Position Paper

Dagstuhl Seminar 9508

Reverse Architecture

John Vlissides

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA

vlis@watson.ibm.com

Abstract

This paper articulates the need to assimilate and exposit software architecture effectively. The term "reverse architecture" describes this process. I use design pattern development as an example. I present lessons learned in the development of numerous design patterns to illustrate one approach to reverse architecture.

1 Introduction

Reverse engineering traditionally refers to the analysis of manufactured artifacts for the purpose of reproducing all or part of them. It's a worthwhile activity because it's generally cheaper to reverse engineer something than to develop it from scratch. Reverse engineering capitalizes on the abilities of experienced engineers and their successful efforts. The concept originated in hardware circles, where it's often associated with thievery. But it also denotes the time-honored and legitimate practice of analyzing finished products (your own as well as your competitor's) for the purpose of improving on them.

Reverse engineering concepts have also found their way into software circles. In software, however, reverse engineering techniques are used for program understanding rather than for duplicating the software (which is easy enough to do already!) [3]. Hence the "engineering" moniker is less appropriate for software than for hardware. The software focus is really architectural—on recovering *design* rather than implementation.

A term like "reverse *architecting*" would be more accurate in the software context. The term "reverse architecture" would apply to the art or science behind the activity. Reverse architect*ure* involves reverse architect*ing*. When applied to software, "reverse engineering" refers to analyzing a particular software system in an effort to recover its design. In contrast, "reverse architecture" refers to analyzing many software systems in an effort to recover *recurring* designs and the rationales behind them.

"Reverse architecture" is really just my tongue-in-cheek term for what Erich Gamma, Richard Helm, Ralph Johnson, and I did as we wrote *Design Patterns: Elements of Reusable Object-Oriented Software* [4]. "Design patterns" capture expertise in building object-oriented software. A design pattern describes a solution to a recurring design problem in a consistent and general way. A particular solution by itself is not enough. You need a deeper understanding to tailor the solution to your variant of the problem. Hence a design pattern also explains the applicability, trade-offs, and consequences of the solution. It gives you the rationale behind the solution, not just a pat answer. A design pattern also shows you how to implement the solution in standard object-oriented programming languages like C++ and Smalltalk. *Design Patterns* contains a set of 23 design patterns distilled from our experience with numerous object-oriented systems.

Portions of this paper are adapted from *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides; copyright © 1995 by Addison-Wesley Publishing Company. Used by permission.

This paper is my attempt to relate some of what we learned about reverse architecture in the course of developing these design patterns. Our experience should help others hone their reverse architectural abilities far faster than we did. Hopefully that will translate into a better understanding and dissemination of good software architecture.

2 An Example: The Strategy Pattern

Before we look at some of the reverse architectural principles we have learned, I should show you an example of a design pattern. Let's take a look at Strategy, one of the patterns from the *Design Patterns* book. Although we don't have room for the pattern in its entirety, the excerpt below will give you a good idea of what design patterns are all about.

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also Known As

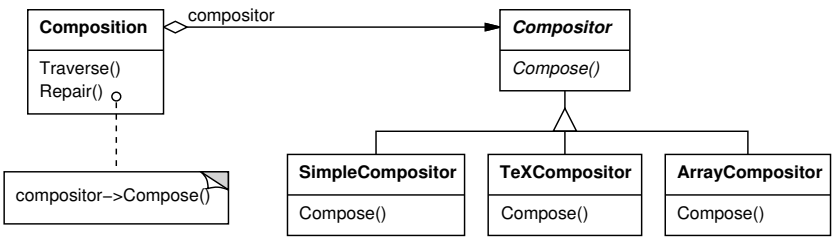
Policy

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that's encapsulated in this way is called a **strategy**.



Suppose a Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class Composition. Instead, they are implemented separately by subclasses of the abstract Compositor class. Compositor subclasses implement different strategies:

- **SimpleCompositor** implements a simple strategy that determines linebreaks one at a time.
- **TeXCompositor** implements the TEX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.

- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

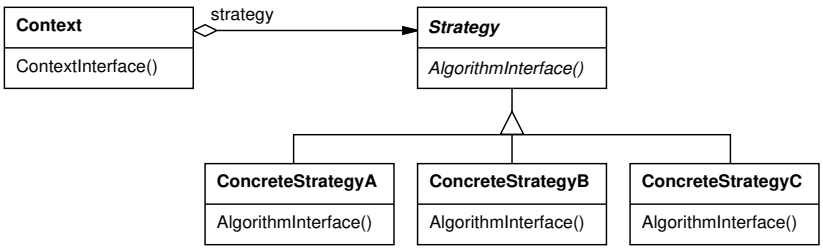
A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

- 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.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

Here are some of the benefits and drawbacks of the Strategy pattern:

1. *Families of related algorithms.* Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
2. *An alternative to subclassing.* Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. *Clients must be aware of different Strategies.* The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients.

Implementation

Consider the following implementation issues:

1. *Defining the Strategy and Context interfaces.* The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.

   One approach is to have Context pass data in parameters to Strategy operations—in other words, take the data to the strategy. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.

   Another technique has a context pass *itself* as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. Either way, the strategy can request exactly what it needs. But now Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.

   The needs of the particular algorithm and its data requirements will determine the best technique.
2. *Strategies as template parameters.* In C++ templates can be used to configure a class with a strategy. This technique is only applicable if (1) the Strategy can be selected at compile-time, and (2) it does not have to be changed at run-time. In this case, the class to be configured (e.g., `Context`) is defined as a template class that has a `Strategy` class as a parameter:

   ```
   template <class AStrategy>
   class Context {
       void Operation() { theStrategy.DoAlgorithm(); }
       // ...
   private:
       AStrategy theStrategy;
   };
   ```

   The class is then configured with a `Strategy` class when it's instantiated:

   ```
   class MyStrategy {
   public:
       void DoAlgorithm();
   };

   Context<MyStrategy> aContext;
   ```

   With templates, there's no need to define an abstract class that defines the interface to the `Strategy`. Using `Strategy` as a template parameter also lets you bind a `Strategy` to its `Context` statically, which can increase efficiency.
3. *Making Strategy objects optional.* The Context class may be simplified if it's meaningful *not* to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If

there is one, then Context uses it normally. If there isn't a strategy, then Context carries out default behavior. The benefit of this approach is that clients don't have to deal with Strategy objects at all *unless* they don't like the default behavior.

Sample Code

We'll give the high-level code for the Motivation example, which is based on the implementation of Composition and Compositor classes in the InterViews user interface toolkit [7].

The `Composition` class maintains a collection of `Component` instances, which represent text and graphical elements in a document. A composition arranges component objects into lines using an instance of a `Compositor` subclass, which encapsulates a linebreaking strategy. Each component has an associated natural size, stretchability, and shrinkability. The stretchability defines how much the component can grow beyond its natural size; shrinkability is how much it can shrink. The composition passes these values to a compositor, which uses them to determine the best location for linebreaks.

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;     // the list of components
    int _componentCount;        // the number of components
    int _lineWidth;             // the Composition's line width
    int* _lineBreaks;           // the position of linebreaks
                                // in components
    int _lineCount;             // the number of lines
};
```

When a new layout is required, the composition asks its compositor to determine where to place linebreaks. The composition passes the compositor three arrays that define natural sizes, stretchabilities, and shrinkabilities of the components. It also passes the number of components, how wide the line is, and an array that the compositor fills with the position of each linebreak. The compositor returns the number of calculated breaks.

The `Compositor` interface lets the composition pass the compositor all the information it needs. This is an example of "taking the data to the strategy":

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

Note that `Compositor` is an abstract class. Concrete subclasses define specific linebreaking strategies.

The composition calls its compositor in its `Repair` operation. `Repair` first initializes arrays with the natural size, stretchability, and shrinkability of each component (the details of which we omit for brevity). Then it calls on the compositor to obtain the linebreaks and finally lays out the components according to the breaks (also omitted):

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...
```

```
    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

Now let's look at the `Compositor` subclasses. `SimpleCompositor` examines components a line at a time to determine where breaks should go. `TeXCompositor` uses a more global strategy. It examines a *paragraph* at a time, taking into account the components' size and stretchability. It also tries to give an even "color" to the paragraph by minimizing the whitespace between components. `ArrayCompositor` breaks the components into lines at regular intervals.

These classes don't use all the information passed in `Compose`. `SimpleCompositor` ignores the stretchability of the components, taking only their natural widths into account. `TeXCompositor` uses all the information passed to it, whereas `ArrayCompositor` ignores everything.

To instantiate `Composition`, you pass it the compositor you want to use:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

`Compositor`'s interface is carefully designed to support all layout algorithms that subclasses might implement. You don't want to have to change this interface with every new subclass, because that will require changing existing subclasses. In general, the Strategy and Context interfaces determine how well the pattern achieves its intent.

Known Uses

Both ET++ [9] and InterViews use strategies to encapsulate different linebreaking algorithms as we've described.

In the RTL System for compiler code optimization [6], strategies define different register allocation schemes (RegisterAllocator) and instruction set scheduling policies (RISCscheduler, CISCscheduler). This provides flexibility in targeting the optimizer for different machine architectures.

The Booch components [2] use strategies as template arguments. The Booch collection classes support three different kinds of memory allocation strategies: managed (allocation out of a pool), controlled (allocations/deallocations are protected by locks), and unmanaged (the normal memory allocator). These strategies are passed as template arguments to a collection class when it's instantiated. For example, an UnboundedCollection that uses the unmanaged strategy is instantiated as `UnboundedCollection<MyItemType*, Unmanaged>`.

Related Patterns

Flyweight: Strategy objects often make good flyweights.

3 Discovering Patterns in Software

Design patterns such as Strategy document a recurring piece of object-oriented design in a way that lets you understand and apply it in your own context. Design patterns also let you talk about design at a higher level than classes and objects. You can say things like, "Let's use a Strategy here," rather than, "Let's encapsulate our algorithms in separate classes and make those subclasses of a common base class that defines a common interface for all algorithms so that...." As a result, you can communicate design information more quickly and accurately—a boon to both the design process and to documentation.

But where did the Strategy pattern come from? How did we find it? What made us recognize its importance?

The single most important activity in reverse architecture is reflection. You must take time off periodically to reflect on what you've done. Think about the systems you've built, the problems you've had, and how you solved (or didn't solve) them.

Such diversions are all but unthinkable in this day of ever-shortening development times. But reflection is critical. There is no better way to get into a creative rut than to hack mindlessly. You may produce lots of working code, but code mass is a poor measure of one's productivity. The mark of a good design is precisely the opposite—it is small and elegant. It does much without bulk. It is also flexible; large bodies of code generally aren't.

Obviously, it's impractical to expect people to take a month off per year for contemplation. What you *can* do, however, is record your experiences *incrementally*. When you have a non-trivial problem to solve, try to write it down immediately. Jot some notes about what the problem is and why it's difficult. Then start working on it. Whenever you try a new approach, jot it down; when it fails, jot that down, too, along with *why* it failed. Do the same if it succeeds. Almost everyone can spend 5% of their time jotting down experience. It just takes discipline.

If you do this religiously, you'll be amazed at the written experience you accumulate. This is the raw material of patterns. There's still much work to be done, certainly, but you've got the all-important nuggets of wisdom from which to refine the gold.

Another important activity is to look at as many other systems as you can, systems designed by other people. The best way to learn from other systems is to actually build with them. But if you don't have the time or money to use them, then at least read about them. Seek an understanding of the problems they address and how they address them. Look at specifications and documentation. Read papers on research systems. Look at *OOPSLA* and *ECOOP* proceedings. *Software—Practice & Experience* is another good source of design and implementation information.

When you examine a system, glean everything you can from it. Try to identify patterns you already know. Evaluate how the solutions you find vary from those of published patterns. Be on the lookout for novel design solutions—they may represent new patterns. But keep in mind that relatively few design solutions are truly new. More often, people use variations on known solutions. A new and unique solution might not be widely applicable enough to be cast in pattern form.

If you do find something that seems new, make sure it is applicable in other contexts before you try to write it up as a pattern. We had one inviolable rule as we developed *Design Patterns*: we had to find *two* existing examples of a problem and its solution before we would write a pattern for it. This was a particularly important rule to follow, because we were exploring unfamiliar territory, and we wanted to make sure what we wrote was grounded in reality. We didn't want to end up with a set of solutions to problems nobody had. Ultimately we discarded many patterns that seemed quite appealing and potentially useful but hadn't seen real use.

4 Principles of Pattern Writing

Once you have the raw materials, how do you go about writing them up in pattern form?

Given that a pattern is a structured exposition of a solution to a problem in a context, the first thing to decide on is its structure. The more information your average pattern has, the more important it is to give it structure. Consistent structure lends uniformity to patterns, letting people compare them easily. Structure also helps people search for information. Less structure means more prose, which might be fine for casual reading but unacceptable for comparison and reference purposes.

Design Pattern Template

The Strategy pattern shown earlier illustrates the structure we have adopted. Each design pattern is divided into sections according to the following template. The template gives the information a uniform structure, making design patterns easier to learn, compare, and use.

Pattern Name The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As Other well-known names for the pattern, if any.

Motivation A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [8].

Participants The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations How the participants collaborate to carry out their responsibilities.

Consequences How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Be Concrete Early

Notice how the motivation for a design pattern comes up-front. People seem to understand concepts better when they are presented in concrete terms first and then more abstract terms. The concrete example in the Motivation section gives the reader a frame of reference for the problem and its solution. Another thing this section demonstrates is why other approaches to the problem fail, again in concrete terms. With the Motivation section as an introduction, the reader is better able to appreciate the general solution.

A corollary to concreteness is the need for lots of examples from the real-world. Examples should *not* be the sole estate of the Motivation section. Use examples throughout the pattern to illustrate key points. Even the most abstract sections in our template (i.e., Applicability, Structure, Participants, and Collaborations) sometimes include examples. For example, the Collaboration sections of some patterns include interaction diagrams [5, 1] to show the collaborations between objects at run-time [4]. Refer to such examples when discussing the abstract aspects of the pattern—be concrete even when you're being abstract.

Keep Patterns Distinct and Complementary

There's a common tendency to avoid as you work on multiple patterns. Patterns tend to grow both in detail and in scope. It's easy to forget about other patterns when you're working on one of them. The distinction between patterns blurs as a result, making it hard for others to understand the patterns as a whole. The patterns start to overlap in scope and purpose. It may all seem perfectly clear to the author, but it generally won't be to newcomers. They won't know when to use one pattern and not another, because their differences aren't clear.

Therefore make sure your patterns are orthogonal and that they work synergistically. Continually ask yourself, "How is pattern *X* different from pattern *Y*?" If two patterns solve the same or similar problems, you can probably merge them. Don't worry if two patterns use similar class hierarchies. There are only so many ways to use the relatively few mechanisms inherent in object-oriented programming. Often the same arrangement of classes will yield substantially different object structures that address widely varying problems. Let the *intents* of the patterns be your guide to their differences and not the class structures that implement them.

A good way to test how orthogonal and synergistic your patterns are is to keep separate documents in which to compare and contrast your patterns. In *Design Patterns* we dedicated several sections to this purpose. The simple act of trying to explain pattern relationships in written form gave us a new perspective on our patterns. More than once it forced us to rethink some of them.

My only regret is that we didn't concentrate on relationships earlier in the game. I recommend that you start writing such supplemental material as soon as possible. It may seem like a silly thing to do, especially when you don't have a lot of patterns to compare. But as soon as you have just two patterns, the possibility of overlap emerges. Spending time comparing and contrasting them early on will help keep your patterns distinct and complementary.

Presentation

The quality of your patterns is determined by how well you present them. You could discover the best pattern in the world, but it won't help anybody unless you can convey it effectively.

By "presentation" I mean two things: typesetting and writing style. Good typesetting is a matter of skill in page layout, typography, and graphics, not to mention printer quality. Use the best software tools (word processor, drawing editor, etc.) you can. Make liberal use of drawings to illustrate key points. Not all drawings have to be formal class and object diagrams; informal drawings and even sketches often convey just as much information and more. If you are "artistically challenged," then have someone else do the drawings for you.

Good writing style is even more important than good typesetting. Write clearly and unpretentiously. Favor a down-to-earth style rather than a stuffy, academic one. People understand and appreciate a conversational tone. It makes them more receptive to the material. This is important in all writing, but it's especially important for pattern writing. The concept is new enough and the subject matter complicated enough that some people have a hard time seeing the point of it all. Everything that can be done to make the pattern approachable should be done.

The best way to learn how to write conversationally is to try your hand at it. Make sure that everything you write is something you could hear yourself saying to friend. Avoid the passive voice. Break up long sentences and paragraphs. Use everyday words and contractions whenever possible. Keep it light.

Another thing everyone should do at some point in their lives is read a book or two on good writing style. There are many to choose from. Two of my favorites are Strunk and White's *Elements of Style* and Joseph M. Williams' *Style: Ten Lessons in Clarity and Grace*. Books like these are packed with tips and techniques for good, clear writing. They can help you improve your patterns independent of their technical content.

Iteration

You won't get a pattern right the first time. You won't get it right the first ten times. In fact, you'll probably never get it totally right. Pattern writing is an on-going process. The fact that the field is new doesn't help matters. But even if it weren't...even if there were lots of examples of good patterns and books to help you write them, pattern development (like any other kind of development) would still be an iterative process.

Expect to write and re-write you patterns many times. Don't look for perfection in one pattern before you begin work on the next. Patterns don't exist in isolation; they affect one another. A significant change to one could very well impact others. As with any iteration, your efforts should converge at some point, but that's just the point where the patterns have stabilized enough that other people can read, understand, and comment on them.

Feedback

No pattern can be trusted unless it is evaluated by someone other than its author. Patterns have the insidious property that they are usually perfectly understandable to those who are familiar with the problem they address and its solution. Such people have used the pattern unconsciously before. Thus when they see the pattern, they can recognize it immediately, even if it isn't presented very well. The real challenge is to make the pattern understandable to people who have never run across the problem before. There's no way to do that without getting and incorporating feedback from just such people.

Use the Pattern

Cervantes was right: "The proof of the pudding is in the eating." The acid test of a pattern comes when it's put to use. It's no trick to use the pattern if you are its author. It's more important to have others try to use it. Encourage your colleagues to discuss design in terms of your patterns. (It's OK for the author to participate in such discussions.) Look for opportunities to use the patterns in your day-to-day work. Try to disseminate your patterns as widely as you can. You might even submit them to conferences like Pattern Languages of Programs (PLoP) or publications like the *Journal of Object-Oriented Programming*. Such exposure will maximize the use of your patterns.

5 Conclusion

I started this paper with an appeal to reverse engineering as an activity to emulate in the realm of software architecture. I introduced the term "reverse architecture" to describe a specific activity I had in mind, namely design pattern development. But patterns are just one of many possible fruits of reverse architecture. There may be other, more effective media for recording and expressing architectural knowledge. What is clear, however, is that software architects often follow unwritten rules and instincts. Reverse architecture is unearthing, characterizing, recording, and disseminating these qualities to broaden the impact of good design. It's something we should do more of.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.

[2] Grady Booch and Michael Vilot. The design of the C++ Booch components. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 1–11, Ottawa, Canada, October 1990. ACM Press.

[3] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

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

[5] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering—A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[6] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. In Robert Giegerich and Susan L. Graham, editors, *Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, 1992. Springer-Verlag.

[7] Mark Linton, Paul Calder, John Interrante, Steven Tang, and John Vlissides. *InterViews Reference Manual*. CSL, Stanford University, 3.1 edition, 1992.

[8] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[9] André Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 46–57, San Diego, CA, September 1988. ACM Press.