AT&T Bell Laboratories

Curiously Recurring Template Patterns

James O. Coplien
Software Production Research Dept., AT&T Bell Laboratories
1000 East Warrenville Rd., Naperville, IL 60566 USA
(708) 713-5384
cope@research.att.com

Abstract

Software design patterns capture recurring good practice in a domain. Many good patterns enjoy independent discovery by different people at different times. I have now seen three distinct uses of what I thought was an obscure template pattern. Each use arose in a unique domain; one instance was outside the C++ community. We can capture the technique in a pattern that solves a problem of factoring circular dependencies in code structure and behavior. The pattern form makes an otherwise opaque framework more accessible.

My earlier columns have started with a discussion *about* patterns, followed by an exemplary pattern. This month, I'm weaving the two together to explore how we find and record patterns. This is a story of how I came to understand an interesting family of patterns that have been independently invented by a telecommunications software designer at AT&T, by some scientific programmers at IBM, and by a computer linguist at Oregon State University. I'll use this pattern to explore the way we think about software and abstraction and to introduce multi-paradigm design, a stepping-stone between object-oriented design and patterns. I feel like a cultural anthropologist who has discovered a grand unified theory of, well, something, and invite you to join me as I re-live the quest for the underlying abstractions of this architectural construct.

The Genesis of an Idea

Four or five years ago, a good friend and co-worker of mine, Lorraine Juhl, showed me a piece of code that has intrigued me to this day. The code combined templates and inheritance in a powerful pattern that I've seen at least three other people invent independently. What was remarkable about this pattern was that it used templates so powerfully so early after their introduction into the language. The code implemented a finite-state machine. FSMs are a big part of our business in telecommunications (I've often said that everyone discovers FSMs at least once during their career), so a robust FSM abstraction was a big deal. Programmers defined their contribution to the FSM like this:

```
class myFSM: public BaseFSM
  <myFSM,
  /*State=*/     char,
  /*Stimulus=*/  char> {
public:
  void x1(char);
  void x2(char);
  void init() {
    addState(1);
    addState(2);
    addTransition(EOF,1,2,&myFSM::x1);
    . . . .
  }
};
```

What's that again? The class `myFSM` is derived from a base class that is instantiated from a template, but the derived class is passed as a parameter to the template instantiation. Why? Let's look at the template used to build the base class:

```
template <class M, class State,
  class Stimulus>
class BaseFSM {
public:
  virtual void addState(State) = 0;
  virtual void addTransition(
    Stimulus, State, State,
    void (M::*)(Stimulus)) = 0;
  virtual void fire(Stimulus) = 0;
};
```

We'll talk more about the base class later; for now, think of it as an abstract base class in the conventional sense, one that provides an interface to a family of derived class state machines.

This framework is completed by a template that captures the FSM state and the common machinery for a state machine. The application programmer uses this template to create the FSM object:

```
template <class UserMachine, class
  State, class Stimulus>
class FSM: public UserMachine {
public:
  FSM() {  init(); }
  virtual void addState(State);
  virtual void addTransition(
    Stimulus, State from, State to,
    void (UserMachine::*)(Stimulus));
  virtual void fire(Stimulus);
private:
  State nstates,*states,currentState;
  Map<Stimulus,
    void(UserMachine::*)(Stimulus)>
    *transitionMap;
};

int main() {
     FSM<myFSM,/*State=*/char,
    /*Stimulus=*/char> myMachine;
     . . . .
}
```

The pattern resurfaces

It was several years later, and I had almost forgotten about this pattern, when Addison-Wesley asked me to review a book manuscript draft. The book was to become *Scientific and Engineering C++* by John Barton and Lee Nackman of IBM).[1] It contained lots of code that looked like this:

```
template<class DerivedType>
class EquivalentCategory {
  friend Boolean operator==(
    const DerivedType &lhs,
    const DerivedType &rhs) {
    return lhs.equivalentTo(rhs);
  }
  friend Boolean operator!=(
    const DerivedType &lhs,
    const DerivedType &rhs) {
      return ! lhs.equivalentTo(rhs);
  }
};

class Apple:
  public EquivalentCategory<Apple> {
public:
  Apple(int n): a(n) { }
  virtual Boolean equivalentTo(
    const Apple &an_apple) const {
      return a == an_apple.a;
  }
```

1. The book is published by Addison-Wesley, ©1994, with ISBN 0-201-53393-6. These are the same guys who write "Scientific & Engineering C++" here in the *C++ Report*, a column that's a must-read whether you do scientific and engineering programming or not.

```
private:
  int a;
};
```

Though it's unimportant for our purposes here, they go on to define a class `Orange` and, well, you can guess the rest. Sure enough, this was the same pattern that Lorraine had discovered a few years earlier. The Barton and Nackman manuscript raised the FSM "trick" to new heights by regularizing it and using it in many different ways. They had recognized a class of problems that called for this solution again and again.

Barton and Nackman are interested in scientific and engineering programming. They dwell on abstractions like numbers, vectors and matrices, groups and other structure categories, etc. They are also interested in efficiency, which is one reason the book leverages templates as much as it does (the term "template" takes a full column in the book's index, almost the same amount as for the term "type"). This pattern is important to them because common category properties (what we usually use inheritance for) and common code structure (what we usually use templates for) can be factored out of many of their designs. Used together, templates and inheritance support a design construct that we can capture as a pattern. The problem is:

- Factoring circular dependencies in code structure and behavior.

The context is a language that supports inheritance and templates, used with an object-oriented focus. The forces are interesting:

- We want a single base class that ties together the semantics of equality and inequality in the most general sense, so that one is guaranteed to be the logical negation of the other: in other words, the implementation should be type-restricted;
- Classes exhibiting the behaviors of an equivalence category should be derived from this class: in general, the derived class behaviors are derived from the base behaviors;
- The base class must know the derived class type, so it can dispatch the computation of equality to its derived class part: in other words, the implementation is type-dependent;
- The parameter list signature of the derived class must be compliant with that of the base class, so if the base class wants to call a derived class function through a base class virtual function interface, the function's interface must be invariant with respect to the derived type;
- The derived class member function that computes equality has a parameter list that is sensitive to the derived type;
- The derived classes of interest cover a broad range of otherwise unrelated types.

The type restriction of the equality equivalence class and type dependency in the derived class implementations leads to a circular dependency between the two. The solution:

- "To obtain both type-restricted and type-dependent functions, we combine the features of implementation and template categories. Specifically, we create an implementation base class, `EquivalentCategory<DerivedType>`, parameterized by the type of the derived class." (Barton and Nackman, p. 352)

That is, we encode the circular dependency directly using inheritance in one direction and templates in the other. The first time I saw the Barton and Nackman code, I told them that their compiler was broken and that this shouldn't compile: a compiler could never reduce the dependency of a base class on its derived class. They very politely cited chapter and verse in the ARM to show where I was wrong. C++ allows such a dependency as long as the *structure* of the base class doesn't depend on a derived class type parameter.

Note that this is an outsider's view of the solution; I'm looking forward to an opportunity for the authors to set the record straight with me. I also defer to them the honor of naming this pattern.

Third time's the charm

Most recently, I attended Tim Budd's OOPSLA tutorial on multi-paradigm programming. I had been looking forward to this tutorial for some time, as I've actively been researching how multi-paradigm design techniques might be used to regularize hybrid designs. (Tim's book, "Multiparadigm Programming in Leda," will soon be on the market as an Addison-Wesley book). It turned out that Tim's talk didn't focus too much on design, but provided some fascinating lessons in programming language. He illustrated multi-paradigm programming using the Leda language that he and his students have developed at Oregon State University. I looked through his extensive notes after the tutorial (his notes include generous excerpts from his forthcoming book) and found the following code snippets:

```
class ordered [T : ordered] of
  equality[T];
  . . . .
end;

class integer of ordered[integer];
  function asString ()->string
    begin . . . . end
  function equals (arg : integer)
    -> boolean;
    begin . . . . end;
end;

class string of ordered[string];
  function asString ()->string;
    begin return self; end;
  function equals (arg : integer)
    -> boolean;
    begin . . . . end;
end;
```

Deja vu all over again, and it's not even C++. What was particularly remarkable about this example is that it relates to equality tests, as does one of the earliest uses of this pattern in the Barton and Nackman book. By the time I discovered the third, independently derived example of this style, I became convinced that this is a pattern, not just an isolated trick. Since then, I've encountered colleagues using this same pattern for finite state machines (Ralph Kolewe at Eridani in Ontario), and in a human-machine interface library (Paul Lucas at AT&T).

If this is a pattern, it must solve the same problem—at some level—in all these examples. It must derive from a single, common set of underlying forces. So what is going on here?

Let's look at the FSM example in Figure 1. Notice that this class diagram shows the same circular dependency we described for the Barton and Nackman example. This pattern attempts to solve a problem:

- Separate the common FSM mechanisms into a library of generic abstractions, so the programmer need write code only for the FSM behaviors (or policies) that are of interest to the application.

(An exercise left to the reader: compare this to the problem statement for the Barton and Nackman example.) Let's jump ahead to the rationale. We have broken the design into three parts. `BaseFSM` is a template that generates a family of abstract base class interfaces. More interestingly, class `FSM` contains all the state machine machinery: it is a generic state machine. What *is* a generic state machine? Its interface defines responsibilities for defining the semantics of a particular FSM: It can learn new states (`addState`) and new transition arcs between the states (`addTransition`). The `fire` member function causes the machine to cycle, processing the input provided as the member function argument. The `FSM` template also provides a generic implementation for all state machines: a state count, a current state, a vector of

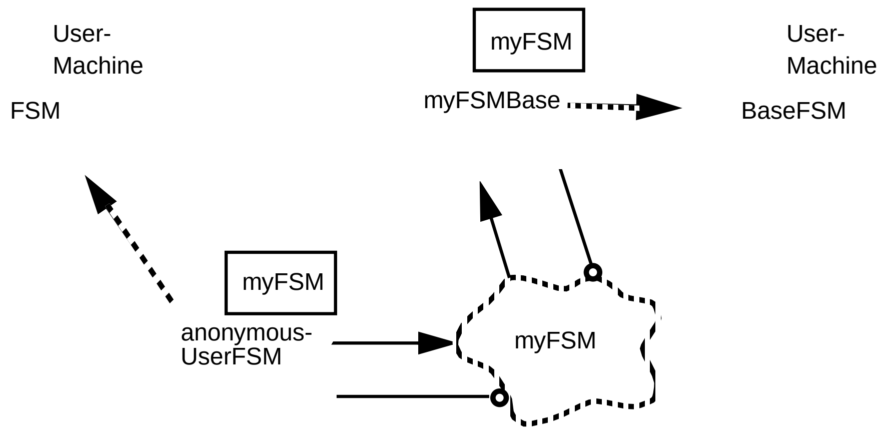

Figure 1. Some abstractions that make a Finite State Machine

states, and a map from the current state to the transition function for the current state. (I have simplified the design for the sake of presentation; a production version would probably map a 2-tuple comprising the current state and an input, to a transition function.)

Programmers can develop their own classes like `myFSM` that capture the behavior of a state machine (its states and transitions) without worrying about the mechanics. The `BaseFSM` class provides a framework for the programmer to fill out: the class `myFSM` provides a home for an obligatory `init` function to set up the machine, as well as a home for the functions that do the work of the machine at run time. All FSMs share the same data structure; usually, we would factor those into a common base class. Here, we factor the common data structures into a common *derived* class! Why? Because:

- •The FSM should initialize itself without the user explicitly calling `init`;
- •A constructor should orchestrate the initialization;
- •Because `init` references virtual functions, it should not be called until after the `myFSM` constructor completes (so that virtual function dispatching works for `myFSM` functions);
- •Therefore, `init` cannot be called from the constructor for `myFSM` or any of its base classes;
- •We want to create only as many classes as necessary (Occam’s razor).

The solution is:

- •Call `init` from a derived class. The `FSM` template can generate a derived class that takes `myFSM` as a base class, with a constructor that calls `myFSM::init`. Capture the common data structure in the same class to avoid multiplying classes unnecessarily.

This is a pattern, at least in the sense that it documents the architect’s rationale for this rather involved data structure. This pattern creates a new context where we face a new problem:

- •We want the `fire` member function to be accessible through the base class interface.

However,

- •Common member functions should be declared in the base class, usually as virtual functions;
- •In this context (defined by the output of the previous pattern), the base class is the one provided by the user;
- •Declaring `fire` in `myFSM` is make-work that a programmer might easily forget to do.

Therefore:

- Define a new base class, `BaseFSM`, that declares `fire` as a pure virtual function. Users are obligated to derive their state machines from `BaseFSM`. Calls to `fire` through a generic base class pointer will be dispatched around class `myFSM` to the derived class generated by the FSM template. Template type parameters tie the three classes together.

These patterns together form a small pattern language that documents this `FSM` framework. The framework code is hopelessly opaque by itself—as the author of the code, even *I* have trouble explaining the implementation from the code alone. The patterns organize my thoughts about the design so I can understand how to use the framework, and perhaps teach others how to use it. I suspect we'll see a proliferation of patterns to document existing and emerging frameworks.

Pattern, idiom, or coincidence?

Is this trick with templates just a low-level idiom specific to C++? I suspect that a broader design pattern lurks here; we already have one non-C++ data point in Tim Budd's language. We will find this pattern only in languages that have both templates and inheritance. That is why we haven't seen the pattern in Ada®, which lacks inheritance (I'll bet we see the pattern in Ada 9X); that's why we don't see it in Smalltalk, which lacks the compile-time binding semantics of templates. These language properties become a crucial component of the pattern context, but they don't limit the pattern to C++ alone.

The FSM pattern is a variant of the technique used by Barton, Nackman, and Budd. Do all these pattern belong in the same pattern language? Does the FSM pattern generalize as much as the Barton/Nackman/Budd technique, or does it just cover an obscure corner of design? Time will tell. As we gain experience with templates in practice, and as more people come up to speed with the Barton and Nackman material, we'll know better how to shape and assemble C++ template pattern languages.

We can broaden the question even further. Budd's code arose from a culture interested in multi-paradigm design (though he doesn't count templates as a real paradigm). I, too have been exploring multi-paradigm design techniques for C++ (see my paper, "Multi-Paradigm Design for C++" in the proceedings of the 1994 SIGS OOP/C++ World in Munich, January, 1994). My technique is based on commonality and variability analyses pioneered by my coworker David Weiss and his colleagues. Multi-paradigm design has steps and notations that point the way to structures such as those discussed here. Curiously, we find that Barton and Nackman have made commonality and variability part of their vocabulary as they describe these designs. Might there be a way to regularize such designs, and to provide a way of thinking about design that makes them more intuitive? If so, patterns might be overkill. Again, time will tell.

Signposts

John Vlissides (co-author of "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994, now available at bookstores carrying professional computer science texts) will soon be adding his pattern perspective to the C++ Report. Keep an eye out for his column just a few pages from here, starting in the March/April issue of C++ Report.

Start readying your submissions for PLoP/95.

Jim Coplien is a Member of the Software Production Research Department at AT&T Bell Laboratories in Naperville, Illinois, and is a member of the Hillside Generative Patterns group. He can be reached at cope@research.att.com. To subscribe to the patterns electronic mail discussion group, send

mail to listserver@cs.uiuc.edu with a message body of “subscribe patterns-discussion your@internet.address”.