Leveraging the Roles and Responsibility Model

by Mark Richards

Whether starting out from scratch or maintaining an existing application, there will always come a time when you need to add new business functionality and capabilities to an application. Which classes and components should contain the new functionality? This question may sound simple and obvious but in most cases it isn't. All too often applications end up becoming overly complex and unmaintainable due to one missing component: the simple but powerful roles and responsibility model. This article shows how the roles and responsibility model can be leveraged to build robust and maintainable software applications.

Introduction

The roles and responsibility model is a simple technique for assigning every class and component in your application a statement about its roles and responsibilities. It sounds simple and easy (and for the most part it is), yet not enough architects and developers use it. Rather, architects and developers implicitly assign roles and responsibilities to classes and components solely based on the class or component.

For example, it may seem like overkill to assign a roles and responsibility statement to an Order class because the nature of the name implies that it handles all of the business logic associated with an order. While this may in part be true, here are some questions that may not be so obvious. Should the Order class contain logic associated with shipping the order? Should it contain logic to calculate shipping charges and sales tax? Should it be responsible for notifying the warehouse when the item quantity falls below a certain threshold? Should it be responsible for emailing the customer based on the various stages of the order (e.g., ordered, shipped, etc.)? Without a roles and responsibility statement, it is difficult to determine exactly what the class actually does or is supposed to do.

The roles and responsibility model is not new. It originated from the CRC (Class Responsibility Collaborators) Card Methodology first introduced by Ward Cunningham and Kent Beck back in 1989 in a paper they presented at the OOPSLA conference titled "A Laboratory for Teaching Object-Oriented Thinking". In the CRC Card Methodology, small index cards were used to record the class name, the super and sub classes, the class collaborators and the responsibility of the class. The use of small index cards was not accidental; they didn't require any special technology or computers, and the small card size kept the responsibility of the class at a minimum, which was one of the goals of the CRC Card Methodology. An example of a typical CRC card is illustrated in Figure RIC-1.

Class Name: Customer	
Superclasses: Person	
Subclasses:	
Responsibilities	**Collaborators**
Maintains name	
Maintains address	
Maintains contact information	
Maintains customer number	
Knows order history	Order
Knows reward points	Order
Knows website preferences	SitePreference

Figure RIC-1

With Agile and Lean methodologies taking the lead these days, the CRC Card Methodology is not as relevant as it was a couple of decades ago. However, the roles and responsibility model that originated from the CRC methodology is still relevant today, particularly for Agile methodologies or applications that are developed without an initial clear and

definitive object model. In these situations it is typical to "let the application evolve" with only an initial notion of what the object model looks like. As a result, you typically end up with over-bloated classes or classes that contain unrelated methods and attributes.

Benefits

One of the major benefits of using the roles and responsibility model is that it provides you with guidance as to where new functionality should be placed within your application. This takes away guesswork and significantly reduces incorrect decisions, resulting in robust applications that are highly maintainable with little or no duplication of functionality.

For instance, going back to the Order class example from the prior section, consider the scenario where you need to add new functionality to your order processing system to accumulate points for a customer every time they place an order for any item over $100.00. Should this new functionality go into the Order class or the Customer class? Without the roles and responsibilities model, there is little guidance as to where the developer should add the new functionality, potentially impacting the original object model and object design (and the overall maintainability of the application as well).

If this indecision continues, you will soon find yourself deep within the bowels of what is known as the *Blob Anti-Pattern*. The Blob Anti-Pattern occurs when you have objects in your application that do too much or know too much about the application. It also occurs in situations where you have classes in your application that contain lots of unrelated methods and attributes.

The Blob Anti-Pattern gets its name from the 1958 Steve McQueen movie *The Blob* where a giant amoeba-like alien begins consuming everything in its path, which causes it to grow until it gets so big it cannot be stopped. A diner owner accidentally discovers that the blob cannot withstand cold, and so with lots of CO2 fire extinguishers they manage to freeze the blob. The movie ends with the classic scene of the blob being dropped out of a military jet over the Arctic wastelands with the words "The End?" splashed across the screen. Spooky, given all the talk about global warming these days.

Like the movie, you can stop the Blob Anti-Pattern from occurring in your application by simply freezing your source code. While this analogy is somewhat humorous, it is unfortunately not realistic. Fortunately, you have an even better (and cheaper) tool at your disposal—the roles and responsibility model.

Using the Roles and Responsibility Model

One common example of a blob that may exist in your application is the infamous custom utility class. Let's call the class Utility. Most of the time this type of custom utility class contains methods such as custom string formatting routines, custom date manipulation routines, mathematical calculations, and so on—in other words, a large group of unrelated methods.

Without a roles and responsibility statement, you might guess that the implicit responsibility of the Utility class is to provide a common location for any utility-related method used by one or more classes. The eventual problem is that it starts to become ambiguous as to what methods really belong in this class. For example, if you need a new string comparator, you would most likely add it to the Utility class. What if you then need a new method to parse a name into the first, middle, and last name parts? You might add that to the Utility class as well since it has to do with String manipulation. What about a method to validate that the customer entered the first, middle, and last name? Since that is related to parsing the name field, you might add that to the Utility class as well. Pretty soon, your Utility class gets so large that it contains a majority of your business logic too and it consumes most of the application up in a single bite (queue scary music).

Why are classes like the Utility class so common? The answer is easy: because such classes are usually injected into almost every business logic class in your application. Therefore, methods added to the Utility class magically become available in any other class in your application. Another reason is because it is easier to add functionality that seems like a utility into a Utility class since you don't have to think about it much or create a new class (along with all the ceremony that goes with it) to implement the new functionality.

This is where the roles and responsibility model comes to the rescue. The litmus test for any class that might be too big or might already be a blob is as follows:

If you cannot create a clear and concise one or two sentence description of what the class is doing or what it is responsible for, then it is probably doing too much.

Going back to our Utility example, if you tried to create a roles and responsibility statement for that class, it would take two to three paragraphs to complete the responsibility statement. Behold, you have identified a blob in your application!

The simple solution to this problem is to break up the Utility class into multiple classes, each containing a specific responsibility. For example, you can create a StringUtil and move all of the String-related methods into this new class. Then you would create a new responsibility statement something like "*this class is responsible for holding and making available all String-related utility functions for the application*". While this responsibility statement is still a little broad, it nevertheless is better than the prior responsibility statement primarily because this one is better scoped. Now do the same thing for the date-related functions, validation-related functions, and so on. It quickly becomes clear where new functionality should (and shouldn't) be added. Voila, you have just destroyed a blob in your application!

While the Utillity class example is common, it is also a simple example. A not-so-simple example is something like an Order class that handles all of the order-related methods in the application. Let's assume this class has a large number of methods (over 60) that are used to process an order, making it a good candidate for a blob. If you were to try and create a roles and responsibility statement for the Order class, you might write something like this: "*this class is responsible for all order-related processing*". Right. All too often I see these sorts of responsibility statements in large classes, reminding me of the following source code comments I frequently find in Java code such as in Listing RIC-1.

```
//set the customer number
customer.setCustNum(request.getCustNum);
```

Listing RIC-1

I am convinced there is an Anti-Pattern for this sort of comment, but I haven't found it yet. A more accurate responsibility statement for the Order class is as follows:

"This class is responsible for placing book orders, CD orders and third-party merchandise orders through the system. It is also responsible for handling the processing involved with validating that book orders, CD orders and third-party orders are of the correct format, and that all data required by these types of orders was entered correctly. Order cancellation is also handled in this class, as well as all inventory adjustments made when the order is confirmed. Payment processing is handled by this class in the form of store credits, credit cards, gift cards, and PayPal transactions. This class is also responsible for all return processing for Books, CDs and Third-Party items, as well as all customer notifications via email..."

Wow—no wonder this class has over 60 methods! By using a more meaningful roles and responsibility statement, it becomes very clear this class is doing far too much. The roles and responsibility model can be used to identify blobs like this oversized Order class. Like the prior Utility class example, the solution to this problem is to either create an abstract order class that contains common order functionality or create a separate book order class, CD order class and so on. Another solution would be to split this class up into multiple classes having distinct responsibilities, such as order validation, order payment processing, order email communications, order shipping, and so on.

When adding a responsibility statement to a class, I typically leverage the top portion of the class file prior to the package statement (you know the place I'm talking about - the one that contains useless statements about copyright information and other meaningless info). Why not make this section useful by placing the roles and responsibility statement there *for every class in your application*? Even transfer objects should have a roles and responsibility statement to ensure they don't contain unrelated attributes not associated with the context in which they were originally intended for.

By creating a standard of placing the roles and responsibility statement at the very beginning of each class file, you always have a clear understanding of what each class is doing (and is supposed to do) in the application. Use the class name as an initial guide, then analyze the roles and responsibility statement at the top of the class; then ask yourself whether it still makes sense for the class to be responsible for the new functionality.

Evolution

The roles and responsibility model provides you with guidance for where additional functionality should (and shouldn't) be placed in the application. However, sometimes the initial responsibility statements don't contain enough information for you to make an informed decision. In these cases, it is natural for the responsibility of a particular class to evolve along with the added functionality.

To illustrate this point, consider a simple example of a stock trading validation system illustrated in Figure RIC-2. The system receives a stock trade and validates the trade to make sure it doesn't violate any rules such as restricted stocks and trader limits. Since this is a trading system, these validations must be fast. As such, the validation system will be threaded to allow for concurrent stock trade order validation.

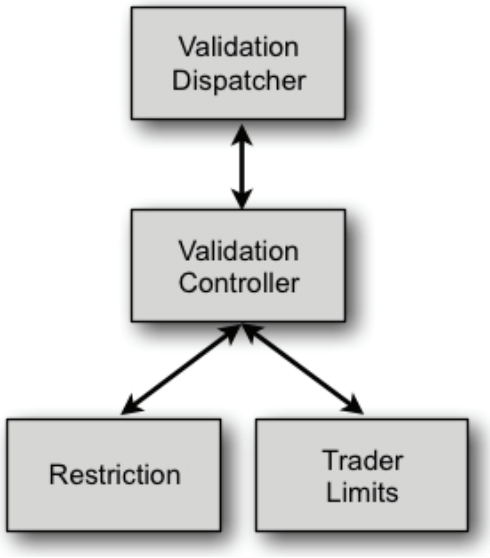

Figure RIC-2

If we assign roles and responsibility statements to each class in this system, they might be stated as shown in Listings RIC-2, RIC-3, RIC-4 and RIC-5 on the next page.

Let's say you already have the basic classes written with the necessary data needed by the validation modules hardcoded in each module. Your first order of business is to now add additional functionality to retrieve the validation data (e.g., the list of restricted stocks) from the underlying database for each validation module. The question is should this new functionality go in the validation modules or in the ValidationController class?

After consulting the roles and responsibility statements, you find there is little in the way of guidance to help you with this decision. You would like to have each validation module be self-contained and responsible for its own data, but since the validation modules are invoked asynchronously and most likely will grow in numbers, you elect to have the ValidationController handle this functionality to save on database connections. Therefore, you extend the responsibility of that class to handle all data retrieval as shown in Listing RIC-6 on the next page.

You now receive a new requirement stating that all validation errors must be persisted to an error table in the database before the request is returned to the caller. The question you now face is whether each validation module should be responsible for persisting its errors or whether the ValidationController should handle the new functionality. This time, the roles and responsibility statements for each class help you make this decision. Since the ValidationController is the only component in this subsystem currently accessing the database, it would make sense to have that component handle *all* of the database activity. Therefore, you further evolve the responsibility of the ValidationController as follows (Listing RIC-7 on page 20).

Now, rather than having multiple components accessing the database, all database activity associated with the validation process is consolidated into a single class, thereby simplifying the object model and source code.

Let's take this example one step further to illustrate how the roles and responsibility model can help identify when a class is doing too much. Assume you have an additional requirement stating that the data for each validation module must be cached to improve performance, and that a cache refresh mechanism must be added as well in the event the validation data is updated. Looking at the roles and responsibility statements for each class, it appears that the ValidationController would be a good candidate to contain this new functionality since it is currently responsible for obtaining the data needed by the validation modules. So, you first modify the ValidationController's responsibility statement to include this new functionality as shown in Listing RIC-8 on page 20.

Notice anything odd about the updated responsibility statement for this class? Its original responsibility was to orchestrate the execution of the validation modules in an asynchronous fashion, which in and of itself is a

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This class is responsible for dispatching a trade order to the next
  * available validation controller thread and sending the results back to the
  * calling component.
  */
public class ValidationDispatcher {…}
```

Listing RIC-2

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This threaded class is responsible for receiving a trade order from the
  * validation dispatcher and orchestrating the calls to the individual
  * validation modules in an asynchronous fashion. Once all validation modules
  * have completed processing, it combines the results and sends them back to
  * the validation dispatcher.
  */
public class ValidationController {…}
```

Listing RIC-3

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This validation module is responsible for ensuring the stock being traded
  * is not on the restricted stock list and that the number of shares being
  * purchased does not exceed a certain amount for technology stocks.
  */
public class Restriction {…}
```

Listing RIC-4

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This validation module is responsible for ensuring that the total amount of
  * the trade does not exceed the preset limits imposed on the trader placing
  * the trade.
  */
public class TraderLimits {…}
```

Listing RIC-5

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This threaded class is responsible for receiving a trade order from the
  * validation dispatcher and orchestrating the calls to the individual
  * validation modules in an asynchronous fashion. Once all validation modules
  * have completed processing, it combines the results and sends them back to
  * the validation dispatcher.  Also responsible for retrieving all of the data
  * needed by each validation module.
  */
public class ValidationController {…}
```

Listing RIC-6

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This threaded class is responsible for receiving a trade order from the
  * validation dispatcher and orchestrating the calls to the individual
  * validation modules in an asynchronous fashion. Once all validation modules
  * have completed processing, it combines the results and sends them back to
  * the validation dispatcher.  Also responsible for database activity,
  * including data retrieval and error posting related to each validation
  * module.
  */
public class ValidationController {…}
```

Listing RIC-7

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This threaded class is responsible for receiving a trade order from the
  * validation dispatcher and orchestrating the calls to the individual
  * validation modules in an asynchronous fashion. Once all validation modules
  * have completed processing, it combines the results and sends them back to
  * the validation dispatcher.  Also responsible for all database activity,
  * including data retrieval and error posting related to each validation
  * module as well as data caching. This class is responsible for listening for
  * cache update events, and upon receiving those events retrieve the updated
  * data from the database.
  */
public class ValidationController {…}
```

Listing RIC-8

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This threaded class is responsible for receiving a trade order from the
  * validation dispatcher and orchestrating the calls to the individual
  * validation modules in an asynchronous fashion. Once all validation modules
  * have completed processing, it combines the results and sends them back to
  * the validation dispatcher.  Also responsible for  persisting errors related
  * to each validation module.
  */
public class ValidationController {…}
```

Listing RIC-9

```
/**
  * Roles & Responsibility Statement
  * ========================
  * This class is responsible for retrieving and caching all data needed by the
  * validation modules. It is also responsible for listening for cache update
  * events, and upon receiving those events retrieve the updated data from the
  * database.
public class ValidationCacheManager {…}
```

Listing RIC-10

complex process considering the class is threaded. However, now it is starting to also become a database manager, isn't it? By modifying the responsibility statement first, you can immediately identify a potential blob before it has a chance to grow. As our application has evolved it has now become clear that we need separate classes for data retrieval and cache management, particularly when it is unlikely that this class will be the only one needing this sort of cache refresh logic.

The appropriate action to take at this point is to refactor the ValidationController to remove the database retrieval logic and add it to a new class responsible for data retrieval and caching as shown in Listing RIC-9 and Listing RIC-10 above.

Now each class has a separate and distinct roles and responsibility statement in the application, making it much more maintainable and robust. As such, the ValidationCacheManager can now evolve to include other data caching at some point.

Summary

Although the examples presented in this article are simple, they nevertheless illustrate the power and usefulness of the roles and responsibility model. One word of caution however; the roles and responsibility model should not be used as a replacement for a good object model design. Rather, it should serve as a self-documenting guide to developers and architects for ensuring that objects in your application contain the appropriate functionality and don't turn into blobs that take over your entire application. Oh, and if you choose to ignore the roles and responsibility model and freeze your code instead, be sure and ask yourself one last question: "The End?"

About the Author

Mark Richards is an Independent Consultant involved in the architecture, design, and implementation of event-driven architecture, service-oriented architecture, messaging systems, and enterprise service bus technologies. Prior to becoming an independent consultant Mark was an Executive IT Architect with IBM, where he worked as an SOA and enterprise architect in the financial services area. Having served in the IT industry since 1984, he has significant experience in the architecture and design of small to large systems on a wide range of languages and platforms. He is the author of several technical books (including the recently published *Java Message Service, 2nd Edition* by O'Reilly), and has spoken at over 85 technical conferences worldwide. You can read more about Mark by visiting his website at http://www.wmrichards.com.