An Introduction to Process Patterns

An AmbySoft Inc. White Paper

Scott W. Ambler
Object-Oriented Consultant
AmbySoft Inc.

Material for this paper has be excerpted from:

Process Patterns: Building Large-Scale Systems Using Object Technology
Scott W. Ambler
SIGS Books/Cambridge University Press, July 1998
and
More Process Patterns: Delivering Large-Scale Systems Using Object Technology
Scott W. Ambler
SIGS Books/Cambridge University Press, September 1998

This Version: June 27, 1998

Table Of Contents

Not only do the same type of problems occur across domains, problems whose solutions are addressed by design patterns and analysis patterns, but the strategies that software professionals employ to solve problems recur across organizations, strategies that can be described by *process patterns*. A process pattern describes a collection of general techniques, actions, and/or tasks for developing object-oriented software. Process patterns are the reusable building blocks from which your organization will develop a tailored software process that meets its exact needs.

1. What is a Process Pattern?

The repetition of patterns is quite a different thing than the repetition of parts. Indeed, the different parts will be unique because the patterns are the same.
– Christopher Alexander

To define what a process pattern is, I would first like to explore its two root words: *process* and *pattern*. A process is defined as a series of actions in which one or more inputs are used to produce one or more outputs. Defining a pattern is a little more difficult. Alexander (1979) hints at the definition of a pattern by pointing out that the same broad features keep recurring over and over again, although in their detailed appearance these broad features are never the same. Alexander shows that although every building is unique, each may be created by following a collection of general patterns. In other words, a pattern is a general solution to a common problem or issue, one from which a specific solution may be derived.

Coplien (1995), in his paper "A Generative Development-Process Pattern Language," hints at the definition for the term "process pattern" in his statement that "the patterns of activity within an organization (and hence within its project) are called a process." For the purposes of this paper, I define a process pattern to be a collection of general techniques, actions, and/or tasks (activities) for developing object-oriented software. An important feature of a process pattern is that it describes what you should do but not the exact details of how you should do something. When applied together in an organized manner, process patterns can be used to construct software process for your organization. Because process patterns do not specify how to perform a given task, they can be used reusable building blocks from which you may tailor a software process that meets the specific needs of your organization.

Related to process patterns are something called organizational patterns, patterns that describe common management techniques or organizational structures. The fact is that process patterns and organizational patterns go hand-in-hand, although the focus of this white paper, however, is not organizational patterns.

2. Types of Process Patterns

Patterns can exist at all scales.
– Christopher Alexander

One feature which I believe is important for process patterns is that it be possible to develop them for all aspects of development. The point to be made is that the scope of a single process pattern may range from a high-level view of how applications are developed to a more-detailed view of a specific portion of the software process.

I believe that there are three types of process patterns. In order of increasing scale they are:

1. **Task process patterns.** This type of process pattern depicts the detailed steps to perform a specific task, such as the Technical Review and Reuse First process patterns

2. **Stage process patterns.** This type of process pattern depicts the steps, which are often performed iteratively, of a single project stage. A project stage is a higher-level form of process pattern, one that is often composed of several task process patterns. A stage process pattern is presented for each project stage of a software process, such as the Program stage presented in Figure 2, of the Object-Oriented Software Process (OOSP) of Figure 4.

3. **Phase process patterns.** This type of process pattern depicts the interactions between the stage process patterns for a single project phase, such as the Initiate and Delivery phases. A phase process pattern, the Construct pattern is depicted in Figure 3, is a collection of two or more stage process patterns. Project phases are performed in a serial manner, this is true of both structured development and of object development. A common myth within the object industry is that object-oriented (OO) development is iterative in nature. Although this may be true of the small, pilot projects prevalent in the early 1990s, it is not true of today's large-scale, mission-critical applications. The reality is that OO development is serial in the large, iterative in the small, delivering incremental releases over time (Ambler, 1998b). Phase process patterns are performed in serial order, made up of stage process patterns which are performed iteratively.

To date, the vast majority of the work in process patterns has been in what I would consider task process patterns, and very little work in phase and stage process patterns (although you could easily argue that some organizational patterns encroach on this territory). In the three following sections I will provide an example each of a task process pattern, a stage process pattern, and a phase process pattern, all taken from the OOSP process pattern language (Ambler, 1998b; Ambler 1998c). In Section 6 I present the Object-Oriented Software Process (OOSP), a life cycle composed of phase and stage process patterns, which in turn are enhanced by task process patterns. My experience is that when you look at process patterns from the point of view of defining/tailoring a software process for an organization then you need the three types of process patterns described in this section to be effective. I believe that task process patterns are a key component of a software process, but that phase and stage process patterns are needed to organize them and to put them into a meaningful context for your organization.

3. Task Process Pattern – Technical Review

The deliverables created during development need to be validated to ensure that they meet the needs of your user community and the quality standards of your organization. The Technical Review task process pattern describes how to organize, conduct, and follow through on the review of one or more deliverables. This pattern is of the same theme as the Validation by Teams (Harrison, 1996), Review (Coplien, 1995), Creator-Reviewer (Weir, 1998), and Group Validation (Coplien, 1995) task process patterns.

3.1 Forces

There are several applicable forces motivating the Technical Review process pattern. First, the deliverables (models, prototypes, documents, source code, …) produced during the development process help to define the software and related products to be released to your user community, therefore you should validate that each deliverable is of sufficient quality before building on it. Second, because the cost of fixing defects increases the later they are detected in the development life cycle (Ambler, 1998a) as a result of the error snowballing throughout your work, you want to detect defects as early as possible so you may fix them early (and inexpensively). Third, because it is difficult to review your own work you want "a second set of eyes" to review a deliverable. Fourth, you want to communicate your work to others, and one way to do that is to have your teammates review the deliverables that you produce.

3.2 Initial Context

There are one or more deliverables to be reviewed, the deliverables are ready to be reviewed, and the development team is ready to have the deliverables reviewed.

3.3 Solution

Figure 1 shows that there are six basic steps to the Technical Review process pattern (model reviews, document reviews, prototype reviews, requirement reviews, and code inspections are all specific processes that follow the Technical Review process pattern). The steps of a technical review are as follows:

1. **The development team prepares for review.** The item(s) that are to be reviewed are gathered, organized appropriately, and packaged so that they may be presented to the reviewers.
2. **The development team indicates that they are ready for review**. The development team must inform the review manager, often a member of quality assurance, when they are ready to have their work reviewed as well as what they intend to have reviewed.
3. **The review manager performs a cursory review**. The first thing that the review manager must do is determine if the development team has produced work that is ready to be reviewed. The manager will probably discuss the development team's work with the team leader and do a quick rundown of what they have produced. The main goal is to ensure that the work to be reviewed is good enough to warrant getting a review team together.
4. **The review manager plans and organizes the review**. The review manager must schedule a review room and any equipment needed for the review, invite the proper people, and distribute any materials ahead of time that are needed for the review. The potential contents of a review package are discussed in the next section.
5. **The review takes place**. Technical reviews can take anywhere from several hours to several days, depending on the size of what is being reviewed, although the best reviews are less than two hours so as not to overwhelm the people involved. The entire development team should attend, or at least the people responsible for what is being reviewed, to answer questions and to explain/clarify their work. There are typically between three to five reviewers, as well as the review manager, all of whom are responsible for doing the review. It is important that all

material is reviewed. It is too easy to look at something quickly and assume that it is right. It is the job of the review facilitator to ensure that everything is looked at, that everything is questioned.

6. **The review results are acted on**. A document is produced during the review describing both the strengths and weaknesses of the work being reviewed. This document should provide both a description of any weakness, why it is a weakness, and provide an indication of what needs to be addressed to fix the weakness. This document will be given to the development team so that they can act on it, and to the review manager to be used in follow-up reviews in which the work is inspected again to verify that the weaknesses were addressed.

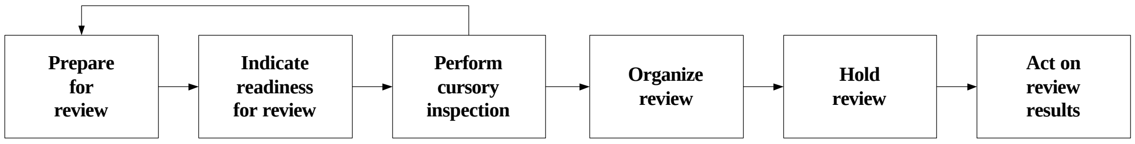

Figure 1. The Technical Review process pattern.

3.4 Resulting Context

Senior management is assured that the development team has produced quality deliverables that meet the needs of their user community. The development team, and the reviewers, have a better understanding of the deliverables that they are building and how their work fits into the overall software project. Individual team members and reviewers are likely to learn new techniques during the review, either techniques applied to the deliverable itself, management techniques applied during the review, or development techniques suggested during the review to improve the deliverable.

4. Stage Process Pattern – Program

An important aspect of software development, one of many to be exact, is the actual development of the source code. As experienced developers know, there is far more to programming than just sitting down at a computer and typing in source code. The Program stage process pattern describes the iterative tasks/activities of programming.

4.1 Forces

Programmers need to develop software that meets the needs of their user communities, needs that have been reflected in the form of models and documents developed during the Model stage and the Define and Validate Initial Requirements stage (Ambler, 1998b). The developed source code should reflect the information contained in these deliverables, yet at the same time may drive changes to them as programmers gain a detailed understanding of the domain (usually more detailed than the modelers have). Furthermore, most organizations want software to be developed in a timely and efficient manner but at the same time want it to be maintainable and extensible so that changes in the future may be made efficiently and quickly.

4.2 Initial Context/Entry Conditions

Several conditions must be met before coding may begin. First, your design models should be in place for the code that you intend to write. Second, your project infrastructure should be in place, defined during the Define Infrastructure stage of the Initiate phase (see Figure 4). The infrastructure includes the development and supporting tools that your programmers will use as well as the standards and guidelines that they will follow. Third, programmers must be available to do the work.

4.3 Solution

Figure 2 depicts the process pattern for the Program stage, showing that there is more to this stage than simply writing source code: you need to understand the models, then seek out reusable artifacts to reduce your work load, then document what you are going to write, then write the code, then inspect and improve it, then test and fix it, and then finally package it.

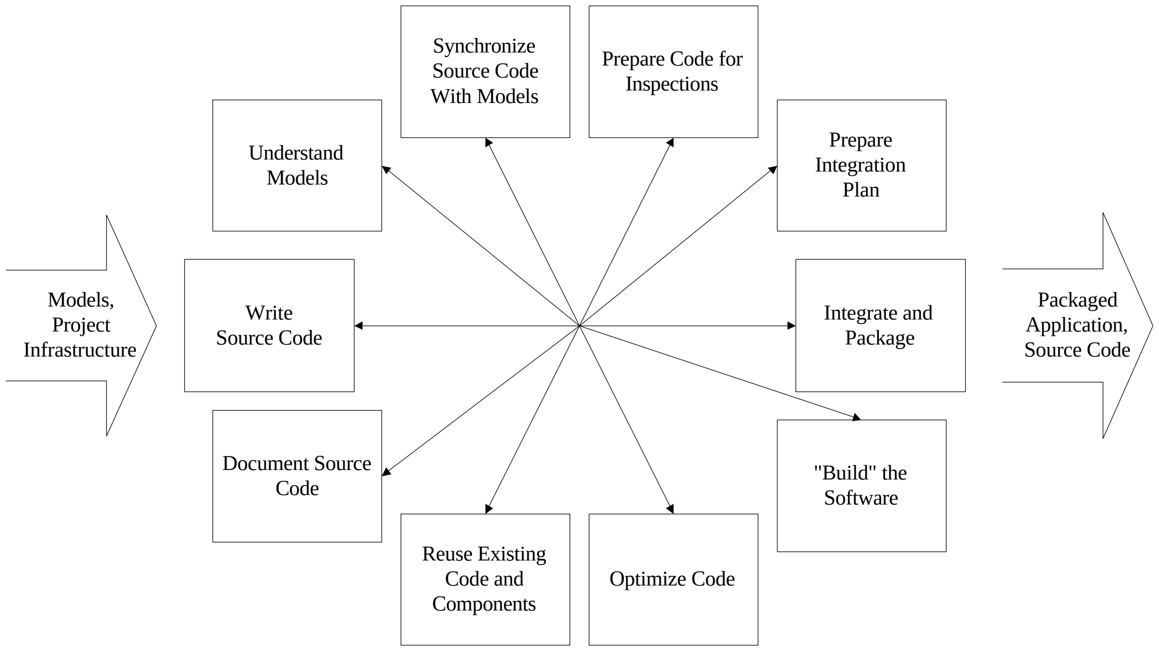
Synchronize Source Code With Models
Prepare Code for Inspections
Understand Models
Prepare Integration Plan
Models, Project Infrastructure
Write Source Code
Integrate and Package
Packaged Application, Source Code
Document Source Code
"Build" the Software
Reuse Existing Code and Components
Optimize Code

Figure 2. The Program process pattern.

The very first thing that should occur before coding begins is that programmers must take the time to understand the models that define what they are to build. Although this sounds straightforward and obvious, it very often is not done. If programmers do not take the time to understand the model before they begin coding, then why bother creating the models in the first place (ideally the programmers were involved in the development of the design models)? The goal is for programmers to understand the design issues and trade-offs involved, and how their code will fit into the overall application. They must invest the time to understand both the problem and the solution before they write the code.

Once your programmers have taken the time to review the models and to understand what needs to be built, they should then look to see what has already been built. One of the promises of object-oriented development is that of increased reuse, but you need to recognize the fact that you can only increase reuse on your projects if you choose to do so. This means that somebody has to make an effort to reuse existing artifacts, and the best way to do so is to look for reusable artifacts before you start writing code. Your design should have also considered reuse issues.

Before you write your source code, you should first document it. Although this seems non-intuitive at first, experience shows that programmers who start by writing brief comments that describe their coding logic are significantly more productive than programmers who do not. The reason for this is simple: the hard part about programming is getting the logic right, not writing the actual code that implements that logic. Writing source code before documenting it is a common process antipattern, one that you want to avoid if possible. By documenting your logic first in the form of brief comments, also called pseudocode, you invest the time to get your logic in place, avoiding the problem of writing a lot of code that later needs to be scrapped because it does not work right. Think first, then code.

Once programmers have invested the time to understand the models that they are implementing, searched for reusable components to reduce their workload, and then written at least initial documentation for their code, they are ready to actually begin writing object-oriented source code. The code that is written should conform to the standards and guidelines defined and selected for your project – conformance that will be verified by code reviews.

Throughout the coding process programmers must constantly take the time to synchronize their source code with the model. During coding it often becomes clear that the models/documentation doe not include all of the information needed by the coder. If this is the case, then the coder should first talk to the modeler to determine a solution, and then both the model(s) and the code should be updated to reflect the solution. The important thing is that the code reflects the information contained in the models and vice versa.

The source code produced by a development team will be inspected, in whole or in part, as part of the Test in the Small stage (Ambler, 1998b). To prepare for a code review, which will apply the Technical Review task process pattern, a programmer should be reasonably assured that their code will pass inspection. This means that the code satisfies the design, follows standards, is well documented, is easy to understand, and is well written.

My experience is that you want to leave optimization to the end because you want to optimize only the code that needs it: very often a small percentage of your code results in the vast majority of the processing time, and this is the code that you should be optimizing. A classic mistake made by inexperienced programmers it to try to optimize all of their code, even code that already runs fast enough. Personally, I prefer to optimize the code that needs it and then move on to more interesting things than trying to squeeze out every single CPU cycle.

Creating a build is the act of compiling and linking your source code in compiled languages such as Java and C++, or packaging your code in languages like Smalltalk. As you would expect, you use tools called compilers, linkers, and packagers to create a build, which I will refer to as “builders” for the sake of our discussion. Successful builds produce software that can be run and tested, whereas unsuccessful builds produce an indication of what your builder did not like about your code. Builds are important because they help to find integration problems in your code; if your code does not integrate well, it will likely break your build, and the build shows that your code does in fact compile – often an incredible morale booster for your programming team.

Long before you attempt to integrate and package your application you must first have a plan to do so. Three key deliverables are needed by the developers responsible for integrating and packaging your application: an integration plan that describes the schedule, resources, and approach to integrating the elements of an application, a version description document (VDD) that describes the elements of an application and their intrarelationships, and a deployment diagram that shows how the software will be deployed to the user community. To integrate and package your application, you must build and smoke-test your software, produce the supporting documentation, and develop the installation process for your software. If you perform daily builds, then the first step is straightforward, as you merely need to follow your existing build procedure to gather and then build the source code files that make up your application.

4.4 Resulting Context/Exit Conditions

The following conditions must be met before the Program stage may be considered complete. First, your code should have passed inspection. Second, the code should work (it passed testing). Third, the code should have been optimized sufficiently. Fourth, if applicable the software should be integrated and packaged for delivery.

5. Phase Process Patterns

The main goal of the Construct phase, the second serial phase of the Object-Oriented Software Process (OOSP) of Figure 4, is to build working software that is ready to be tested and delivered to your user community. This software will be accompanied by the models and source code that was used to develop it, a test plan to verify that the software works, any reusable artifacts that can be used on future projects, and the initial documentation and training plans supporting the software.

5.1 Forces

There are several forces applicable to the Construct Phase, including a lack of understanding of how to work the phase by both senior management and by developers; an unwarranted focus on programming to the neglect of modeling, testing, and generalization; and a penchant by everyone involved to cut corners and take shortcuts that more often than not result in poor quality software that is late and over budget anyway.

5.2 Initial Context/Entry Conditions

The Construct phase can be entered two different ways, either from the Initiate phase or from the Maintain and Support phase (see Figure 4). Regardless, there are several conditions that must be met before the Construct phase may begin. First, the key project management documents (project plan, estimate, schedule, risk assessment, …) should be available and up-to-date. Second, the project infrastructure should be defined, or at least a good portion of it is defined, so that the tools, processes, and standards are available to your development team. Third, the high-level requirements for your software should be in place as well as the project charter for your team. Fourth, maintenance changes applicable to the software you are constructing should be allocated to the release that you are working on (this is applicable only for existing software that is being updated). Finally, your development team should be selected and made available (as best as possible) for when they are needed by your project.

5.3 Solution

Figure 3 presents the Construct phase process pattern. An important implication of Figure 3 is that you are not starting from scratch when you enter the Construct phase – important management documents such as the project plan and initial risk assessment have been defined, the initial requirements for your application should have been defined, the project infrastructure is (mostly) defined, and the funding and charter for the project have been obtained. The four iterative stages of the Construct phase are highly interrelated. The Model stage concentrates on the abstraction of the technical and/or problem domain via the use of diagrams, documents, and prototypes. The Program stage (see Figure 2) focuses on the development and documentation of program source code. The Generalize Stage is critical to your organization’s reuse efforts as it focuses on identifying reusable items, or items that may become reusable once modified, from a software project. This is effectively “opportunistic reuse” because you attempt to develop reusable items by harvesting your work after the fact, instead of “systematic reuse” in which you design software during modeling to be reusable. The goal of the Test In The Small Stage is to verify and validate the deliverables developed by the other stages of the Construct Phase. In many ways this stage is the equivalent of unit testing from the structured world combined with quality assurance techniques such as code inspections and technical reviews.

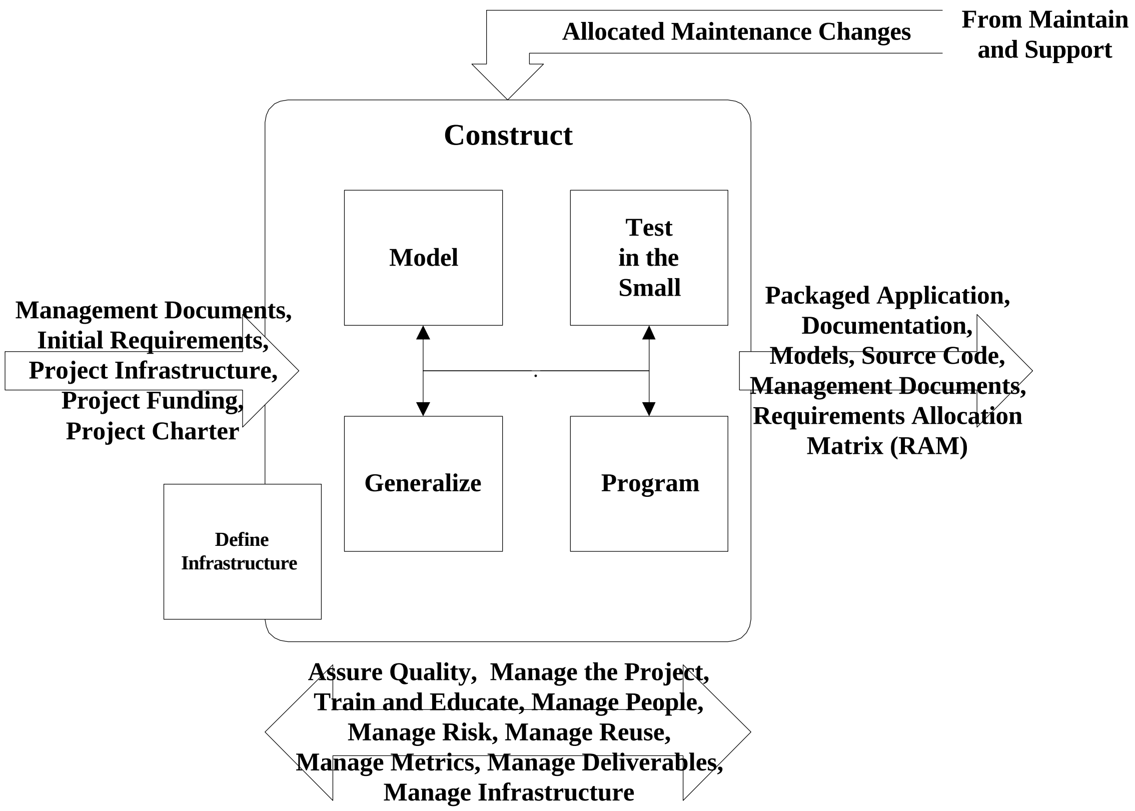
Allocated Maintenance Changes
From Maintain and Support
Construct
Model
Test in the Small
Generalize
Program
Define Infrastructure
Management Documents, Initial Requirements, Project Infrastructure, Project Funding, Project Charter
Packaged Application, Documentation, Models, Source Code, Management Documents, Requirements Allocation Matrix (RAM)
Assure Quality, Manage the Project, Train and Educate, Manage People, Manage Risk, Manage Reuse, Manage Metrics, Manage Deliverables, Manage Infrastructure

Figure 3. The Construct process pattern.

Just because the Construct Phase is iterative in nature, it DOES NOT imply that your developers are allowed to start hacking. The reality of software development is that you must first identify and understand the requirements for something, then you must model them, and then code them. If you have not defined the requirements, then why are you building something? Do you honestly believe that it is more productive for you to start writing code before investing the time to think about and to model what you are building? Truly top-notch developers also know that they must verify their work before moving on to the next task. There's no value modeling requirements that are invalid, or writing source code based on an incorrect model. This means that you need to test your work as you develop it, not at the end when it is often too late to fix discovered problems. I'm not saying that you have to define all of the requirements first, then do all of the modeling, then write all of the code. What I am saying is that any code that you write should be based on a validated model, and that any modeling you do should be based on one or more validated requirements.

5.4 Resulting Context/Exit Conditions

The Construct phase effectively ends when a code/development freeze has been declared. For a code/development freeze to be official, the following deliverables *must* be in place (when applicable): Models (Class Model, Use-Case Model, Sequence Diagrams, …), Requirements Allocation Matrix (RAM), Source code, Master Test/QA Plan, User Documentation, Operations Documentation, Support Documentation, the software itself, Training Plan, Release Plan, and Lessons Learned. At this point your software is ready to move on to the Deliver phase (see Figure 4) where it will be tested in the large, reworked as needed, and released to your user community.

6. Why Process Patterns?

Process patterns are an excellent mechanism for communicating approaches to software development that have proven to be effective in practice. Furthermore, process patterns are the reusable building blocks from which your organization may tailor a mature software process. For example, in Figure 4 we see a depiction of the Object-Oriented Software Process (OOSP), which is composed of four serial phases that in turn are composed of iterative stages (Ambler, 1998b; Ambler, 1998c). The "big arrow" at the bottom of the diagram indicates important tasks critical to the success of a project that are applicable to all stages of development. The phase and stage process patterns, as well as the "big arrow tasks," are in turn enhanced by task process patterns. Process patterns, in the form of the OOSP, have been used to form a mature software process for the development of large-scale, mission-critical software using object technology.

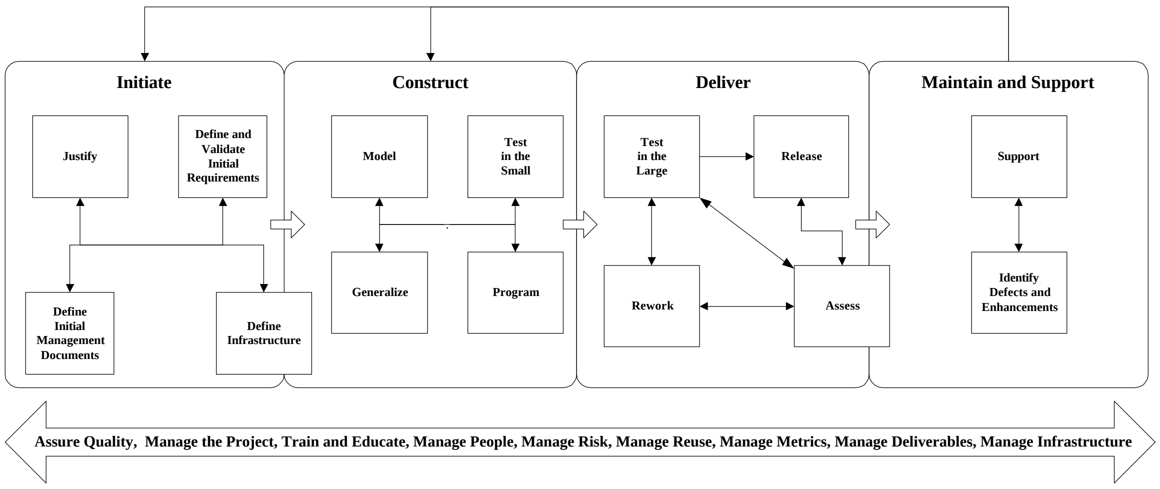
Initiate
Justify
Define and Validate Initial Requirements
Define Initial Management Documents
Define Infrastructure
Construct
Model
Test in the Small
Generalize
Program
Deliver
Test in the Large
Release
Rework
Assess
Maintain and Support
Support
Identify Defects and Enhancements
Assure Quality, Manage the Project, Train and Educate, Manage People, Manage Risk, Manage Reuse, Manage Metrics, Manage Deliverables, Manage Infrastructure

Figure 4. The Object-Oriented Software Process (OOSP).

As you can see in Figure 4, there are four project phases within the OOSP – Initiate, Construct, Deliver, and Maintain and Support – each of which is described by a phase process pattern. Also in Figure 4, you see that there are 14 project stages in the OOSP – Justify, Define and Validate Initial Requirements, Define Initial Management Documents, Define Infrastructure, Model, Program, Test In The Small, Generalize, Test In The Large, Rework, Release, Assess, Support, and Identify Defects and Enhancements – each of which is described by a stage process pattern. Project stages are performed in an iterative manner within the scope of a single project phase. Project phases, on the other hand, are performed in a serial manner within the OOSP.

As indicated above, I believe that process patterns are a key enabler for tailoring/defining a mature software process for your organization. The reality of process improvement, however, is that you cannot make all of the changes that you want to immediately; it is simply too great a change for your organization to absorb at once. This is why we have efforts such as the Software Engineering Institute's (SEI's) Capability Maturity Model (CMM) efforts (SEI, 1995) and the Software Process Improvement Capability Determination (SPICE) efforts (Emam, Drouin, and Melo, 1998) of the International Standards Organization (ISO). Both of these organizations suggest that you prioritize the process improvements that your organization needs to make, expect that it will take several years to make the needed changes, and expect that you will experience difficulties while doing so. Experience shows that organizations that try to make immediate, large-scale process changes are likely to fail doing so.

7. Process Antipatterns

Just as there are process pattern, approaches to development that are proven to work in practice, there are also process antipatterns, approaches to development that are proven to be ineffective in practice. An example of a process antipattern is hacking, an approach to development where little or no effort is spent on analysis and design before coding begins. Unfortunately hacking is probably the most common approach in use today to develop applications. Fundamentally, you first need to do some analysis, then some design, then some coding. There is no way around it, and there is not a single piece of software that can justifiably be developed any other way. How can you possibly develop software without first identifying requirements for it? If there are no requirements then why are you building the software? Second, what is easier to do, first draw a few diagrams which help you to iron out your design before you begin coding, or to just start to madly develop code, throwing portions of it away and then rewriting it when you discover that it does not work? I think that you will agree that there is something to be said for starting out with a few bubbles and lines. It never makes sense to start with coding, you first have to do a little analysis and design.

Having stated that, there is always a few developers who think that the software they are currently working on is the one that breaks this rule. Because they are working on technical software, often for the system or persistence layer, they delude themselves into thinking that they do not need to start with analysis. Pure hogwash. I have been involved in the development of some very technical software, and every time we began by documenting the requirements (i.e. we did analysis), we then modeled our design, and then we coded. Throughout this process we discovered that we missed some requirements, forcing us to update our design and then our code. The point to be made is that we did not hack out the code without first thinking about what we wanted to build and how we wanted to build it. Yes this is common sense, yet many developers do not seem to grasp the concept.

8. Summary

Process patterns are reusable building blocks from which your organization can tailor a mature software process. In this white paper I described three types of process patterns: task process patterns that describe the detailed steps for a specific task; stage process patterns that describe a series of iterative tasks; and phase process patterns that describe a collection of iterative stages. Process patterns describe proven approaches to developing software, approaches that can be used within your organization to increase the quality, maintainability, and extensibility of software.

9. References and Recommended Reading

Alexander, C. (1979). *The Timeless Way of Building*. New York: Oxford University Press.

Ambler, S.W. (1995). *The Object Primer: The Application Developer's Guide To Object Orientation*. New York: SIGS Books.

Ambler, S.W. (1998a). *Building Object Applications That Work – Your Step-by-Step Handbook for Developing Robust Systems With Object Technology*. New York: SIGS Books/Cambridge University Press.

Ambler, S. W. (1998b). *Process Patterns: Building Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press.

Ambler, S. W. (1998c). *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press.

Baudoin, C., Hollowell, G. (1996). *Realizing The Object-Oriented Life Cycle*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Boehm, B.W. (1988). *A Spiral Model Of Software Development And Enhancement*. IEEE Computer, pp. 61-72, 21(5).

Coplien, J.O. (1995). *A Generative Development-Process Pattern Language*. Pattern Languages of Program Design, Addison Wesley Longman, Inc., pp. 183-237.

DeLano, D.E. and Rising, L. (1998). *Patterns for System Testing*. Pattern Languages of Program Design 3, Addison Wesley Longman, Inc., pp. 503-525.

Emam, K. E., Drouin J., and Melo, W. 1998. *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. Los Alamitos, California: IEEE Computer Society Press.

Foote, B. and Opdyke, W.F. (1995). *Life cycle and Refactoring Patterns That Support Evolution and Reuse*. Pattern Languages of Program Design, Addison Wesley Longman, Inc., pp. 239-257.

Graham, I. (1995). *Migrating To Object Technology*. Reading, MA: Addison-Wesley Publishers Ltd.

Graham, I., Henderson-Sellers, B. & Younessi, H. (1997). *The OPEN Process Specification*. New York: ACM Press Books.

Harrison, N.B. (1996). *Organizational Patterns for Teams*. Pattern Languages of Program Design 2, Addison-Wesley Publishing Company., pp. 345-352.

Rational (1996). *Rational Rose: A Rational Approach To Software Development Using Rational Rose*. Rational Software Corporation, CA: Santa Clara.

Software Engineering Institute (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading MA: Addison-Wesley Publishing Company.

Weir, C. (1998). *Patterns for Designing in Teams*. Pattern Languages of Program Design 3, eds. Martin, R.C., Riehle, D., and Buschmann, F., Addison Wesley Longman, Inc., pp. 487-501.

10. Glossary

Antipattern – The description of an approach to solving a common problem, an approach that in time proves to be wrong or highly ineffective.

Object-Oriented Software Process (OOSP) – A collection of process patterns that together describe a complete process for developing, maintaining, and supporting software.

Pattern – The description of a general solution to a common problem or issue from which a detailed solution to a specific problem may be determined. Software development patterns come in many flavors, including but not limited to analysis patterns, design patterns, and process patterns.

Phase process pattern – A process pattern that depicts the interactions between the stage process patterns for a single project phase.

Process – A series of actions in which one or more inputs are used to produce one or more outputs.

Process antipattern – An antipattern which describes an approach and/or series of actions for developing software that is proven to be ineffective and often detrimental to your organization.

Process pattern – A pattern which describes a proven, successful approach and/or series of actions for developing software.

Project phase – The large components of the OOSP which are performed in a serial manner. The four project phases are Initiate, Construct, Deliver, and Maintain & Support.

Project stage – The components of a project phase, performed in an iterative manner, that make up a project phase. For example, the project stages that make up the Construct Phase are Model, Test In The Small, Program, and Generalize.

Stage process pattern – A process pattern which depicts the steps, often performed iteratively, of a single project stage.

Task process pattern – A process pattern that depicts the detailed steps to perform a specific task, such as detailed modeling or performing a technical review.

11. About the Author

Scott W. Ambler is a object development consultant living in the village of Sharon, Ontario, which is 60 km north of Toronto, Canada. Scott is the author of *The Object Primer* (SIGS Books/Cambridge University Press, 1995), *Building Object Applications That Work* (SIGS Books/Cambridge University Press, 1998), *Process Patterns* (SIGS Books/Cambridge University Press, July 1998), and *More Process Patterns* (SIGS Books/Cambridge University Press, September 1998). He has worked with OO technology since 1990 in various roles: Business Architect, System Analyst, System Designer, Project Manager, Smalltalk programmer, Java programmer, and C++ programmer. He has also been active in education and training as both a formal trainer and as an object mentor. Scott is a contributing editor with *Software Development* (http://www.sdmagazine.com) and writes columns for *Object Magazine* (http://www.sigs.com) and *Computing Canada* (http://www.plesman.com). He can be reached via e-mail at **scott@ambysoft.com** and you can visit his personal web site **http://www.ambysoft.com.**

About The Object Primer

The Object Primer is a straightforward, easy to understand introduction to object-oriented analysis and design techniques. Object-orientation is the most important change to system development since the advent of structured methods. While OO is often used to develop complex systems, OO itself does not need to be complicated. This book is different than any other book ever written about object-orientation (OO) – It's written from the point of view of a real-world developer, somebody who has lived through the difficulty of learning this exciting new approach. Readers of *The Object Primer* have found it to be one of the easiest introductory books in OO development on the market today, many of whom have shared their comments and kudos with me. Topics include **CRC modeling**, **use cases**, **use-case scenario testing**, and **class diagramming**.

About Building Object Applications That Work

Building Object Applications That Work is about: **architecting** your applications so that they're maintainable and extensible; analysis and design techniques using the **Unified Modeling Language (UML)**; creating applications for stand-alone, **client/server**, and **distributed** environments; using both **relational** and object-oriented (OO) databases for persistence; OO **metrics**; applying OO **patterns** to improve the quality of your applications; OO **testing** (it's harder, not easier); **user interface design** so your users can actually work with the systems that you build; and **coding** applications in a way that makes them **maintainable** and **extensible**.

Uses the

Unified Modeling Language

About Process Patterns and More Process Patterns

Process Patterns (available July 1998) and More Process Patterns (available September 1998) are ground-breaking texts, describing proven, reusable techniques for developing large-scale, mission-critical object-oriented software that is robust and extensible. A process pattern describes a collection of general techniques, actions, and/or tasks for developing object-oriented software. The focus of the books is The Object-Oriented Software Process (OOSP), presented as a collection of process patterns that are geared toward medium to large-size organizations that need to develop software that support their main line of business. Process patterns are the reusable building blocks from which your organization will develop a tailored software process that meets its exact needs. This book can now be pre-ordered from Cambridge University press (see *http://www.ambysoft.com/processPatterns.html* for details)!

Uses the

Unified Modeling Language

About the AmbySoft Inc. Java Coding Standards

The *AmbySoft Inc. Java Coding Standards* summarizes in one place the common coding standards for Java, as well as presents several guidelines for improving the quality of your code. It is in Adobe PDF format and can be **downloaded** from http://www.ambysoft.com.

12. Change History of This Document

May 30th, 1998:

This document was originally posted on May 4th, 1998 and it mirrored the first part of an article that I wrote for the May 1998 issue of Object Magazine. In both the original version of this paper and the article I argued that I believe that there are five types of process patterns, the three presented here plus life cycle process patterns and approach process patterns. Although I still believe that these high-level types of process patterns exist, it was too radical of a departure from the current work in process patterns, which for the most part focuses on task process patterns, and as a result I took a lot of heat from other pattern developers. The short story is that for now I will back off from the concept and be content with pushing phase and stage process patterns which are the focus of my book *Process Patterns* anyway. The second major change is that I present the process patterns in this paper as templated patterns, patterns which are presented in a specific format, instead of degenerate patterns, patterns that are not presented using a template. Common practice is to present templated patterns, which is how phase and stage process patterns are presented in my book (but not task process patterns for readability reasons), so I decided to flesh out the paper by presenting the example patterns using a template. In the article and the previous version of the paper I concentrated for the most part on the solution aspect of the patterns and not on the context and forces surrounding the patterns. This misjudgment has been rectified.

June 27th, 1998:

References to *More Process Patterns* added.

Index