An Ontology for Microarchitectural Design Knowledge

Javier Garzás, *mCentric*
Mario Piattini, *Alarcos Research Group*

Our ontology can help you better understand how to implement and refactor your OO design knowledge, thereby improving quality, reducing costs, and saving time.

Twenty years ago, Samuel Redwine and his colleagues commented that "an expert in a field must know about 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived,"[1] adding that in mature areas this takes about 10 years. Since then, many authors have cited a need for such chunks in software engineering,[2] and the software engineering community now has many defined chunks of knowledge—standards, methodologies, methods, metrics, techniques, languages, patterns, processes, concepts, and so on. Nevertheless, knowledge in the field still lacks a structured classification,[3] and not all knowledge is transmitted, accessible, or studied in the same way. For example, where exactly is the enormous amount of practical knowledge accumulated regarding object-oriented (OO) microarchitectural design? In other words, where is the knowledge accumulated from working with the inherent properties of software—knowledge that's generally accepted in most projects as being valuable and useful?[4]

In this article, we present an ontology that structures and unifies this accumulated OO microarchitectural design knowledge. This ontology differentiates between declarative and operative knowledge, and encompasses rules, patterns, and refactorings. It also encompasses the differences and relationships between these types of knowledge.

Elements of OO microarchitectural design knowledge

An essential item of knowledge in OO design is the concept of *patterns*. But this is just the tip of the iceberg. Suppose we want to specialize as software engineers in OO design. Projects such as SWEBOK can help us ascertain what *design* is, learn how it's subdivided, find the main bibliographical references and so on, and clearly see how to acquire a sound theoretical knowledge of the subject.[4] If we concentrate part of our professional activity on design, we must study the practical experience of other experts in the area, and thus the concept of a pattern occurs to us. Yet after examining the main pattern references in OO design, we still think something's missing. Besides pat-

0740-7459/05/$20.00 © 2005 IEEE

Table 1

Examples of OO design knowledge

Knowledge element	Example
Principle	The dependency inversion principle: Depend on abstractions, not concretions.[6]
Heuristic	"If two or more classes only share common interface (i.e. messages, not methods), then they should inherit from a common base class only if they will be used polymorphically."[7]
Best practice	"See objects as bundles of behavior, not bundles of data."[8]
Bad smell	Refused bequest (a literal name that Martin Fowler used for a bad smell): These are subclasses that don't use what they inherit.[5]
Refactoring	Extract interface: "Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. Extract the subset into an interface…."[5]
Pattern	Observer (a design pattern): Intent is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically…."[9]

terns, missing elements for formulating a good microarchitectural design include principles, heuristics, best practices, "bad smells" (which indicate that a problem may exist that you need to address[5]), and refactorings. Table 1 gives an example of each of these elements.

However, the differences between these elements aren't clear. Many of them concern a single concept with different names. Others don't always contain knowledge gained from experience. Still others are simply vague concepts. This confusion leads to a less efficient use of knowledge, so concepts such as principles and heuristics are still unknown to some software engineers and are completely understood by only a few.[10] There's been considerable progress in the accumulation of experience-based knowledge of OO microarchitectural design, but considerably less progress in its exploitation and classification. This could be an example of the *Feigenbaum Bottleneck*, which Carlo Pescio describes as follows: "As domain complexity grows, it becomes very difficult for human experts to formulate their knowledge as practical strategies ...."[11]

Our ontology

An ontology describes domain knowledge in a generic way and provides an understanding of it. Using an ontology can help

- Structure and unify accumulated essential knowledge
- Improve communication
- Teach concepts and their relationships
- Share knowledge
- Resolve terminological incompatibilities

Thus, defining an ontology that structures and unifies OO microarchitectural design knowledge is important. However, before we describe our ontology, we first explain its scope.

Scope of OO microarchitectural ontology

Referring to the SWEBOK guide,[4] we determine that our ontology falls under Software Design. However, it's not clear in which of the six subareas of this category it belongs. There are places for two of the elements of knowledge we've been considering: principles ("enabling techniques" in the SWEBOK guide) under Software Design Fundamentals, and design patterns under Software Structure and Architecture. However, the place for other concepts such as refactoring isn't so obvious (the SWEBOK guide only briefly mentions refactoring in the Software Maintenance section). Our ontology concentrates on microarchitectures. Therefore, we consider the best fit to be Software Structure and Architecture because this is the focus of microarchitectural design strategies. We propose including a more generic area within Software Structure and Architecture called Microarchitectural Knowledge, as Figure 1 shows. We divide this area into OO and functional knowledge, and our ontology structures the OO part.

Entities of knowledge

Many elements can be associated with essential knowledge gained from experience with OO microarchitectural design, but most are vaguely defined and confusing, and it's often unclear whether they constitute knowl-

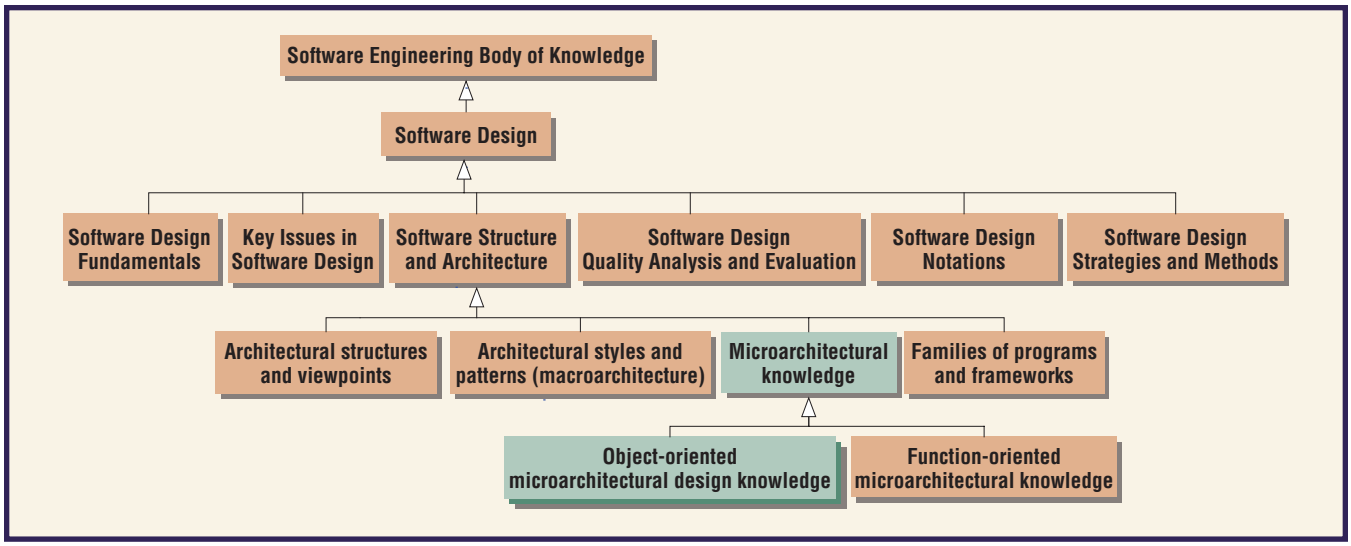

Figure 1. Context and situation of the ontology of object-oriented microarchitectural design knowledge according to the SWEBOK guide.[4]

edge. To clarify the situation, we divide the elements related to OO design knowledge into two initial groups:

- *Declarative knowledge* involves concepts (heuristics, patterns, bad smells, best practices, and so on) describing what to do with a problem.
- *Operative knowledge* concerns operations or processes for making design changes (parameterized transformations of a program preserving its functionality[12]) and includes concepts such as refactoring. Here, we mean design refactoring, not code refactoring, which is more common. Although refactoring falls under the Software Maintenance category in the SWEBOK guide, grouping together all knowledge related to OO microarchitectural design is important, and this improves its localization and utilization.

We can thus establish an initial classification of essential experience-based knowledge of software design with two parts: declarative and operative. The former indicates what to do; the latter how. Both cases of knowledge are based on experience. Figure 2 shows a unified modeling language (UML) class diagram expressing our ontology.

Considering only declarative knowledge without regard to design patterns, we observe that principles, heuristics, bad smells, and so on have a common structure with no substantial difference between them. They all have the structure and form of a rule: They posit a condition and offer a recommendation. A recommendation is not a solution like that of a pattern. Patterns are more formalized than rules, and pattern descriptions are always broader. Patterns propose solutions to problems, whereas rules are recommendations that a design should fulfill. Unlike patterns, rules use mainly natural language, which can be more ambiguous.[11]

Finally, to complete the description of entities, we must concentrate on their attributes. We based our determination of attributes on the terms used in the catalogue that Erich Gamma and his colleagues used to describe a design pattern: *name, intent, also known as, motivation*, *structure*, *applicability*, *participants*, *collaborations*, *consequences*, *implementation*, *sample code*, *known uses*, and *related patterns*.[9] Many of these items are common to other elements of knowledge, and these common attributes are located in the top knowledge entity in Figure 2.

As to the other attributes under consideration, *structure* is a synonym for a solution in a pattern, and we created the *recommendation* attribute for rules (which would be close to the pattern's solution[11]). We also created the *mechanics* attribute for refactorings—our name choice here was based on Fowler's refactoring catalogue.[5] The *participants* (the classes or objects participating in the design pattern and their responsibilities) and *collaborations* (how the participants collectively carry out their responsibilities) attributes

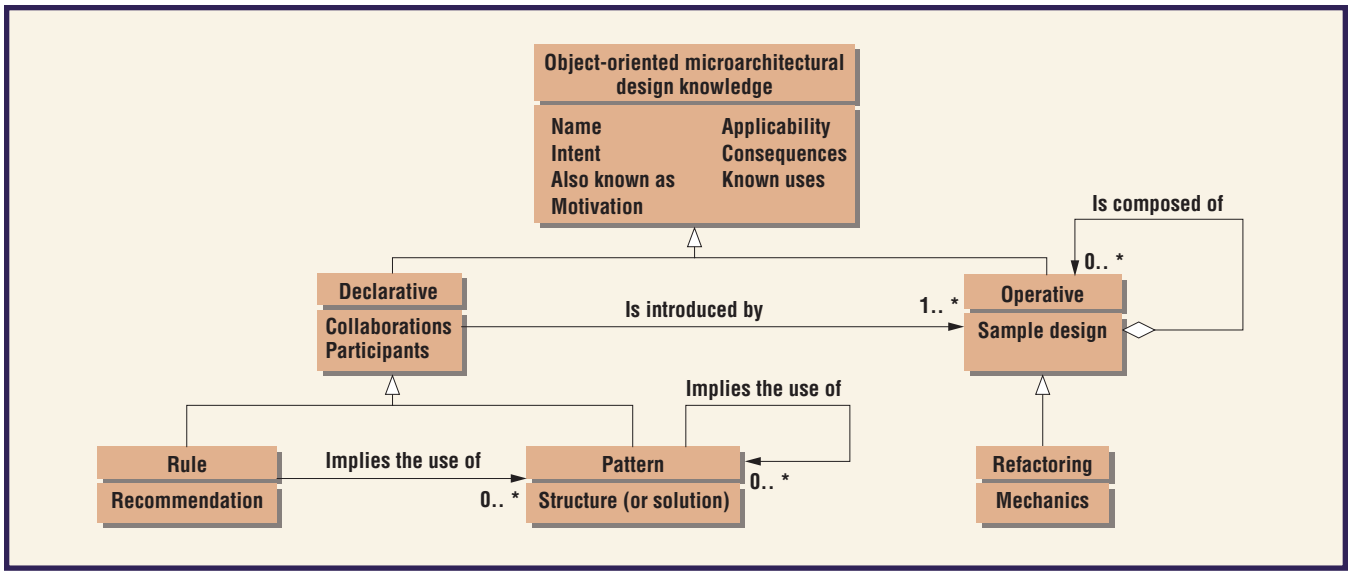

Figure 2. Unified modeling language (UML) class diagram expressing our ontology for object-oriented microarchitectural design knowledge.

concern declarative knowledge. The *sample design* attribute concerns operative knowledge. We substituted the *implementation* attribute for *mechanics*. We generalized the *related patterns* attribute, and it appears in each of the relationships between entities.

Relationships

We now focus on the relationships between entities (see Figure 2).

Applying a rule implies using a pattern. Introducing a rule often leads to a new design, which requires a pattern. For example, consider the Dependencies on Concrete Classes Rules (DCCR) design rule. (The "Example of an Object-Oriented Microarchitectural Design Rule" sidebar gives the attributes for such a design rule.) This rule introduces an abstract class or an interface,[6] which in turn necessitates a creational pattern to create instances and associate objects in the new situation.[9] Not all rules imply the introduction of a pattern, so cardinality is 0 to *n*. A clear case occurs when we apply rules that work only inside a module—for example, the "long method" rule, a bad smell according to Fowler.[5]

Applying a pattern implies using another pattern. Catalogues and pattern languages have featured this relationship for some time. An example is the map of relationships between patterns presented by Erich Gamma and his colleagues.[9] Once again, cardinality is from 0 to *n*; we can see in the work of Gamma and his colleagues how adapter, proxy, and bridge patterns are isolated.[9]

Introducing declarative knowledge using operative knowledge. In the design, each element of declarative knowledge (rules and patterns) is introduced by an element of operative knowledge (a refactoring), with cardinality from 1 to *n*. This is obvious because it doesn't make sense for an element of declarative knowledge to exist if it can't be introduced.

First, we can implicitly observe the relationship between patterns and refactorings by reading some of the refactoring catalogues that concentrate on the design level.[5] Gamma's catalogue states that "design patterns provide the refactorings with objectives."[9] Moreover, there's a natural relationship between patterns and refactorings: Patterns can constitute the objectives; and refactorings, the way to achieve those objectives. In fact, there should be catalogues of refactorings for all design patterns.[5] In this way, refactorings (such as "Replace type code with state/strategy" or "Form template method") can focus on introducing patterns within a system (again to emphasize that these are design refactorings, not code refactorings).

Second, researchers haven't studied the relationship between rules and refactorings as much as between patterns and refactorings. Rules can be introduced in the design just like patterns by using refactorings. Such refactorings store knowledge about how to introduce elements in designs in a controlled way. Continuing with the DCCR example, to resolve a violation of this rule, we insert an abstract entity into the design, which software engineers would normally handle using refactorings.

Example of an Object-Oriented Microarchitectural Design Rule

In line with our ontology, we developed a unified catalog composed of 20 rules. To improve the detection of rules, we named them according to their *condition*. So, for example, here we define the rule called "If there are dependencies on concrete classes."

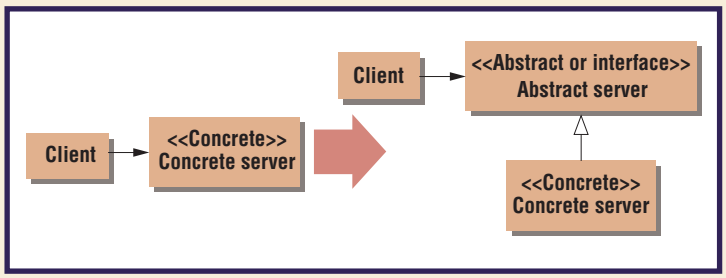

Figure A. Recommendation if there are dependencies on concrete classes.

Intent

The strategy of this rule depends on interfaces or abstract classes rather than on concrete elements.

Also known as

This rule is also known as a dependency inversion rule, or "programming to an interface, not an implementation."

Motivation

The structural design shows a particular type of dependency in which high-level modules depend on low-level ones. High-level modules handle policies with low-level modules. Generally speaking, these policies have little to do with the details of their implementation. So, why do high-level modules depend directly on implementation modules?

Object-oriented architectures show dependencies mostly on abstractions; the modules that contain implementation details also depend on these abstractions, and not vice versa. Thus, the dependency has been inverted.

This rule implies that each design dependency's objective must be an interface or an abstract class; the dependencies must not have concrete classes as objectives. Concrete things are far more liable to change than abstract ones.

Applicability

Use this rule when you find dependencies on or associations with concrete classes that may change.

Don't use this rule when a dependency exists on a concrete class that's unlikely to change—for example, a library class such as String.

Recommendation

If there are dependencies on concrete classes, these dependencies should be on abstractions, as in Figure A.

Participants

The participants include the client, concrete server (implementation), and abstract server class (concrete server interface).

Collaborations

The client communicates with the abstract server class, and the concrete server implements the abstract server.

Consequences

This rule has the following benefits: It allows the introduction of abstractions to extend the design without modifying it; limits the effect of design variations; lets all subclasses respond to interface requests; and ensures that clients aren't aware of the specific types of objects used.

Known uses

This rule is used in many design patterns, frameworks, and component models.

Implies the use of (patterns)

One of the most common situations in which a design depends on concrete classes is when instances are created. By definition, it's impossible to create instances of abstract classes. Concrete classes must be instanced, and the creational patterns (abstract factory, builder, factory method, prototype, and singleton) allow this instantiation. These patterns provide different ways to associate an interface with its implementation. Creational patterns ensure that your system is written in terms of interfaces, not implementations.

Is introduced by (refactorings)

Extract interface refactoring can introduce this rule.

An element of operative knowledge comprises other elements of operative knowledge. Examples of this composition are available in refactoring catalogues such as Fowler's,[5] which show, for instance, that the "Extract method" refactoring is not composed but is used by others (cardinality 0 to *n*).

Why some relationships aren't included in the ontology. Finally, we must explain why other relationships are not included:

- *A pattern implies the use of a rule.* Introducing a pattern must not violate a rule; applying a pattern shouldn't reduce design quality.

- *A rule implies the use of another rule*. Similar to the previous case, introducing one rule must not violate another; applying a rule shouldn't reduce design quality.
- *Operative knowledge implies declarative knowledge*. A refactoring doesn't know which rule or pattern it uses.

Currently, we are working on metrics associated with OO microarchitectural knowledge. We also plan to investigate how elements of knowledge can influence OO design quality, discover and catalogue more knowledge elements, and carry out several empirical studies about these topics.

Acknowledgments

This research is part of the Agile Software Maintenance project TIC 2003-02737-C02-02, funded by the Spanish Ministry of Science and Technology. We thank the anonymous reviewers for their valuable comments and insights.

References

1. S.T. Redwine et al., *DOD-Related Software Technology Requirements, Practices, and Prospects for the Future*, tech. report P-1788, Inst. Defense Analyses, US Dept. of Defense, 1984.
2. M. Shaw, "Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 7, no. 6, 1990, pp. 15–24.
3. S. McConnell, *Professional Software Development*, Addison-Wesley, 2003.
4. A. Abran et al., eds., *Guide to the Software Engineering Body of Knowledge: SWEBOK*, IEEE CS Press, 2004; www.swebok.org.
5. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
6. R.C. Martin, "The Dependency Inversion Principle," *C++ Report*, vol. 8, no. 6, 1996, pp. 61–66.
7. A.J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.
8. B. Venners, *Interface Design: Best Practices in Object-Oriented API Design in Java*, Artima Software, 2004; www.artima.com/interfacedesign/contents.html.
9. E. Gamma et al., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.
10. C. Schwanninger, "Patterns as Problem Indicators," *Proc. Workshop on Beyond Design Patterns (Mis)Used*, ACM Press; www.schwanninger.com/OOPSLA2001/#PositionPapers.
11. C. Pescio, "Principles Versus Patterns," *Computer*, vol. 30, no. 9, 1997, pp. 130–131.
12. W. Opdyke, *Refactoring Object-Oriented Frameworks*, doctoral dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.

About the Authors

Javier Garzás is a project manager at mCentric in Madrid, where he drives the software process improvement program, and he is a lecturer at Rey Juan Carlos University. His research interests include the Capability Maturity Model, object-oriented design, and software process and project management. He received his PhD in computer science from the University of Castilla-La Mancha. Contact him at javier.garzas@m-centric.com or jgarzas@gmail.com.

Mario Piattini is a full professor and leads the Alarcos Research Group at the University of Castilla-La Mancha. His research interests include advanced database design, database quality, software metrics, object-oriented metrics, and software maintenance. He received his PhD in computer science from the Polytechnic University of Madrid. Contact him at mario.piattini@uclm.es.