

Describing Process Patterns with UML

Harald Störrle

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

Abstract. Patterns are now widely used for describing software designs. However, they may also be used to describe *process* structure rather than the product structure. This may be accomplished by adapting the well known pattern description schemes to the software process domain. Within this scheme, I use the description techniques and notions of the UML wherever suitable. This, and superior degrees of detail, precision, and expressiveness set this paper apart from previous approaches in this area.

1 Introduction

1.1 Motivation

Workflows concerning software processes have been described in a variety of formalisms and styles (see e.g. [16] 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 [18]). The vast majority of traditional processes like the “Unified Process” (UP, cf. [14]), 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.

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. [24, 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 this obstacle can be overcome quite easily (see Section 2).

On the other hand, UML is now rapidly being developed into a body of mathematically precise formalisms (cf. [22, 17, 25]), 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. [3, 13, 9, 19]). 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. Then the descriptive schema laid down in [11] (and very similarly in [5], 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 a little 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. [23]). 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 [3, 10, 26] which already use the term process pattern, but remain very 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. [8]), 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.

Compared to all the above mentioned approaches, our approach is the only one which seriously tries to incorporate the UML. 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 scheme at all.

2 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. However, describing workflows is still not very well catered for by the UML (cf. [15, 1]). The standard proper contains only activity graphs (and a few comments in one of the "UML Standard Profiles", see [20, part 4]). The UP offers a little more in terms of notations and concepts, and, while technically not part of the standard proper, is a kind of quasi-standard in this area.

The UP extends the UML notation by two elements. First, there is a twin cog wheel-symbol for SubactivityStates¹, which is just a syntactic stereotype. Then, there are symbols for workers and resources (see [14, pp. 25, 145]).

I also add an another element that, while not part of the UML, is universally understood anyway: an icon that looks like a rectangle with a dog's ear that denotes documents (see Figure 1 for an abstract example). The document type may be put inside the rectangle, either as plain text or as a pictogram. The contents of a diagram may be defined in different ways: document types representing a particular UML diagram type or ModelElement are completely specified by this fact

¹ The most important part of the UML is its metamodel. When talking about UML in detail, referring to the metamodel is inevitable—like in this case: SubactivityState is a metaclass. In order to be very clear about what I refer to, metaclasses will always be printed in the font used here.

alone. Documents representing programs are specified by the programming language grammar. Other kinds of document types may or may not be specified explicitly, e.g. by a XML DTD, or plain natural language. Conceptually, I assume that a document represents a Document, which is taken to be a kind of Classifier, e.g. such as may be associated as the type of an ObjectFlowState (see Section 3.4 below). In order to distinguish diagrams using these symbols from ordinary UML activity diagrams, I call them workflow diagrams. Note that, syntactically, workflow diagrams are just UML activity diagrams with special PresentationElements for SubactivityState and Document. This kind of diagrams is found frequently in the UP, where it is used informally (but not properly defined).

Observe that workflow diagrams may be presented in a coarse version, where all SubactivityStates are presented by the UP's twin cog wheel icon. Conceptually, they may be refined by an arbitrary ActivityGraph. If this refinement were resolved and spelt out explicitly, a graph like in Figure 1 (bottom) would arise. Observe that, again, syntactically this is an activity diagram. Similarly, the document types could be refined if that were desired (this step is omitted here).

Observe also, that the coarse version of workflow diagrams exhibit a close correspondence (nit quite an isomorphism) to a Petri-net process.

3 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 [11, 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 [3] for a discussion of the requirements of process vs. result structure patterns).

3.1 Overall description schema

The two best known books on design patterns [5, 11] propose very similar schemas for the description of design patterns: there are only minor differences in terminology. 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 [5, 11] and here.

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 [5], and appears as part of the name aspect in [11]. See Section 3.2 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.

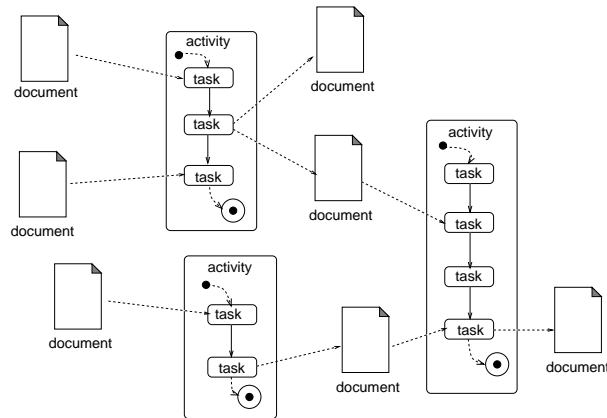
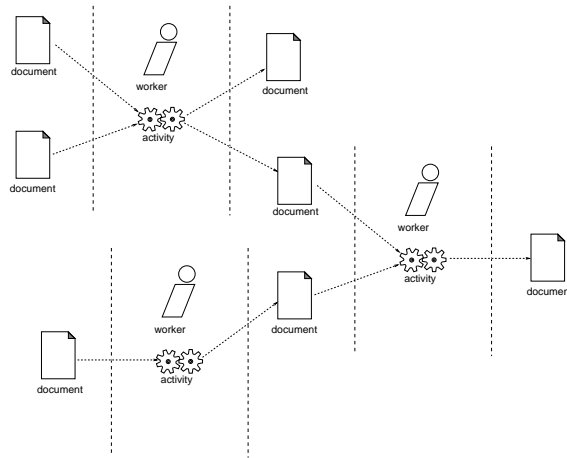


Fig. 1. An activity diagram describing a workflow, coarse (top) and refined (bottom).

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 pre-conditions

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 3.3 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 3.4 below.

POSA [5]	GOF [11]	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.

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 3.7 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 [11] (called "Dynamic Aspects" in [5]): 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.

3.2 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. [21]). 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.² Buschmann et al. propose to use the abstraction level and the problem category ([5, p. 362ff]). The Gang-of-four propose to use the purpose and the scope (see [11, 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 [5], 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.

purpose In [11], 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.

Note that these are exactly the classification *facettes* used in [5, 11]. 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. There is no hard and fast discrimination between the different *simple classes* of these *facettes*.

3.3 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 rather than a static structure diagram³.

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

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

³ In the UML, class diagrams, object diagrams, and package diagrams are abstracted to static structure diagrams.

language-like notations. None of these, however, reaches the degree of acceptance the UML has (and will have for some time to come).

3.4 Participants aspect

There are two kinds of participants of a process pattern, documents and resources. In the terminology of the UP, resources refers to tools and machines as well as to people ("workers", see Figure 1).

A document is taken to be a Classifier, e.g. such as may be associated as the type of an ObjectFlowState. Documents may be either prerequisites or deliverables of a process pattern, or both. They may be represented graphically by the classical symbol, a rectangle with a dog's ear. This icon may be inscribed by the document type in plain text or a pictogram. Document types representing an UML diagram type are specified by this fact entirely already. Compound document types are defined by the structures of the document types they contain. Other kinds of document types may or may not be specified explicitly.

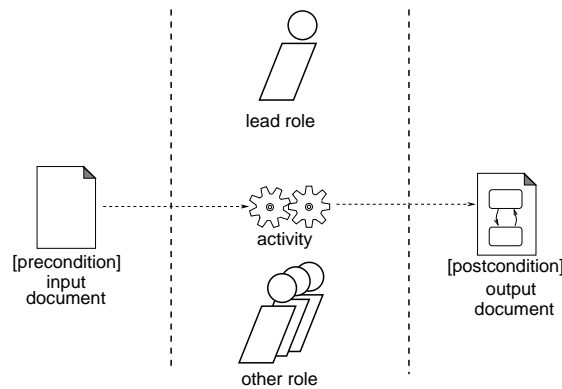


Fig. 2. Schematic notation of an activity in the notational schema of the UP, see [14, pp. 25, 145].

I use the UP's schema for activities (see Figure 2) to represent the participants of a pattern. Note that the twin cog wheel-icon maps to a SubactivityState, that is, it may be refined by a complete ActivityGraph.

3.5 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`. 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, a document 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.

3.6 Consequences 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.

3.7 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. [25]) might be appropriate.

3.8 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. [12, 6]).

4 Using process patterns

In this section I shall discuss how process patterns may be used. Suppose that a language of patterns were available in the scheme outlined above. Starting from

some predefined initial state (that is, a set of documents), 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. Applying one of these patterns results in the modification of existing and creation of new documents, according to the schema provided in the participants aspect. That is, applying a process pattern is, superficially, the composition of schemas like in Figure 2. Looking closer, the details of the activity and the documents' state have to be taken into account, too.

In a very similar way, a traditional process may be re-constructed using a suitable language of process patterns. Here, the schemas of the participants aspect are merged together at the right document-nodes, see Figure 3. 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.

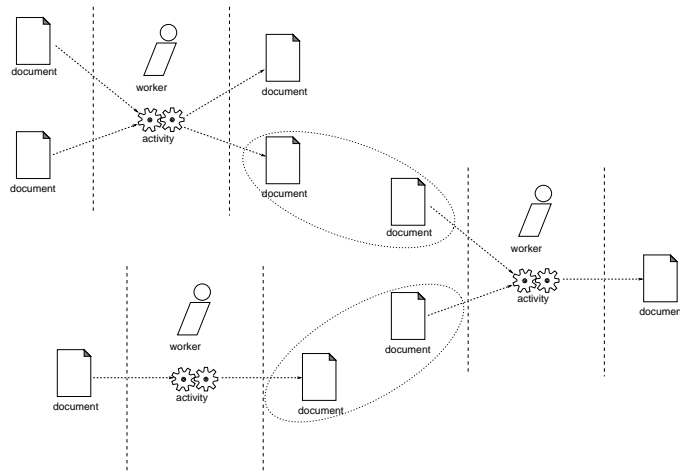


Fig. 3. Individual patterns may be merged to form a development graph.

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.

5 Conclusions

5.1 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.

5.2 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 [25], 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. [4, 2]) 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. 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. [24], pages 62–77.
3. Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A Componentware Development Methodology based on Process Patterns. In Joseph Yoder, editor, *Proc. 5th Annual Conf. on the Pattern Languages of Programs (PLOP)*, 1998. Available at www4.informatik.tu-muenchen.de/~rausch/publications/.

4. Christie Bolton and Jim Davies. On Giving a Behavioural Semantics to Activity Graphs. In Reggio et al. [22], pages 17–22. Also appeared as Technical Report No. 0006 of the Ludwig-Maximilians-Universität, München, Fakultät für Informatik, October 2000.
5. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., 1998. Also published in German by Addison-Weseley, 1998, under the title "Pattern-Orientierte Software-Architektur", to which all citations refer.
6. James O. Coplien. A Generative Development-Process Pattern. In Coplien and Schmidt [7], pages 183–238. Based on the proceedings of PLoP'94. Available also at www1.bell-labs.com/user/cope/Patterns/Process/index.html.
7. James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995. Based on the proceedings of PLoP'94. Available also at www1.bell-labs.com/user/cope/Patterns/Process/index.html.
8. Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.
9. 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. Available at <http://www.laputan.org/frameworks/fractal.html>.
10. Brian Foote and William F. Opdyke. Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In Coplien and Schmidt [7], pages 239–258. Available at www.laputan.org/lifecycle/Lifecycle.html.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. Neil B. Harrison. Organizational Patterns for Teams. Monticello, Illinois, 1995. Available via <http://st-www.cs.uiuc.edu/~plop/>.
13. Wolfgang Hesse. From WOON to EOS: New development methods require a new software process model. In A. Smolyaninov and A. Sheshialynow, editors, *Proc. 1st and 2nd Intl. Ws. on OO Technology (WOON'96/WOON'97)*, pages 88–101, 1997.
14. Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
15. Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using UML for Software Process Modeling. Number 1687 in LNCS, pages 91–108. ACM, 1998.
16. Marc I. Keller and H. Dieter Rombach. Comparison of Software Process Descriptions. pages 7–18, Hakodate, Japan, October 1990. IEEE Computer Society Press.
17. Alexander Knapp. *A Formal Approach to Object-Oriented Software Engineering*. PhD thesis, Ludwig-Maximilians-Universität München, Institut für Informatik, May 2000.
18. Ralf Kneuper. Requirements on Software Process Technology from the Viewpoint of Commercial Software Development. Number 1487 in LNCS, pages 111–115. Springer Verlag, 1998.

19. Meir M. Lehman. Programs, life cycles, and laws of software evolution. *IEEE Transactions on Software Engineering*, 68(9), September 1980.
20. OMG Unified Modeling Language Specification (version 1.3). Technical report, Object Management Group, June 1998. Available at uml.shl.com.
21. Ruben Prieto-Diaz. *Classification of Reusable Modules*, volume I - Concepts and Models, pages 99–124. ACM Press, 1989.
22. 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.
23. Colette Rolland and Naveen Prakash. Reusable Process Chunks. Number 720 in LNCS, pages 655–666. Springer Verlag, 1993.
24. Bran Selic, Stuart Kent, and Andy Evans, editors. *Proc. 3rd Intl. Conf. <<UML>> 2000—Advancing the Standard*, number 1939 in LNCS. Springer Verlag, October 2000.
25. Harald Störrle. *Models of Software Architecture. Design and Analysis with UML and Petri-nets*. PhD thesis, Ludwig-Maximilians-Universität München, Institut für Informatik, November 2000. to be published.
26. Bruce Whitenack. RAPPeL: A Requirements-Analysis-Process Pattern Language. In Coplien and Schmidt [7], pages 259–292. Based on the proceedings of PLoP'94. Available also at www1.bell-labs.com/user/cope/Patterns/Process/index.html.