The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods

Erdun E

September 2024

Abstract

Interest in object-oriented methods has been rapidly increasing, as software developers and project managers try to reduce escalating development and maintenance costs. There is an increasing need to determine if there are differences in effectiveness between various methods of object-oriented software development, and whether techniques from more successful methods can be extracted and applied to improve other methods.
This paper reports on research to compare the effectiveness of two methods for the development of object-oriented software. These methods are representative of two dominant approaches in the industry. The methods are the responsibility-driven method and a data-driven method that was developed at The Boeing Company and taught in a course available to the public.
Each of the methods was used to develop a model of the same example system. A suite of metrics suitable for object-oriented software was used to collect data for each model, and the data was analyzed to identify differences.
The model developed with the responsibility-driven method was found to be much less complex, and specifically to have much less coupling between objects and much more cohesion within an object.

1 Introduction

2 Object-Oriented Software Development

3 Approach

A single system was chosen to be developed with both methodologies. Each author of this article had experience with one of the methods and used that method to create a design for the system.
The system chosen is a brewery control system that is used as an example in a course on object-oriented development [5]. It was originally chosen because it involves processes that are easily understood from everyday experience. It is also sufficiently complex to be a realistic case study.
A detailed description of the procedures that were followed to create the designs and to collect the metrics can be found in a technical report [25]

3.1 The Brewery Problem

The brewery control system is composed of an inventory system, a production system, and a recipe library.
The production system is intended to supervise the creation of beer, its movement in the brewery, the monitoring of its condition, and its eventual bottling.
Beer is made according to a recipe, which specifies ingredients and their amounts. The ingredients are mixed in a mixing vat to create a batch of beer, which is then moved to a fermenting vat where readings of its gravity will be taken daily.

While in a fermenting vat, and until it is finished fermenting, the temperature will be monitored and kept within a tolerance of the temperature specified in the recipe by means of refrigeration.
After it has finished fermenting, the beer is moved to another vat to settle. When finished, the beer is moved again, this time to a bottling vat where it may be bottled by an external bottling line.
As specified, the brewery control system includes a network of interconnected pipes and vats of various kinds, an inventory of ingredients, a collection of recipes, and batches of beer both in production and already bottled.
The system monitors beer in production and transfers batches of beer between vats by the shortest clean path.
The above describes the "essential" brewery system, i.e. ignoring the user interface, the features of the operating system, and issues concerning performance or the use of system resources.
For the purposes of this study, this system is primarily specified by the scenarios itemized below. The next section considers other issues that serve to complete the specification.

3.2 Assumptions of Environment and Design Guidelines

Since the example system is a control system, there were certain assumptions to be made concerning the interface to the system being controlled.
There were also issues that had to be resolved to ensure that the two designs would become comparable. These issues can be defined with respect to the capabilities of the programming language, the features of the execution environment, and guidelines to ensure the completeness of the metrics.
The programming language was assumed to have basic object-oriented features, such as information hiding, data abstraction, dynamic binding, and inheritance [26].
Moreover, the language was assumed to be strongly typed, that is it adheres to the principle that every object is of a certain declared type which determines the context in which it may be used. The language also provides for the definition of methods and attributes for classes as well as for instances.
The implementation environment was assumed to be based on a single process system. Therefore, only one instruction of a single method will be executed at a given time, although other methods that invoked the current method may be pending. No asynchronous error handling or other processing was allowed.
In addition, extensive object management capabilities such as those provided by object-oriented database management systems are not available.
The external systems that must be controlled are to be represented by objects in the designs that have direct access to those systems.
These objects are only to be modeled to the degree necessary to account for their interaction with other objects in the system.
Interaction with the user is to be through an existing object-oriented user interface subsystem. The subsystem was assumed to be constructed in a manner convenient for each of the designs considered independently. Since the metrics that were used were intended to measure only object-oriented features, the following guidelines were adopted to avoid the use of other features:

- Variables can either be defined from one of the built-in types or can be references to objects created from the classes defined in each design. The built-in types are numbers, Boolean values, characters, and character strings. Therefore, data structures other than objects, e.g., arrays or records, cannot be used.
- Special meanings are not to be ascribed to specific values of variables.• Functions or procedures other than methods of objects or classes are not to be used.• The attributes of objects of one class are to remain hidden from objects of another class. Thus, an object cannot directly manipulate the attributes of objects of a different class.
- Global vmSables, even references to objects, are not to be used.
- No more than one item of information can be requested by or provided with the invocation of a method unless the additional information is necessary to provide the service represented by the method or is created as a

necessary consequence of performing the service.

3.3 The Scenarios

A scenario is a narrative of a part of the overall behavior of a system. A complete collection of scenarios can be used to completely specify a system under development [27]. As a model of a system is developed, it can be checked with respect to the scenarios to verify that it satisfies the requirements. The set of scenarios that follows serves to de-fine the requirements for the brewery control sys-tem as the subject for the present study. They define the scope of the problem to be modeled using the two development methods. The authors strived to capture as much of the functionality of the system as possible with a reasonable number of scenarios. It is possible to define additional scenarios that to some extent would result in additional contributions to the metrics.
The scenarios are:

- Add to Inventory - An additional amount of an ingredient is added to the stock on hand and the resulting amount is compared to the reorder level.
- Bottle a batch - A batch of beer is in a bottling vat and the bottling line connected to the vat is notified that it is "ok" to begin bottling. When the bottling operation is complete, the batch is marked as bottled.
- Clean a Container - A user notifies the system that a vat or pipe has been cleaned. The new status is remembered by the system.
- Create a Batch - The user instructs the system that a new batch of beer is to be created. The batch will be of an indicated size, made according to a given recipe, and will be mixed in a specified mixing vat. The system must check insufficient amounts of the ingredients are present in the stock on hand, and account for the use of these ingredients.
- Create a Vat - The user adds a Vat and its specifications to the existing system. The user indicates which pipes should connect to the new vat.
- Create a Recipe - The user adds a new recipe to the library. The brewmaster specifies the characteristics of the recipe and indicates which ingredients to use and their amounts for a standard yield.
- Monitor a Batch - Periodically the system monitors each active batch contained in a fermenting vat to verify that its temperature is within a certain range given by its original recipe and its stage of development.
- Record a Daily Reading - Daily, the system records a reading of the characteristics of active batches in fermenting vats. The specific gravity of a batch is supplied by a production person.
- Schedule a Transfer - The user schedules a transfer for a batch to a new vat at a future time. The system checks if the transfer is possible.
- Transfer a Batch - The system will periodically check if it is time to begin a previously scheduled transfer. When a transfer is started, the system will determine the shortest available path, if any, allocate the pipes, and open and close valves to establish a dedicated connection between the source and destination vats. Backfill valves are opened and pumps are started if appropriate. When the transfer is completed, the pumps are stopped; the connection is closed; and the pipes and vats are marked as dirty.

3.4 Description of the Metrics

In order to compare the two designs, the set of six software metrics for object-oriented design proposed by Chidamber and Kemerer [28] was used. These metrics are based on measurement theory, reflect the viewpoints of experienced object-oriented software developers, and are not biased to any particular approach to OOSD. To date, these are the only metrics that directly measure the complexity of the features of object-oriented software. The basic set of metrics is:

- Weighted Methods per Class (WMC) - The complexity of a class is given by the complexity of its attributes and its methods. WMC is the sum of the complexities of the methods of a class. The complexity of individual methods can be measured by cyclomatic complexity, lines of code, or as in this study it can be considered unity.
- Depth of Inheritance Tree (DIT) - The deeper a class is in the inheritance hierarchy, the greater the number of methods it will inherit, making it more complex. DIT for a class is defined as the number of its ancestor classes
- Number of Children (NOC) - NOC for a class is the number of its immediate sub-classes. This is an indication of the potential influence a class can have on the system.
- Coupling between objects (CBO) - Coupling can exist between classes that are not related through inheritance. One class is coupled to another if its methods use the methods or attributes of the other class. CBO for a class is the total number of classes in which such couples exist.
- Response For a Class (RFC) - The response for a class is the sum of the number of its methods and the total of all other methods that they directly invoke. This measures a combination of the complexity of a class through the number of its methods and the amount of communication with other classes.
- Lack of Cohesion in Methods (LCOM) - The cohesion of the methods in a class increases with the degree of their similarity. Methods are more similar if they operate on the same attributes. This metric attempts to measure the degree of similarity by counting the number of dis-joint sets produced from the intersection of the sets of attributes that are used by the methods.

Since it is not certain that the above form a complete set, the following two metrics suggested by the author of the data-driven method were also applied:

- Weighted Attributes per Class (WAC) -Attributes and methods are both properties of a class, and both contribute to its complexity. Whereas WMC, described above, measures the contribution of the methods, this metric measures the contribution of the attributes. WAC is defined as the number of attributes weighted by their size.
- Number of Tramps (NOT) - The signature of a method, including the types and a number of its parameters, gives an indication of its function. Extraneous parameters or tramps, i.e., those not referred to by the body of the method, both in-crease complexity by increasing the number of parameters, and give a misleading indication of the processing done by the method. This metric is defined as the total number of extraneous parameters in the signatures of the methods of a class.

Finally, the following metric derived from the law of Demeter [29], which is an established rule of style for object-oriented software, was included:

- Violations of the Law of Demeter (VOD) - The Law of Demeter attempts to minimize the coupling between classes. If a class follows this law, then its methods can only invoke the methods of a limited set of other classes. This set is composed of the classes of the attributes of the object, the classes of the parameters of the method, or the classes of objects created locally during the execution of the method.

All of the above metrics basically measure complexity. In an object-oriented model, complexity may be defined with respect to the components of object-oriented software as discussed in Section 2. These components are the objects and classes, the interactions via client-server relationships, and the relationships due to inheritance.
Metrics could be based on either objects or classes. Since objects are the components of the system that exist when the system is actually running, metrics based on objects would be influenced by the conditions of execution, and therefore the measurements must be made empirically. The number of objects that are created and the processing that is triggered by the environment would affect the measurements. Furthermore, ob-jects not only exhibit the behavior of their immediate defining class but also that of all their superclasses. Since any given superclass may be shared by objects of many

different subclasses, a great deal of redundancy would inevitably exist in the measurements.
Metrics that are based on classes, do not measure any particular execution of the system but rather produce a measurement that applies to all possible scenarios of execution. Since classes are formal descriptions, measurements can be produced deductively from an analysis of the design of the system. Whereas objects are the components that exist at run-time, classes are the components that exist during maintenance and are the units of extension and reuse.
Therefore, since we are interested in producing a comparison of two designs for a system, especially with respect to the complexity that affects maintenance, extension, and reuse, then metrics based on class descriptions are to be preferred. Then, for the purposes of this study, the complexity of object-oriented software can be given by the complexity of the classes, the complexity of the interaction of classes via client-server relationships, and the complexity of the relationships between classes as given by the inheritance hierarchy.Table 3.4-1 lists each metric with the corresponding component that it measures.

3.5 Collection of the Metrics

The values for the metrics were collected by analyzing the interaction of classes through which each of the designs modeled the scenarios of Section 3.3. For details see [25].

4 Results

The metrics described in Section 3.4 were applied to both designs of the brewery control system in the manner outlined in the previous section. In the following sections, the metrics are summarized for the two designs. Those metrics that measure coupling and cohesion are discussed in detail, and the complexity of the inheritance hierarchy is examined. Some additional measures for the scenarios are also discussed; and finally, some causes for the differences are investigated.

4.1 Summary of the Metrics

As discussed in Section 3.4, the metrics were collected for each class of each design. Table 4.1-1 shows the totals of each metric for the designs when the values for their individual classes are summed. Summing over the classes is consistent with the use of metrics that are based on classes as described in Section 3.4.
The value for NOT for both designs is zero, and the responsibility-driven design also had a value of zero for VOD. The differences shown by the measurements are striking. Overall, the data-driven design is decidedly more complex than the responsibility-driven design.
To examine how this complexity can be attributed to the three components of class, interaction, and inheritance, the measurements are also shown in Figure 4.1-1. In this figure, the metrics are first grouped according to the components that they measure. The axes for the metrics are arranged radially, and the value for each metric is plotted along the respective axis. The values are then connected to form a polygonal area.
The polygon formed from the values for the re-sponsibility-driven design is shown superimposed on the figure. In this manner, Figure 4.1-2 shows what might be called a "relative complexity signature" for the two designs. As shown in the figure, the component with the greatest difference is the complexity of the interactions (RFC, VOD, CBO), followed by the complexity of the classes (RFC, LCOM, WAC, WMC), and finally the complexity of the inheritance relationships (D1T, NOC). The first component includes two measures of coupling, and the second includes a measure of cohesion.

4.2 Coupling and Cohesion

The greatest differences in the measures for the two designs are for VOD, CBO, and LCOM.VOD measures the coupling of a class to the structure of a composite class, while CBO measures the coupling of a class to the interface of another class. LCOM measures the lack of cohesion among the methods of a class, or how many unrelated activities a class is performing.

A high value for VOD implies that the overall system is highly dependent on the structure of the composite classes. Changes to the composition of classes should have effects that are limited to the class itself and possibly its subclasses. However, the relatively high value of VOD for the data-driven design implies that changes to the composition of a class will have wide-ranging effects throughout the system.

Values for LCOM and CBO are directly related to the encapsulation of objects. As discussed in Section 2, encapsulation of intelligence is one of the two fundamental principles of object-oriented development. An object with good encapsulation has all the information it needs to carry out its natural activities and does not need to ask for basic information from other objects. Also, a well-encapsulated object does not perform services that are unnatural to it, i.e., foreign to its intrinsic nature.

When an object has poor encapsulation, it may have to acquire information from objects of other classes. The number of these other classes contributes to the value of CBO. A poorly encapsulated object may also be keeping information for other objects, rather than for its own purposes. Each distinct grouping of such foreign information contributes to the value of LCOM.

Thus the values for LCOM and CBO may im-ply differences in the support for encapsulation, a fundamental principle of OOSD. An example from the brewery system may help to understand how these very different values for LCOM and CBO arise.

"Transfer a Batch" is the title of the final scenario described in Section 3.3. As specified, this scenario involves many actions happening in concert. Valves must be opened and closed, pumps started and stopped, and the shortest clean path through the brewery must be found. This last action involves the most complex algorithm in the system. Both designs model this scenario, and two similar sets of classes carry out the major activities. Table 4.2-1 shows the correspondence between these roughly similar classes from the two designs. The corresponding classes are not identical, however, as can be seen from examining the communications in which they are involved.

Figures 4.2-1 and 4.2-2 show the paths of communication among the various classes for the data-driven and the responsibility-driven brewery designs respectively. Messages are sent between classes in the direction of the arrow. The information in these figures is abstracted from the detailed descriptions of these scenarios which can be found in Appendices A and B. These figures also show the location of the data and the algorithms for finding the shortest clean path. The names of the classes that contain the necessary data are circled in the figures, while the names of classes that contain parts of the algorithm appear in boxes.

Figure 4.2-1 shows that for the data-driven design, there is one class, Scheduled Transfer, that communicates with all seven other classes involved in the scenario. This class contains the algorithm for finding the shortest clean path but does not have the necessary data which instead must be acquired by communicating with Container, Interconnect, and Pipe. Similarly, Scheduled Transfer interrogates In Process Batch and Vat to determine which Valves and Pumps to control, and then opens and closes the Valves and starts and stops the Pumps. Scheduled Transfer could be said to be "controlling" other"controlled" classes.

In contrast, Figure 4.2-2 shows that in the responsibility-driven design the classes Manifold and Valve each have part of the algorithm and part of the necessary data. The communications necessary for opening and closing valves and starting and stopping pumps has also been distributed among the set of classes.

These features of the two designs directly correlate to the values for LCOM and CBO for each class, which are shown in Figure 4.2-3. Classes such as Scheduled Transfer that control other classes exhibit a correspondingly high value of CBO, while classes that are being controlled have a high value of LCOM.

Controlling classes must acquire necessary information from other classes and controlled classes keep information for other classes to use. Thus the presence of "controlling" and"controlled" classes is essentially a manifestation of the poor encapsulation of those classes. Therefore, the high values of LCOM and CBO for the data-driven design are seen to derive from an apparent lack of support for the principle of encapsulation in the data-driven method. Therefore, the high values of LCOM and CBO for the data-driven design are seen to derive from an apparent lack of support for the principle of encapsulation in the data-driven method.

4.3 Complexity in the Treatment of the Scenarios

As mentioned in Section 3.5, a step in the process of compiling the class-based metrics discussed above is the modeling of the communications among the classes for each scenario. The amount of communication required to model a scenario is an indication of the complexity of a design in its treatment of that scenario.
Figure 4.3.1 shows the total communications for the scenarios of Section 3.3, for the designs derived from both methods. This figure shows that the data-driven design requires significantly more class-to-class communications to model each of the ten scenarios.
A large amount of class-to-class communication is a symptom of poor organization of data and processing. When an activity and the information necessary to perform it are not collected together into a single object, the result is an increase in the number of messages that request information from other objects.
For the scenario entitled "Transfer a Batch," there is a ratio of more than two to one for the communications of the data-driven model compared to the responsibility-driven one. This correlates well with the lack of support for encapsulation in the data-driven method as discussed in the previous section.

4.4 Complexity in the Inheritance Hierarchy

The data-driven design has a relatively high value of NOC compared to its value of DIT and compared to the value of the responsibility-driven design. NOC is a measure of the breadth of the inheritance hierarchy, and D1T is a measure of its depth. Generally, it is better to have depth than breath, since this promotes reuse and reduces redundancy in the system.
A high value of NOC indicates that classes high up in the inheritance hierarchy can potentially influence a large number of other classes. This increases the amount of testing required to verify the system originally and makes it much more difficult to safely modify the system later.

4.5 Causes of the Differences

Focusing on an object's responsibilities in the responsibility-driven method seems to provide a natural grouping of data and processing. Objects built around responsibilities automatically contain both processing and the data necessary for the processing. Specifying responsibilities follows and reinforces the principle of encapsulation.
In contrast, the data-driven method is actually composed of several separate models of the internal characteristics of classes. It has been pointed out elsewhere [20] that methods of this kind"...deteriorate when they try to integrate the different views into an object-oriented design." The type of method provides no rationale for the grouping of data and processes, and the individual internal models ignore, and their use actually de-feats, the principle of encapsulation.
Responsibilities are external features of a project. Describing objects in terms of responsibilities leaves internal details unspecified until very late in the design process. This discourages the formation of other objects that are dependent on that detail. The data-driven method describes an object primarily in terms of internal details, beginning with the data attributes. This encourages the formation of highly interdependent objects that have high coupling and low cohesion.
In the data-driven method, the data attributes of classes are the primary basis for the construction of the inheritance hierarchy. Responsibilities, which in a sense combine data attributes and methods, provide a more complete basis for inheritance and allow inheritance to better express the principle of classification.
The differences in the measurements summarized above are seen then to derive from differences in the fundamental nature of each approach. The responsibility-driven approach intrinsically supports the principles of encapsulation and classification, while the data-driven approach inherently defeats encapsulation, and can give only limited support to classification.