

Recasting Class Patterns as Semantic Patterns

by Cliff Berg

Background

The concept of naming and cataloging software patterns is one of the most significant advances in practical software development. Patterns provide both a universal framework and a lexicon for use by practicing software programmers. For example, the Java language standard libraries contain of the order of a hundred uses of the Factory pattern [GOF], such as the `ContentHandlerFactory` in package `java.net`. The `ContentHandlerFactory` is actually a Java interface, which is significant, because programmers who must implement the interface must anticipate the behavior and usage of a proper implementation of `ContentHandlerFactory`. The fact that `ContentHandlerFactory` identifies a standard pattern makes the programmer's job easier, because they have as a guide other uses of factories. That is, the term "factory" is well understood by now, and examples of factory implementations abound.

Patterns have their origin in object-oriented programming. A primary focus of object-oriented programming is reuse. As a result, patterns historically focus on the decomposition of structures in order to generalize them so that they can be simplified and reused. Simplification has many benefits beyond reuse, including increased reliability. Another primary focus of object-oriented programming is information hiding [Parnas]. Information hiding facilitates reuse because it reduces the coupling between components. However, information hiding also benefits reliability. In order to model aspects such as decomposition and information hiding, patterns often employ class diagrams. Class diagrams incorporate features and relationships such as encapsulation of instance state, cardinality between instances, and decomposition via inheritance.

Problem

Allowing for the substantial benefits of class diagrams, there are other aspects of patterns that are often more important than decomposition, and potential reuse is but one consideration in terms of design robustness. Reliability derives from more than simplification and information hiding: it also derives from the controlled flow of information, the complete handling of errors, and from correct concurrency design, among many other things. Patterns that focus on these aspects therefore are often best represented using notation other than class notation. In particular, in many cases, the dynamic relationships between components is of more interest than the static structural relationships. Such dynamic relationships might include access rights, visibility, data flow, identity, trust, processing rules

and algorithms, state, kinds of actors, roles, and reentrancy.

Let's consider the widely used GOF patterns [GOF], which is arguably the most well-known set of software patterns, and which Jon Erickson has described as "simple and elegant solutions to specific problems in object-oriented software design". Many of these patterns are indeed structural in nature. For example, the Adapter pattern serve to structurally adapt one kind of interface to another. However, what if the interface includes temporal behavior, such as a protocol? In that case, the adapter pattern, cast in its common structural representation, is inadequate. However, the concept of the adapter pattern may still apply: adapting one interface to another for the purpose of joining two systems. If the adapter pattern were cast differently – say, in terms of an abstract concept of interface, rather than through an interface class – then it might still apply.

Discussion

As a more interesting example, let's consider the Strategy pattern [GOF], which is shown in the figure below. The Strategy pattern shows a set of classes that facilitate this kind of substitution. It is basically an indirection mechanism, defined in terms of classes.

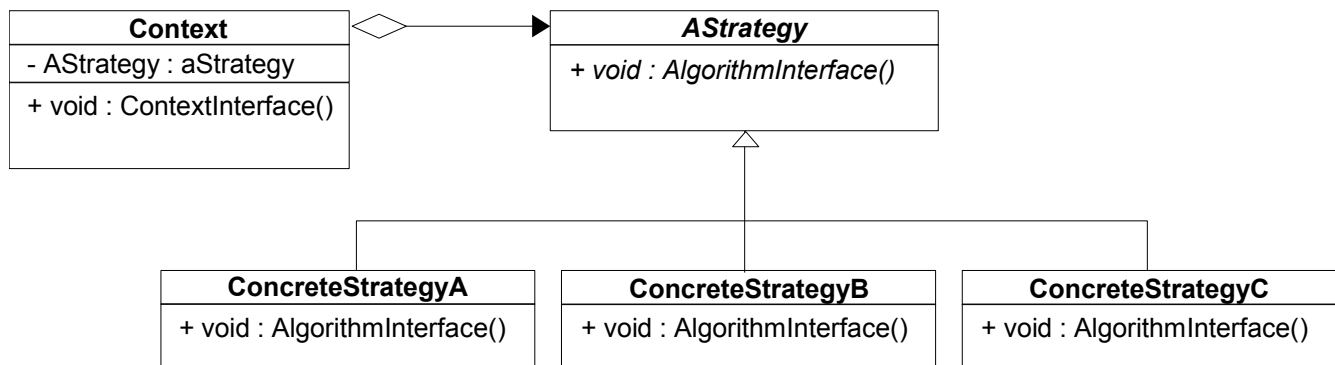


Illustration 1: The Strategy Pattern - Class View

The Strategy pattern represents a simple idea: A template for how to choose or switch out behavior at runtime, without the client of the behavior having to know about it. For example, if a component called Client accesses a component called Context, the functionality that Context provides might need to depend on the kind of object that the Context is currently operating on; or, the functionality of Context might depend on factors that are unknown to Client. These are two very different situations, but in both situations there is a need to decouple the implementation of the context's functionality from the usage of the context, so that the functionality can be chosen depending on the circumstances. Who does the choosing – the client or the Context's environment – depends on the application.

The original specification of the Strategy pattern identified the first of the two cases above. That is, it assumed that the client would make the choice of which strategy was appropriate. In many applications

of the Strategy pattern, however, the second case is more appropriate. For example, if the client is external to the application domain, and is therefore less trusted, then perhaps it should not be selecting the strategy. In that case, the Context's environment can decide which variant of behavior is appropriate for Client, and plug that behavior in.

There are implicit semantic relationships embodied in the Strategy pattern. One such implicit relationship is which component domain should be trusted to choose which strategy implementation to plug in: the client domain, or the Context's domain. Let's assume the second choice, that the Context's domain chooses the strategy. That is, the Context can access only the strategy that is provided to it by its environment. That is the only way that the environment can be sure that the correct strategy is used.

Two additional implicit relationships are that the environment owns the strategy implementations, and that the implementations are private to the environment. That is the only way that the environment can ensure the integrity of the strategy implementations. The original specification of the Strategy pattern does not mention these considerations. The only access control consideration that was mentioned was that the reference from Context to a strategy should be private to the Context. The omissions are likely the result of the absence of suitable instance access control mechanisms in the languages that were prevalent at the time, particularly C++, which was used by the GOF to code the examples.

Another implicit relationship is that the Context must have or be given the right to invoke the strategy that it is provided with. Again, the original GOF Strategy pattern does not mention this, probably due again to the assumption that Context would have to have access to all strategies in order to have access to any of them – again, the result of the absence of instance-level access control in C++.

The essence of the Strategy pattern is therefore the set of relationships. These are depicted in Illustration 2 below.

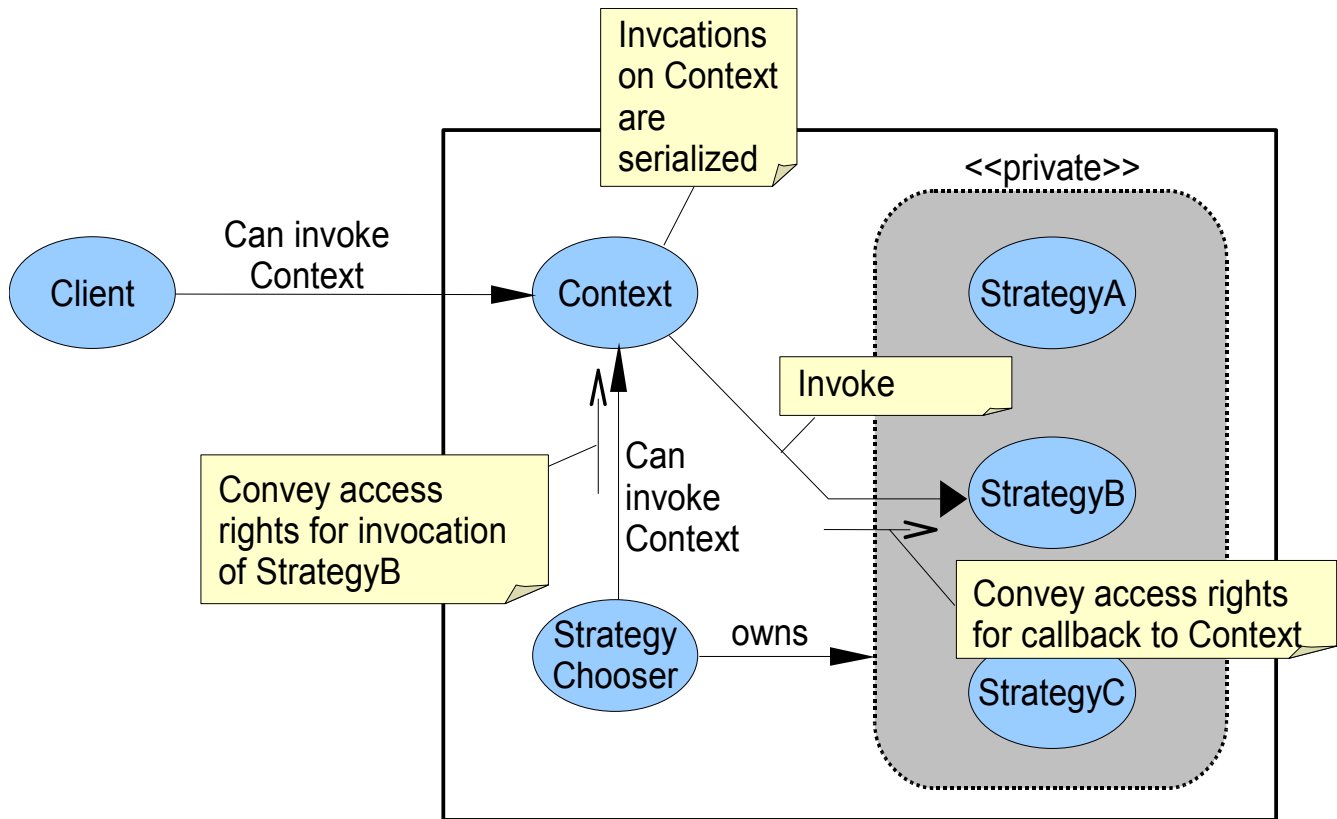


Illustration 2: Strategy Pattern - An Access Rights View

Note that the revised Strategy pattern does not specify any classes, methods, or inheritance. Rather, it defines component instance roles, and relationships between those components. Such roles can be considered to be classes, but note that it is not necessary to define method signatures or instance variables. This representation captures the semantics of the Strategy pattern, rather than proposing a possible structure with method names or class names that most likely would not be used in practice.

There is another relationship that is shown in the figure that I have not mentioned: When the Context invokes a chosen strategy (StrategyB in the figure), it conveys the right for that strategy to call back to the Context into obtain data from the Context or to update the Context's state. This interaction is discussed in the GOF specification of the Strategy pattern: "Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required."

This makes it clear that the intent of the Strategy pattern is to allow two choices for design, and both have to do with the ability of a strategy to access the Context. One choice is to establish that communication in a lazy manner, whenever a strategy is called. The other choice is to establish it up

front, when a strategy is chosen. In the Strategy pattern shown in Illustration 2, the first approach is used. Illustration 3 below shows how the access rights view would change if the second approach were used.

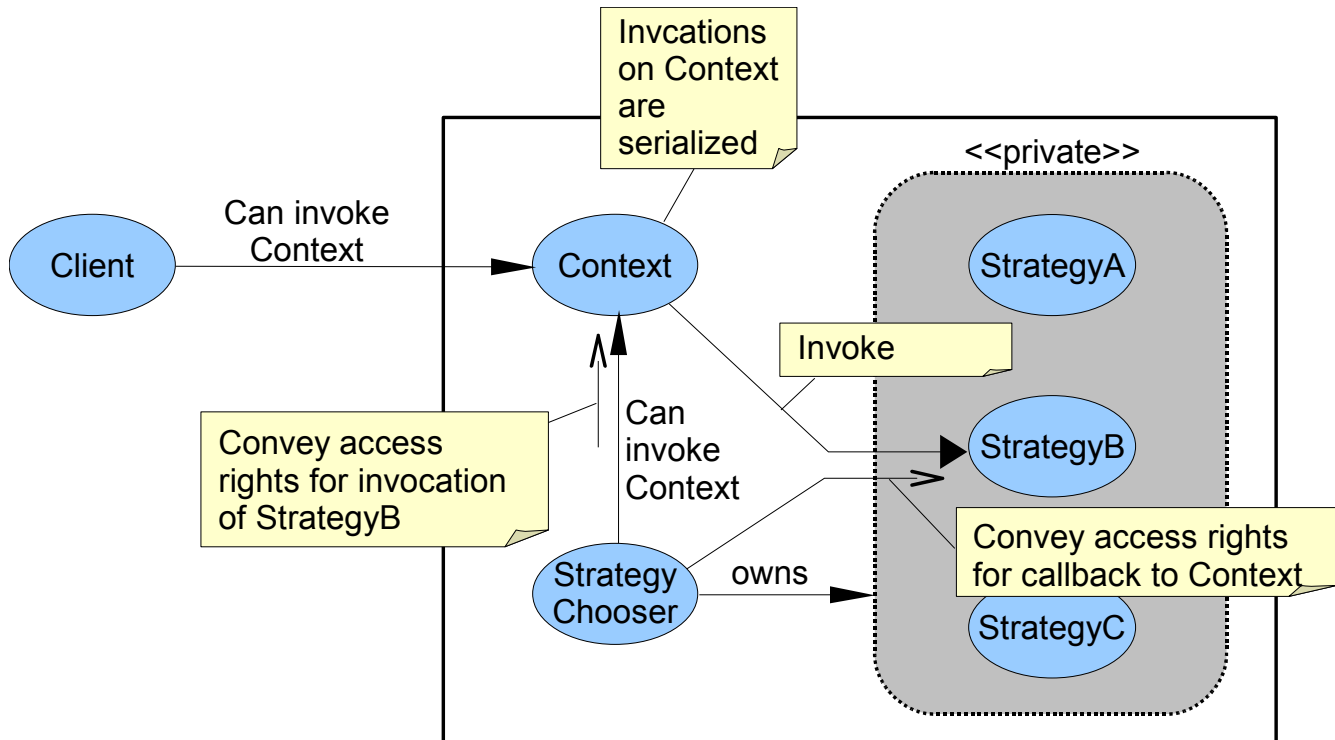


Illustration 3: Strategy Pattern - An Access Rights View, With Access From Strategy to Context Established Up Front

The GOF patterns are divided into three categories: Creational, Structural, and Behavioral. The Strategy pattern is listed as a Behavioral pattern. But as we have seen, its specification relies on a structural view. The behavioral semantics are described textually in the pattern specification.

The access-rights-based view of the Strategy pattern utilizes access rights semantics. There are many other kinds of semantic relationships and attributes that are useful for modeling. The next section discusses different kinds of model semantics.

It turns out that most of the GOF patterns can be recast in terms of semantic relationships – without specifying structure or inheritance. The relationships are actually key, and in the traditional representation of these GOF patterns, inheritance is used to indicate certain relationships that would be better called as they are. For example, the Strategy pattern needs to indicate that an object can provide variations in its behavior. The original expression of the Strategy pattern achieves this by having the behavioral variants implement the same interface. However, this is not necessary. In fact, there are many structural ways to achieve behavioral variants, depending on the language and other design considerations. The key concept in the Strategy pattern is the ability to plug in new behaviors, and this can be expressed through other means, as has been shown in this article.

If alternative representations are useful, the question arises, what are the kinds of representations that are more useful? There are many modeling and design languages to draw from, and these are beyond the scope of this article. Some interesting recent efforts include the ArchJava programming language [ArchJava] and the Acme design language [Acme]. Both of these provide features that address domain integrity, which is an important consideration for security and reliability.

Summary

Design patterns are much more useful if their semantic intent is captured in a precise way. Most of the design patterns that are useful for software are semantic in nature, not structural. Adding structure that is not really germane to a pattern results in unnecessary complexity that software designers usually have to remove or modify when instantiating a pattern. Therefore it is better if structure is omitted. Names of methods or classes also represent complexity that is not germane to a pattern, as evidenced by the fact that most uses of a pattern require that names be changed. Every boundary or line in a design pattern implies a rule of some type, and such rules should be identified if they are truly pertinent to the pattern, or else they are extraneous and should be eliminated. Following these practices will lead to patterns that are more reusable and possibly the creation of pattern libraries. The creation of a pattern design language that is semantic rather than structural, and that provides the needed precise semantics rather than imprecise descriptions, is a necessary step toward the creation of libraries of directly reusable patterns.

References

GOF – Design Patterns, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley Professional, January 1995. See <http://hillside.net/patterns/DPBook/DPBook.html>.

Parnas – Parnas “On the criteria to be used in decomposing systems into modules,” D.L. Parnas. Communications of the Association of Computing Machinery, December 1972.

ArchJava – “Using Types to Enforce Architectural Structure,” Jonathan Aldrich. PhD Thesis, University of Washington, 2003. See also <http://archjava.fluid.cs.cmu.edu/>.

Acme – “Acme: Architectural Description of Component-Based Systems,” by David Garlan, Robert T. Monroe, and David Wile. In Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000, pp. 47-68. Available online at <http://www-2.cs.cmu.edu/~able/publications/acme-fcbs/>.