Patterns, Classes, and Derivation

Joseph K. Cross
Lockheed Martin Tactical Systems
P.O. Box 64525, M.S. U2N27
St. Paul, MN 55164-0525, USA
(651) 456-7316
joseph.k.cross@lmco.com

Abstract

This paper explores the analogies between patterns and classes, and in particular the parallel in the patterns world to class derivation. These considerations enable a new approach to the design of the interface between application software and its supporting infrastructure. This new approach may provide large cost savings, especially in the maintenance of systems in which the applications software is intended to remain stable for far longer than the lifetime of its supporting infrastructure components.

1 Introduction

The author of a software pattern is today faced with a difficult choice: how much problem detail and solution specificity to include. If too much detail is specified, it become unlikely that the pattern will be actually reused. If too little is specified, then the pattern may be too vague to be useful. Pattern authors make reasonable middle-ground choices, and users of patterns are often forced to cite their use of a pattern with qualification, as in, "We use the Wrapper Façade pattern. Sort of."

The response presented in this paper considers the hierarchy of patterns on varying levels of specificity. This hierarchy exhibits some analogies to the derivation hierarchy of classes. When these analogies are explored, a new approach to the design of the interface between application software and its supporting infrastructure presents itself. This new approach may provide large cost savings, especially in the maintenance of systems in which the applications software is intended to remain stable for far longer than the lifetime of its supporting infrastructure components.

The first section of this paper defines the software design problem we are addressing. The second section explores pattern derivation and its analogies to class derivation. The third section describes the options that the preceding considerations offer to the design of interfaces between applications and their (necessarily evolving) infrastructures.

2 Problem Definition

2.1 Maintaining Long-Lived Systems

New and planned commercial and military distributed real-time and embedded (DRE) systems take input from many remote sensors, and provide geographically-dispersed operators with (1) the ability to interact with the collected information and (2) to control remote effectors. In circumstances where the presence of a human in the loop is either too expensive or too slow, these systems must respond autonomously and flexibly to unanticipated combinations of events at run-time. Moreover, these systems are increasingly networked to form long-lived "systems of systems" that must run unobtrusively and autonomously, shielding operators from unnecessary details, while simultaneously communicating and responding to mission-critical information at heretofore infeasible rates.

It is possible in theory to develop these types of complex DRE systems from scratch. However, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so in practice. The proportion of DRE systems made up of "commercial-off-the-shelf" (COTS) hardware and middleware has therefore increased dramatically, which helps reduce the initial non-recurring cost of these systems. In the context of this paper, middleware is software that functionally bridges the gap between application programs and the lower-level underlying operating systems and network protocol stacks.

Therefore, middleware provides services to the applications whose qualities of service – such as latency of message delivery – are critical to DRE systems [Quorum].

In many commercial application domains, such as e-commerce or consumer electronics, application software evolves faster than middleware. As a result, most mainstream COTS middleware products focus on presenting a powerful set of services that are attractive to new applications, so that existing applications can evolve freely. Long-lived DRE systems, however, often have the reverse problem, i.e., how to write applications that can remain stable, while permitting and exploiting the relatively rapid evolution of the underlying infrastructure.

In the DRE domain, applications are often maintained over long periods, e.g., 20 to 30 years. When combined with free-market economics, the inevitable evolution of the infrastructure has far-reaching technical consequences. For years, software designers hoped that while the infrastructure components might change, at least the functional interface between them and the applications might remain invariant. Now, the rate of infrastructure evolution is so rapid that even this slim reed is slipping from our hands. This fact is gaining increasing recognition:

> The middleware environments that are most visible today are CORBA, Enterprise JavaBeans, message-oriented middleware, XML/SOAP, COM+ and .NET. However, over the past decade or so, the middleware landscape has continually shifted. For years we've assumed that a clear winner will emerge and stabilize this state of flux, but it's time to admit what we've all suspected: The sequence has no end! And, in spite of the advantages (sometimes real, sometimes imagined) of the latest middleware platform, migration is expensive and disruptive. (We know an industry standards group that, having migrated their standard

infrastructure twice already, is now moving from their latest platform *du jour* to XML.) [OMG-MDA]

2.2 Specifying and Applying Patterns

The software community has become skillful at specifying patterns [GOF, POSA1, POSA2]. We have difficulty in cleanly applying existing patterns. It is common to hear a developer say, "We're using the Foobar pattern. Sort of."

For example, the Wrapper Façade pattern "encapsulates the functions and data provided by existing non-object-oriented APIs within … object-oriented class interfaces." [POSA2] This is, without doubt, a righteous and useful pattern. However, there are common situations in which something similar to a Wrapper Façade pattern is called for, but in which the Wrapper Façade does not perfectly apply. Similar patterns, such as the Adapter [GOF], might be referenced, but with similar tradeoffs between precision of specification and accuracy of fit to its application.

In addition, there is danger of a proliferation of similar but distinct patterns, such as the Wrapper Façade, Adapter, and Proxy [POSA1] (of which the last has seven "variants," such as the Cache Proxy). If such a proliferation of patterns continues, we are certain to develop a morass of numerous overlapping but distinct patterns, and this will act against the very goals of developing a common vocabulary and facilitating getting up to speed in software architecture that the use of patterns is intended to serve.

3 Patterns and Classes

This section explores the analogy between patterns and classes. The conclusion is that while the analogy is by no means complete, its consideration raises some important possibilities.

3.1 Derivation of Patterns and Classes

A class `Lo_c` is derived from a class `Hi_c` if, roughly, `Lo_c` is more specific than `Hi_c`.

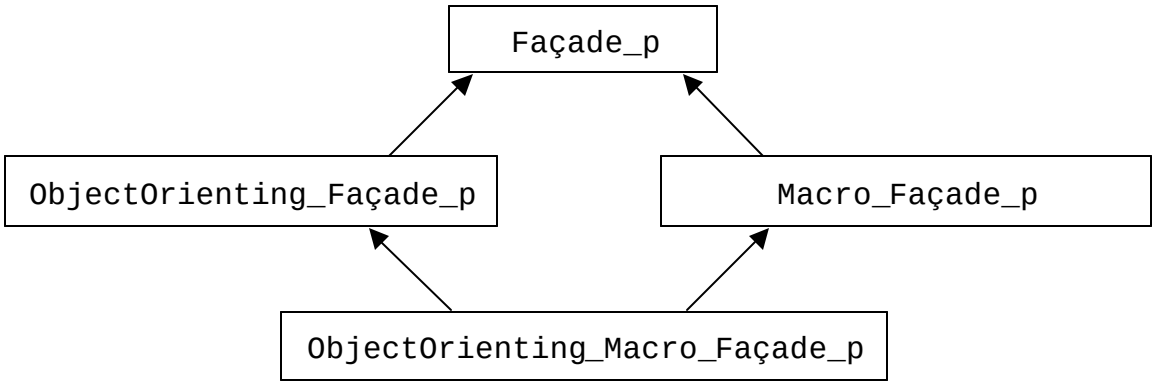
Let us say that a pattern `Lo_p` is derived from a pattern `Hi_p` either `Lo_p` solves a subset of the problems that `Hi_p` solves, or the solution in `Lo_p` is more specific than is the solution of `Hi_p`. In addition, the context of `Lo_p` should be a (not necessarily proper) subset of the context of `Hi_p`.

Consider a simple example from building architecture. "Courtyard" is a pattern; "OpenCourtyard" and "RoofedCourtyard" are patterns derived from it. (Note that while it's technically correct to say that all three *Courtyard patterns are variants of one another, it is pointlessly inexpressive – 'variant' is a potentially symmetric relationship, while the derivation relationship expresses the actual one-way refinement relations.)

For an example from software engineering, let us consider the "`Façade_p`" pattern that solves the problem of providing a different (presumably better) interface to a pre-existing set of data and/or operations.

We can derive from an `ObjectOrienting_Façade_p` pattern that solves the problem of providing an object-oriented interface to pre-existing non-object-oriented facilities. Hence `ObjectOrienting_Façade_p` solves a more specific problem than does `Façade_p`.

Additionally, we can derive from `Façade_p` a `Macro_Façade_p` pattern that implements its façade by means of macros that expand into references to the pre-existing facilities. Hence `Macro_Façade_p` specifies a more specific solution than does `Façade_p`.



The reader will note the intimation of multiple inheritance.

3.2 Instances and Values

If a pattern is analogous to a class, then one use of a pattern in a system is analogous to an instance of a class – i.e., an object.

Returning to the Courtyard analogy, each physical courtyard in a building would be an instance of the Courtyard pattern.

Instances of a class can contain values, and the class definition specifies the names and types of those values. Analogously, it is desirable for patterns to specify names and types of values that its instances contain.

For example, consider the (highly abstract) pattern `Connector_p`, which solves the problem of communicating data and/or control between distinct (abstract) processes.

From `Connector_p`, are derived patterns such as `Stream_p`, `ClientServer_p`, and `Event_p`. Although `Connector_p` is too abstract to support any values, its sub-patterns can specify values, such as `last_event_delivery_latency` for the event pattern, and `bit_error_rate` for the stream pattern. Then every instance of these patterns in a system would have the values[1] specified by its pattern.

3.3 Methods

Classes specify methods. The analogy in patterns is the operations that are permitted or required in utilizing an instance of the pattern.

For example, the `ClientServer_p` pattern might expose methods such as `obtainProxy()`, `oneWayCall()` and `roundTripCall()`.

The best mechanism for specifying pattern methods, and specifying correctness conditions on their invocation, is a subject for future research; perhaps the OMG's Object Constraint Language (OCL) [OCL] will provide inspiration.

3.4 Overriding, Hiding, and Visibility Control

Classes have (sometimes elaborate) rules governing overriding, hiding, and visibility control of their members in a class hierarchy. Your humble scribe sees no use for the analogous concepts for patterns.

4 Impacts on Software Productivity

As noted above, the infrastructure that underlies long-lived software applications will change during the lifetime of the application, and that change will probably entail a change to the interface between the application and the infrastructure. Rather than focusing on a fixed infrastructure and attempting to define its interface in a way that might help maintain its invariance as long as possible (e.g., POSIX profiles), this paper proposes addressing the interface from the viewpoint of the application.

4.1 Visions of Hierarchies

Suppose that suitable organization maintains and publishes a set of software patterns, defined as described above – with derivation relations and attribute definitions. Then the following benefits would accrue:

- A clear organization of recognized patterns would be promulgated. Relations other than inheritance, such as uses, can and should be supported.
- A person looking for an off-the-shelf solution to a design problem could follow the problem-derivation links (as opposed to the solution derivation links) down the hierarchy to match a relevant piece of the existing architecture.
- A person considering defining a (possibly) new pattern would have a central source to inspect; hence the proliferation of arbitrarily overlapping patterns would be inhibited
- The act of clearing defining attribute values (such as the `last_event_delivery_latency` of an event service) would provide a major benefit to developers and evaluators of such services.
- Providers of services could reference the pattern that their service supports.

[1] Note that these values are not required to be scalars.

- Developers could cite patterns as requirements on infrastructure components. E.g., "We need an implementation of `ClientServer_p` on <some platform> with a `last_event_delivery_latency` of at most 1 millisecond."
- In addition to specifying patterns with values as procurement requirements, an application program can specify a pattern (such as `Event_p`, and specify values for each instance value as system-build time, load-time, or run-time requirements. This amounts to specifying a functional interface and its associated qualities of service.
- It is possible that higher level patterns will enable better software architecture, by enabling one to design an architecture of a system by constructing a number of higher-level patterns that abstract away from many details, then plugging in the right derived patterns, which then get implemented.

4.2 Promising Direction: The Pattern Hierarchy and Declarative Programming

It has been suggested that we have enough procedural languages, and that the most promising direction for programming to evolve is toward declarative programming: the programmer states what is to be achieved, more or less as data, and the translation into processing steps is performed automatically [2].

If we use a procedural language, there remains a fair amount of work to be done after requesting an implementation of a pattern (such as `ClientServer_p`) and before the resulting communication capability is usable. E.g., the programmer must decide what sort of POA will hold the server, and when to create the proxies.

It seems that both the number of such decisions and the number of alternative choices for each are manageably small (e.g., 1 to 10, not hundreds). For example, the alternative choices for the decision about when to create proxies might include "upon initialization" and "hard to describe – give me a parameterless function called `createProxies()`."

If these decisions and alternative choices could be included with the pattern definitions, as are the attribute values described above, then we would be well along the path to a purely declarative implementation of at least infrastructure services.

5 Acknowledgements

The author is pleased to acknowledge the inspiration of Joe Loyall, of BBN, through his discussion of the connector pattern and its sub-patterns (although not exactly in these terms).

This work was funded in part by DARPA/ITO contract number F33615-01-C-1847, under direction of Dr. Douglas Schmidt.

Neither of the acknowledgees has endorsed the viewpoints presented herein.

Bibliography

[GOF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[OCL] http://cgi.omg.org/cgi-bin/doc?ad/00-09-03.pdf

[OMG-MDA] R. Soley and the OMG Staff Strategy Group, "Model Driven Architecture", Object Management Group White Paper, Draft 3.2, November 27, 2000.

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley and Sons, 1996.

[POSA2] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture,: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[Quorum] The Quorum Project, DARPA Information Technology Office.
http://www.darpa.mil/ito/research/quorum/index.html

[2] This is a much more abstract definition than that used by the artificial intelligence and functional programming communities. Our goals for declarative programming are more modest.