Software Design Patterns: Common Questions and Answers

James O. Coplien
Software Production Research Department
cope@research.att.com
AT&T Bell Laboratories

Abstract

Software design patterns are an emerging tool for guiding and documenting system design. This paper is a beginner's guide to patterns, answering common questions about patterns, pattern languages, generativity, and Alexandrian form. The paper describes patterns in terms of objects, rules, idioms, and paradigms in general. The paper also provides sources for patterns, and provides a short bibliography of patterns papers that have appeared in recent literature.

1: What is a pattern?

The term *pattern*, as adopted by contemporary software designers exploring its benefits, is both part of a system and a description of how to build that part of the system. Patterns usually describe software abstractions used by advanced designers and programmers in their software. As abstractions, patterns commonly cut across other common software abstractions like procedures and objects, or combine more common abstractions in powerful ways. For example, using one object as an iterator that non-destructively visits every element of a collection object, is a simple pattern.

The term *pattern* applies both to the thing (for example, a collection class and its associated iterator) and the directions for making a thing. In this sense, software patterns can be likened to a dress pattern: the general shape is in the pattern itself, though each pattern must be tailored to its context.

2: Where does this use of the term 'pattern' come from?

In the 1960s, building architects were exploring the issues of design in its broadest sense, and investigating automated, computerized building design. There was keen interest in rules and algorithms that could transform requirements into a configuration of building modules. The mainstream of this movement moved on to what we know as modular construction. The architect Christopher Alexander broke with that movement, noting that the great architectures of history were not made from rigorous, planned designs, but that their pieces were custom-fit to each other and to the building's surroundings. He also noted that some buildings were more aesthetically pleasing than others, and that these aesthetics were often attuned to human needs and comforts. He found recurring themes in architecture, and captured them into descriptions (and instructions) that he called patterns. The term 'pattern' appeals to the replicated similarity in a design, and in particular to similarity that makes room for variability and customization in each of the elements. Thus, *Window on Two Sides of Every Room* is a pattern, yet it prescribes neither the size of the windows, the distance between them, their height from the floor, nor their framing (though there are other patterns that may refine these properties).

Software designers have discovered (and rediscovered) analogies between Alexandrian patterns and software architecture patterns over the past decade.

3: What is the difference between a non-generative pattern and a generative pattern?

We observe patterns everywhere as we interact with the world around us. There are patterns in building architecture, patterns in nature, and patterns in the software people write. Recent work in software visualization is striving to bring out the patterns in software.[1] Not all these patterns are ''good'': for example, such research has found that people copy code from one place in a system to another, instead of generalizing the original code and reusing it in-place.

Patterns that we observe in a system that has already been built are non-generative, and are sometimes called *Gamma patterns*, after the pioneering work of Erich Gamma.[2] Such patterns are descriptive and passive.

That we can find these patterns doesn't imply any rationale behind them, and not every pattern leads to desirable results. What we want to do is capture the patterns that are ''good,'' and codify them, so people can use them when building systems. These patterns *generate* systems, or parts of systems. We can observe the patterns (in other words, see their structure or their effects) in the systems they generate. Patterns designed to shape system architectures are called *generative patterns*. They are prescriptive, and active.

4: What is 'Alexandrian form'?

We express most patterns in a format called *Alexandrian form*. Alexandrian form tells what is contained in a pattern. As described above, a pattern is both the thing and the instructions for making the thing. The form of pattern expression makes them particularly useful. The form draws on constructs used by Christopher Alexander to describe his patterns.

There are variants on Alexandrian form, but most look like this:

- *The pattern name*: A name by which the pattern is called. For example, we might create a pattern called 'C++ Type Promotion,' just so we can refer to it conveniently.
- *The problem the pattern is trying to solve*: If people know what problem the pattern solves, they'll know when to apply it. For example, *Type Promotion* solves the problem of promotion between objects of different C++ types, zero or one of which is a built-in type, or a type exported by a library for which the programmer does not have source.
- *Context*: A pattern solves a problem in a given context, and it may not make sense elsewhere. For example, C++ type promotion applies to C++, and potentially to other object-oriented languages with strong compile-time type systems; the decision of which promotion to apply is made at compile time; and the context is inadequate to apply built-in translation rules (e.g., from a derived class to one of its public base classes).
- *Forces, or tradeoffs*: Not all problems are clear-cut. Forces make clear the intricacies of a problem. For example, in *Type Promotion*, we need to point out the tension between using constructors and using conversion operators. We must also point out that we can redefine some properties for class types that we cannot redefine for built-in types. A good pattern resolves one or more forces.
- *Solution*: This describes the structure, behavior, etc., of the solution, which is often tantamount to telling how to build the solution. For *Type Promotion*, a program should promote class objects to built-ins using a member conversion operator, and should use constructors for all other promotions.
- *Examples* are present in all good patterns. *Visual analogies* are also powerful and frequent adornments for a pattern: you can draw a picture for most (but not all) good software patterns.
- *Force resolution, or resulting context*: Few patterns are perfect or stand on their own. A good pattern tells what forces it leaves unresolved, or what other patterns must be applied, and how the context is changed by the pattern.
- *Design rationale*: This tells where the pattern came from, why it works, and why experts use it. Good designers are most effective when they apply patterns insightfully, and that means being able to understand how the patterns work.

5: What is a 'Pattern Language'?

A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture. It is not a programming language in any ordinary sense of the term, but is a prose document whose purpose is to guide and inform the designer. Patterns rarely stand alone. Each pattern works on a context, and transforms the system in that context to produce a new system in a new context. New problems arise in the new system and context, and the next ''layer'' of patterns can be applied.

We usually use (or imply) the term ''generative'' when speaking of pattern languages. Just as the English language can generate all possible papers in conference proceedings, so a pattern language can generate all sentences in a given domain. Good pattern languages guide the designer toward useful architectures and away from architectures whose literary analogies are gibberish or unartful writing. Good architectures are durable, functional, and aesthetically pleasing, and a good combination of patterns can balance the forces on a system to strive toward these three goals. A good pattern language gives designers freedom to express themselves and to tailor the solution to the particular needs of the context where the patterns are applied.

Pure Alexandrian pattern languages uniquely specify the order in which patterns can be applied (though one may choose any one from a fixed set of patterns at any point in a given design). For software design patterns, we are finding that a more iterative approach makes better

sense.

6: Why am I hearing about patterns at object conferences?

Probably because that's where the focus of software design lies today, which in turn owes to the tremendous economic forces behind object-oriented programming languages, tools, and methods. Patterns are riding the wave of the object movement, driven by individuals commonly associated with object-oriented analysis and design (Beck, Cunningham, Johnson, Coad, Booch), but there is nothing intrinsically object-oriented about patterns. Patterns aspire to rise above object-level abstractions to more subtle structures, threads of execution, arrangements, and transformations in software. Such patterns are present in most object-oriented systems, but good patterns can be found in programs written in most programming languages under many design regimens.

Many current design patterns are expressed in terms of objects or object-oriented constructs, but good designers should introspect about how they use other paradigms effectively as well.

7: Why should I be interested in them?

Because they are gaining visibility, applicability and momentum in the object movement, which is where the action seems to be right now in software. People advocating patterns also think they are valuable (me among them), but the newcomer to patterns shouldn't expect to find volumes of material they can take home and use immediately. The best advice right now is to stay tuned, so you'll be better positioned to use patterns when they come of age.

8: What is the difference between 'patterns' and 'paradigms'?

The pattern approach is a paradigm only in the most abstract, almost philosophical sense of the word. The goal of most popular paradigms is to make a complex work understandable by organizing it into separable parts, each of which can be separately understood. Patterns try to describe relationships within and between the parts, not just the parts themselves. This is why patterns are truly an architectural technique, not just a divide-and-conquer technique.

The insightful might note that Webster gives *pattern* as one definition of *paradigm*.

9: Aren't patterns just rules?

Rules aren't commonly supported by a rationale, nor put in context. A rule may be part of the solution in a pattern description, but a rule solution is neither sufficient nor necessary. Patterns aren't designed to be executed or analyzed by computers, as one might imagine to be true for rules: patterns are to be executed by architects with insight, taste, experience, and a sense of aesthetics.

10: How are patterns and idioms related?

Idioms are special kinds of patterns that are tied to a specific programming language. I have discovered that recasting idioms in pattern form often makes them more accessible to moderately experienced C++ programmers. For example, compare the pattern form of *C++ Type Conversion* in Appendix 1, with the less structured discussion on pages 54 through 58 of my advanced C++ text. [3]

11: How do patterns relate to creativity?

Design is a creative act; so is the creation or application of a design pattern. If design is codified in patterns, does the need for creativity go away? Can we replace high-priced expensive designers with less sophisticated programmers who are guided by patterns? The answer is that creativity is still needed to shape the patterns to a given context. Just as a dressmaker tailors a pattern to an individual customer, and perhaps to a specific event where the dress is to be worn, so designers must be creative when using patterns. Patterns channel creativity; they neither replace nor constrain it.

12: Where can I find patterns?

Numerous articles and papers have now appeared both in published journals and in the trade press; in particular, see those by Beck, [4] Booch, [5] Gabriel, [6] and Johnson. [7] Watch for my forthcoming column on pattern in the C++ Report. Also, get plugged into the upcoming Patterns Conference in August in Illinois: PLoP (Pattern Languages of Programs). Send me an electronic mail inquiry if you are interested in more information.

13: How do I get started writing patterns?

Introspect about what makes your designs great in areas where you have recognized expertise or specialization, *and write them down* in Alexandrian form. Try them out on your friends.

A Pattern: C++ Type Promotion

Pattern Name: Type Promotion

Problem: Promotion between objects of different but related C++ types, zero or one of which is a built-in type, or a type exported by a library for which the programmer does not have source.

Context: The pattern applies to C++ and potentially to other object-oriented programming languages. The decision of which promotion to apply is made at compile time. The context is inadequate for the compiler to apply built-in translation rules, as would be possible for conversion between built-in types, or between a derived class and one of its base classes.

Forces:

- The implementation of promotion from an object of one type to an object of another type is usually coupled to the implementation of both types.
- The C++ language lets the programmer associate such an implementation with only one of the participating types.
- The type containing the conversion implementation must be a class object type, since the programmer cannot redefine the implementation of built-in types or types exported by object-code-only libraries.
- Two language mechanisms support user-defined conversions: constructors and conversion operators. Individually, each is an equally suitable solution in some circumstances, but use of both leads to an irreconcilable ambiguity.

Solution:

- A program should promote a class object type to a built-in type or a type exported from a library using a member conversion operator:

```
class RationalNumber {
public:
    operator float() const;
    . . . .
};
```

- A program should use constructors for all other promotions:

```
class Complex {
public:
    Complex(const RationalNumber&);
    Complex(double);
    . . . .
};
```

Force Resolution:

- Coupling between types (and, in general, `friend` relationships) is still necessary between types; the force is resolved only to the extent that the conversion is firmly associated with a single type. The pattern does guarantee that the type bearing the conversion is always a class object type, however.
- The pattern avoids most conversion ambiguities. An additional pattern must deal with the case:

```
struct S {
        operator int() const;
        operator float() const;
};

void f( int );
void f( float );
```

```
main() {
        S s;
        f( s ); // error: ambiguous call: f ( struct S )
}
```

Design rationale: A given type cannot know about every (more general) type in the universe whose instances might be created as generalizations of itself; the onus is on the type of the newly created object to understand its own initialization parameters. Primitive types form an exceptional sub-pattern because their semantics are built into the compiler for efficiency, and their semantics are not as generally subject to change as for user-defined types.

REFERENCES

1. Church, Kenneth Ward, and Jonathan Isaac Helfman. ‘‘Dotplot: A Program for Exploring Self-Similarity in Millions of Lines for Text and Code.’’ American Statistical Association, Institute of Mathematical Statistics and Interface Foundations of North America, Vol. 2, No. 2, pp. 153-174.
2. Gamma, Erich. ‘‘Design Patterns — Abstraction and Reuse of Object-Oriented Designs.’’ in *Proceedings of the European Conference on Object-Oriented Programming*, Oscar Nierstrasz, ed. Berlin: Springer-Verlag, ©1993.
3. Coplien, James O. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley, ©1992.
4. Beck, Kent. ‘‘Patterns and Software Development.’’ *Dr. Dobb’s Journal*, Vol. 19, No. 2, pp. 18-23.
5. Booch, Grady. ‘‘Patterns.’’ *Object Magazine*, Vol. 3, No. 2, SIGS Publications, New York, NY.
6. Gabriel, Richard. Pattern Languages. *Journal of Object-Oriented Programming*, Vol. 6, No. 2, p. 14, Jan. 1994. SIGS Publications, New York, NY.
7. Johnson, Ralph E. ‘‘Documenting Frameworks using Patterns.’’ *ACM SIGPLAN Notices*, Vol. 27, No. 10, Oct. 1992, ff. 63.