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

Robert C. Sharble†
Samuel S. Cohen

Boeing Computer Services
PO Box 24346, Mailstop 6M-23
Seattle, Washington 98124-0346
email: shabcg00@ccmail.ca.boeing.com
cohen@kgv1.profs.ca.boeing.com

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

In the 1980's object-oriented design (OOD) first evolved as a formalization of object-oriented programming (OOP). Many complete methods of software development have since arisen, each claiming to be object-oriented. These methods have different approaches, come from diverse origins, and are supported by distinct communities of experts.

Two approaches to the development of object-oriented software have gained wide acceptance in the past few years, responsibility-driven methods which evolved from object-oriented programming, and data-driven methods which arose from structured analysis and design methods and from methods used in the database world [1].

In this article these two approaches to object-oriented development are described and compared. Both methods were used to develop models of a single system, and the models were analyzed using metrics that have been devised for object-oriented software.

2. Object-Oriented Software Development

By object-oriented software development (OOSD) is meant the process of developing object-oriented software, and it may be considered to be composed of the processes of object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). (Here OOP is used to denote the implementation phase of development. However, historically, the term refers to the entire development process). There are various approaches to OOSD each giving rise to distinct methods, some formal and others informal. Of course, all such approaches are based on the concept of an object, but they may differ in their ability to employ that concept to improve software quality.

The very term "object-oriented" seems to have various interpretations. Before presenting an overview of the goals, strategies, and underlying principles of OOSD, some definition for this term must be given.

Most researchers accept Wegner's definition [2] for object-oriented languages, and this may be adapted for OOSD. OOSD will ultimately pro-

†Member of the ACM, SIGSOFT, SIGPLAN, and the IEEE Computer Society

Any opinions expressed or implied are those of the authors alone, and not necessarily those of The Boeing Company.

duce a model of a system. The components of an object-oriented model are objects, which are grouped into classes for specification purposes. In addition to familiar client-server dependencies, inheritance relationships exist between classes and serve to express the generalization and specialization of the concepts represented by the classes.

From the beginning, the object-oriented paradigm was expected to revolutionize the software development process, to make the process more natural and to make its products easier to understand. The concepts of objects and classes were considered more "natural" than data and processes. Smalltalk itself was originally intended as a programming language for children to use. Today the purpose of OOSD remains the same, i.e., to be a natural and coherent process, and to produce software that is easy to understand. The recognized achievement of OOSD is the production of software that is less complex, and is therefore easier to maintain and extend, and can be more easily reused.

A software system is *maintainable* if it can be modified to adapt to external changes, such as changes in hardware or operating systems, or to correct deficiencies and improve performance. To achieve a high degree of maintainability, a system should have weak coupling between parts and have interactions that rely only on the external interfaces of those parts.

If a system can be changed to support changes to its original purpose, then it is *extendible*. To facilitate this it should be possible to add new components and to capture and isolate changes to existing components. Also, it is important that references to components can be readily updated throughout the system.

A system is *reusable* if its parts or its overall design can be utilized to build a new system. Reusability is enhanced when each component is designed around a single purpose. A reusable design should define generalized patterns for the interaction of its components.

To achieve these qualities, OOSD relies on four strategies to reduce complexity. These are: increase cohesion, reduce coupling, increase polymorphism, and eliminate redundancy.

Uniting together related parts of a system into a single component increases cohesion for that component and for the system as a whole. A component with high cohesion is easier to understand and more likely to be reused. A system of cohesive components is easier to understand and simpler to maintain.

Interaction between dissimilar parts of a system increases coupling. Coupling binds components together. Decreasing coupling frees components, thus increasing the feasibility of extending the system. Decreasing coupling also simplifies maintenance by isolating the effects of changes to components. By reducing the coupling of objects in a system, the system becomes easier to extend and maintain.

Polymorphism is defined as the ability to take different forms. For example, a procedure with polymorphic formal parameters, can have different types of actual parameters each time it is called. In addition, a polymorphic reference can point to different types of components at different times. Increasing polymorphism makes it easier to extend a system.

When the same activity is described or the same concept defined in many different places in a system, it becomes difficult to modify the activity or concept. Reducing redundancy in a system makes it easier to extend and maintain, and also makes it easier to understand.

These strategies are united and directed by two basic principles of the object-oriented paradigm, the principles of *encapsulation* and *classification.*

As mentioned earlier, OOSD has its roots in object-oriented programming and has evolved bottom up from programming to design and finally to analysis. Central to OOP was the principle of encapsulation as defined by Tim Rentsch [3]:

> "The first principle of object oriented programming might be called intelligence encapsulation: view objects from outside to provide a natural metaphor of intrinsic behavior."

As illustrated in Figure 2-1, an object encapsulates intelligence, which is ultimately represented in software by data and processes. An object-oriented model is built from objects that are constructed in a manner such that each object is viewed from the outside and each is understood in terms of its intrinsic behavior which is expressed in terms of the services that it provides. Intrinsic behavior denotes behavior that derives from the very nature of the object, i.e., behavior that is natural to it. As a corollary, an object would not be

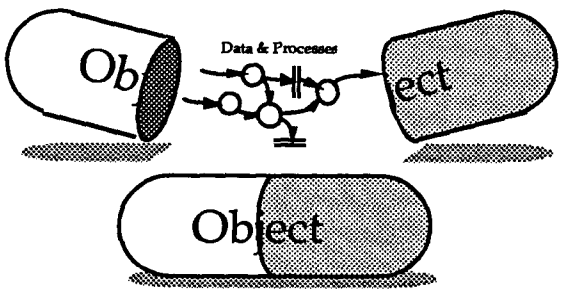

Figure 2-1 An Object encapsulates Intelligence

called upon to perform actions that would be unnatural to it.

The principle of encapsulation drives the creation of classes. A class made following this principle represents a single concept in the problem domain and has a well-defined and cohesive purpose in the model domain. This principle increases the cohesion of classes and reduces the coupling between them.

Historically, inheritance was introduced into programming in SIMULA I where its purpose was to model the principle of classification, i.e., the arrangement or distribution of classes according to the generalization and specialization of behavior. This principle drives the creation of inheritance structures where behaviors that are shared by a group of classes are described only once, in a class that is a common ancestor to each class in the group. Adherence to this principle reduces redundancy, while preventing the formation of indiscriminate inheritance relationships between classes. When inheritance is used indiscriminately, it can severely compromise the benefits of encapsulation [4].

There are currently four main approaches to object-oriented development: data-driven, process-driven, event-driven, and responsibility-driven. These approaches are named for the particular characteristic of objects on which they initially concentrate. Data-driven methods start with the data encapsulated by an object. Process-driven methods start with the processes encapsulated by an object. Event-driven methods start with the states encapsulated by an object and the events that can trigger changes in those states. Finally, responsibility-driven methods start with the responsibilities that each object has as part of the system, helping to carry out the overall purpose of the system.

Each of these approaches differs in its expression of and support for the principles of encapsulation and classification and may introduce additional principles from outside the object-oriented paradigm. The methods based on these approaches can also be expected to differ in the effectiveness of their use of the strategies described above.

In recent years both the data-driven approach and the responsibility-driven approach have received increasing interest and use at The Boeing Company. Boeing currently offers a course on OOSD [5] which describes a data-driven method [6,7,8,9] and has arranged for courses to be taught in responsibility-driven OOD [10].

Data-driven methods include those that evolved from abstract data-type analysis and design as practiced by the Ada and Modula-2 communities (e.g. [11]), as well as the methods that come from the relatively new field of object-oriented domain analysis (e.g. [12]).

The responsibility-driven approach began as a description of the processes that evolved from building commercial Smalltalk systems and applications at Tektronix [13, 14, 15]. Recent work [16] has extended this approach, especially in the use of scenarios to elucidate responsibilities and ensure traceability.

In the following sections, data-driven methods are represented by the methodology described in [6]. Discrepancies between this and other methods will be noted where applicable. The responsibility-driven method is that described in [17].

2.1 Object-Oriented Analysis

Analysis and design are two fundamental parts of software development. Simply put, analysis says what the system is supposed to do, and design says how it is supposed to do it. The goal of analysis is to describe a real-world system that will be modeled in software. Object-oriented analysis abstracts from concepts in the real-world (the problem-domain) to form objects as parts of an object-oriented model (in the model domain). Classes are formed from these objects through *generalization.*

> "Generalization is the act or result of removing certain types of distinctions between types of objects so that we can see commonalities." [18]

Analysis can involve two views of a system, an external view and an internal view. The external view describes objects in terms of their externally visible activities, while the internal view describes the inner-workings of the objects that enable the activities.

2.1.1 Data-Driven Analysis

The data-driven approach combines separate models of a system using traditional techniques. Data-driven object-oriented analysis consists of four different models that taken together provide both an internal view and an external view of the objects in a system. The four models are: the information model, the state model, the process model, and the communication model. Figure 2.1.1-1 shows what features of a system are detailed in these models for both views.

The cornerstone [1] of data-driven object-oriented analysis is information modeling [19], which originated from the study of database systems. This model describes the relationships between objects and the data, or attributes, contained

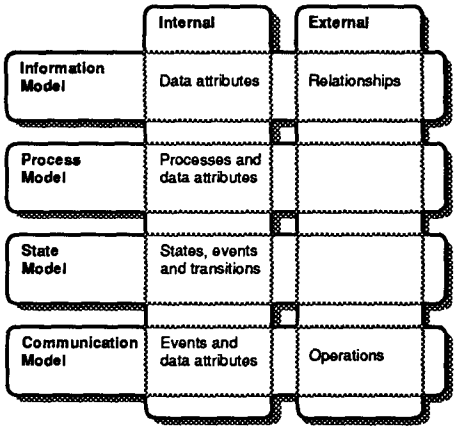

Figure 2.1.1-1 Features that correspond to the Intersections of the four Models and the two views of Data-Driven OOA

within each object. Inheritance is represented as a relationship, with subclasses formed by adding to the attributes of their superclasses.

The information model is supplemented by the process model and the state model. The process model describes the internal activity of each object in terms of internal processes acting on the data attributes of the object itself or on the attributes of other objects in the system.

The state model uses states, events, and the transitions between states triggered by the events, to describe the internal mechanisms that underlie the reactions of an object to its environment. Events can be generated from within the object or can originate in other objects. Similarly, an event can have purely local effects confined to the object that generated it, or can affect other objects in the system.

Finally, the communication model is created to describe the global or system level view of the interactions between objects. Objects request the value of attributes or communicate the occurrence of events by invoking externally visible operations defined for other objects. Operations can also provide the results of a computation based on the attributes of an object [11].

Each of the models details some internal aspect of an object (its data, its processes, or its states). Together they constitute a detailed internal view as well as an overall, external view of the objects in the system that can be checked for coherence, consistency, and completeness.

2.1.2 Responsibility-Driven Analysis

The responsibility-driven approach is a "pure" object-oriented approach to software development [20]. In responsibility-driven analysis the goal is:

> "...first, to understand the problem description and, second, to formulate this description in terms of multiple interacting objects. These objects fill system roles and responsibilities by both providing and contracting for well-defined services that carry out system behaviors." [16]

The major tasks are: (The tasks are iterative, but are presented sequentially to aid in explanation.)

- Determine the overall responsibilities of the system and how the system is used.
- Discover the objects required to model the system.
- Assign individual responsibilities to specific objects.
- Determine the collaborations between objects that are necessary to carry out the assigned responsibilities.
- Define classes of objects based on the responsibilities they share and specialize.

Scenarios can be used to help accomplish the first task, and can be defined with the aid of interviewing techniques. The descriptions of the scenarios are the raw material for the accretion of objects. As depicted in Figure 2.1.2-1, related responsibilities, which are derived from the scenarios are grouped together, and are assigned to objects. The objects must be able to support the responsibilities by virtue of their natural behaviors.

The responsibilities and collaborations of an object, together with its class name combine to define its role in the system. As illustrated in Figure 2.1.2-2, these items form the three dimensions with respect to which the role of an object can be completely mapped. Objects provide a model of a system in terms of their externally visible roles. This "object" model expresses the application of the principle of encapsulation.

Objects that have the same role are also instances of the same class. The responsibilities thus

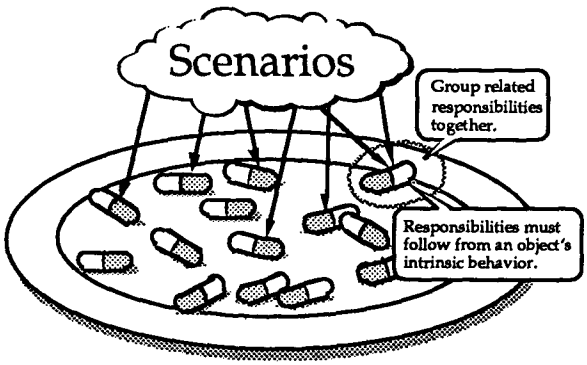

Figure 2.1.2-1 Objects are created to Support Responsibilities

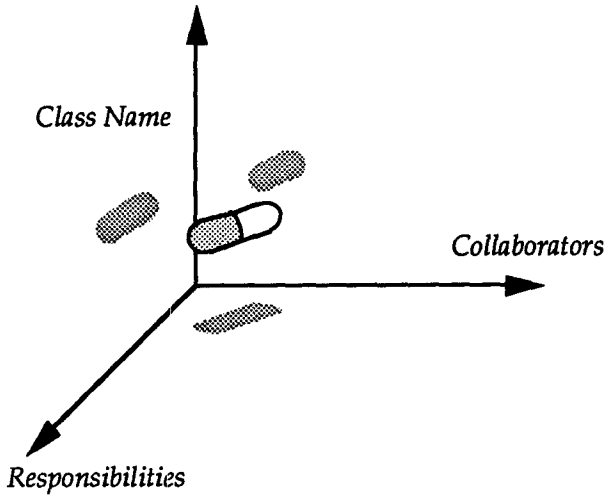

Figure 2.1.2-2 An Object's Role is Described in Terms of Its Class Name, Its Responsibilities, and its Collaborators

derived for a class are the basis for the application of the principle of classification. Subclass and superclass relationships are defined to correspond to generalization and specialization of responsibilities across classes. These relationships form a "class" model of a system. This model expresses the application of the principle of classification.

Thus the two models utilized in the responsibility-driven method correspond to and reinforce the two fundamental principles of OOSD. Figure 2.1.2-3 summarizes these two models in terms of the features of a system that they describe. In contrast to data-driven OOA, responsibility-driven analysis describes a system only in terms of features that are visible from outside the objects. Internal features are intentionally absent. Since collaborations are determined by responsibilities, and responsibilities form the basis of inheritance, the definition and assignment of responsibilities remains the continuing focus of the method.

Together, the objects and their classes establish

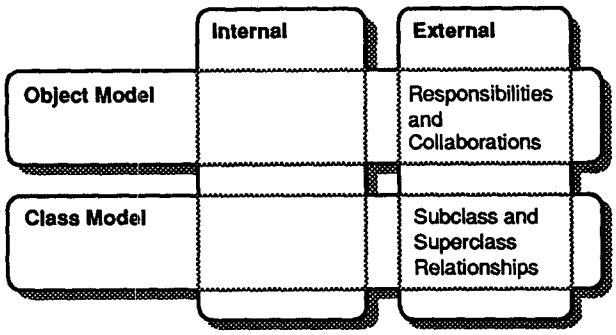

Figure 2.1.2-3 Features that Correspond to the Two Models and the Two Views of Responsibility-Driven OOA

a vocabulary for discussing the system. The scenarios that specify the purpose of the system can be reexamined and explained in terms of this vocabulary. By modeling the scenarios in detail, and performing walk-throughs, the object and class models can be checked for coherence, consistency, and completeness.

2.2 Object-Oriented Design

Object-oriented design is mainly the process of analyzing the model produced during analysis; optimizing it to reduce coupling, increase cohesion, increase polymorphism; and factoring common behavior into an inheritance hierarchy. In addition, the model is refined and formalized. Additional classes may be defined to address issues that arise simply as a result of constructing a software model, such as initializing the run-time system, saving the state of the system in a file, using external libraries, etc.

2.2.1 Data-Driven Design

Data-driven object-oriented design as articulated by Peter Coad [21], and others [22, 23, 24], is the systematic addition of design-dependent detail to the OOA results. The classes created in analysis pass through design essentially unchanged except for modifications to improve performance or compensate for a lack of language support. There are additional requirements to address, such as: hardware allocation, interfaces to system resources and devices, and human-interface displays and controls.

2.2.2 Responsibility-Driven Design

In the responsibility-driven approach design entails the redefinition of objects, streamlining of collaborations, the redistribution of responsibilities, the creation of hierarchies, and the formalization and refinement of responsibilities.

Subsystems of objects are identified. Subsystems are groups of objects that collaborate to fulfill a common set of responsibilities. These responsibilities are delegated to individual objects. The number of objects that directly provide services to objects outside the subsystem is minimized.

Responsibilities are redistributed and collaborations redesigned and refined to improve cohesion and reduce coupling. For each class cohesive sets of responsibilities are grouped into formal client-server contracts. The number of different contracts supported by each class is minimized as are

the number of other classes with which each class collaborates.

Classes are arranged in an inheritance hierarchy to model generalization and specialization of responsibilities. Abstract classes are defined, and common responsibilities are moved as high in the hierarchy as possible.

Finally, individual responsibilities are refined into methods and attributes. The signatures (name and parameter types) of methods that perform similar functions are made as similar as possible to increase polymorphism.

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 be 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 executing at a given time, although other methods which 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 variables, 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 define the requirements for the brewery control system 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 if sufficient 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] were used. These metrics are based in 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 an 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 to 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 disjoint 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 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 increase 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 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, objects 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, espe-

Table 3.4-1 Metrics and the Components of Complexity that they Measure

Metric	Component
WMC	Class
DIT	Inheritance
NOC	Inheritance
CBO	Interaction
RFC	Class and Interaction
LCOM	Class
WAC	Class
NOT	Interaction
VOD	Interaction

cially 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 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 component 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 responsibility-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 complexity of the classes (RFC, LCOM, WAC, WMC), and finally the complexity of the inheritance relationships (DIT, 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 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-

Table 4.1-1 Totals of the Metrics for Both Designs

	WMC	DIT	NOC	CBO	RFC	LCOM	WAC	NOT	VOD
Data - Driven	130	21	19	42	293	64	64	0	40
Responsibility - Driven	71	19	13	20	127	21	44	0	0

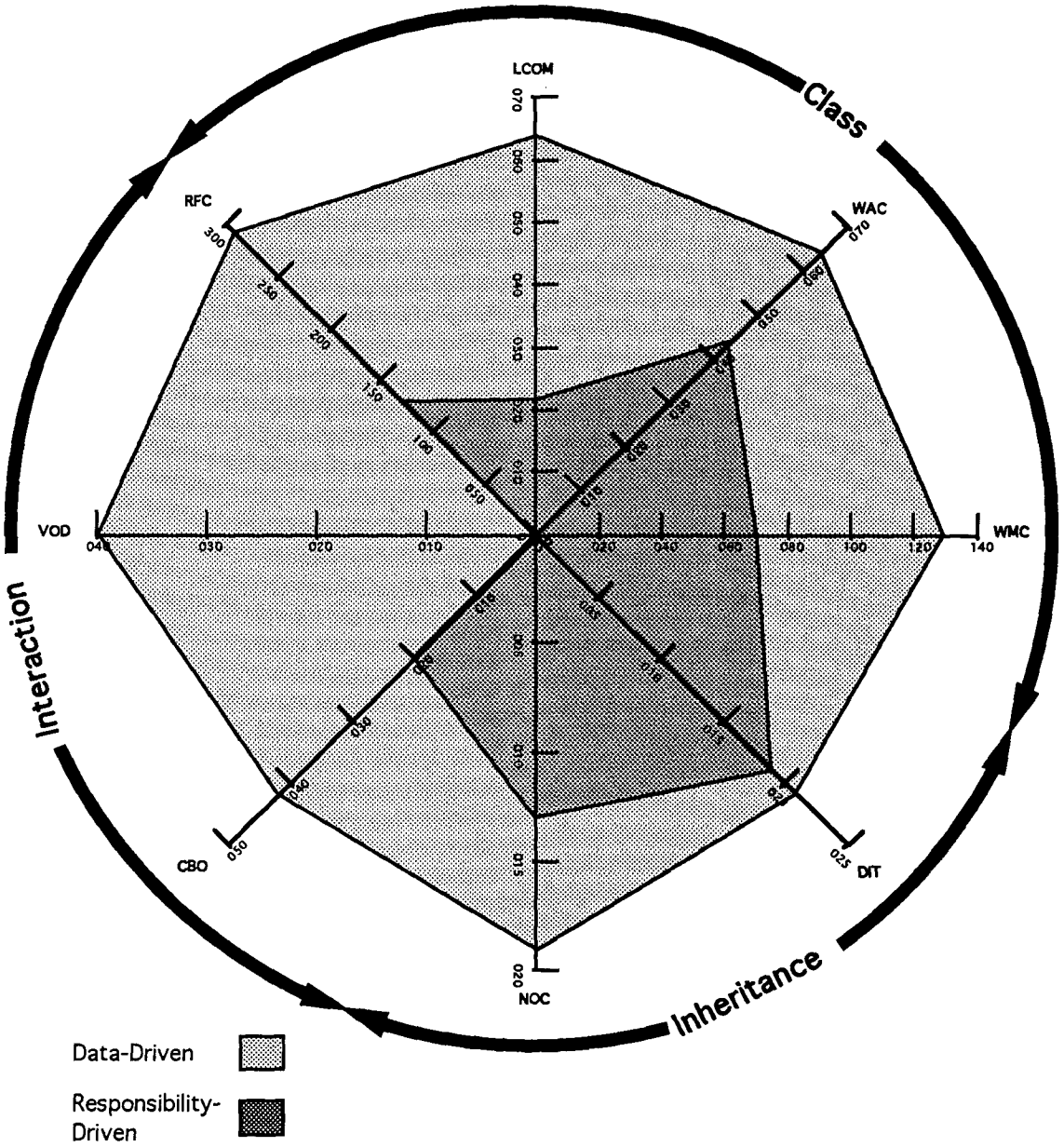

Figure 4.1-1 The Metrics arranged According to the Components of Complexity

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 imply differences in the support for encapsulation, a fundamental principle of OOSD. An example

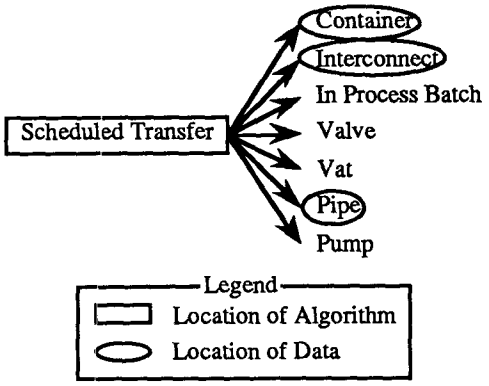

Figure 4.2-1 Transferring a Batch in the Data-Driven Design

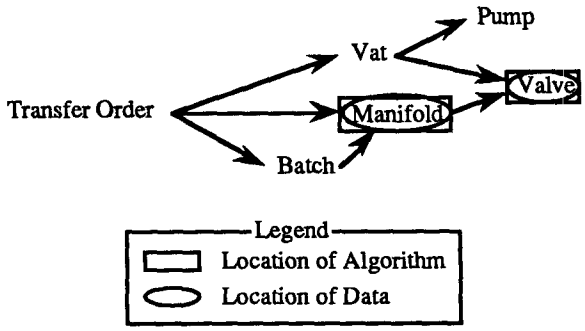

Figure 4.2-2 Transferring a Batch in the Responsibility-Driven Design

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.

Table 4.2-1 Corresponding Classes in the Two Designs

Responsibility-Driven		Data-Driven
Transfer Order	↔	Scheduled Transfer
Manifold	↔	Container
Vat	↔	Vat
Valve	↔	Valve
		Interconnect
Batch	↔	In Process Batch
Pump	↔	Pump
Pipe	↔	Pipe

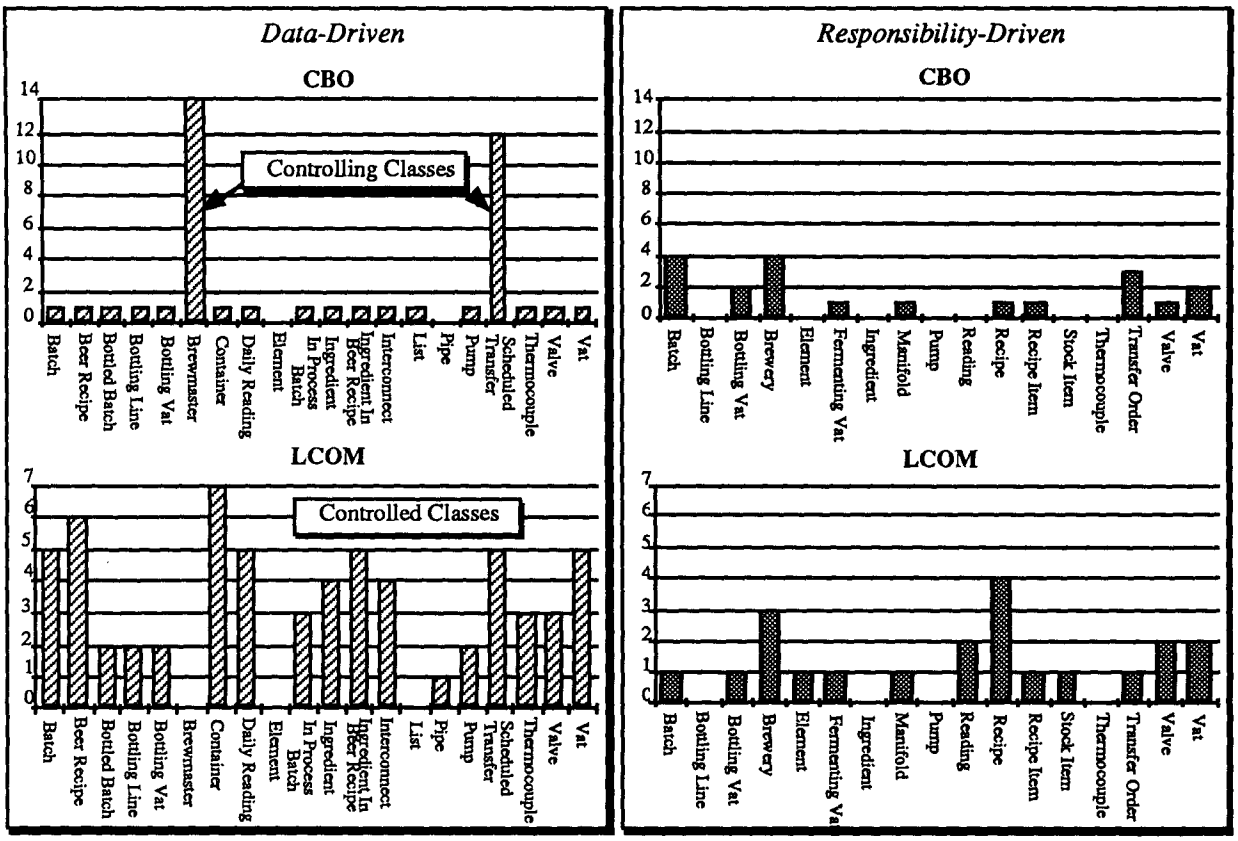

Figure 4.2-3 Measurement of Class Cohesion and Coupling for both Versions of the Brewery

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

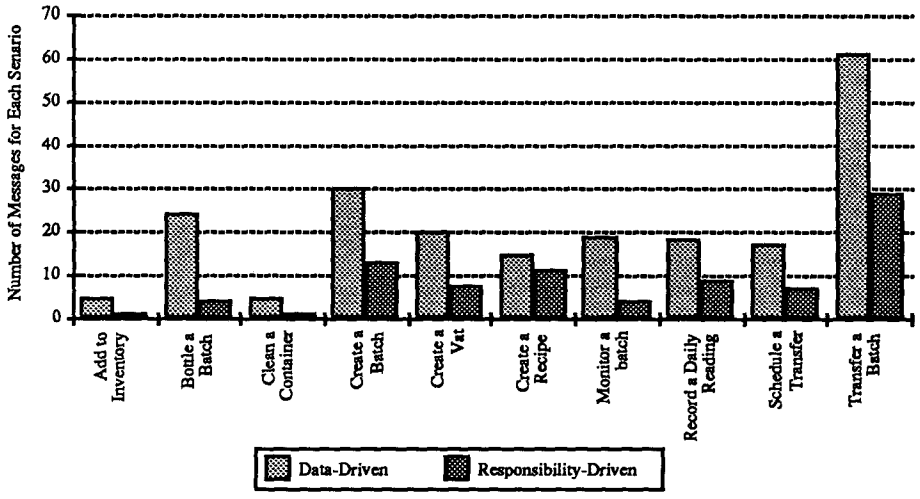

Figure 4.3-1 Total Communications for the Scenarios

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 communications 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 for the responsibility-driven design. NOC is a measure of the breadth of the inheritance hierarchy, and DIT 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." This type of method provides no rationale for the grouping of data and processes, and the individual internal models ignore, and their use actually defeats, the principle of encapsulation.

Responsibilities are external features of an object. 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.

5. Conclusion

The responsibility-driven method was shown to produce a design which was much less complex than that produced by the data-driven method. In particular, the responsibility-driven design exhibited much greater cohesion of classes and much less coupling between them. Also, the responsibility-driven method produced a better organized inheritance hierarchy, with greater potential to reduce redundancy.

When examined in detail, it was seen that the data-driven method failed to support the principle of encapsulation, as evidenced by its creation of "controlling" and "controlled" classes. Also it was found that this method only provides limited support for the principle of classification. Furthermore, it was shown that the failure to support these fundamental principles was inherent in the data-driven approach itself.

With respect to every one of the metrics (with the exception of NOT, which was zero for both designs), the data-driven design was shown to be more complex. Based on this data, the data-driven method therefore appears to have no advantages when compared to the responsibility-driven method. Moreover, since the concepts and basic procedures of the data-driven method seem to be the sources of its poor performance (as discussed in Section 4.5), and are fundamentally different from those of the responsibility-driven method, only additional complexity could be expected to

result from an attempt to combine the features of the two methods.

In summary, the responsibility-driven method produced a much less complex design and exhibited stronger support for encapsulation and classification, greater ability to increase cohesion and reduce coupling, and better use of inheritance to reduce redundancy. The responsibility-driven approach seems to be significantly more effective for the production of software that can be maintained, extended, and reused.

Acknowledgements

The authors wish to express thier thanks to Amad Yaghoobi and Mike Armstrong for reviewing earlier versions of this paper, and especially to Maxine Ogino for her continuous and invaluable support.

References

[1] S. Shlaer and S. Mellor, "Understanding Object-Oriented Analysis", Design Center Magazine, Hewlett-Packard Company, Jan. 1989

[2] P. Wegner, "Dimensions of Object-Based Language Design", *OOPSLA'87* Conference proceedings, published as a special issue of *SIGPLAN Notices*, 22(12), Dec. 1987, pp. 168-182

[3] T. Rentsch, "Object Oriented Programming", *SIGPLAN Notices*, 17(9), Sept. 1982, p. 51

[4] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", in *OOPSLA'86 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 21(11), Nov. 1986, pp. 38-45

[5] *An Object Oriented Approach to Software Development*, Course #GE-B0435, Boeing Computer Services, Seattle, Washington, 1991

[6] M. K. Smith and S. R. Tockey, "An Integrated Approach to Requirements Definition Using Objects", *Proceedings of the 10th Structured Development Forum*, Aug., 1988

[7] S. Tockey, "Using Off-the-Shelf CASE Tools to Build Object Oriented Analysis Specifications", *Proceedings of the 11th Structured Development Forum*, May 1990

[8] B. J. Hoza, M. K. Smith, S. R. Tockey, "An Introduction to Object Oriented Analysis", *Proceedings of the 5th Structured Techniques Association*, May, 1989

[9] S. Tockey, B. Hoza, S. Cohen, "Object Oriented Analysis: Building on the Structured Techniques", *Proceedings of the Software Improvement Conference*, Nov. 1990

[10] *Designing Object-Oriented Software: An Introduction*, Digitalk, Portland, Oregon, 1992

[11] E. Seidewitz and M. Stark, *Principles of Object-Oriented Software Development with Ada*, Millennium Systems, 1992

[12] S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988

[13] J. J. Ewing, "An Object-Oriented Operating System Interface", in *OOPSLA'86 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 21(11), Nov. 1986, pp. 46-56

[14] M.S. Miller, H. Cunningham, C. Lee, S.R. Vegdahl, "The Application Accelerator Illustration System", in *OOPSLA'86 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 21(11), Nov. 1986, pp. 294-302

[15] R. J. Wirfs-Brock, "An Integrated Color Smalltalk-80 System", in *OOPSLA'88 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 23(11), Nov. 1986, pp. 71-82

[16] K. S. Rubin and A. Goldberg, "Object Behavior Analysis", *Communications of the ACM*, (35)9, Sept. 1992, pp. 48-62

[17] R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990

[18] J. Odell, "Managing Object Complexity, Part I: Abstraction and Generalization", *JOOP*, 5(5), Sept. 1991, pp. 19-22

[19] P. Pin-Shan Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, Vol. 1, March 1976

[20] D. E. Monarchi and G. I. Puhr, "A Research Topology for Object-Oriented Analysis and Design", *Communications of the ACM*, 35(9), Sept. 1992, pp. 35-47

[21] P. Coad, "OOA & OOD: A Continuum of Representation", *JOOP*, February 1991, pp. 55-56

[22] S. Shlaer and S. Mellor, "Recursive Design and its Effect on the Project Life Cycle", Project Technology 1989

[23] R. Hill, "Object-Oriented Design in Ada: A Transformational Approach Based on OOA", Project Technology, 1989

[24] R. J. Norman, "Object-Oriented Systems Design: A Progressive Expansion of OOA", *Systems Management*, Aug. 1991

[25] R. C. Sharble, S. S. Cohen, Boeing Company Report #BCS-G4059, The Boeing Co., Seattle, Wa., 1992

[26] G. A. Pascoe, "Elements of Object-Oriented Programming", *BYTE*, 11(8), Aug. 1986, pp. 139-144

[27] Capt. H. Holbrook II, "A Scenario-Based Methodology for Conducting Requirements Elicitation", *Software Engineering Notes*, 15(1), Jan. 1990, pp. 95-104

[28] S. R. Chidamber and C. F. Kermerer, "Towards a Metrics Suite for Object Oriented Design", *OOPSLA'91 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 26(11), Oct. 1991, pp. 197-211

[29] K. Lieberherr, I. Holland, and A. Riel, "Object-Oriented Programming: An Objective Sense of Style", *OOPSLA'88 Conference Proceedings*, published as a special issue of *SIGPLAN Notices*, 23(11), Nov. 1988, pp. 323-334