# *Implementing Design Responsibilities: Guidelines for Constructing Usable Objects*

by Rebecca Wirfs-Brock

An object designer's job isn't over when she starts coding. We can ruin perfectly good work by translating crisp designs into murky implementations.

"Consider the books, radios, kitchen appliances, office machines, and light switches that make up our everyday lives. Well-designed objects are easy to interpret and understand. They contain visible clues to their operation. Poorly designed objects can be difficult and frustrating to use. They provide no clues or sometimes-false clues. They trap the user and thwart the normal process of interpretation and understanding. Alas, poor design predominates. The result is a world filled with frustration, with objects that cannot be understood, with devices that lead to error." - Donald Norman, *The Design of Everyday Things*

Donald Norman's words of advice apply equally to the design of software as well as physical objects. To implement an object's responsibilities that we have designed we specify one or more method signatures. A method signature consists of a method name, arguments, and return value. We then code the body of each method and also declare necessary internal object storage. As we implement our designs we should take care not to create brittle, breakable objects. Meilir Page-Jones cautions against creating objects having awkward, illegal, irrelevant, redundant, incomplete or inappropriate behavior. There are safeguards we can take to protect against these ills. Let's look at some guidelines for designing interfaces and implementations that make objects more usable.

Suppose we design a rectangle having responsibilities for knowing its dimensions and its location. During implementation we transform these responsibilities into four instance variables to represent our rectangle's corners, and provide methods for setting and moving its corners. We provide the public method

moveCorner(oldCoordinate, newCoordinate)

that relocates a single corner. What's wrong with this picture?

**Guideline**: Avoid messages that break objects.

Don't permit unchecked changes to an object's instance variables. Providing these interfaces leaves an object open to a flurry of incorrectly ordered messages that will break it.

If our moveCorner() method blindly moves one corner independently of the other three, our rectangle becomes a trapezoid in a single message! We need to preserve our object's property of rectangleness: having four corners and four right angles.

Let's fix our vulnerable rectangle to be more robust. To retain its rectangle shape, one corner cannot move independently of the other three. We can fix our rectangle by disallowing direct, unguarded corner changes. A more robust moveCorner() implementation would check and preserve the constraints between each corner. We can do this by adding the distance between oldCoordinate and the newCoordinate to all four corners. Our new, improved moveCorner() method moves all corners in response to a request to relocate a single corner. This is a better implementation because it prevents inconsistent state changes.

**Guideline**: Don't allow messages which cloak hidden limitations

These kinds of messages don't cause an object to be inconsistent, yet its users still thinks it is broken!

Let's imagine our rectangle to be constrained to a minimum size: we cannot create a rectangle smaller than one unit per side. How should we implement a

createRectangle (lowerLeftCorner, width, height)

method to support this constraint?

One startling way to do so would be to allow the sender to think she was creating a small rectangle, but to silently return a minimum-size rectangle, regardless of the requested width and height. Another, still somewhat unsatisfying solution would be to return an error or throw an exception in response to a request to create too small a rectangle. Ideally there shouldn't be arbitrary restrictions on an object's legal states: why limit a rectangle's size?

If there are such limitations, and there will be, the world is full of imperfections, how best can we convey these limits in our implementation? Does it make our design better to describe these limits in our method comments? Someone attempting to create a too small rectangle might be better informed: Aha! It isn't a bug, merely a design limitation! But she still won't be satisfied. Documenting limitations improves understanding, but not an object's usability or maintainability. Unless the reasons why such limitations exist are documented, maintainers won't be able to discriminate between gratuitous implementation limitations, design

quirks or mandatory restrictions.

A better solution is to provide more than one way to create rectangles. We still can’t avoid generating exceptions to exceptional arguments in

createRectangle (lowerLeftCorner, width, height),

but a second method could provide an implementation that constrains without throwing exceptions the rectangle's size to its minimum size,

createConstrainedRectangle (lowerLeftCorner, width, height).

This improves the usability of our rectangle for situations when supplied values might be slightly smaller than the minimum. Rather than have the client code constrain the values of with and height, these can be performed once within createConstrainedRectangle().

We might even include a third method to create a minimum sized rectangle, which allows a user to create small rectangles without having to know minimal dimensions:

createMinimumSizeRectangle (lowerLeftCorner).

If we embed enough information about rectangle creation behavior in each method's name, these clues to help avoid surprising behavior and can lead users to find details in documentation or method comments.

**Guideline**: Avoid implementation revealing messages

Providing public accessing methods to every instance variable can cause an even more insidious problem. Client code will tend to rely upon the details, making it hard to change them without breaking client code. Fortunately, the fix for this is very simple. Provide methods to set and/or retrieve the values of instance variables only if you believe they should be publicly visible throughout the object’s lifetime. For example, while it is appropriate to provide methods to retrieve the values of our rectangle's corners, e.g.,

- lowerLeftCorner()
- lowerRightCorner()
- upperLeftCorner()
- upperRightCorner(),

we don't provide setter methods to independently change corner values.

**Guideline**: Avoid spreading an atomic responsibility across multiple messages

When multiple messages are required to carry out a single responsibility, the

possibility exists that the client will not finish directing our object to do its proper work. This is especially bad if one or more of those messages put our object into an inconsistent state. For example, if to move a rectangle we had to send it four messages to move its four corners, our rectangle would be misshapen if only three messages were sent. We've seen how to fix our rectangle's interface, above. We can remove this problem by providing a single method that in one atomic operation moves the rectangle to a new location. This trivial example is easy to fix; however it is tempting to provide power interfaces for power users in addition to well-behaved atomic methods (power users know what they are getting into, so let have fine control over an object). Whenever you expose minute details of an object's behavior to external control question whether it is possible to abuse this power, and whether the resulting control outweighs the potential danger.

**Guideline**: Don't arbitrarily place responsibilities.

Designing an object to support a responsibility that is tangentially related (at best) is simply not a good idea. Question whether an object's responsibilities are appropriate. When you cannot find an appropriate existing object to place a responsibility, consider adding a new object to your existing design.

Should you burden a Product object with determining the best means of shipment?

To answer this question we need to analyze why a suspect responsibility landed in this object, and what other objects are involved in fulfilling the responsibility. We need to ship an Order (consisting of one or more Product Items) reliably to a customer's shipping location. Determining the best means of shipment involves knowing the size, weight and quantity of the Product, the shipping location (represented by an Address object), any restrictions/constraints on shipping (for example when the customer wants the order), how many other Product Items are in the Order, etc. A single Product object is just a minor player in this complex relationship between customer Order, Product Items and Means of Shipment. It is reasonable to expect a Shipment object (which is either a completed or a partial order) to know its shipment details, but even then, it might collaborate with another, as yet undetermined object, perhaps a Shipment Planner, who knows the rules for shipping.

**Guideline**: Avoid overly restrictive or incomplete implementations of a responsibility.

Design an object to support all behavior needed by an application. Question arbitrary restrictions

One example of this was an initial design of a Customer object for a banking application. Once a customer is activated (moved from a pending to an active

status), it was thought that the customer object could not or should not be returned to a pending status. This simply wasn’t so. If a Bank Agent incorrectly changed a customer’s registration status, she needed to reset it! The initial design was too restrictive, not allowing mistakes to be corrected.

**Guideline**: Avoid redundant messages

Don't support multiple ways to accomplish the same request. In the design of a result object, which conveyed information between distributed components in an internet banking application, the Result objects supported over a dozen messages for checking on and setting the result value. Did the interface designer go overboard?

ifError: actionBlock

" If the receiver indicates error or fatal error, answer the result of evaluating the actionBlock without arguments; otherwise, answers the receiver. "

isContinuableError

" Answers true if the receiver indicates a non-fatal error; otherwise, answers false. "

isError

" Answers true if the receiver indicates error or fatal error; otherwise, answers false. "

isFatalError

" Answers true if the receiver indicates a fatal error; otherwise, answers false. "

isSuccess

" Answers true if the receiver indicates success; otherwise, answers false. "

isWarning

" Answers true if the receiver indicates warning; otherwise, answers false. "

setCorruptDataError: aTemplateID

" Sets the receiver's condition to #CorruptDataError and sets its templateID from the argument and answers the receiver. "

setDeveloperError

" Sets the receiver's condition to #DeveloperError and answers the receiver. "

setDeveloperError: aTemplateID

" Sets the receiver's condition to #DeveloperError and sets its templateID from the argument and answers the receiver. "

setError

" Sets the receiver's condition to #Error and answers the receiver. "

setError: aTemplateID

" Sets the receiver's condition to #Error and sets its templateID from the argument and answers the receiver. "

setFatalError

" Sets the receiver's condition to #FatalError and answers the receiver. "

setFatalError: aTemplateID

" Sets the receiver's condition to #FatalError and sets its templateID from the argument and answers the receiver. "

setProcessingError

" Sets the receiver's condition to #ProcessingError and answers the receiver. "

setProcessingError: aTemplateID

" Sets the receiver's condition to #ProcessingError and sets its templateID from the argument and answers the receiver. "

setSuccess

" Sets the receiver's condition to #Success and resets its templateID to nil and answers the receiver. "

setSuccess: aTemplateID

" Sets the receiver's condition to #Success and resets its templateID to nil and answers the receiver. "

setWarning

" Sets the receiver's condition to #Warning and resets its templateID to nil and answers the receiver. "

setWarning: aTemplateID

" Sets the receiver's condition to #Warning and sets its templateID from the argument and answers the receiver. "

The Result class designer chose to provide a unique message to set each possible result value, and another message for setting each status with a message template. This resulted in thirteen messages instead of one. An alternative design would have been to provide a single message with two arguments, status and message template, e.g.:

setStatus:aSymbol message: aTemplateID.

The designer chose to not do this. He felt that the internal details of how status was represented should remain hidden: all a client should be able to do is to set a status with a specific message, and then ask the object if it were in that state.

If he had modified his design to provide one method to set the status with an argument, he felt this implied that status was stored directly in an instance variable whose type matched the status argument. The designer vigorously argued for his implementation that did not reveal nor imply any details about status was internally stored. This led to a verbose implementation of a single responsibility.

Sometimes we overwork our designs. Designers get protective and insist that their way is the right way. Egos clash and cloud good judgement. In truth there is no one correct way to implement a design. I argued for a simpler implementation of the Result object. It was one argument I lost. Fortunately the consequences weren't disastrous. We turned out to use only a very small fraction of the elegant Result interface. Most messages ended up never being sent! Wonderful design, moderately complex implementation, but simple usage prevailed. In conclusion I'll calmly interject a guideline which serves me well in everyday life as well as software design.

**Guideline**: Favor simplicity and comprehension over completeness and complexity.

Don't be enamored with complexity for complexity's sake. When you find yourself admiring an elegant implementation, consider whether your object's users think as highly of it as you do.

*References*

D. Norman, *The Design of Everyday Things*, Bantam Doubleday, 1988 ISBN 0-38-52677-64

M. Page-Jones, *What Every Programmer Should Know about Object-Oriented Design*, Dorset House Publishing, 1995 ISBN 0-932633-31-5