elicitation. Neither UML nor patterns in the modern sense are used.

Then there is a number of papers like [4, 11, 27] which already use the term process pattern, but remain superficial. There, even less detail is provided to describe the patterns. Neither a general scheme nor interesting individual aspects (e.g. classification, process, participants) are described in any detail.

A more serious approach is presented in Catalysis (cf. [9]), where there are 60 different patterns, each of which is described by three to five aspects (name, intent, strategy, and sometimes considerations and benefits) using natural language and ad hoc sketches. While presenting the first real example of a language of patterns, Also, Catalysis does *not* use UML, except claims of the opposite, and even the notation it does use (which is vaguely reminiscent of UML) is not used in the process patterns. Neither the roles and responsibilities involved, nor the document types required and provided, nor the actual design activities are described in any great detail.

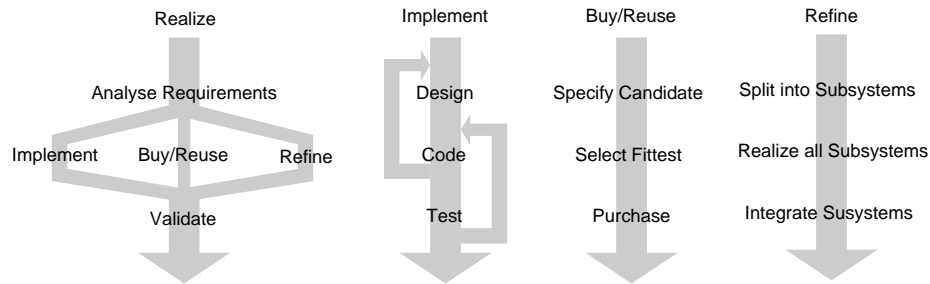
Finally, there are the books by Ambler (see e.g. [2]). They deal with what in the terminology presented here would be organizational, or process phase patterns. What I call process pattern is termed task pattern by Ambler. Also, Ambler does not really use the UML (despite the cover page claim of the opposite): there is a handful of use case diagrams, and that's it. Also, there is no elaborated schema (like we present it here), much less a classification schema.

In fact, Amblers books are really general purpose introductory textbooks on software process (though comparatively good ones).

Compared to all the above mentioned approaches, our approach is the only one which seriously tries to incorporate the UML (and certainly the first one to take into account the UML 1.4). Also, all other mentioned approaches use much less elaborated and precise descriptive schemes for patterns. With the exception of Rolland et al., they don't even present a schema at all.

## 2 An example

As an example, consider the following family of process patterns, consisting of four process patterns called *Realize*, *Implement*, *Buy/Reuse* and *Refine*. Figure 1 presents an informal sketch of the tasks involved in these process fragments, and the control flow.



**Fig. 1.** A simple language of process patterns (informal sketch).

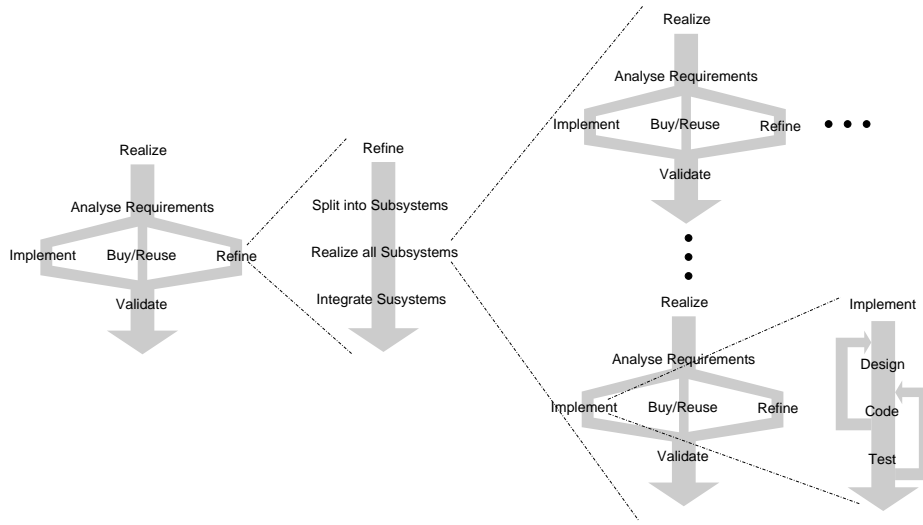
**Realize** The overall goal is to realize some system. Whichever strategy will be chosen for the realization proper, it is certain that the task will need definition before, and validation afterwards.

**Implement** In some cases, a system may implemented right away without further fuzzing about. This is the case for very small systems, for a quick-and-dirty prototype, or if there is a lot of experience and adequate support of tools and libraries.

**Buy/Reuse** Alternatively, one may purchase a prefabricated component from a kind of component market , that is, either buying it from a third party, or reusing it from internal sources.

**Refine** If a system can be neither implemented directly nor purchased from a third party, one may chose to split it up into smaller subsystems, and try to tackle these individually.

Obviously, applying this set of patterns gives rise to a recursive structure: consider each pattern as a rule in a grammar, and the application as a derivation (cf. Figure 2).



**Fig. 2.** Applying the pattern language (informal sketch).

Admittedly, this pattern language is somewhat simplistic. A practical example, however, would by far exceed the space limit set for this article.

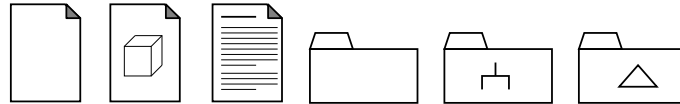
### 3 Using UML to describe software processes

The UML is *the* most widely understood design notation in software engineering. Obviously, it would be helpful to use it for the description of processes, so as to benefit from the tools, knowledge, and research results concerning UML. Up to version 1.3, the UML contained very little for describing software processes (cf. [16, 1]). The UP did present some notations, but their relationship to the UML metamodel was undefined. Technically, the UP is not part of the standard proper, but it is a kind of quasi-standard in this area. The UML 1.4 now provides the new metaclass **Artifact**<sup>1</sup>, with stereotypes like `<<document>>`, `<<file>>` and `<<executable>>` (cf. [21, p. 2-17–2-21]). It also provides a much more detailed standard profile for software process modeling which focuses on the product structures. In the remainder, I will compile a set of notations and concepts (plus additions, where necessary) from UML 1.4 and UP that should be sufficient for most tastes.

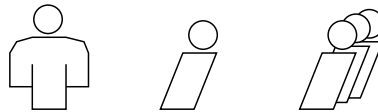
**Documents** Though the concept has been included in the move from version 1.3 to 1.4, there are no notations presented. I propose to use the folklore symbols (also used in UP), that is, an upright rectangle with a dog's ear for documents. Different types may be distinguished by additional inscriptions or

<sup>1</sup> When referring directly to the metamodel, metaclasses will always be printed in sans-serif font.

icons. UML diagrams should be represented by a new stereotype `<<diagram>>` with a string-valued tagged value `kind` indicating the type of diagram. **Models**, **Subsystems**, and all kinds of **Package** in general should be represented as iconified package-symbols, possibly with further inscriptions or icons attached to differentiate between different kinds. See the following sketch for some examples (left to right): any document, a text, a deployment diagram, any package, a subsystem, a model.

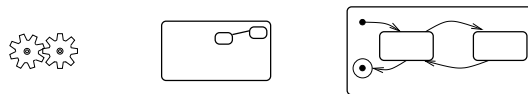


**Resources** Neither the UML proper nor the standard profiles provide a concepts for resources like tools or people: the notion of **Actor** serves only to represent a neighboring system. The UP, however, distinguishes between *Resources* (roles) and *Workers* (concrete people or tools that are capable of playing these roles). The UP introduces the symbols below for resources, workers, and groups of workers.



Further extensions might become necessary in the future, such as for describing organization structures, but this is beyond the scope of this paper.

**Tasks** The structure of a task may be described as an activity diagram. The UP introduced a symbol consisting of a pair of cog-wheels for coarse-grained activities. These may be considered as syntactical sugar for **SubActivityStates**. In the UML, it is not legal to present the refinement of a **SubActivityState** by graphical containment (like it is done for **CompoundStates**). This can be quite useful however, so we propose this extension (task, standard **SubActivityState**, refined **SubActivityState**).

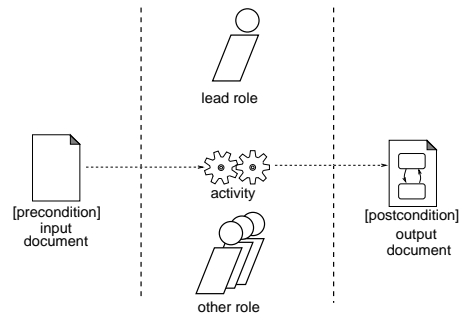


Note that, since **Artifact** is a **Classifier**, it may occur in activity diagrams to symbolize the type of an **ObjectFlowState** (see Section 4.1 below). Observe also, that now workflow may be described in coarse and refined versions.

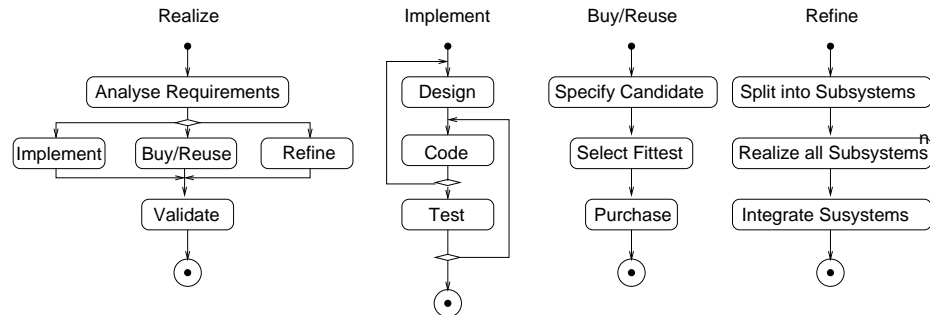
Note that the extensions proposed en passant are conservative and mainly only syntactical, that is, they could be easily included into the conceptual framework of the UML metamodel.

Now, the UP introduces a new kind of diagram. This simple chart shows the artifacts and resources required and produced by a task. It is found throughout the UP, used informally, but not defined properly. With these notations it is

now possible to present e.g. the process fragment Refine from above in greater detail, and with more precision (see Figure 4). Note that we have presented only the control flow, but not the data flow. A more worked example is presented in Section 5.



**Fig. 3.** The UP schema for tasks.



**Fig. 4.** Presenting tasks by activity diagrams: the language of process fragments of Figure 1 as UML activity graphs. Note the usage of `dynamicMultiplicity` in the Refine-task. We have omitted the data-flow here for simplicity (cf. Fig. 5, right).

## 4 Describing process patterns

Process patterns can be described by a schema similar to that known from result patterns. The overall descriptive scheme for process patterns and the terminology are taken from [12, p. 5f]. In the following subsections I will first describe the overall schema, and then the modifications resulting from transferring it to the process domain (see [4] for a discussion of the requirements of process vs. result structure patterns).

#### 4.1 Overall description schema

The two best known books on design patterns [6, 12] propose very similar schemas for the description of design patterns. In order to increase the acceptance of our approach, I adopt this approach as far as possible. See Table 1 for a comparison of the schema used in [6, 12] and here.

POSA [6]	GOF [12]	this paper
Name	Pattern Name	Name
n.a.	Classification	Classification
Problem	Intent	Intent
Also known as	Also known as	Also known as
Example	Motivation	Motivation
Context	Applicability	Applicability
Structure	Structure	Process
Dynamic Aspects	Collaborations	n.a.
n.a.	Participants	Participants
Implications	Consequences	Consequences
Implementation	Implementation	Implementation
Sample Solution	Sample Code	Sample Execution
Applications	Known Uses	Known Uses
References	Related Patterns	Related Patterns

**Table 1.** Comparison of pattern schemas.

**Name** First, each pattern has a name. For process patterns, this is usually identical to the name of the task that is supported by this pattern.

**Classification** As pattern catalogs may grow rather large, there must be a systematic way of retrieving them, which is facilitated by this aspect. This aspect is not present in [6], and appears as part of the name aspect in [12]. See Section 4.1 below for details.

**Intent** A intuitive account of the rationale of the pattern, its benefits and application area.

**Also known as** Sometimes, there is not a unique best name of a patterns, but a number of equally adequate names, which are given here.

**Motivation** A scenario that illustrates the problem, applicability conditions and purpose of the pattern.

**Applicability** The prerequisites for applying a pattern, the context where it may be applied. In the scope **Capsule**, this may be specified by the required Views and possibly consistency conditions established. includes preconditions

**Process** In design patterns, this aspect is called “structure” and contains a OMT class diagram. For a process pattern, the equivalent of “structure” is the process, that is the causal structure of activities. See Section 4.1 below.

**Participants** In result patterns, this aspect refers to the classes involved. In process patterns, the participants are the documents and resources (including people) that play a role in the pattern. See Section 4.1 below.

**Consequences** Discussion of the advantages and disadvantages of using the pattern, e.g. some quality goal or other end that is achieved by this pattern.

**Sample Execution** The implementation of a design patterns is a program, that is, an expression of a programming language. The implementation of a process pattern is not a program for some computer, but a procedure in an organization, involving people, tools, organizational facilities and an array of personal and interpersonal techniques. See Section 4.1 below.

**Implementation** Discussion of implementation-specific problems, i.e. problems related to the realization of the pattern. As process patterns are realized by organizations (see previous aspect), this aspect refers to problems related to organizational and tool problems.

**Known Uses** Applying a pattern may be easier when guided by concrete examples where the pattern has been applied to general acclaim.

**Related Patterns** Relationships to other patterns, i.e. alternatives, companions, or prerequisites and discussion of differences.

This schema does not contain the aspect **Collaborations** of [12] (called “Dynamic Aspects” in [6]): it is used to represent the behavior (and dynamic structure) of the Structure. This is already catered for by the new aspect **Process**, and may be omitted.

**Classification aspect** Patterns always exist only as part of a *language* of patterns—this is one defining criterion. These pattern languages may become rather large, and so cataloging and retrieval become important practical problems. These problems have been studied in the reuse community (see e.g. [22]). There, the terms *facette* and *simple class* have been coined for the dimensions of classification, and their respective values.

There are several classification schemes for result patterns.<sup>2</sup> Buschmann et al. propose to use the abstraction level and the problem category ([6, p. 362ff]). The Gang-of-four propose to use the purpose and the scope (see [12, p. 9ff]). For process patterns, these schemes obviously need adaptation, both concerning do not apply. In their stead, I have identified four classification *facettes*: abstraction level, phase, purpose and scope. I shall now explain the simple classes of these *facettes*.

**abstraction level** In [6], the three abstraction levels of idioms, design patterns, and architectural patterns are distinguished. Analogously, process patterns may be classified as techniques, process patterns proper, and development styles.

**phase** Design patterns are attributed to a problem category. The analog for process patterns is the development phase (in the classical sense), that is, specification, design, realization, and maintenance. Others are also possible, of course.

---

<sup>2</sup> The popular patterns-books do not use the terminology known from *facette*-classification, but the ideas are identical.

**purpose** In [12], result patterns are classified as “creational”, “structural”, or “behavioral”. For processes, there are other purposes, such as the administration, the construction proper, and quality assurance.

**scope** Finally, the design patterns may be distinguished according to whether their scope is an object or a class. Since this is a bias towards object-oriented technology which is soundly out of place in the context of processes, these simple classes are replaced by such that refer to the entities occurring in traditional processes, e.g. “architectural style”, “product line architecture”, “reusable component”, “implementation module”, or “test case” may be distinguished.

The simple classes given in each facette may differ from project to project—the ones mentioned here are just plausible defaults. Also, there is usually no hard and fast dividing line between the simple classes of the respective facettes. Note that granularity is not a useful classification criterion for a fractal process.

**Process aspect** Many people seem to think that the structure aspect (i.e. an OMT class diagram) of a design result pattern *is* the pattern. Process patterns obviously do not have a structure in this sense: they specify a (part of a) process. Consequently, this aspect has been renamed to “process”, and it consists of an UML workflow diagram (typically the refined variant) rather than a static structure diagram.

Instead of activity diagrams, any of the other UML dynamic notations could be used. In fact, *any* other notation for the description of software processes, for example, Petri nets, Rules, activity trees/transaction graphs, or programming language-like notations. None of these, however, reaches the degree of acceptance the UML has (and will have for some time to come).

**Participants aspect** There are two kinds of participants of a process pattern, artifacts and resources. In the terminology of the UP, resources refers to tools and machines as well as to people. Artifacts may be either prerequisites or deliverables of a process pattern, or both. To represent the participants aspect, the UP’s schema may be used.

**Applicability aspect** Apart from the traditional contents of these aspects (i.e. natural language descriptions of the applicability conditions of using a pattern), in our approach, this aspects may be refined by formal preconditions on participants. This is particularly relevant for documents, of course, but may also be applied to resources. This opens the road to formal analysis of software processes and automatic enactment.

At this point, a little digression into the UML metamodel is called for. First observe that all notational elements of the UML correspond to a metaclass, i.e. a state chart diagram corresponds to a `StateMachine`, an activity diagram to an `ActivityGraph`, an action state node to an `ActionState`, an object flow node to an `ObjectFlowState` and so on. I have introduced `Document` as a special kind of

**Classifier**, and thus it may have a **StateMachine** to describe its lifecycle. In the context of **ActivityGraph**, an **ObjectFlowState** has a **type**, that is, an association to a **Classifier** such as a **ClassifierInState** (and **Artifact**). This may be used to represent the current state of the document’s lifecycle.

Or, in other words (and ending the digression), documents and other resources have lifecycles, and a particular state of these lifecycles may be used to inscribe document (or resource) icons in workflow diagrams. For instance, artifact may be in states “checked out”, “tested”, or “approved” and so on. So, this state may be used as a specification of a precondition of a resource. Also, any other side conditions (including timing constraints, or complex logical predicates) may be expressed using OCL.

**Consequence aspect** All that has been said in the previous section applies analogously to the consequences aspect: the state attached to a document or other resource is a postcondition, rather than a precondition.

**Sample Execution aspect** As I have said before, design result patterns are realized as programs, and process patterns are realized by people in organizations, equipped with tools and applying techniques. So, the process pattern analog of sample code is a sample execution, giving an example of how the respective process pattern might be realized by an organization, that is, who is responsible for what, which techniques and tools are appropriate, how parts of the work are to be synchronized and integrated, which change and version control system may be used and so on.

For example, an initial class model may be generated by conducting structured interviews with domain experts, and then extracting nouns and verbs from such interviews to define analysis level classes and methods, respectively. For analysis level integration testing, role playing with paper prototypes; for user interface design and validation, storyboards are useful, and so on. On the design level, techniques like walk-throughs, inspection, and automatic consistency checking (cf. [26]) might be appropriate.

**Related Patterns aspect** In design patterns, this may only refer to other design result patterns. Process patterns, however, may refer to any kind of patterns, including result patterns, other process patterns, and organizational patterns (cf. [13, 7]).

## 5 Using process patterns

Consider now the pattern-representation of the process fragment Refine. This example is abstracted to the bare minimum—real life examples would be more elaborated.

**Name** Refine

**Classification** process pattern / realization / construction / architecture

**Intent** Realize a subsystem by a divide-and-conquer strategy.

**Also known as** Top-down, divide-and-conquer.

**Motivation** When a system is too complex to be implemented directly, and if purchasing it from another source is not possible, one might approach the problem by splitting up the system in simpler subsystems.

**Applicability** Whenever a complex system is to be realized. Requires the functionality of the system to be specified.

**Process** See Figure 5, right.

**Participants** See Figure 5, left. Artifacts used: system specification consisting of use cases, scenarios, other requirements, plans for budget, resources, and schedule. Artifacts created: system implementation consisting of executable, documentation, and validation report. Resources: varying.

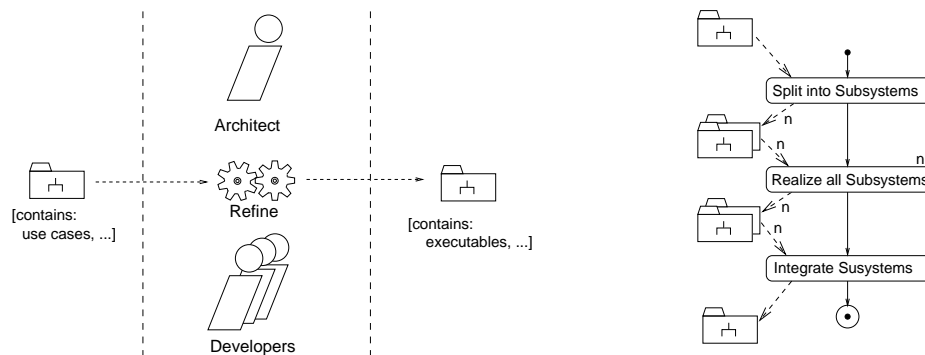
**Consequences** The only realistic choice for complex systems that can not be purchased. Requires that the system in question can be divided into smaller parts at all (not necessarily the case e.g. for time-critical systems). Once the system has been split up, it may be difficult to refactor it. Requires considerable architectural expertise on behalf of the people doing the split.

**Sample Execution** omitted here due to space restrictions

**Implementation** Currently no adequate CASE-tool support available.

**Known Uses** Plenty, e.g. compiler construction.

**Related Patterns** Usually applied as a consequence of the Realize pattern, which in turn is called again for each of the new subsystems.

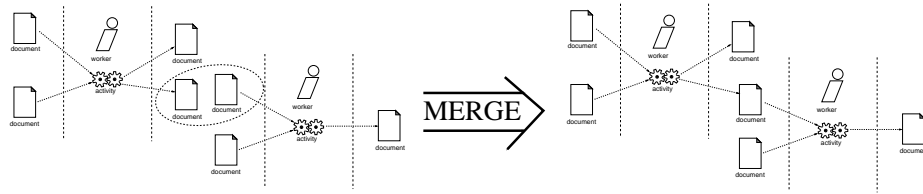


**Fig. 5.** The Refine pattern: participants (left) and process (right).

Suppose that the language of patterns sketched first in Figure 1 were available in this scheme. There are two fundamentally different ways of applying languages of patterns, namely after-the-fact, to document some result, or generatively to instantiate some process. In the first case, there may be steps in the process that are not documented by patterns, so the developers can focus their effort on those

parts of a project's process that needs their attention most. In the remainder, we consider only the second kind of process pattern application.

Starting from some predefined initial state (that is, a set of artifacts), developers may now apply any of those patterns, whose applicability aspect is satisfied, that is, all participants are available in the required state, and all OCL constraints are satisfied. Observe that applying pattern  $y$  on (some of) the artifacts created by pattern  $x$  amounts basically to a sequential composition by merging these artifacts (cf. Figure 6).




---

**Fig. 6.** Individual patterns may be merged to form a development graph.

---

This also shows, how a traditional process may be re-constructed using a suitable language of process patterns. Some of the benefits of pattern languages for processes may be realized even here, e.g. a more compact representation, usage of UML, and better control over where to demand which level of formality.

Both activity and document-oriented process descriptions may be achieved using process patterns: to achieve an activity-oriented style of process description, the document-types and states are simply left abstract. To mimic a document-oriented style, the documents are filled in with all necessary detail, but the activities are left coarse.

In a rigid development process, both of these aspects would be fully specified. Only process patterns provided with the process could be used to create or modify documents. In a looser development process, the developer would also be allowed to create and modify documents in an ad hoc style. Later on, these activities might again be cast into the framework of the process patterns language, should this be desired, as a form of commenting.

## 6 Conclusions

**Summary** In this paper, I have defined a description scheme for process patterns very similar to the one used for design result patterns. Within this scheme the UML is used to describe the aspects process, participants, applicability, and consequences.

In a nutshell, process patterns may be used to describe processes at an arbitrary level of precision. They are now generally understood, as is the UML. The

kind of process patterns I have presented here can easily accommodate traditional and component-based (i.e. fractal) process models. In general, they offer a very flexible, yet precise and generally understood description formalism.

Other approaches are much less elaborated, and, by not using the UML, also of smaller practical use.

**Future work** Our group is currently working on a CASE-system to support usage of process patterns. In particular, cataloging and retrieval, enactment, and checking of satisfaction of abstract design-guidelines will be supported in this tool. With this tool, it will then be possible to conduct field studies of realistic size. So far, I have used the kinds of process patterns proposed here only in the classroom, though with encouraging results. In [26], I have proposed a process language for architectural modeling, but other application areas, including those of classical workflows are possible, but have not yet been examined.

One other important other strand of our work is the definition of a formal semantics for UML ActivityGraphs. This will be necessary to enact and formally analyze workflow graphs. As of writing this, there is no generally agreed upon semantics available. However, there is a plethora of more or less complete formal semantics for UML StateMachines, and ActivityGraph is a much simplified case of these. Recent advances in this area (cf. [5, 3]) and our own work should amount to a satisfying formal semantics pretty soon now. Such analysis methods would apply as well to process patterns as to traditional process descriptions.

## References

1. Thomas Allweyer and Peter Loos. Process Orientation in UML through Integration of Event-Driven Process Chains. In Pierre-Alain Muller and Jean Bézivin, editors, *International Workshop «UML» '98: Beyond the Notation*, pages 183–193. Ecole Supérieure des Sciences Appliquées pour l'Ingénieur—Mulhouse, Université de Haute-Alsace, 1998.
2. Scott W. Ambler. *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.
3. Alistair Barros, Keith Duddy, Michael Lawley, Zoran Milosevic, Kerry Raymond, and Andrew Wood. Processes, Roles and Events: UML Concepts for Enterprise Architecture. In Selic et al. [25], pages 62–77.
4. Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A Componentware Development Methodology based on Process Patterns. In Joseph Yoder, editor, *Proc. 5<sup>th</sup> Annual Conf. on the Pattern Languages of Programs (PLoP)*, 1998.
5. Christie Bolton and Jim Davies. On Giving a Behavioural Semantics to Activity Graphs. In Reggio et al. [23], pages 17–22. Also appeared as Technical Report No. 0006 of the Ludwig-Maximilians-Universität, München, Fakultät für Informatik, October 2000.
6. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., 1998.

7. James O. Coplien. A Generative Development-Process Pattern. In Coplien and Schmidt [8], pages 183–238.
8. James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.
9. Desmond Francis D’Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.
10. Brian Foote. A Fractal Model of the Lifecycle of Reusable Objects. In James O. Coplien, Russel Winder, and Susan Hutz, editors, *OOPLSA ’93 Workshop on Process Standards and Iteration*, 1993.
11. Brian Foote and William F. Opdyke. Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In Coplien and Schmidt [8], pages 239–258.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. Neil B. Harrison. *Organizational Patterns for Teams*. Monticello, Illinois, 1995.
14. Wolfgang Hesse. From WOON to EOS: New development methods require a new software process model. In A. Smolyaninov and A. Shestialtynov, editors, *Proc. 1<sup>st</sup> and 2<sup>nd</sup> Intl. Ws. on OO Technology (WOON’96/WOON’97)*, pages 88–101, 1997.
15. Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
16. Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using UML for Software Process Modeling. Number 1687 in LNCS, pages 91–108, 1998.
17. Marc I. Keller and H. Dieter Rombach. Comparison of Software Process Descriptions. pages 7–18, Hakodate, Japan, October 1990. IEEE Computer Society Press.
18. Alexander Knapp. *A Formal Approach to Object-Oriented Software Engineering*. PhD thesis, LMU München, Institut für Informatik, May 2000.
19. Ralf Kneuper. Requirements on Software Process Technology from the Viewpoint of Commercial Software Development. Number 1487 in LNCS, pages 111–115. Springer Verlag, 1998.
20. Meir M. Lehman. Programs, life cycles, and laws of software evolution. *IEEE Transactions on Software Engineering*, 68(9), September 1980.
21. OMG Unified Modeling Language Specification (draft, version 1.4). Technical report, Object Management Group, February 2001. Available at <http://cgi.omg.org/cgi-bin/doc?ad/01-02-14>.
22. Ruben Prieto-Diaz. *Classification of Reusable Modules*, volume I - Concepts and Models, pages 99–124. ACM Press, 1989.
23. Gianna Reggio, Alexander Knapp, Bernhard Rumpe, Bran Selic, and Roel Wieringa, editors. *Dynamic Behavior in UML Models: Semantic Questions. Workshop Proceedings*, Oktober 2000. Also appeared as Technical Report No. 0006 of the Ludwig-Maximilians-Universität, München, Fakultät für Informatik, October 2000.
24. Colette Rolland and Naveen Prakash. Reusable Process Chunks. Number 720 in LNCS, pages 655–666. Springer Verlag, 1993.
25. Bran Selic, Stuart Kent, and Andy Evans, editors. *Proc. 3<sup>rd</sup> Intl. Conf. <<UML>> 2000—Advancing the Standard*, number 1939 in LNCS. Springer Verlag, October 2000.
26. Harald Störrle. *Models of Software Architecture. Design and Analysis with UML and Petri-nets*. PhD thesis, LMU München, Institut für Informatik, December 2000. In print, ISBN 3-8311-1330-0.
27. Bruce Whitenack. RAPPeL: A Requirements-Analysis-Process Pattern Language. In Coplien and Schmidt [8], pages 259–292.