Software Patterns

Vladan Devedzic

FON - School of Business Administration, University of Belgrade, Yugoslavia

Software patterns are attempts to describe successful solutions to common software problems [26]. Software patterns reflect common conceptual structures of these solutions, and can be applied over and over again when analyzing, designing, and producing applications in a particular context.

The purpose of this chapter is to gradually introduce the concept of software patterns and describe most frequently used classes of patterns. Key ideas are first presented by describing some practical needs and experiences of software designers. Then a commonly used classification of software patterns is shown, followed by an informal example that illustrates the presence of patterns in different fields of software engineering and knowledge engineering. The introductory part of the chapter closes with a brief history of software patterns. The central part begins with a discussion of how the information and the knowledge of patterns are organized in practice and how designers should use that information/knowledge. Then the chapter dedicates one entire section to each one of the frequently used classes of patterns. Within any such a section, the objectives and the scope of the corresponding class of patterns are stated, and an example pattern is described in detail. The remainder of the chapter discusses the concept of pattern languages and some important research issues.

Keywords. Design patterns, analysis patterns, architectural patterns, organizational patterns, process patterns, pattern languages.

Introduction

Patterns are important because they help us to understand how people perceive the world [17]. It is valuable to base a computer system's analysis, organization, and design on this perception. In their own way, patterns represent knowledge and experience that underlies many redesign and reengineering efforts of developers that have struggled to achieve greater reuse and flexibility in their software.

The Key Ideas of Software Patterns

Software patterns contain useful models, their design rationale, and the assumptions and constraints of using the models. They facilitate reuse and sharing of the models and design knowledge by allowing software engineers to adapt the models to fit a specific problem.

It is extremely important to understand that developers *do not invent* software patterns. Rather, they *discover* patterns from experience in building practical systems. Upon discovery, they describe and document the patterns. Each pattern description states explicitly the general problem to which the pattern is applicable, the prescribed solution, assumptions and constraints of using the pattern in practice, and often some other information about the pattern, such as the motivation and driving forces for using the pattern, discussion of the pattern's advantages and disadvantages, and references to some known examples of using that pattern in practical applications. Once the pattern description is made public, other developers can apply it in their projects and use effectively the knowledge and experience it contains.

It is also of primary importance for developers to understand that using software patterns does not require knowledge of specific programming tricks or languages. Patterns only require a little extra effort in order to understand the recurring nature of solutions to specific problems, recognize instances of such problems in building specific software systems, and find a suitable way to apply already known solutions. The pay-off is definitely much larger that the extra effort: increased flexibility, modularity, and reuse of software, reduced development time, and efficient, elegant, and effective design solutions.

Classes of Software Patterns

Patterns exist in several phases of software development. The software patterns community has first discovered, described and classified a number of *design patterns*. They name, abstract, and identify the key aspects of common design structures that are useful in creating reusable object-oriented design [18]. Design patterns identify the participating classes and objects, as well as their roles, collaborations, and distribution of responsibilities in recurring, stereotypical problems of object-oriented design. They are relatively low-level abstractions, in the sense that they are concerned with classes, their instances, and relationships that must be eventually implemented as programs.

Note, however, that there are also lower-level patterns than design patterns. Novice practitioners in the world of software patterns will easily get used to the idea of software patterns starting from *idioms*, which are the patterns of source code levels [8], [18]. They express generally accepted conventions of certain programming languages or cultures, thus representing reuse in the small. For example, returning an integer value as the indicator of success/failure of some function is quite common in C/C++ programming. Some other examples of idioms include naming conventions, the use of exceptions, and the style of writing and documenting class interfaces. Such idioms represent commonly accepted programming style inside the culture.

More recent developments have also identified many patterns that transcend programming and go beyond software design per se [11]. *Analysis patterns* are reusable object models resulting from the activities of object-oriented analysis applied to common business problems [17]. They contain a lot of domain knowledge and experience, yet can be used in almost all kinds of business software.

Patterns for software architectures are concerned with an overall structure of a software system or subsystem that is appropriate to the problem domain and clarify designer’s intentions about the organization of the system or subsystem. [27]. Software architecture patterns are based on selected types of components and connectors (the generalized constituent elements of all software architectures), together with a control structure that governs execution.

Another frequently used class of software patterns is that of *organizational and process patterns* [3], [9], [12], [32]. These patterns apply to software development processes and organizational pragmatics, i.e. to software developers and users, relationships between them, and relationships between people and software [11].

There are many other classes of software patterns as well, although they are narrower in focus than the classes mentioned above. See the Web pages listed in the end of the chapter for a good insight into the full span of software patterns.

An Example

The following example illustrates the existence and use of patterns at the level of software design. It shows how two specific design problems from different domains of software engineering and knowledge engineering abstract to a generalized design problem that can be solved successfully using a design pattern.

Suppose the designer of an intelligent software system wants to develop an explanation generator that can generate explanations for different users. In general, current level of understanding the system's domain is different for different users at any given moment. In intelligent software systems, that fact is reflected in the user model of each user. Novice users should get more general and easy explanations, while the system should generate more complex and detailed explanations to more advanced users. The problem is that the number of possible explanations of the same topic or process is open-ended. It should be anticipated that during the system maintenance another set of knowledge levels could be introduced in order to describe the user model more accurately. The explanation generator should not be modified each time another set of knowledge levels is introduced. On the contrary, it should be easy to add a new knowledge level easily.

The designer of the format converter in a document-processing system would face an analogous problem. A document in a certain reference format should be converted to many other different formats, and it should be easy to extend the converter when it has to support another new format.

The solution for the explanation generator problem is shown in Figure 1. The solution for the format converter problem is quite similar. The general solution is shown in Figure 2, and is well known in the patterns community as the *Builder* pattern [18]. The notation of Unified Modeling Language (UML) [6] is used in both figures to illustrate classes of objects and their relationships.

In Figure 1, the explanation generator can be configured with an ExplanationBuilder, an object that converts a specific knowledge level from the user model to an appropriate type of explanation. Whenever the user requires an explanation, the explanation generator passes the request to the ExplanationBuilder object according to the user's knowledge level. Specialized explanation builders, like EasyExplanationBuilder or AdvancedExplanationBuilder, are responsible for carrying out the request. Note that their concrete implementations of the functions like CreateText and CreateGraphics provide polymorphism in generating explanations.

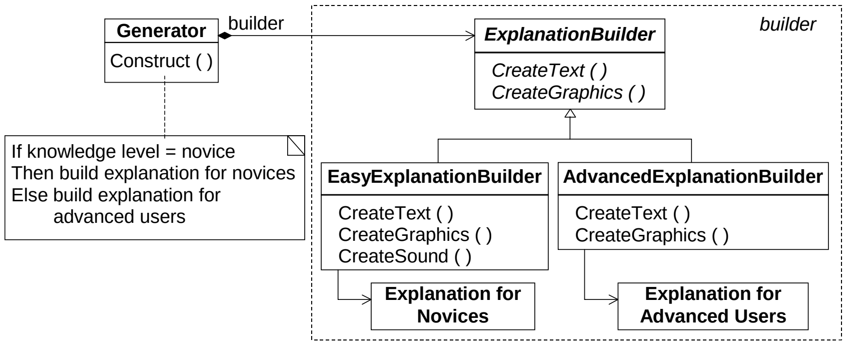

Figure 1 - Design of an explanation generator using the *Builder* pattern

Generally, in both the explanation generator and format converter problems the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled. Also, different representations are needed for the object that's constructed. Using the solution from Figure 2 (the *Builder* pattern) provides both these features in the two problems considered here, as well as in all the similar problems. Since that solution is a fairly general one, it requires some minor adaptation in all specific cases. The abstract class Builder (e.g., ExplanationBuilder in Figure 2) provides the interface for creating parts of the Product object (e.g., text and graphics of the explanation). Each ConcreteBuilder implements that interface. The designer configures the Director object (explanation generator) with the desired Builder object (types of explanation, according to the user's knowledge level). Director notifies the Builder each time a part of the product should be built. The Builder handles the requests from the Director and adds parts to the Product.

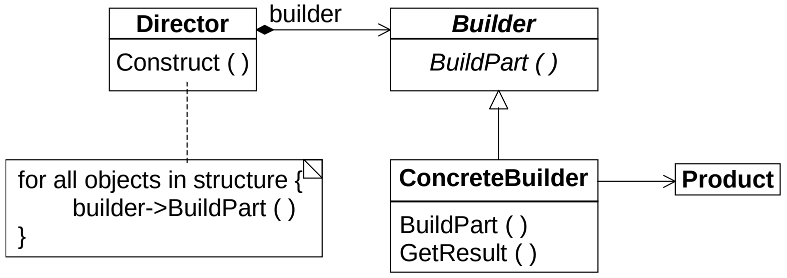

Figure 2 - The *Builder* pattern (after [18])

The point is that there are *many* patterns like the one in Figure 2 that provide general solutions to common problems in software development. Familiarity with such patterns helps developers do their job more efficiently.

History of Software Patterns

Software patterns have evolved from several initiatives. In the late 1980s and early 1990s, object-oriented software development community has come across the idea of applying patterns in architectural design, proposed by Christopher Alexander [2]. It has become clear that similar approach could be used in software design as well. A number of software designers and researchers have started exploring that approach. As the first major result of these efforts, the classical ``Gang of Four" book has appeared in mid-1990s [18]. It has covered a number of patterns discovered in object-oriented software design, and is actually still the best-known catalogue of design patterns to date.

Later efforts have shown that patterns exist in other activities of software development, e.g. in object-oriented analysis [17], in defining software architectures [27], as well as in organizing the entire development process [12]. Many researchers have also noticed that *collections* of inter-related patterns make sense as well. It has led to development of many *pattern languages* [11], [13]. Recently, patterns and pattern languages are used even in creating software documentation [21], as well as in modeling variability in families of software products [20].

Patterns in Practice

Prespecified families of pattern-based solutions to common software engineering problems are available from *pattern catalogues*. Catalogues provide selected and readily usable descriptions of specific patterns. In practice, using patterns from the catalogues also requires some degree of customization and adaptation to the specific project. Developers can browse patterns in the catalogues in search of ideas for already existing solutions to many problems that are common in software engineering. Catalogues are well organized, so browsing them essentially means scanning the relevant patterns' sections that describe the pattern's purpose and how the pattern interrelates with other patterns.

Describing patterns

Individual patterns are described in catalogues using *templates* that provide uniform and consistent representation style. There is, however, no single format of such templates. Typically, there are several different templates for each class of software patterns mentioned above, but there is also the most frequently used one within each class of patterns. For example, many people from the design patterns community use the format proposed in [18]. The format has the sections shown in Figure 3.

- the pattern's ***name*** and ***classification***;
- the pattern's ***rationale*** and ***intent*** - what particular design issue or problem does it address;
- ***alternative name(s)*** of the pattern (if any);
- ***motivation*** - a scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem;
- ***applicability*** - typical situations in which the pattern can be applied;
- ***structure*** - usually a class diagram depicting relations between the classes in the pattern;
- ***participants*** - the classes and/or objects participating in the pattern and their responsibilities;
- ***collaborations*** - how the patterns collaborate to carry out their responsibilities;
- ***consequences*** - what are the results and trade-offs of using the pattern;
- ***implementation*** - pitfalls, hints, techniques, and other issues to be aware of;
- ***sample code*** - illustrations of how the pattern might be implemented;
- ***known uses*** - examples of the pattern found in real systems (at least two examples);
- ***related patterns*** - what other design patterns are related to the one considered.

Figure 3 - A template for describing design patterns (after [18])

On the other hand, catalogues of analysis, organizational and process patterns often follow some form the template shown in Figure 4 [11], [17]. In that template, specifying the solution often involves showing a *diagram*. Most of these "parts" of a pattern are usually shown in the form of simple statements. Along with the pointers to related patterns, sometimes the pattern's *variants* are also described, and a *discussion* of the pattern's usefulness and limitations is provided.

- ***problem*** - the problem that the pattern addresses;
- ***context*** - the context where the pattern is useful;
- ***forces*** - the forces that drive the process of forming a solution;
- ***solution*** - the solution that resolves the forces;
- ***related patterns*** - what other patterns are related to the one considered.

Figure 4 - A more general template for describing software patterns (after [11], [17])

Using patterns

Within the catalogues, patterns are classified so that it is possible to talk about families of related patterns. Moreover, many catalogues show *graphically* how other patterns from the same catalogue relate to the one being described, thus further refining the corresponding class of patterns. The classification helps designers find their way around the catalogues and also find the candidate patterns to be used in solving specific design problems. For example, within the class of design patterns there are *creational patterns* (concerned with the process of object creation), *structural patterns* (dealing with the composition of classes or objects), *behavioral patterns* (characterizing the way in which classes or objects interact and distribute responsibility), etc.

Using patterns in practice has several aspects. First, the designer of a specific system must know (or learn) what roles patterns play in developing an application of that type. For example, design patterns help solve design problems and increase reusability in different ways for different types of systems (e.g., for application programs, toolkits, and frameworks). Second, the designer should use pattern catalogues in order to select the appropriate pattern to solve a specific problem. This means considering:

- what family of patterns is related to that problem;
- in what way patterns from that family solve problems;
- what are their contexts;
- what are the consequences of using them;
- how they are interrelated;
- what kind of reuse could be supported;
- in the reuse sense, what aspects of the particular design is variable.

Finally, once the right pattern is selected, its description should be read thoroughly, paying particular attention to the sections describing its context, structure, and limitations. Then the pattern should be adapted to the particular application domain and software engineering problem.

Design Patterns

Design patterns were the first class of software patterns that has been discovered and that has attracted software developers. They still remain the most widely used kind of software patterns in practical developments.

Objectives and Scope

Design patterns are "simple and elegant solutions to specific problems in object-oriented software design" [18]. They capture static and dynamic structure of these solutions in a consistent and easily applied form. They contain knowledge and experience that underlies many redesign and recoding efforts of developers that have struggled to achieve greater reuse and flexibility in their software. Design patterns show generalized, domain-independent solutions of stereotypical problems that can be used many times without ever doing it the same way twice. Examples of such problems include representation of part-whole hierarchies, dynamic attachment of additional responsibilities to an object, accessing the elements of an aggregate object sequentially without exposing its underlying representation, and many more.

Software designers have discovered dozens of design patterns so far (see the URLs at the end of the chapter, as well as [Gamma et al., 1994], [19], and [22]). It is possible to use design patterns in object-oriented software development in any application domain by adapting the general design solutions that they prescribe to the problems in specific application domains.

Examples

The pattern description format used in this Section follows the one from Figure 3. However, due to space limitations, some sections from that template are omitted.

As the first example, consider the *Builder* pattern [18], [19] already described informally in the introductory Section. It has no alternative names.

Classification and intent. *Builder* is a creational design pattern. Its intent is to help separate the construction of a complex object from its representation. Such a separation makes it possible to create different representations by the same construction process.

Motivation. For a scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem see the example in the introductory Section.

Applicability. The *Builder* pattern is useful when an object must be constructed using different representations at different object instances and when the process of creating the object should be independent of the object's parts and the way they are assembled.

Structure, participants and collaborations. See Figure 2 and the text that explains it.

Consequences. Using the *Builder* pattern lets designers vary the Product's internal representation (e.g., the contents of the explanation in Figure 1). The pattern provides isolation of the code for representation from the code for construction. Construction of the Product is a step-by-step process, and is under the Director's control.

Implementation. The Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. Building the product step-by-step usually means appending results of construction requests to the product.

Known uses. Some examples of using the *Builder* pattern in knowledge engineering include different generators, such as explanation generator, exercise generator, and hint generator [14]. Parsers in various compilers are also designed using the *Builder* pattern.

Related patterns. Builder goes together well with the *Composite* pattern (see below), and is similar to the *Abstract Factory* pattern (see, for example, [19] for details of *Abstract Factory*). In fact, the Products that *Builder* constructs are often *Composites*.

The second example is the *Composite* pattern [18], [19], [22]. It has no alternative names.

Classification and intent. *Composite* is a structural design pattern. Its intent is to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

Motivation. Lesson presentation planner of an educational system for individualized learning may decide to build an agenda of the topics to be presented during the lesson. Complex topics can be divided into simple elements, like concepts, text, graphics, and the like, or into a sequence of subtopics (simpler topics). Each subtopic in turn can be further subdivided into a lower level sequence of simple elements and other subtopics, producing an agenda like in the following example:

```
1.  Topic 1
          Text
          Graphics
          Concept A
          1.1.    Subtopic 1.1
          1.2.    Subtopic 1.2
                  Text
                  Concept B
                  1.2.1.  Subtopic 1.2.1
                  1.2.2.  Subtopic 1.2.2
2.  Topic 2
. . .
```

A simple implementation could define classes for simple elements, such as text and graphics, plus additional classes for subtopics as containers of simple elements. But in that case, the code using these classes would have to treat simple and container objects differently, which would be inefficient from the design point of view. Instead of that, the *Composite* pattern shows how to use recursive composition so that clients don't have to make this distinction. Moreover, using *Composite* makes it easier to achieve any desired depth of subtopics nesting.

Structure. The key to the *Composite* pattern is an abstract class that represents *both* primitive elements (concepts, text, and graphics) and their containers (subtopics). Figure 5 shows the general structure of the *Composite* pattern using the UML notation.

Participants and collaborations. The abstract class Component in Figure 5 corresponds to the contents to be presented during the lesson in the example of lesson presentation planner. It is responsible for providing a common interface to all objects in the composition, both simple (concepts, text, and graphics) and complex (topics and subtopics). The Add and Remove functions in Figure 5 are examples of functions from such a common interface. It also defines the interface for accessing the child components of a complex object (a topic or a subtopic), e.g. the functions Operation and GetChild. The concrete classes Leaf (corresponding to simple contents of a lesson in the example) and Composite (corresponding to topics and subtopics) are derived from Component. The Leaf class represents leaf objects in the composition and has no children. It defines behavior for primitive objects in the composition. The Composite class implements child-related functions from the Component interface, stores child components, and defines behavior for components having children. A client object (e.g., the lesson presentation planner) interacts with any object in the composition through the common interface provided by the Component class. The client's requests are handled directly by Leaf recipients, and are usually forwarded to child components by Composite recipients.

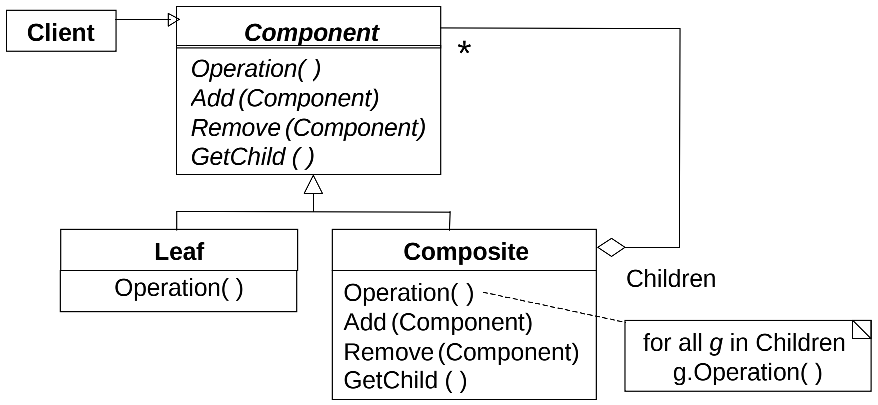

Figure 5- Structure of the *Composite* pattern (after [18])

Applicability. Typical situations in which the *Composite* pattern can be applied in software design include representing part-whole hierarchies of objects (such as composing topics, lessons, and curricula), and those when it should be possible to treat all objects in the composite structure uniformly by a client (such as a lesson presentation planner).

Consequences. Using the *Composite* pattern lets designers define hierarchies consisting of both primitive and composite entities, makes the client simple, also makes it easier to add new kinds of components, but can make the design overly general because it is hard to restrict the components in such a design.

Implementation. There are many things to consider when implementing the *Composite* pattern. Due to space limitations, only one of them will be mentioned here. It is related to child to parent references in a composite structure. Maintaining such references can make the design much more efficient, since it simplifies traversal and management of the composite structure. For example, having appropriate references from graphics objects to topics of a lesson is essential, especially when a graphics object can be presented during more than one topic presentation.

Known uses. Using the *Composite* pattern in designing several kinds of planners as parts of knowledge-based systems is described in [14]. Almost every user interface toolkit uses *Composite* for representing complex screen objects containing text, graphics, images, and other elements. Generally, *Composite* is used very often in object-oriented design.

Related patterns. *Composite* is often used with the patterns called *Chain of Responsibility*, *Iterator*, and *Decorator* (see [18], [19], and [22] for details of these patterns).

Analysis Patterns

These patterns describe the models of business processes that result repeatedly from the analysis phase of software development. Unlike design patterns, analysis patterns don't reflect actual software implementations. They convey conceptual structures of business processes [17]. However, one unusual feature of this class of patterns is that they are hard to classify into traditional vertical domains, such as finance, manufacturing, health care, and so on - they are often useful in more than one vertical domain.

Objectives and Scope

Software analysis is not just listing requirements in use-cases - it involves creating a model of the domain as well. However, after creating a number of models analysts often find that many aspects of a particular project revisit problems they have seen before [17]. Ideas they have used in the context of a previous project happen to be useful in the actual one, so the analysts can improve them and adapt them to new demands. Analysis patterns are groups of concepts and their relationships that represent such ideas, i.e. common constructions in business modeling. An analysis pattern may be relevant only to a single domain, but it may span several domains as well. For example, many models from health-care domain are also applicable to financial analysis. Hence an abstract form of these models actually defines some analysis patterns.

To represent the structure of analysis patterns, people also use graphical notation as in the case of design patterns. There are also catalogues of analysis patterns. However, analysts generally don't use a strict template form for describing this kind of software patterns. They only vaguely follow the format shown in Figure 4, but descriptions of analysis pattern look more like discussions of specific characteristics of business models, one or more pages in length. They often use several examples from specific business models to enhance the pattern presentation.

Examples

Analysis patterns are grouped into a number of groups, but again - each pattern from a certain group can be useful outside its group and outside the domain in which it was originally discovered. Some groups of analysis patterns are as follows:

- Accountability
- Observations and Measurements
- Observations for Corporate Finance
- Referring to Objects
- Inventory and Accounting
- Planning
- Trading
- Application Facades
- …

The two examples of analysis patterns presented here come from [17].

The *Party* pattern is the basic pattern in the Accountability group. It is based on the generalization principle of object-oriented analysis. Party is the first more general concept from which specific concepts like person, company, organization, and the like can be derived directly to form the corresponding hierarchy. Hence this pattern applies to modeling all hierarchical organizations.

For example, consider the problem of modeling the address book. Typical attributes of each person-related entry in any address book are the person's address, phone number, and email address. Likewise, the same attributes feature company-related entries in the address book as well. This could be modeled as in Figure 6, but such a model would contain unnecessary duplications.

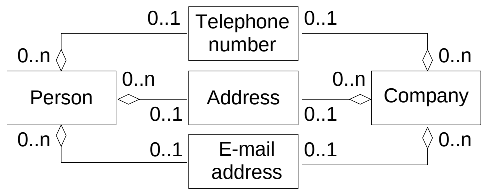

Figure 6- Initial model of an address book

Using generalization, the analyst can introduce the Party class to encompass the Person and Company classes. This leads to the model in Figure 7, which is much simpler and easier to maintain than the one from Figure 6. The duplication is eliminated by ``pulling up" the common attributes from more specific classes to the general one. The general analysis principle underlying the *Party* pattern is that in cases when identical attributes or identical behavior feature several object types (classes), these specific types should be combined into a general one and use inheritance to model the relationship between the general and specific types.

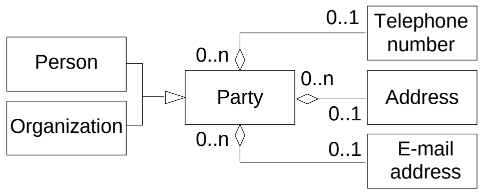

Figure 7- The model of an address book using the *Party* pattern

The *Party* pattern is related to several other patterns from the Accountability group (e.g., patterns like Organization hierarchies, Organization structure, Accountability, Accountability knowledge level, and Party type generalizations). As a matter of fact, *Party* participates as a part of more complex patterns in its group. The same holds for some other analysis patterns as well.

The *Quantity* pattern belongs to the Observations and Measurements group. Many software systems record and update various kinds of measurements. The simplest way to do it is to record the measurements as numbers. It is shown in the example in Figure 8. But there is a problem with this approach: what does it mean, for example, to say that a person's *height* is 5, or 155? It is the measurement unit that is missing here. It is possible to use attributes with more descriptive names, such as *height_in_cm*, but the representation remains awkward.

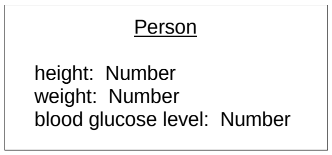

Figure 8- Measurements as Number attributes

The *Quantity* pattern suggests a much better solution: introduce Quantity, a special-purpose type for representing measurements, which combines numbers, units, and operations. The modified Person type might look as in Figure 9. With this modification, it becomes easy to specify that someone's weight is 80kg. Over time, using various forms of the Quantity type becomes as frequent as using built-in types like integer or double.

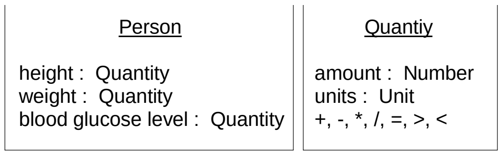

Figure 9- Measurements as Quantity attributes

A typical example of using the *Quantity* pattern is currency conversion in financial applications. Here the monetary values are represented as Quantities and use currencies as their units. Currency conversions can be specified as operations in the Quantity class.

The general analysis principle underlying the *Quantity* pattern is that in cases when identical behavior features different attributes of a data type (a class) and that behavior can be useful in other data types (classes) as well, the attributes should be combined into a unified fundamental data type (class).

Architectural Patterns (Patterns for Software Architectures)

Software systems are composed from identifiable *components* and *connectors* of various distinct types [27]. The components (e.g. compilation units or data files) interact in identifiable, distinct ways, and the connectors (e.g. table entries, dynamic data structures, system calls, and the like) mediate interactions among components. It is possible to define an *architectural style* for a collection of related systems. The style determines a coherent vocabulary of components and connectors (such as pipes, filters, clients, servers, parsers, databases, etc.) and rules for their composition. It structures the design space for a family of related systems and reduces costs of implementation through reusable infrastructure. Styles provide guidance and analysis for building a broad class of architectures in a specific domain.

Objectives and Scope

Patterns for software architectures can be identified by abstracting from the details of architectural styles. The main objective of this abstraction is to make the informal pattern clear, thus laying ground for subsequent formalization. In fact, these patterns impose an overall structure for a software system or subsystem that is appropriate to the problem domain and clarify the designer's intentions about the organization of the system or subsystem. They also provide information about the structure and help establish and maintain internal consistency, as well as perform appropriate style-specific analyses and checks.

Examples of patterns for software architectures include pipeline, layered architecture, data abstraction, repository, and many more.

Some of these patterns govern the overall architectural style that organizes the components [27]. Other architectural patterns identify an abstraction for component interaction, i.e. kinds of interactions among the components. In practice, designers shape up the system's initial overall design using one or more of these patterns - they may decide to elaborate a component of one pattern using some other pattern. For example, in a layered system some layers may be elaborated as pipelines and others as data abstractions. By continuing such an elaboration progressively and repeatedly, architectural issues eventually get resolved.

A typical template for describing architectural patterns is shown in Figure 10. The fields of the template can be filled only with simple statements, or can contain longer descriptions.

- ***problem*** - the problem that the pattern addresses, i.e. the characteristics of the application requirements that lead the designer to select this pattern;
- ***context*** - the context where the pattern is useful, as well as the constraints in using it;
- ***solution*** - the system model captured by the pattern (the intuition about how the pattern's elements are integrated), together with the components, connectors, and control structure that make up the pattern;
- ***diagram*** - a figure showing the pattern's structure, components, and connectors;
- ***significant variants*** - major variants of the basic pattern;
- ***examples*** - references to examples that apply this pattern.

Figure 10 - A template for describing architectural patterns (after [27])

Examples

The examples in this section are presented after [27].

The first example is the *Pipeline* pattern. It is a useful architectural choice for systems that perform a series of computations incrementally on a data stream.

Problem. A series of computations should be performed on ordered data. Sometimes these computations can proceed simultaneously.

Context. It is possible to decompose the problem into a set of transformations, each one processing an input data stream and generating an output data stream. A separate process performs each transformation. Each process takes its input data stream from one or more other processes and delivers its output stream to one or more other processes.

Solution. The system model of this pattern is data flow between the components (processes) called the *filters*. The components incrementally map data streams to data streams, thus making the data streams connectors of the architecture and data flow its control structure.

Diagram. See Figure 11.

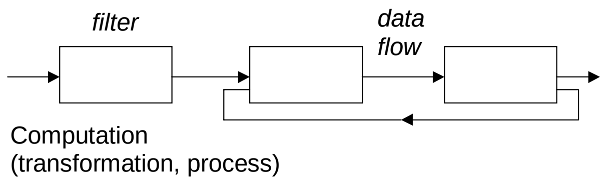

Figure 11- The *Pipeline* architectural pattern (after [27])

Significant variants. ``Pure" filters, with local processing and little state. Alternatively, loops can be present in the topology, such as in Figure 11. If a particular filter requires the entire input data stream before it can start its processing, then loops can significantly slow down the entire system.

Examples. Unix programmers often use this pattern for prototyping, due to Unix' natural pipeline style of processing. Compiler designers also often use *Pipeline* to represent a series of translators.

The second example is the *Implicit Invocation* (*Event-Based*) pattern. It is suitable for systems that include a number of loosely coupled components, each of which carries out some operations that may (or may not) trigger other operations.

Problem. Architectural design of so-called ``reactive systems", containing components that signal significant events without knowing the recipients. Other components that are interested in the events react by invoking some processes automatically upon receiving the signals.

Context. Reactive systems usually require an event handler. It registers components' interest in receiving events and notifies them when the events arise.

Solution. The system model of this pattern is a collection of independent reactive processes. The components are processes that signal significant events. The connectors are automatic invocations of other processes upon the appropriate events. The control structure is event-based and decentralized, since individual processes are not aware of the recipients of the signals they rise.

Diagram. See Figure 12.

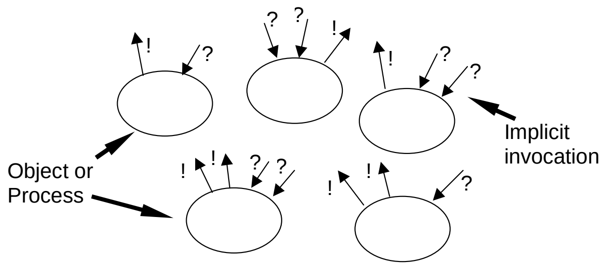

Figure 12- The *Implicit Invocation* (*Event-Based*) architectural pattern (after [27])

Significant variants. There are no significant variants of the pattern itself, but the level of sophistication of the event handler can vary a lot. Its reasoning about the correctness of the system depends on the components, the kinds of events, and the collective effect of processing the events.

Examples. Event processing in user interfaces, in operating systems, and in telecommunications.

Organizational and Process Patterns

Organizational and process patterns capture successful management practices of software development [3], [11]. They are much different from other kinds of patterns. They cover general development problems that all software organizations must face, hence they actually *support* software design, not *address* it directly [11].

Objectives and Scope

Organizational patterns are about software-development organizations and about people who work in such organizations. Some organizational patterns can be used to shape a new organization and its development processes [12]. Others can be applied to evolution of existing organizations. Organizational patterns capture suitable management techniques and organizational structures in software development, thus affecting the ability of people to do their work. It should be noted that organizational patterns are usually drawn from case studies of organizations with high productivity and from common sense. It is hard to prescribe a suitable and simultaneously simple metrics for evaluating such patterns. For that reason, the value of organizational patterns is always judged empirically by managers of successful development organizations.

Templates for organizational patterns vary from quite informal descriptions to strictly structured ones, depending on the pattern's nature [9], [11]. An example of structured template for organizational patterns is shown in Figure 13.

- ***problem*** - the problem that the pattern addresses in building, managing, or evolving a software-development organization;
- ***context*** - the context where the pattern is useful (e.g., the goals of building a new organization, the product's characteristics, typical roles in the development team, and the like);
- ***forces*** - the motivation and driving forces for using the pattern;
- ***solution*** - the organizational/managerial technique or practice captured by the pattern;
- ***resulting context*** - discussion of how applying the pattern changes or improves the organizational pragmatics;
- ***rationale*** - examples of organizations and situations where the pattern has been successfully used.

Figure 13 - A template for describing organizational patterns (after [12])

Process patterns relate to the strategies that software professionals employ to solve problems that recur across organizations [3]. Each process pattern describes a collection of general techniques, actions, and/or tasks for developing object-oriented software. Figure 14 shows a typical template. An important feature of process patterns is that they suggest *what* to do, but not details of *how* to do something. Hence process patterns are reusable building blocks from which an organization can develop a tailored software process that meets its exact needs. There are three types of process patterns:

- *task process patterns* **-** they are process patterns of the narrowest scope; these process patterns specify detailed steps to perform a specific task, such as technical review, component reuse, or software configuration management;
- *stage process patterns* **-** these patterns prescribe the steps of a single project stage, such as requirements analysis, modeling, programming, and the like; not surprisingly, such patterns are often composed of several task process patterns;
- *phase process patterns* **-** patterns of this type depict interactions among several stage process patterns and a single project phase, such as the inception, elaboration and construction phases.

- ***forces*** - the problem that the pattern addresses in the software development process and the driving forces for resolving the problem using that pattern;
- ***context*** - the initial context and entry conditions;
- ***solution*** - the techniques, actions, and/or tasks (activities) to be used to solve the problem, usually supported by a suitable diagram;
- ***resulting context*** - discussion of how applying the pattern changes or improves the software development process and possibly of what exit conditions must be met for the solution to be complete.

Figure 14 - A template for describing process patterns (after [3])

Most organizational and process patterns support iterative and incremental software design. They have started to appear in mid-1990s as a reflection of the fact that software development processes used in software organizations have gradually moved away from the traditional, linear ``waterfall" model towards iterative and incremental development practices. Researchers have noticed many recurring practices in highly productive and successful organizations, and it is exactly such practices that form the skeleton of these kinds of patterns.

Also, most organizational and process patterns never come as individual patterns. They usually go together as groups of related patterns, i.e. as pattern languages (see the next section).

Examples

The first example is the organizational pattern called *Domain Expertise in Roles*, and it is defined within James Coplien's generative development-process pattern language [12].

Problem. How to match staff (people in the organization) to roles?

Context. Key atomic process roles and a characterization of the Developer role are known.

Forces. All the roles must be staffed with qualified individuals, but the more expertise is spread across roles the more complicated the communication between the people in the organization.

Solution. Hire experienced domain experts. Any given actor may fill several roles. Domain training is more important than process training.

Resulting context. The roles can be carried out more successfully and are more autonomous.

Rationale. In practice, many successful organizations hire deeply specialized experts. Experience shows that the roles of System Engineer and System Tester are often poorly staffed and should be paid special attention.

As the second example, consider the *Technical Review* task process pattern [3]. It addresses the problem of reviewing deliverables created during software development in order to meet the user's needs and ensure quality standards.

Forces. It is necessary to organize, conduct and follow through the review of one or more deliverables, such as models, prototypes, documents, or source code. Each deliverable should be validated for quality before building further on it. Defects in deliverables should be detected early, in order to decrease the costs of fixing them. The work of an individual developer must be communicated to the other team members, and one way to do it is to have the other team members review the work of that developer.

Context. The deliverables and the team are ready for the review process.

Solution. Figure 15 prescribes how to do the review process successfully. The team first prepares the items to be presented to the reviewers and then inform the review manager (usually the software quality assurance manager) that the items are ready for review. Then the review manager does a short, cursory overview of the work that has been done in order to ensure that the work to be reviewed is good enough to deserve gathering the review team. The next step is to plan, schedule, and organize the whole process, meaning to invite the proper people and hand out the necessary materials ahead of the review sessions. During the review process, it is necessary for the developers that have actually done the work to attend and explain/clarify details of their work. It is important to review everything thoroughly; hence the review process can take from several hours to several days depending on the number and complexity of deliverables to be reviewed. In the end, the review team produces a document that describes strengths and weaknesses of the deliverables. The development team should analyze the document and act properly in order to eliminate the weaknesses detected and explained in the document.

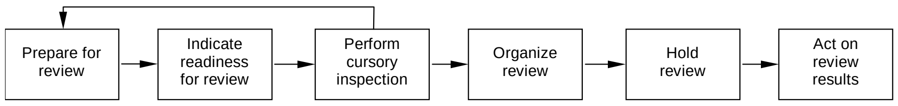

Figure 15 - The *Technical Review* process pattern (after [3])

Resulting context. Managers are ensured that the development team has produced deliverables that satisfy the organization's quality standards. The developers have a better understanding of what is missing, what is unclear, and what must be improved in their work. Both team members and the reviewers usually learn or at least get initiated to new techniques during the review process.

Pattern Languages

Patterns have a context in which they apply. When several related patterns are woven together, they form a *pattern language* [26]. A pattern language is not a formal language. Rather it is a structured collection of interrelated patterns that provides vocabulary for talking about a particular problem.

Objectives and Scope

By providing specific design and development vocabularies pattern languages help software developers communicate better. They cover particular domains and disciplines, such as concurrency, distribution, organizational design, business and electronic commerce, human interface design and many more [11]. Pattern languages help developers communicate architectural knowledge, help analysts avoid pitfalls and traps that other people have learned painfully by their own experience, and help designers learn a new design paradigm or architectural style [26].

It is important to note the difference between pattern catalogues and pattern languages. In pattern catalogues, individual patterns represent relatively independent solutions to common software design and development problems. In pattern languages, groups of related patterns are *integrated* on the grounds of experience in using individual patterns from catalogues - patterns in the same group often go together in different practical problems and are all suitable for particular areas of software design.

The number of patterns in the existing pattern languages varies from about half a dozen to several dozens, depending on the problem the language addresses.

In some pattern languages patterns are structured. Structures of pattern languages can take different forms, such as networks or trees. For example, in the G++ pattern language for design of large software control system [1], all the patterns are structured in a tree. A part of that tree is shown in Figure 16. The nodes of the tree represent individual patterns, i.e. design decision points, while the arcs represent the temporal sequence of decisions.

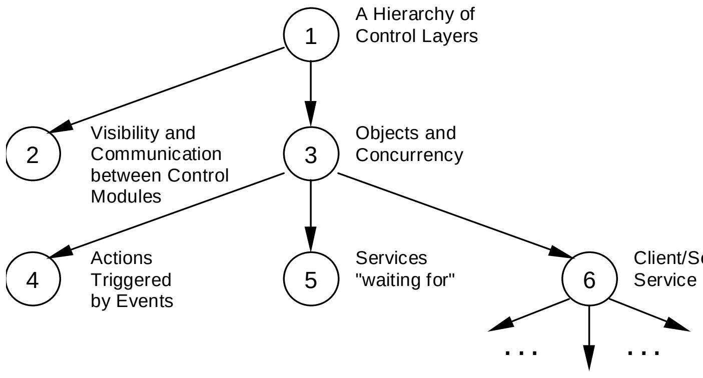
1
A Hierarchy of Control Layers
2
Visibility and Communication between Control Modules
3
Objects and Concurrency
4
Actions Triggered by Events
5
Services "waiting for"
6
Client/Server/ Service
...
...

Figure 16 - A part of the G++ pattern language (after [1])

Examples

The first example is the G++ pattern language, Figure 16. Since it addresses design of large software control system, its 11 patterns cover issues such as the organization of control modules in a hierarchy, the visibility requirements between the control modules, concurrency and distribution. For the sake of brevity, only the names and related problems of three selected patterns from the language are listed here. Interested readers should refer to [1] for a complete coverage of G++.

Pattern 1: A Hierarchy of Control Layers - it is necessary to arrange individual control modules in multiple layers and create the communication hierarchy between them.

Pattern 2: Visibility and Communication between Control Modules - these mechanisms must be established for both imperative and reactive communication.

Pattern 3: Objects and Concurrency - it is desirable to adopt the correct model for representing concurrency at different granularities.

The second example is the RAPPeL - a Requirements-Analysis-Process Pattern Language for object-oriented development [32]. It provides a direction and rationale for analysts, developers, and managers in requirements engineering for business applications. More than 100 patterns have been discovered in that area and integrated in RAPPeL, but the language is still under development. Some of the patterns are as follows.

Pattern 1: Building the Right Things - it is necessary to properly and thoroughly capture, communicate, understand, and validate the customer's real needs in order to create the requirements specification for the system that will "do the right things".

Pattern 9: Customer Rapport - how to build and establish a good relationship with a customer?

Pattern 20: Problem Domain Analysis - it is important to determine and define the essential nature of the problem domain in which the system is to be built?

Pattern 24: Finding and Defining the Domain Objects - how to best determine the objects in the problem domain and define their roles and responsibilities?

Pattern 50: User Interface Requirements - what is the best way and time to determine the requirements for the user interface?

Research Issues

Although software patterns have first and foremost attracted people from the world of industrial software development, many researchers from the broad field of software engineering have been involved in patterns from the very beginning. Research on software patterns is nowadays best presented on the dedicated pages on the Web, as well as on annual PLoP conferences (Pattern Languages of Programs) held from 1994.

From the PLoP conferences and the relevant Web pages it is easy to see that recent research efforts and trends in software patterns span across many areas. The following avenues are just a brief selection from such areas.

- *Risk Management Patterns*. These patterns capture strategies of project management by risk reduction [10]. The idea is to transfer the knowledge of successful project leaders to new project leaders by describing the successful risk management practices in a consistent, common format.
- *Patterns for component-based software*. Many projects today use a component-based approach to software development [31]. The main issues in component-based development are language and platform interoperability and separation of interface from implementation. Existing and newly constructed components are used on clients and servers to build flexible, reusable solutions on different platforms with different frameworks. Interested researchers from the software patterns community are trying to capture some of the recurring themes and best practices in component-based development that lead to winning software projects.
- *Patterns of social forces affecting software development*. Social forces of software development, such as skill mix, movement of people, and team structuring, do introduce some amount of bias in design decisions [9]. For example, a designer may make some decisions based on some privately evolved set of principles, and other designers have no way to judge what depends on such principles and what doesn't. The usual form of presenting software patterns hides that bias. The idea behind the patterns of this kind is to consistently document the interactions of social issues and software architecture. For example, these patterns stress such principles as creating interfaces around predicted points of variation, separating subsystems by staff skill requirements, ensuring there is an owner for each deliverable, and so on.
- *Domain-specific patterns.* Such patterns resemble analysis patterns to an extent, because they describe common solutions to typical problems of domain analysis. However, unlike analysis patterns they are strictly specific to a vertical domain. Each vertical domain has its own concepts, relations, structure, and organization, in addition to more general principles common to some other domains and that are captured by analysis patterns. In some cases, domain-specific patterns are possible to express (at least partially) using the knowledge of other classes of patterns. An example of such domain-specific patterns in the domain of education is presented in [14] and [15].
- *Software patterns and knowledge engineering.* In knowledge engineering, an *ontology* represents the basic structure or armature around which a knowledge base can be built [30]. It is well known that ontologies are "specifications of conceptualizations", they provide

vocabulary of a problem domain, they also clarify the structure of knowledge, and they enable knowledge sharing. In other words, ontologies generally provide upper-level guidance and analysis for building sharable knowledge bases. Software patterns and ontologies are *not* the same concepts, but they have much in common. For example, design patterns also represent a kind of specifications, since they represent design solutions. They also provide some abstract, common, and general, though small vocabulary of specific design stereotypes. By clearly showing the intent, motivation, applicability, structure, participants and their collaborations, as well as consequences of applying them, design patterns also clarify parts of design knowledge and experience. Finally, design patterns are one way of sharing design knowledge and experience. For more comprehensive recent research on the relation between software patterns and ontologies, see [14].

- *Anti Patterns*. Anti patterns can be considered as an extension of design patterns. They provide the patterns of bad solutions and suggest refactoring techniques to improve the situations [7]. An anti-pattern is either a pattern that tells how to go from a problem to a bad solution, or a pattern that tells how to go from a bad solution to a good solution. A good anti pattern also shows why the bad solution looks attractive (e.g., it actually works in some narrow context), why it turns out to be bad, and what positive patterns are applicable in its stead. As an example, consider the *Design By Committee* anti pattern. It says that if no one single person in the team can be influential enough to present a design for a system and get it approved, to get the design done a big committee of designers is assembled to solve the problem. They battle it out amongst themselves and finally take whatever comes out in the end. The result is a mish-mash of features in which everybody gets their share put in, since there was no unifying vision. Another example is the *Analysis Paralysis* pattern. It refers to the situations where a team of analysts gets stuck in building the application model forever. Other team members, such as designers and developers, have no work to do before the analysts deliver and are given busy work or some training. It happens almost always when the management insists on completing all analysis before beginning design, or when the project lacks clear goals. The common cures for this paralysis are keeping the models small, developing prototypes and making testing drive analysis, and minimizing the number of "analysts" to a single architect, turning everybody else into a developer.

Other notably attractive research topics include patterns for configuration and change management, business process reengineering, software testing, refactoring, and organization diagnostics.

Important Practical Issues

There are reports from actual development projects where software patterns have been applied. Such reports are very useful for researchers as well as practitioners, because they include advantages and disadvantages of the patterns in addition to experiential findings. They also present ways to avoid common traps and pitfalls of applying design patterns in practical software development processes. Many such reports can be found on the Web, starting from sites like Cetus Links (see the next section). Some others are available in published literature. For example, Kent Beck and his colleagues have described their experiences and lessons learned from applying patterns in a number of different industrial settings, including Hewitt Associates, Orient Overseas Container Limited, AT&T, Motorola, BNR, Siemens, and IBM [5]. Enumerating their lessons learned here should give novice practitioners an idea of what does it really mean to work with patterns in real-world projects:

- patterns serve as a good team communication medium;
- patterns are extracted from working design;
- patterns capture the essential parts of a design in a compact form;
- patterns can be used to record and encourage the reuse of "best practices";
- patterns are not necessarily object-oriented;
- the use of pattern mentors in an organization can speed the acceptance of patterns;
- good patterns are difficult and time-consuming to write.

Another example is shown in [25]. The author describes how design patterns have been used in several projects in the domain of communication software, in order to enable widespread reuse of communication software architectures, developer expertise, and object-oriented framework components. The projects included the Motorola Iridium global personal communications system, a family of network monitoring applications for Ericsson telecommunication switches, and a system for transporting multimegabyte medical images over high-speed ATM networks.

It is also important for practitioners to know that there are efforts related to formally modeling software patterns. For example, the meta-pattern of W. Pree is considered as formally modeling design patterns to categorize a family of the patterns [24]. In a similar endeavor, J. Soukup has developed a special class, called the Pattern class, that encapsulates all the behavior and logic of individual patterns [28]. The Pattern class makes possible to clearly decouple application classes from patterns. Laudar et al. have discussed modeling design patterns from the viewpoints of role, type and class, and representing these models visually [23]. There is also some work related to the formalization of filling hot spots of design patterns by using UML parametrization facility (e.g., see [19] and [29]).

Recent popularity of autonomous agents and agent-oriented software engineering has led to the discovery of specific *agent design patterns*, that capture good solutions to common problems in agent design. For example, Aridor and Lange report on several design patterns they have found in mobile agent applications [4].

There are some CASE tools where design patterns are embedded and/or which have the functions to manage design patterns. Apart from some earlier examples (e.g., [16]), a popular modern product suite of such tools includes Together Solo, Together Enterprise, Together Control Center from TogetherSoft (www.togethersoft.com).

Summary

The software patterns movement has emerged in the object-oriented community, from the studies of software reuse and from observations that ordinary, recurring solutions to typical software design and development projects should be given priority over using the latest technology. However, the movement has soon transcended the object-oriented community and spread all over the software development world, leading to the rapid development of patterns as an important software engineering problem-solving discipline per se and to the specific patterns community and culture. The goal of the pattern community is to build a body of literature and other resources to support software design and development in general. There is less focus on technology than on a culture to document and support sound design.

The roots of the discipline of software patterns stem from many disciplines, including literate programming, but most notably from Alexander's work on urban planning and building architecture [2]. The landmark work that has determined the real beginning of the patterns movement in software engineering has been the famous *Gang of Four* book on object-oriented design patterns [18]. Although the majority of the software community is still using patterns largely for software design, from the times of the *Gang of Four* book patterns have been used for domains as diverse as software analysis, development organization and process, exposition and teaching, software architecture, component-based software development, and many more. The discipline continues to grow and to attract more and more people to the community.

Software Patterns on the Web

There are many software patterns resources on the Web. The short list of URLs shown below has been composed according to the following criteria:

- the number of useful links following from that URL
- how comprehensive the site is
- how interesting the URL is for researchers

As for the first two criteria, the first four URLs below provide the best starting points. The other sites listed satisfy all three criteria but are focused on specific classes of software patterns and pattern languages.

- Cetus Links:
  http://www.objenv.com/cetus/oo_patterns.html
- Patterns Home Page 1:
  http://hillside.net/patterns/patterns.html
- Patterns Home Page 2:
  http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- Patterns Archive:
  http://www.DistributedObjects.com/portfolio/archives/patterns/
  index.html
- Design Patterns Home Page:
  http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- Analysis Patterns:
  http://www.aw.com/cseng/titles/0-201-89542-0/apsupp/index.htm
  http://www.martinfowler.com/
  http://www.martinfowler.com/ap2/index.html
- Organizational Patterns:
  http://www.bell-labs.com/cgi-user/OrgPatterns/
  OrgPatterns?ProjectIndex
- The Process Patterns Resource Page:
  http://www.ambysoft.com/processPatternsPage.html
- Design Patterns and Pattern Languages:
  http://siesta.cs.wustl.edu/~schmidt/patterns.html
- Jim Coplien's Process Pattern Language:
  http://www.bell-labs.com/people/cope/Patterns/Process/index.html

References

1. Aarsten, A., Brugali, D., Menga, G., Designing Concurrent and Distributed Control Systems, *Communications of The ACM* 39 October 1996., pp. 50-58.
2. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., *A Pattern Language*. New York: Oxford University Press, 1977.
3. Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge, MA: Cambridge University Press, July 1998.
4. Aridor, Y., Lange, D.B., Agent Design Patterns: Elements of Agent Application Design. Proceedings of Autonomous Agents '98, Minneapolis, MN, 1998, pp. 108-115.
5. Beck, K., Coplien, J.O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., Vlissides, J., Industrial Experience with Design Patterns. Proceedings of 18th International Conference on Software Engineering. Berlin, Germany, February 1996, pp. 103-114.

6. Booch, G., Rumbaugh, J., Jacobson, I., *Unified Modelling Language User's Guide*. Reading: Addison-Wesley, 1999.
7. Brown, W.J., Malveau, R.C., Brown, McCormick, H.W., III, Mowbray, T.J., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. NY: John Wiley & Sons, 1998.
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., *Pattern-Oriented Software Architecture - A System of Patterns*. NY: John Wiley & Sons, 1996.
9. Cockburn, A., The Interaction of Social Issues and Software Architecture. *Communications of The ACM* 39 October 1996., pp. 40-46.
10. Cockburn, A. (ed.), *Surviving Object-Oriented Projects*. Reading: Addison-Wesley, 1998.
11. Coplien, J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Reading: Addison-Wesley, 1995.
12. Coplien, J., A Generative Development-Process Pattern Language. In: Coplien, J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Reading: Addison-Wesley, 1995.
13. Coplien, J., Vlissides, J., Kerth N. (eds.), *Pattern Languages of Program Design - Vol.2*. Reading: Addison-Wesley, 1996.
14. Devedzic, V., Ontologies: Borrowing from Software Patterns. *ACM intelligence Magazine* 10, Fall 1999., pp. 14-24.
15. Devedzic, V., Intelligent Tutoring Systems - Using Design Patterns. *International Journal of Knowledge-Based Intelligent Engineering Systems* 4, January 2000., pp. 25-32.
16. Florijn, G., Meijers, M., van Winsen, P., Tool Support for Object-Oriented Patterns. In: Aksit, M., Matsuoka S. (eds), *Proceedings of ECOOP'97*, Jyväskylä, Finland, 1997. *Lecture Notes in Computer Science 1241*. Berlin: Springer Verlag, 1997, pp. 134-143.
17. Fowler, M., *Analysis Patterns: Reusable Object Models*. Reading: Addison-Wesley, 1997.
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.
19. Grand, M., *Patterns in Java - A Catalog of Reusable Design Patterns Illustrated with UML*. New York: John Wiley & Sons, 1998.
20. Keepence, B., Mannion, M., Using Patterns to Model Variability in Product Families. *IEEE Software* 16, July/August 1999., pp. 102-108.
21. Kotula, J., Using Patterns To Create Component Documentation. *IEEE Software* 15, March/April 1998., pp. 84-92.
22. Larman, C., *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. Upper Saddle River: Prentice-Hall, 1998.
23. Laudar, A., et al., Precise Visual Specification of Design Patterns. *Lecture Notes in Computer Science* 1445, Berlin: Springer Verlag, 1998, pp. 114-134.
24. Pree, W., *Design Patterns for Object-Oriented Software Development*. Reading: Addison-Wesley, 1994.
25. Schmidt, D., Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of The ACM* 38, October 1995., pp. 65-74.
26. Schmidt, D., Fayad, M., Johnson, R.E., Software Patterns. *Communications of The ACM* 39, October 1996., pp. 37-39.

27. Shaw, M., Patterns for Software Architectures. In: Coplien, J., Schmidt, D. (eds), *Pattern Languages of Program Design*. Reading: Addison-Wesley 1995., pp. 453-462.
28. Soukup, J., Implementing Patterns. In: Coplien, J., Schmidt, D. (eds), *Pattern Languages of Program Design*. Reading: Addison-Wesley 1995., pp. 395-412.
29. Sunye, G., et. al., Design Patterns Application in UML. *Lecture Notes in Computer Science* 1850, Berlin: Springer Verlag, 2000, pp. 44-62.
30. Swartout, W., Tate, A., Ontologies. Guest Editors' Introduction. *IEEE Intelligent Systems* 14, Special Issue on Ontologies, January/February 1999., pp. 18-19.
31. Szyperski, C., *Component Software: Beyond Object-Oriented Programming*. Reading: Addison-Wesley, 1998.
32. Whitenack, B., RAPPeL: A Requirements-Analysis-Process Pattern Language for Object-Oriented Development. In: Coplien, J., Schmidt, D. (eds), *Pattern Languages of Program Design*. Reading: Addison-Wesley 1995., pp. 259-292.