Ontologies: Borrowing from Software Patterns

Vladan Devedžić

University of Belgrade, FON - School of Business Administration, Department of Information Systems
POB 52, Jove Ilića 154, 11000 Belgrade, Yugoslavia
Phone: +381-11-3971440, +381-11-493492 Fax: +381-11-461221
Email: devedzic@galeb.etf.bg.ac.yu, devedzic@fon.fon.bg.ac.yu
URL: http://galeb.etf.bg.ac.yu/~devedzic/

Note to the reviewers and editors

Blue-colored phrases are candidates for callouts. Also, a couple of phrases in the beginning of the article are shown in larger font on purpose (as lead-in).

When developing ontologies, it helps to pay more attention to some results and achievements of people from other disciplines. Some people may use different terminology, but they may actually mean much the same as we do. To an extent, people from the *software patterns* community are like that. So, the following question naturally arises: can we borrow from them?

There is at least a dozen definitions of ontologies in the literature. One of the most recent ones says that an ontology provides the basic structure or armature around which a knowledge base can be built [Swartout and Tate, 1999]. Although completely informal, this definition captures the central idea of ontologies. Likewise, one definition of software patterns says that they are ideas that have been useful in one practical context and will probably be useful in others [Fowler, 1997] (see the sidebar for more details and a background on software patterns). While it also may look quite loose, this definition closely reveals the underlying motivation of software patterns. We will adopt these definitions in this article.

In spite of the fact that the above definitions may seem not to have much in common, the concepts of ontologies and software patterns partially overlap. Putting them side by side reveals what they have in common, as well as how do they complement each other. Using software patterns along with other tools of ontological engineering can open new perspectives to this growing field of practical AI developments.

Ontologies are not just for knowledge-based systems but for all software systems, since all software systems need some model of the relevant world [Chandrasekaran et al., 1999]. Hence all software-engineering technologies can benefit from developing and using ontologies, and the technology of software patterns is not an exception. Simultaneously, a useful feedback to the field of ontologies can come from software patterns - they are always about some knowledge, just like ontologies are. Differences do exist, but conceptually, practically, and even methodologically software patterns resemble ontologies well enough to generate a handful of useful ideas for building ontologies in practice.

The knowledge level

Ontologies provide the skeletal knowledge and an infrastructure for integrating knowledge bases at the *knowledge level*, independent of particular implementations. The skeletal knowledge typically enables communication between intelligent agents that generally have different internal representations of their knowledge, but need to share that knowledge in order to solve problems effectively.

Software patterns also describe things at the knowledge level. They provide means for effectively describing knowledge of successful solutions to known and recurring problems in software development. Patterns make possible for developers to easily share common ways and techniques of doing things, regardless of any particular project, features of problem domains, or implementation tools and languages.

The heart of ontologies and the quintessence of software patterns

Ontologies allow us to relate things by virtue of the relations between their constituents [Lenat, 1995]. Things, objects and concepts have their meanings, goals, functions, functionalities, components, structure, and other characteristics. In most of cases ontologies have the form of networks of categories/classes of concepts, with an explicit representation of hierarchy/inheritance among them. Along with these abstraction hierarchies, an ontology represents the vocabulary of the problem domain, different kinds of relationships between its

concepts (e.g., *part-of* and *instance-of* relationships), concept properties, constraints (axioms), and rules for extending the vocabulary. All this information is necessary to properly model the domain. For example, the ontology of computer program can specify its concepts using the terms *program*, *identifier*, *declaration*, *expression*, *control structure*, *operator*, and so on. The *greater-than* operator is related by *a-kind-of* relation to the class of *relational-operators*, which are a kind of operators. A property of identifiers is that they are strings of characters, and a constraint is that each identifier must begin with a letter or a special character (such as "_"). Ontologies define generic structure that can be used when building specialized knowledge bases.

Software patterns describe the cores of solutions to analysis, design, architectural and other problems in software engineering that have been used more than once in different systems. Software engineers usually represent these problem-solution pairs graphically, giving appropriate generic names to all participants in the solution, as well as to the patterns themselves. The graphical representation clearly shows hierarchies and relationships among the participants. For example, the pattern in Figure 1 is a design pattern (see the sidebar) called Builder [Gamma et al., 1995]. It abstracts the core of the solution to the frequent problem of separating the construction of a complex object from its representation. Such a separation makes it possible to create different representations by the same construction process. Consider, for instance, generating explanations to different kinds of users of an explanation generator. The client will always ask the *Director* to generate an explanation. However, concrete explanations (*Products*) to novice users and advanced ones may differ considerably, because the *Director* always passes the request to a *ConreteBuilder* specific to the type of the user through the common *Builder* interface.

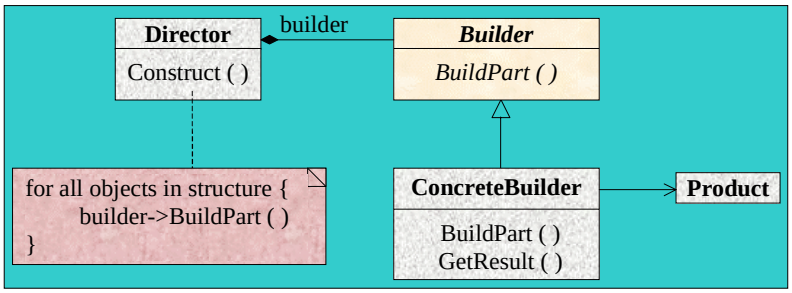

Figure 1. Structure of the Builder pattern

An essential feature of a software pattern is that it is solution to a problem in a *context*. The context specifies the constraints like when the pattern is applicable, what pitfalls, hints and techniques should one be aware when using the pattern, and what positive consequences and trade-offs should be expected when applying the pattern in practice. Using patterns early in the lifecycle, one can avoid many painful modifications at later stages of design. Software patterns are specific to generic problems in software development, but can span a number of application domains.

How do software patterns resemble ontologies?

Both ontologies and software patterns are intended for knowledge sharing and knowledge reuse. Although ontologies are by far more general and software patterns concentrate only on software engineering problems, both ontologies and software patterns have similar approaches to knowledge sharing and reuse, as well similar design methodologies. Hierarchies and relationships among concepts, common vocabularies of the problem domain, representation at the knowledge level, describing core pieces of domain models, and generic knowledge and constraints for fitting that knowledge to a particular application also feature both fields. Moreover, both of them let us identify and capture less obvious abstractions and concepts, such as processes or algorithms. Ontology captures the intrinsic conceptual structure of the domain [Chandrasekaran et al., 1999]. Software patterns capture the intrinsic conceptual structure of their specific domain - that of solutions to common problems in software engineering. Finally, both fields share strong support by object-oriented technology at the design level. Higher up, at the knowledge level, ontologies and software patterns may emphasize different aspects, but both essentially focus on "armatures and skeletons".

On the other hand, neither an ontology nor a software pattern can cover all possible potential uses. Both of them are more appropriate for certain uses than others and are hard to share across widely divergent tasks. Also both of them often require adaptation to fit a particular context.

What is the relation between ontologies and software patterns?

In order to answer this question, it is worth to consider a more general one: What is the relation between ontologies and *any* patterns? A good visual illustration of that relation is the image in Figure 2. In spite of the obvious minor differences in geometry of the repeating curves and shapes at several regions of the image,

most people can easily recognize the patterns that the curves and shapes create. If there were no geometrical differences, i.e. if the patterns were completely regular, we would have a notably different image. However, hypothetical ontologies of both kinds of images (with regular and with irregular patterns) would have much in common. Moreover, important parts of such ontologies would describe the structure of the patterns. Even these parts would overlap to a large extent.

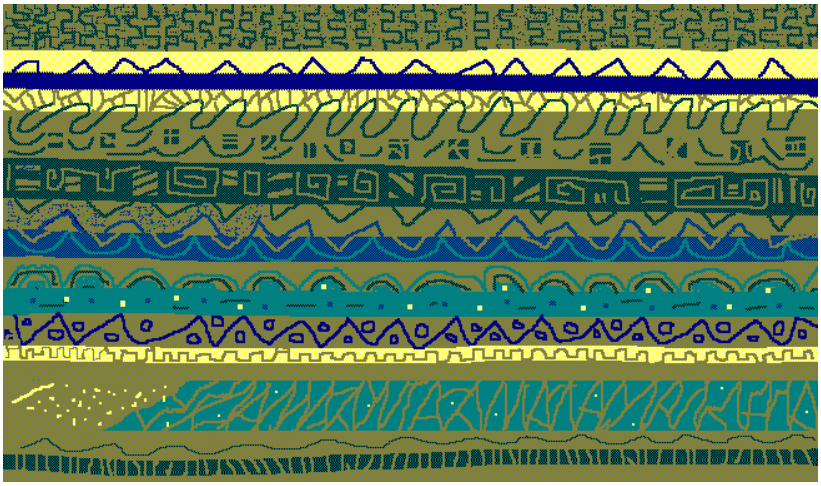

Figure 2. Patterns are important parts of ontologies

As another example, consider the structure of email addresses and URLs. The ontology of such abstract objects would define concepts, vocabulary, and composition constraints. A large part of the ontology would actually describe patterns for generating the addresses. Again, some patterns are obvious in spite of minor differences (e.g., *user1@sigart.acm.org* and *user2@.acm.org*), while others can be expressed only heuristically and with some degree of certainty (e.g., home pages of many companies begin with "www", followed by the company's name in some form, followed by "com", and ending by the country code). The point is, however, that patterns contain significant portion of knowledge to be encoded in the ontology. Coming back to the terrain of *software* patterns, we can treat them as incomplete specifications of software engineering ontologies. They can also be parts of other ontologies, and they can be sources of knowledge for developing other ontologies.

An important step in representing knowledge of software patterns in computers should be development of their ontology. Good starting points in this direction are templates that people use for describing software patterns (see the sidebar). Templates provide means for describing patterns in a consistent format. In spite of the fact that the pattern community uses different templates for describing different kinds of software patterns, most templates one way or another contain four essential parts: the problem the pattern addresses, the forces that play in forming a solution, the solution that resolves those forces, and a statement of the context where the pattern is useful. The ontology of software patterns must also include these four parts.

Vocabularies and meanings

Each ontology specifies the vocabulary of representational terms in the corresponding domain, with agreed-upon definitions of the terms in a declarative form. Definitions may include different axioms, constraints, relations among concepts, and hierarchies of the problem domain. Once the vocabulary and definitions are specified, each knowledge base and agent using the ontology must commit to the semantics of the terms and definitions. Agents exchange queries and assertions using the vocabulary from the ontology. One can build semantically consistent knowledge bases by specializing and instantiating the ontology, using application-specific information. The meaning of the ontology is contained in the conceptualization that the terms in the vocabulary are intended to capture [Chandrasekaran et al., 1999].

It is much the same with software patterns. They provide a common vocabulary for developers to communicate, document, and explore software development alternatives. Although pattern descriptions and their underlying vocabularies are often highly informal, they nevertheless appear to convey high-level software engineering knowledge effectively. They reduce system complexity by naming and defining abstractions, i.e. knowledge that is above ordinary software analysis and design artifacts. A good set of software patterns can effectively describe the engineering semantics of the entire development of a software product. As with ontologies, one of the hardest parts of developing software patterns is that of finding good names for the patterns and their participants, and agreeing upon the names. Hence a software pattern may have more that one well-known name in practice. In ontologies, this problem is sometimes bypassed by allowing customization of the vocabulary.

Reusability and building blocks

Reusability and knowledge sharing are common issues both in ontologies and in software patterns. Software patterns constitute a reusable base of experience for building reusable software [Gamma et al., 1995]. They distill the software development knowledge gained by skillful and experienced practitioners. In software design, patterns act as building blocks for constructing more complex designs. In software analysis, they can be considered micro-architectures that contribute to overall system architecture. Software patterns determine how separate parts of the system are combined, or "woven together". The good thing is that a number of software patterns recur regularly in software systems, thus offering a good armature for practical developments. Moreover, the knowledge (the ontology) of software architectures extends to frequent combinations of two or more software patterns that go together well. For example, Figure 3 shows how designers use four design patterns together in complex systems. In fact, that's how those four patterns are often integrated at the knowledge level.

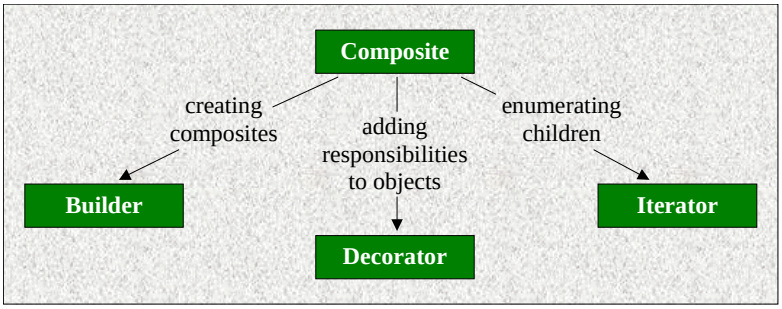

Figure 3. Some design patterns often go together

The design level

After all, somebody always has to build it, be it an ontology, a knowledge base, or an arbitrary application based on software patterns. One of the main points is to decompose the knowledge-level specifications of the domain effectively to produce the design model. Knowledge-level specifications include specifications of the domain contents and specifications of problem-solving tasks and methods. At the design level knowledge is encoded into sentences, such as first-order logic sentences.

Taxonomies and structures

When designing ontologies, developers often translate concept hierarchies into *is-a* taxonomies, having a set of properties and components that are meaningful for each category [Fridman-Noy and Hafner, 1997]. Relations between categories, such as *part-whole* relation, define the internal concept structure. It simplifies the ontology design significantly if the taxonomies of concepts can be represented as a tree of ontologies [Van der Vet, 1998], in which every subtree is itself an ontology. Knowledge bases built around such trees of ontologies are particularly well suited for generating meaningful high-level explanations of the system behavior.

With software patterns, the design level often means specification of classes and objects and their relations to encode the meaning of the patterns. The world of software patterns is full of expressive graphical languages, such as UML [Larman, 1998], that allow designers to precisely express the rich semantics of object and class relationships. Ontological engineers may often benefit from borrowing the notation of such a language to represent an ontology. Such languages offer a variety of options for highly consistent representation of cardinalities, collections of concepts, details of *is-a* and *part-of* relationships, and other important issues and details of ontology design that otherwise might be overlooked.

Design processes

Figure 3 is an illustration of how a more complex design "ontology" based on software patterns derives its value from smaller granularity ones. In fact, this process is always present when designing ontologies using the bottom-up approach [Fridman-Noy and Hafner, 1997], [Van der Vet, 1998].

Other than that, it is often possible to use software patterns as initial, rough ideas for designing an ontology by extending their structure in a kind of the middle-out design. For example, in intelligent human-computer interaction there is often a need to generate different kinds of intelligent system's output to the user, such as explanations, hints, or examples. All the corresponding generators have much in common, hence should be based on a common ontology. They all generate some objects that are generally composite (e.g., an explanation may contain text, graphics, or even links to more detailed explanations). They do it upon request

from an external client. They all use some built-in knowledge in order to generate their objects, and they must do it in different contexts, depending on specific interactions with the user. Also, it is desirable for each such a generator to use the same process of generating objects in all possible contexts. An analysis of these facts can show that it is easy to develop the common *Generator* ontology starting from the Builder pattern shown in Figure 1. In fact, a similar ontology has been already built in the domain of intelligent manufacturing [Devedžić and Radović, 1999].

Relations among the top-level concepts of the Generator ontology at the design level are represented in Figure 4. Comparing Figure 4 to Figure 1 reveals design origins of the Generator ontology. Generating the desired object (the *Result*) starts by creating an instance of a *Concrete generator* that will do the real job, and the *Generator* that "wraps-up" the *Concrete generator*. In fact, by making the *Concrete generator* a part of the *Generator* the *Client* configures the *Generator* with a desired *Concrete generator*. The *Concrete generator* then builds one part of the *Result* after another, getting the requests from the *Generator* through the *Builder* interface. Each time it builds a part, it consults the relevant *Knowledge* and evaluates the relevant *Context*. For example, in an intelligent human-computer interface, the relevant knowledge and context may be parts of the system meta-knowledge (e.g., rules that say how to generate an explanation or a hint of a specific type) and parameters of the user model.

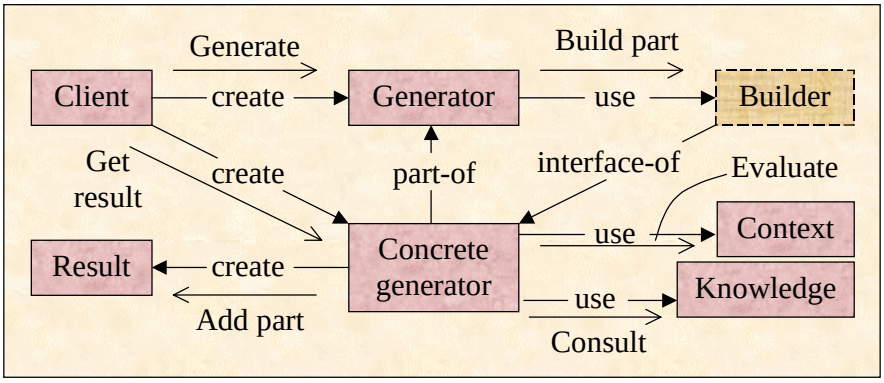

Figure 4. The Generator ontology at the design level

Libraries of ontologies and catalogues of software patterns

Ontology engineers can use prebuilt ontologies as parts of their own design, provided that the prebuilt ontologies fit the application's needs. Such ontologies exist in publicly available libraries of ontologies, which define the common models needed for combining and reusing knowledge bases. Using public ontologies can avoid the time-consuming formalization and representation processes in ontology design. Sometimes, however, using a prebuilt ontology and customizing it to fit a specific project may involve problems of translating it from one representation language to another [Valente et al., 1999].

In the realm of software patterns, prespecified families of 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. Ontology developers can browse patterns in the catalogues in search of ideas for already existing solutions to many problems that are common to ontological engineering and software engineering. Catalogues are well organized, so browsing them essentially means scanning the relevant patterns' sections that describe the pattern's intent and how the pattern interrelates with other patterns.

Complementary mechanisms

If so many issues in ontologies and software patterns look alike, what's different?

To an extent, software patterns and ontologies are complementary mechanisms for enabling knowledge sharing and reuse. Ontologies generally provide upper-level guidance and analysis for building sharable knowledge bases. However, the software patterns community doesn't describe and classify software patterns with knowledge bases in mind. Patterns contain general software engineering knowledge that is useful for developing applications in many domains, but focus on solving smaller, more specific problems of software development. Ontologies are more commonsense-oriented, software patterns are more concrete. Software patterns are often about low-level, Earthly things such as software design, but can be about more abstract activities as well (e.g., organizational patterns and analysis patterns [Fowler, 1997]). On the other hand, it doesn't take a hard mental shift to view ontologies as abstract patterns, or knowledge skeletons of some domains.

Another important difference between software patterns and ontologies is that ontologies are represented and encoded in computers, while software patterns are not. Hence ontologies enable knowledge sharing and reuse between intelligent agents other than humans, while software patterns provide means for knowledge sharing only among software analysts and designers.
These complementary mechanisms of ontologies and software patterns give us an opportunity to use patterns along with the other tools of ontology design. Patterns may be used alone or in combination, either by providing complementary views during initial development of ontologies, or by elaborating parts of ontologies. When used together, ontologies and software patterns can help establish links between the knowledge level of a problem domain and actual application development effectively.

Seeing the World Differently

The concept from the field of software patterns that is perhaps the closest to the world of ontologies is that of pattern languages (see the sidebar). Being a structured collection of interrelated patterns, a pattern language generates a family of related systems in a particular domain or discipline. Just like ontologies, pattern languages provide vocabularies of problem domains that usually go far beyond software analysis and design. For example, there is a pattern language that captures organizational pragmatics and successful management practices of highly productive organizations [Coplien and Schmidt, 1995]. Moreover, that language has taken inspiration from architectural design and models of socio-urban planning! That language not only helps understand existing organizations; it also helps build new ones. The names of the patterns from that language (e.g., Size and Schedule, Form Follows Function, Self-Selecting Team), as well as additional vocabulary from the patterns descriptions (e.g. Activities, Roles, Groups, Market, Phases), reveal that the language is quite a good candidate for the basis of ontology of development organizations.

(This image is an artwork that fits here, it is not referred to in the text)

In practice, developing ontologies by getting initial cues from software patterns, pattern languages, and pattern catalogues as specific knowledge repositories can be rather straightforward. In doing so, it helps to have a framework. An idea of what such a framework might look like is illustrated in Figure 5, adapted from [Devedžić and Radović, 1999]. The framework has its origins in a software pattern as well, the Layered Architecture pattern for software architecture (see the sidebar). The idea is to consider both patterns and ontologies at five *levels of abstraction*, and along several *dimensions* (such as concepts, methods, relations and suitable inference techniques). For example, looking along the *concepts* dimension, *primitives* are components like plain text, logical expressions, attributes and numerical values. They are used to compose *units* like rules, frames, and sets. These are then used as parts of certain building *blocks*, e.g. aggregates, tasks, and strategies. At the *system* level, we have self-contained systems or agents like planners or learning systems, all composed using different building blocks. Finally, at the *integration* level there are distributed intelligent systems, agents, and environments.

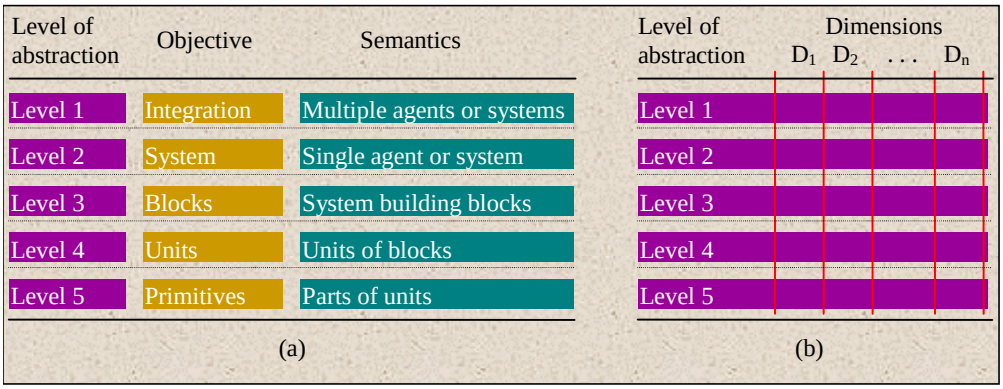

Figure 5. Layers and dimensions of intelligent systems

The hard part in applying such a framework is to map the patterns from catalogues and pattern languages to one or more levels of abstraction, since both ontologies and patterns can span more than one level. However, once this is done ontology engineers get another valuable substrate for their developments. Software patterns can shed another light to ontology development, and can help developers see the world differently.

References

[1] B. Chandrasekaran, J.R. Josephson, V.R. Benjamins, "What Are Ontologies, and Why Do We Need Them?", IEEE Intelligent Systems, Vol.14, No.1, Special Issue on Ontologies, January/February 1999, pp. 20-26.

[2] V. Devedžić, D. Radović, "A Framework for Building Intelligent Manufacturing Systems", IEEE Transactions on Systems, Man, and Cybernetics, Part C - Applications and Reviews, Vol.29, No.3, August 1999 (forthcoming).

[3] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, Reading, MA, 1997.

[4] N. Fridman-Noy, C.D. Hafner, "The State of the Art in Ontology Design: A Survey and Comparative Review", AI Magazine, Fall 1997, pp. 53-74.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, 1995.

[6] C. Larman, "Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design", Prentice-Hall, Upper Saddle River, NJ, 1998.

[7] D.B. Lenat, "CYC: A Large-Scale Investment in Knowledge Infrastructure", Communications of the ACM, Vol.38, No.11, November 1995, pp. 33-38.

[8] A. Newell, "The Knowledge Level", Artificial Intelligence, Vol.18, pp. 87-127.

[9] W. Swartout, A. Tate, "Ontologies", Guest Editors' Introduction, IEEE Intelligent Systems, Vol.14, No.1, Special Issue on Ontologies, January/February 1999, pp. 18-19.

[10] P.E. Van der Vet, N.J.I. Mars, "Bottom-Up Construction of Ontologies", IEEE Transactions on Knowledge and Date Engineering, Vol.10, No.4, July/August 1998, pp. 513-526.

Sidebar

Software Patterns

In software engineering, patterns are attempts to describe successful solutions to common software problems [Schmidt et al., 1996]. 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. Patterns are important because they help us to understand how people perceive the world [Fowler, 1997]. 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. 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.

Patterns exist in several phases of software development. The software patterns community has first discovered, described and classified a number of *design patterns* [Gamma et al., 1995]. More recent developments have also identified many patterns related to other phases and issues of software development, like *analysis patterns* [Fowler, 1997] and *patterns for software architectures* [Shaw, 1995].

Design Patterns

Design patterns are the most widely used kind of software patterns. They are related to the design phase of software development. Design patterns are "simple and elegant solutions to specific problems in object-oriented software design" [Gamma et al., 1995]. They capture static and dynamic structure of these solutions in a consistent and easily applied form. They show generalized, domain-independent solutions of stereotypical design 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 [Design Patterns Home Page, 1998], [Gamma et al., 1995].

There are catalogues of design patterns, in which all of the patterns are described using some prespecified template. For example, the template described in [Gamma et al., 1995] suggests describing each pattern by showing its name, structure, motivation for its use, commenting its applicability and the positive and negative consequences of using it, and discussing its implementation and known uses.

Using design patterns in practice means first considering what family of patterns from the catalogues is related to the particular design problem. After finding such a family, the designer considers in what way the patterns from that family solve design problems, what are their intents, what are the consequences of using them, how they are interrelated, and how they increase reusability. Finally, once the right pattern is selected, its description should be read thoroughly. Then the pattern should be adapted to fit the particular design problem.

Analysis patterns

These patterns describe the models of business processes that result repeatedly from the analysis phase of software development. 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 [Fowler, 1997]. 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 templates for describing this kind of software patterns.

Patterns for software architectures

Software systems are composed from identifiable components and connectors of various distinct types [Shaw, 1995]. 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. A pattern for software architecture is based on selected types of components and connectors, together with a control structure that governs execution. These patterns impose an overall structure for 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. They also provide information about the structure and help establish and maintain internal consistency, as well as perform appropriate checking and analysis. Examples of patterns for software architectures include pipeline, layered architecture, data abstraction, repository, and many more.
Specialists in software architectures represent structure of their patterns graphically, and often describe the patterns using predefined templates. Typical template contains the "problem", "context", "solution", "diagram" and "examples" sections.

Pattern Languages

Patterns have a context in which they apply. When several related patterns are woven together, they form a pattern language [Schmidt et al., 1996]. 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 [Coplien and Schmidt, 1995]. 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. 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.

* * *

For historical and curiosity reasons, it is also worth noting that the fields of ontologies and software patterns have started and flourished almost in exact parallel during the last decade. Also, both fields have their roots and inspiration in some works dating from the late 70's and early 80's. Many authors in the field of ontologies refer to Alan Newell's work on knowledge level as the most important early landmark. Likewise, a lot of people from the software patterns community revers the work of Christopher Alexander on patterns in architectural design as the set of ideas that have triggered their own field.

Sidebar References

[1] J. Coplien, D. Schmidt (eds.): "Pattern Languages of Program Design", Addison-Wesley, Reading, MA, 1995.
[2] Design Patterns Home Page: http://st-www.cs.uiuc.edu/users/patterns/patterns.html, 1999.
[3] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, Reading, MA, 1997.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, 1995.
[5] D. Schmidt, M. Fayad, R.E. Johnson, "Software Patterns", Communications of The ACM, Vol.39, No.10, 1996, pp. 37-39.
[6] M. Shaw, "Patterns for Software Architectures", in: J.Coplien, D. Schmidt (eds), "Pattern Languages of Program Design", Addison-Wesley 1995, pp 453-462.