Object-Oriented Design: A Responsibility-Driven Approach

Rebecca Wirfs-Brock
rebeccaw@orca.wv.tek.com
(503) 685-2561

P.O. Box 1000, Mail Station 61-028
Tektronix, Inc.
Wilsonville, OR 97070

Brian Wilkerson
(503) 242-0725

921 S.W. Washington, Suite 312
Instantiations, Inc.
Portland, OR 97205

ABSTRACT

Object-oriented programming languages support encapsulation, thereby improving the ability of software to be reused, refined, tested, maintained, and extended. The full benefit of this support can only be realized if encapsulation is maximized during the design process.

We argue that design practices which take a data-driven approach fail to maximize encapsulation because they focus too quickly on the implementation of objects. We propose an alternative object-oriented design method which takes a responsibility-driven approach. We show how such an approach can increase the encapsulation by deferring implementation issues until a later stage.

Introduction

The primary benefit of object-oriented programming is its ability to increase the value of a number of software metrics. These metrics include being able to reuse, refine, test, maintain, and extend the code. Yet the value of these metrics has been decreasing as the size of applications, and hence their complexity, has been increasing.

Object-oriented programming increases the value of these metrics by managing this complexity. The most effective tool available for dealing with complexity is abstraction. Many types of abstraction can be used, but encapsulation is the main form of abstraction by which complexity is managed in object-oriented programming.

Programming in an object-oriented language, however, does not ensure that the complexity of an application will be well encapsulated. Applying good programming techniques can improve encapsulation, but the full benefit of object-oriented programming can be realized only if encapsulation is a recognized goal of the design process.

The approach taken by a designer has a profound impact on the degree to which encapsulation is embodied in a design. We will describe the data-driven approach to design and why it fails to maximize encapsulation. We will then describe an alternative design approach, referred to as responsibility-driven, and explain why it results in designs with a higher degree of encapsulation.

Data-Driven Design

Data-driven design is the result of adapting abstract data type design methods to object-oriented programming. The adaptation is straightforward because classes closely resemble abstract data types.

From a purely pragmatic point of view, objects encapsulate behavior (the implementation of an object's responsibilities) and structure (the other objects known directly by that object). This is similar to the definition of an abstract data type.

Before data-driven design is described, let us briefly review abstract data type design.

Abstract Data Type Design

An abstract data type is the encapsulation of data and the algorithms that operate on that data. Abstract data types are designed by asking the questions:

- What data does this type subsume?

  and

- What algorithms can be applied to this data?

The primary focus of these questions is to determine what data is being represented in the system. This can be done initially by identifying the data required by the program (or perhaps only a portion of it). This data can then be grouped into types using cohesion as a guide. (Cohesion, as applied to a group of data, is a measure of how strongly related the parts of the group are.) Finally, identifying the algorithms associated with those types of data often leads to the discovery of other types that are required.

Definition of Data-Driven Design

In a data-driven design, objects are designed by asking the questions:

- What (structure) does this object represent?

  and

- What operations can be performed by this object?

Again, the primary focus is on the structure of the data being represented in the system.

An Example of Data-Driven Design

The following example illustrates the results of a data-driven approach. The example is a class for raster image objects. A RasterImage is an object that stores visual images as two-dimensional arrays of bits. In order to specify a RasterImage object we need to describe the structure represented by a RasterImage and operations that a RasterImage can perform.

The structure of a RasterImage includes

- its bit array (represented as an array of bytes), and
- the dimensions of the bit array (both width and height).

Operations on a RasterImage include basic image manipulation operations such as rotation and scaling; accessing and changing individual bits in the bit array, and combining RasterImages using an imaging operation. We choose not to further specify the imaging operation; it could be performed by a specialized imaging object (such as Smalltalk BitBlt [Gold83]) or a variety of other standard bit image operations.

This overview of the structure and operations on RasterImage lead us to define a RasterImage with the following typed structure:

```
class RasterImage is
    bitArray : ByteArray;
    width : Integer;
    height : Integer;
```

We define protocol for each desired operation. A partial protocol list might include:

```
    rotateBy(degree : Integer) : void;
    scaleBy(x : Integer, y : Integer) : void;
    bitAt(index : Point) : Integer;
    bitAtPut(index : Point, bitValue : Integer) : void;
```

We also define protocol to access and modify each part of the RasterImage structure:

```
    bitArray() : ByteArray;
    bitArray(bits : ByteArray) : void;
    width() : Integer;
    width(aNumber : Integer) : void;
    height() : Integer;
    height(aNumber : Integer) : void;
end RasterImage;
```

A specialization of a RasterImage, called a ShapedImage, includes a clipping mask as well as the bit array. The clipping mask specifies whether each bit in the bit array is opaque (the corresponding clipping mask bit is one) or transparent (the corresponding clipping mask bit is zero). ShapedImages allow display of nonrectangular bit patterns. ShapedImage is a subclass of RasterImage. We define additional structure in ShapedImage to represent the clipping mask.

```
class ShapedImage inherits RasterImage is
    clipMask : ByteArray;
```

And we define protocol to access and modify the clipping mask.

```
    clipMask() : ByteArray;
    clipMask(bits : ByteArray) : void;
end ShapedImage;
```

The Strength of Data-Driven Design

The main benefit of the data-driven approach is that it is a familiar process for programmers experienced with traditional procedural languages. It is relatively easy for such programmers to adapt their previous experience to the design of object-oriented systems.

The Weakness of Data-Driven Design

Even though the goal of data-driven design is to encapsulate data and algorithms, it inherently violates that encapsulation by making the structure of an object part of the definition of the object. This in turn leads to the definition of operations that reflect that structure (because they were designed with the structure in mind). Attempts to change the structure of an object transparently are destined to fail because other classes rely on that structure. This is the antithesis of encapsulation.

Responsibility-Driven Design

The goal of responsibility-driven design is to improve encapsulation. It does so by viewing a program in terms of the client/server model.

The Client/Server Model

The client/server model is a description of the interaction between two entities: the client and the server. A *client* makes requests of the server to perform services. A *server* provides a set of services upon request. The ways in which the client can interact with the server are described by a *contract*: a list of the requests that can be made of the server by the client. Both must fulfill the contract: the client by making only those requests it specifies, and the server by responding to those requests.

In object-oriented programming, both client and server are either classes or instances of classes. Any object can act as either a client or a server at any given time.

The advantage of the client/server model is that it focuses on *what* the server does for the client, rather than *how* the server does it. The implementation of the server is encapsulated — locked away from the client.

A class may have three distinct type of clients:

- external clients,
- subclass clients, and
- the class itself.

Each of these type of clients is described below.

External Clients

An external client of a class is an object that sends messages to an instance of the class or the class itself. The receiver is viewed as a *server*, and the sender is viewed as a *client*. How the message is responded to should not be important to the sender. The set of messages that an object responds to, including the types of arguments and the return type, defines the contract between the client and the server.

Subclass Clients

A subclass client of a class is any class that inherits from the class. The superclass is viewed as a *server*; the subclass is viewed as a *client*. The subclass should not care about the implementation of any behavior which it inherits [Snyd86]. The set of messages that are inherited by the subclass defines the contract between the client and the server.

In most object-oriented languages, subclasses inherit not only the behavior but also the structure defined by their superclasses. This is unfortunate, as it tends to encourage programmers to violate encapsulation in order to increase the amount of code reused.

Self Client

For the purposes of maximizing encapsulation, classes should also be viewed as clients of themselves. The amount of code relying directly on the structure of the class should be minimized. In other words, the structure of the class should be encapsulated in the minimum number of methods. This minimizes the reliance of an object on its own structure, allowing changes to that structure to be made as transparently as possible.

Definition of Responsibility-Driven Design

Responsibility-driven design is inspired by the client/server model. It focuses on the contract by asking:

- What actions is this object responsible for?

  and

- What information does this object share?

An important point is that information shared by an object might or might not be part of the structure of that object. That is, the object might compute the information, or it might delegate the request for information to another object.

An Example of Responsibility-Driven Design

We now illustrate the results of a responsibility-driven design using the same classes as before: RasterImage and ShapedImage. In order to specify a RasterImage object we need to describe the actions a RasterImage is responsible for performing, and the information it shares with its clients.

A RasterImage is responsible for:

- knowing and maintaining the smallest rectangle that completely contains its image,
- knowing and maintaining all individual bit values within its image, and
- rotating and scaling its image.

Given these responsibilities, we translate them into the following protocol:

```
class RasterImage is
    boundingRectangle() : Rectangle;
    boundingRectangle(bounds : Rectangle) : void;
    bitAt(index : Point) : Integer;
    bitAtPut(index : Point, bitValue : Integer) : void;
    scaleBy(x : Integer, y : Integer) : void;
    rotateBy(degrees : Integer) : void;
end RasterImage;
```

A ShapedImage has additional responsibility for knowing and maintaining the visibility of individual bits within its image. To fulfill this responsibility, we define the following protocol.

```
class ShapedImage inherits RasterImage is
    isVisibleAt(index : Point) : Boolean;
    makeVisibleAt(index : Point) : void;
    makeInvisibleAt(index : Point) : void;
```

A variety of ways can be used to specify the shape of a ShapedImage, such as specifying a polygon which encloses all the visible bits. We add the following protocol to describe the shape of a ShapedImage:

```
    shape() : Polygon;
    shape(aPolygon : Polygon) : void;
end ShapedImage;
```

We could even conceive of specifying the shape of a ShapedImage through a function which describes a closed area, if we so desired.

Looking at the protocol for ShapedImage, we observe that it is reasonable to ask a RasterImage whether or not it is visible at a given point (every point within its bounding rectangle is visible, all other points are not). So we move isVisibleAt() protocol to RasterImage as well.

Following this responsibility-driven approach, we have not yet specified, nor are we yet concerned with, the structure of our objects RasterImage and ShapedImage. We choose first to focus on fully specifying the desired behavior and associated sets of responsibilities for each object. We will focus on structure during implementation.

The Strengths of Responsibility-Driven Design

Encapsulation is compromised when the structural details of an object become part of the interface to that object. This can only occur if the designer uses knowledge of those structural details. Responsibility-driven designs maximize encapsulation when the designer intentionally ignores this information.

Polymorphism increases encapsulation, because the client of an object does not need to know not only *how* the object implements the service being requested, but neither *which* class (type) is responding to the request. The responsibility-driven approach can help a designer identify standard protocols (message names) by encouraging the designer to focus on responsibilities independently of implementation. This facilitates polymorphism.

Designing classes without knowledge of their structure encourages the design of the inheritance hierarchy to follow a type hierarchy, because only the types of the classes are known. If the inheritance hierarchy follows a type hierarchy, then the type of every class is a subtype of the type of every superclass of that class [Card85]. This has two advantages.

- It improves encapsulation with respect to subclass clients [Halb86], ensuring that all inherited behavior is part of the contract of the subclass.
- It makes abstract classes easier to identify. One difficulty in identifying abstract classes lies in determining what parts of the protocol of existing classes are parts of the type of those classes, and what parts are implementation details. Because the protocol of a class includes only those messages which form the type of the class, this problem is eliminated.

Comparing Responsibility-Driven and Data-Driven Design

Responsibility-driven design specifies object *behavior* before object *structure* and other implementation considerations are determined. We have found that it minimizes the rework required for major design changes.

Responsibility-driven designs can lead to early definitions of abstract classes. For example, given our responsibility-driven design of the RasterImage object hierarchy, it is reasonable to consider specifying an abstract RasterImage class as the superclass of both ShapedImage and RectangularImage (a new class taking the old role of RasterImage). This allows us to abstractly specify the behavior expected of all raster images and makes the addition of new types of images more straightforward.

Object structure is specified during the implementation of a responsibility-driven design. At that time, performance considerations can lead to adjustments of the object's responsibilities. The goal during implementation is to preserve encapsulation while renegotiating the client/server contracts.

Data-driven design, on the other hand, quickly leads to prescribing object structure. Building abstract classes involves finding the abstraction and lifting it out of a set of concrete classes. Client objects of a data-driven designed object can (and often do) become dependent upon the specific structure of the object, making more difficult the task of developing an abstraction while preserving the contract between clients and servers.

Contrast with the Law of Demeter

The Law of Demeter also tries to reduce dependencies between classes and thus maximize encapsulation [Lieb88, Lieb89]. The Law states that a message can be sent only to:

- message arguments,
- instance variables,
- new objects returned by a message-send, and
- global variables.

The strong form of the Law disallows sending messages to inherited instance variables. The weak form of the Law allows inherited instance variables to be referenced.

The Law of Demeter achieves encapsulation by limiting access to objects. In our view, the Law overly constrains the possible connections between objects. The Law of Demeter results from taking a data-driven view of object-oriented programming. All objects returned from a message-send are regarded as potential encapsulation violations, as those returned objects may reflect the internal structure of the receiver.

This view, which treats all returned objects as Trojan horses, is in direct contrast to our view of a client/server relationship. In the responsibility-driven approach, returned values are part of the client/server contract. There need be no correlation between the structure of an object and the object returned by a message.

While the Law of Demeter does improve encapsulation, it also can confuse the design or implementation of the set of services offered by a class. To avoid sending a message to a returned value (which is known to be part of the structure of the receiver), Lieberherr, Holland and Riel propose adding messages to all intermediary server objects [Lieb88, Lieb89]. These methods exist to relay messages to a component of an object's structure. If this encapsulation technique were carried to an extreme, all possible messages understood by publicly accessible instance variables of an object would have corresponding relay methods. Adding these messages to an object would almost certainly violate the type hierarchy. Maintainability and extensibility are then compromised.

The weak form of the Law of Demeter provides encapsulation for external clients of an object. The strong form of the Law provides encapsulation between subclass clients. No corresponding formulation of the Law exists to realize self-encapsulation. In responsibility-driven design, we address self, subclass, and external client encapsulation through the same technique: limiting variable access [Wirf88b, Scha86].

Language Support

There are a number of forms of language support that can help maintain the high degree of encapsulation achieved during the design phase. Two of these are discussed below: limiting access to variables, and limiting access to behavior. If a language does not provide direct support for either or both of these, stylistic conventions should be followed to achieve the same result.

Limiting Access to Variables

Directly accessing the state (variables) of an object violates the encapsulation of a class with respect to at least one of the three types of contracts defined in the section titled **The Client/Server Model** [Wirf88b]. Which type of contract depends on the type of client that accesses the state.

Smalltalk-80 [Gold83] and Objective-C [Cox86], among others, prevent external clients from directly accessing the structure of a server. Languages such as Trellis/Owl [Scha86], Self [Unge87], and Modular Smalltalk [Wirf88] enforce encapsulation for all types of clients by eliminating instance variables entirely.

Limiting Access to Behavior

In addition to limiting access to state, a language should allow the programmer to limit the access to the behavior of a class. We feel that it is reasonable to assume that subclass clients should have all access permitted to external clients. The access granted to the class itself should similarly include all of the access permitted to subclass clients.

Modular Smalltalk supports only two of these levels of access: visibility to all clients (public) and visibility to self (private). C++ [Stro87], on the other hand, supports all three levels of protection to members (variables or functions): visibility to all clients (public), visibility to subclasses and self (protected) and visibility to self (private).

Eiffel [Meye88] allows a much more finely grained specification of access. Each feature (variable or function) can be given a list of classes that can access it. This list contains the clients whose contract includes the feature.

Conclusion

Encapsulation is the key to increasing the value of such software metrics as reusability, refinability, testability, maintainability, and extensibility. Object-oriented languages provide a number of mechanisms for improving encapsulation, but it is during the design phase that the greatest leverage can be realized.

The data-driven approach to object-oriented design focuses on the structure of the data in a system. This results in the incorporation of structural information in the definitions of classes. Doing so violates encapsulation.

The responsibility-driven approach emphasizes the encapsulation of both the structure and behavior of objects. By focusing on the contractual responsibilities of a class, the designer is able to postpone implementation considerations until the implementation phase.

While responsibility-driven design is not the only technique addressing this problem, most other techniques attempt to enforce encapsulation during the implementation phase. This is too late in the software life-cycle to achieve maximum benefits.

References

Cardelli, Luca, and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, 17(4), December, 1985, pp. 471 522.

Cox, Brad, *Object-oriented Programming, An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.

Golberg, Adele, and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Halbert, Daniel C. and Patrick D. O'Brien, "Using Types and Inheritance in Object-Oriented Languages," Technical Report DEC-TR-437, Digital Equipment Corporation, April, 1986.

Lieberherr, K., I. Holland and A. Riel, "Object-Oriented Programming: An Objective Sense of Style," *SIGPLAN Notices*, 23(11), November 1988, pp. 323 334.

Lieberherr, Karl L. and Ian Holland, "Formulations and Benefits of the Law of Demeter," *SIGPLAN Notices*, 24(3), March, 1989, pp. 67 78.

Meyer, Bertrand, *Object-oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Schaffert, Craig, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt, "An Introduction to Trellis/Owl," *SIGPLAN Notices*, 21(11), November, 1986, pp. 9 16.

Snyder, Alan, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *SIGPLAN Notices*, 21(11), November, 1986, pp. 38 45.

Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1987.

Unger, David, and Randall B. Smith, "Self: The Power of Simplicity," *SIGPLAN Notices*, 22(12), December, 1987, pp. 227 242.

Wirfs-Brock, Allen and Brian Wilkerson, "An Overview of Modular Smalltalk," *SIGPLAN Notices*, 23(11), November, 1988, pp. 123 134.

Wirfs-Brock, Allen and Brian Wilkerson, "Variables Limit Reusability," *Journal of Object-Oriented Programming*, 2(1), May/June, 1989, pp. 34-40.