Interaction and Object Modeling

Scott M. Lewandowski
scl@cs.brown.edu
ScottMLew@aol.com

Final Paper
CS273, Fall 1996

Abstract

Interactive object technology is replacing traditional models of computation that are based on the notion of strict algorithmic expression. Interacting objects provide for more expressiveness within a system than reliance on algorithms alone can. This transition is of paramount importance in light of the rapidly increasing importance of distributed and cooperative computing.

After a discussion of what algorithms, objects, and interaction are, the fundamental difference between objects and algorithm is discussed, with a special emphasis on the greater degree of expressiveness possible using interaction. Interfaces are then treated and the relationship between interaction and object models is established. The Object Modeling Tool (OMT) proposed by Rumbaugh is described and its treatment of interaction is discussed. This leads to a discussion of design patterns, with a focus on granularity, composition, specification, and frameworks.

After discussing the relationship between OMT and interaction and how patterns of interaction and design patterns are related, a new perspective on the object model is presented. Furthermore, a proposal as to how to optimize and improve the object model through the use of a new graph is presented. Finally, the necessity of object models is discussed.

Table of Contents

1. ALGORITHMS VS INTERACTION 4

1.1 What Is An Algorithm? 4

1.2 What Is An Object 5

1.3 What Is Interaction? 7

1.4 Why Objects Are Not Algorithms 8

1.5 Why Interaction Is More Expressive Than Algorithms 10

2. INTERFACES 12

2.1 What Is An Interface 12

2.2 Interfaces And Object Modeling 13

3. INTERACTION AS A TOOL FOR OBJECT MODELING 14

3.1 How They Are Related 14

3.2 Components of the Object Model 15

4. OBJECT MODELING TOOL (OMT) 15

4.1 Description 16

4.2 A Three Level Model 17

5. DESIGN PATTERNS 18

5.1 What is a Design Pattern? 18

5.2 Granularity 19

5.3 Composition 21

5.4 Specification 23

5.5 Frameworks As Extended Design Patterns 25

6. CONCLUSION 26

6.1 Design Patterns and Patterns of Interaction 26

6.2 Relationship between OMT and Interaction 27

6.3 A New Perspective On the Object Model 27

6.4 Optimizing the Object Model 30

6.5 Why the Object Model? 34

7. FUTURE EXPLORATION 35

8. REFERENCES CONSULTED 35

Interaction and Object Modeling

1. Algorithms vs Interaction

The recent shift from workstation computing to distributed computing environments has instigated a transition from computation via algorithms to computation accomplished using interacting objects. In addition to being more conducive to computation in such a distributed environment, the interacting objects offer greater expressiveness than is possible using algorithms alone.

1.1 What Is An Algorithm?

An algorithm can be explained as a set of rules for carrying out a given calculation. More specifically, it has been said that an algorithm is any computational procedure that takes some set of inputs and produces a corresponding set of outputs. The outputs generated should be the same for each distinct set of inputs.

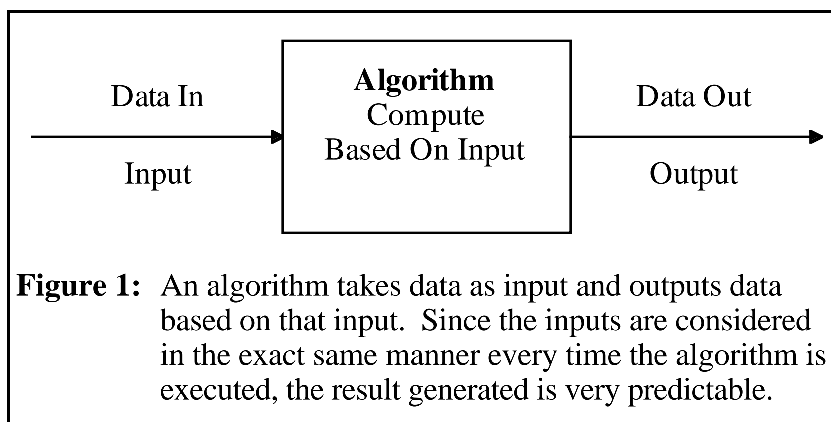


Figure 1: An algorithm takes data as input and outputs data based on that input. Since the inputs are considered in the exact same manner every time the algorithm is executed, the result generated is very predictable.

More formally, we can say that algorithms can be interpreted as mathematical functions. The set of all inputs to the algorithm are the domain of the algorithm, and the set of all corresponding outputs of the algorithm are called the codomain of the algorithm. Like their mathematical counterparts, algorithms pair each instance of inputs to a single solution, which need not be unique (in other words, each set of inputs must generate exactly one output, but more than one set of inputs can generate the same output).

Church's Thesis, which is yet to be proven but also yet to be shown to be in error, asserts that all algorithms can be expressed by Turing Machines. This notion can be extended and we can

define an algorithm as a Turing Machine that eventually halts no matter what input it receives. Therefore, systems that express non-terminating computations are not algorithmic. This idea will later be important when analyzing the expressiveness of objects and algorithms.

The unit of observable behavior for an algorithm is a single execution. That is, the smallest unit of activity produced by an algorithm which we can examine is one full execution. Not only is a smaller unit of observation meaningless since the algorithm has not been able to produce its final output, it may well be impossible since it is possible that no transformation from the inputs to the outputs has occurred.

Since the relation of Turing Machines and algorithms does not provide for a time period in which the algorithm must finish its computation, algorithms are also generally considered to be time independent. If we view algorithms as steps in a larger algorithm that models a particular system or solves a particular problem, the amount of time it takes an algorithm to give its final result is not relevant since the steps of the algorithm are being completed in a sequential order. Dependencies between algorithms are not an issue since it is assured that one sub-algorithm will finish before another begins. For similar reasons, the time at which a given algorithm executes is also not important (so long as all sub-algorithms execute in the correct order relative to each other).

1.2 What Is An Object

Simply put, an object is a mechanism for providing controlled and uniform access to a collection of shared resources. It provides these services by offering clients, which are often other objects, the ability to invoke certain operations which can either examine or modify the internal state of the object receiving the action request. All operations on an object are bound by the same state since they all are basing their actions on the internal information stored by the object.

The operations made available to clients are only invoked when a client issues a request. The requesting client could indeed be the object itself. It is in this way that an object can carry out a large task in a systematic and easily modeled manner. Most object models allow certain operations to be declared as publicly available, meaning that they can be invoked by any client, while others can be declared private. Such private operations can only be invoked by the object itself; external clients need not (and should not) be aware of their existence.

These operations are formally referred to as events. Rumbaugh asserts that events are external stimuli to an object that happens at a given point in time. Since the occurrence of an event takes so little time in comparison to the time scale of a system, it is considered to be instantaneous. The effect of the sequence of events instigated by the component objects in a system is a sequence

of state changes. Without knowing the state of an object, the effect of an event on the object can to be predicted.

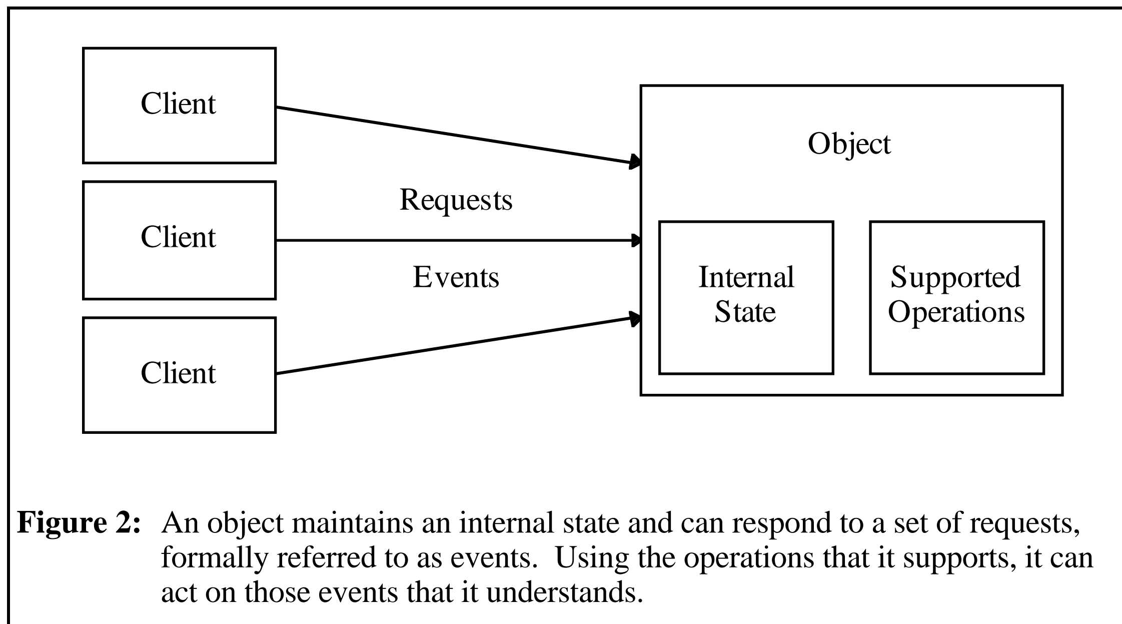


Figure 2: An object maintains an internal state and can respond to a set of requests, formally referred to as events. Using the operations that it supports, it can act on those events that it understands.

Like algorithms, objects have a base unit of observable behavior. This unit is a sequence of interactions and the corresponding changes in state that the objects undergo. This sequence is usually referred to as the object's interaction history. For the behavior of an object to be observably different from that of an algorithm, the unit of analysis must capture the fundamental nature of the object. It must account for the fact that the object provides multiple services that are time dependent. Were an object to be analyzed as a single interaction it would effectively be an algorithm. The reason that this is true is because objects have states, which are inherently time dependent. Although the events that affect object can be considered instantaneous in duration, by nature a state has a duration which provides for a notion of time dependence.

The time at which a given operation begins and its duration all play a critical role, especially when systems of multiple objects are analyzed. Since objects are executing the requested operations based on an internal state, any previously executed operations which affected that internal state could affect the outcome of subsequent operations relying on that state. Therefore, when constructing object systems, it is of extreme importance that time of execution be accounted for. The time dependent nature of these inter-object interactions cannot be modeled by functions alone.

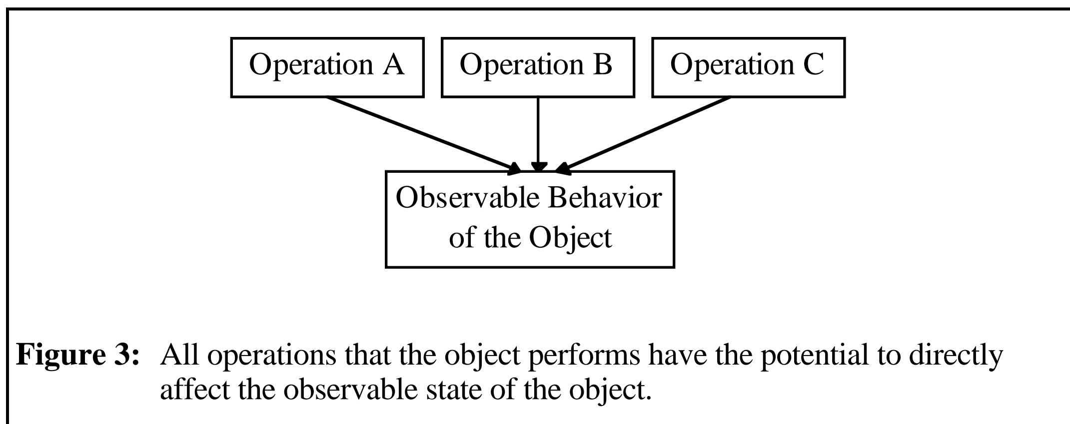


Figure 3: All operations that the object performs have the potential to directly affect the observable state of the object.

Since object behavior is time dependent, there are some interesting issues which arise. One of the most interesting is that unlike algorithms, objects can receive input over the period of time in which they are computing. Since inputs need not be fixed when an operation begins, objects are much better suited for functioning in dynamic environments.

Finally, it should be noted that objects can provide non-functional services to the systems in which they participate. Objects can be reused and adapted to meet the needs of other systems relatively easily. Their client-driven approach to action also makes them particularly suitable for integration into new systems. Objects can simply be added into systems and be counted on to respond to client requests in a predictable manner. Objects can even be created to administrate and manage such systems.

1.3 What Is Interaction?

Interaction is a term used to describe the inter-object communication which occurs over time. The communication which occurs gives rise to a new breed of computational possibilities involving cooperating objects. Furthermore, and of equal importance, with this new model of operation comes the ability for an object to dynamically adapt to its environment and act based on the state of other objects existing in the same domain.

Object systems derive their power and ability to model real world systems by providing a mechanism through which objects can cooperate to perform a given task. For example, if we look at the model of an airplane flight, we see that the crew of the plane plays a significant role in its success or failure. But taken individually, no single crew member could make the flight occur. Even the pilot could not possibly maintain the plane, load the luggage, service customers, and fly the plane. However, when the crew members are allowed to communicate and cooperate with the

intent of reaching a certain goal (in this, provide customers with a successful flight), the team that they form can do everything necessary to ensure a successful flight.

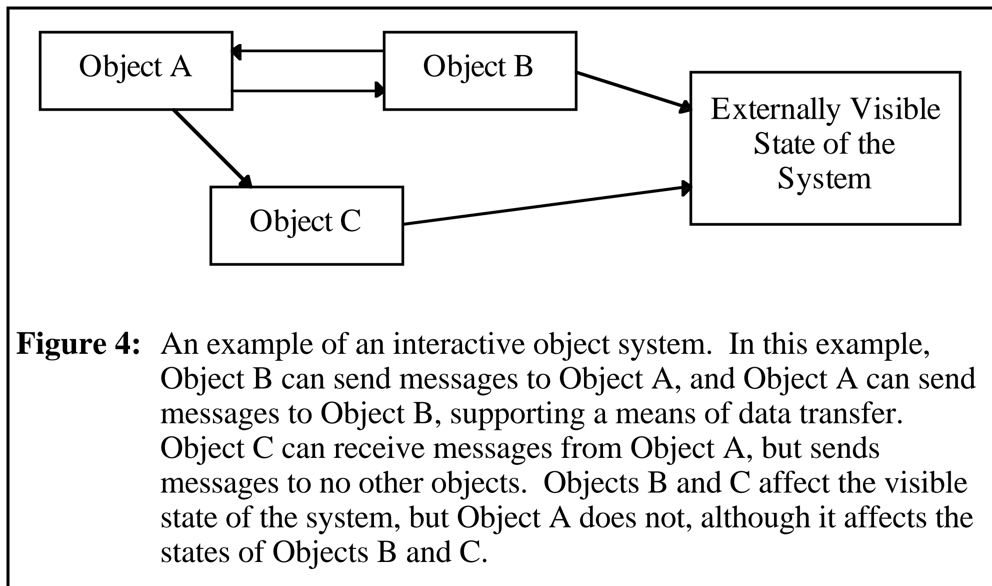


Figure 4: An example of an interactive object system. In this example, Object B can send messages to Object A, and Object A can send messages to Object B, supporting a means of data transfer. Object C can receive messages from Object A, but sends messages to no other objects. Objects B and C affect the visible state of the system, but Object A does not, although it affects the states of Objects B and C.

The model of cooperation should not be taken too literally, however. For cooperation to occur, it is not necessary for multiple objects to respond to events that actually alter the externally visible state of the system as a whole. Cooperation could simply be the sharing of data. Systems in which multiple objects affect the visible state of the system should not be considered to be more dependent on interaction. Systems in which objects exchange data about their state but have only one object changing the external perception of the system are just as interactive as systems in which multiple objects affect the observable nature of the system.

Events are the fundamental building blocks of interaction. We could therefore say that an interaction is an event. This is true; however, since it is necessary to analyze objects from a time-dependent point of view, it is important to also account for the state changes which result. If we ignore the state changes which result from the events when we discuss interaction, the next event might not induce the correct behavior from the system.

1.4 Why Objects Are Not Algorithms

Although it may seem as if the functionality of objects could be simulated through carefully constructed algorithms, this is not so. After all, it is inherent that a system being executed on a computer is ultimately algorithmic since the operation of the computer itself is indisputably algorithmic. However, the fact that objects have time-dependent states preclude this from happening. The response of an object to an event is dependent on the internal state maintained by

the object at the time the event message was received. This internal state could have been affected by any number of possible operation requests issued by other clients. With this known, it may seem logical that we could algorithmically model the interacting clients such that we could predict their sequence of operation requests. This is not possible, however, since we cannot guarantee external activity to be algorithmic in all cases. If the time granularity of states and events was the same, then perhaps the system could be algorithmically modeled. However, since states incorporate time into their observable behavior and algorithms do not, such a model cannot be constructed.

Another fundamental difference between algorithms and objects which precludes modeling objects via algorithms is the fact that objects provide persistent services over time, but algorithms provide response for only a single event at a time. Our present inability to provide a persistent algorithm prohibits the modeling of objects in this way.

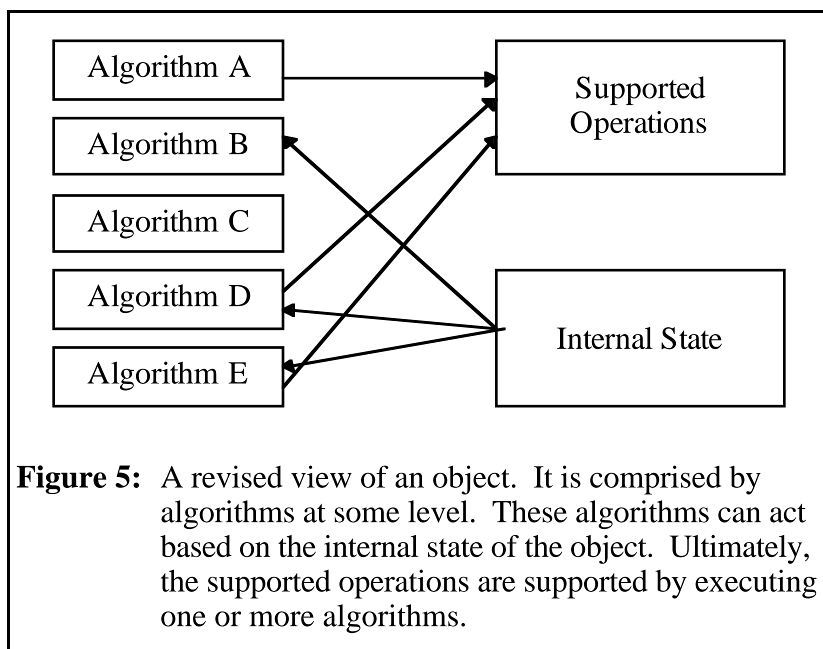


Figure 5: A revised view of an object. It is comprised by algorithms at some level. These algorithms can act based on the internal state of the object. Ultimately, the supported operations are supported by executing one or more algorithms.

It is interesting to describe the low-level functionality of objects as algorithms. This author would maintain that at the lowest level, algorithms are the base of all computation. Although difficult to formally prove, we can assert this by calling on the intuitive notion that anything being modeled on a computer, an algorithmic device, must be algorithmic at some level. The algorithmic nature might be observable before execution of the system on a computer, but it becomes algorithmic at this point in all cases. In all current interactive object-based systems, algorithms serve as the basic structure through which basic data elements are manipulated. From

this stems the notion that at some point in the composition of these algorithms a new behavior emerges. The new behavior is not one that can be algorithmically modeled; yet, moving back just one step in the concatenation process returns us to a system that can be explained in algorithmic terms.

There are many ideas associated with objects that cannot be expressed at a high-level through the use of algorithms. One such idea is the coordination of interactive behavior. The central problem of coordination is constraining the space of interaction. In other words, constraints express the kinds of interactions that are allowed within an object system. Coordination is necessary whenever dependent computational activities occur; this need arises from the time dependence of the interaction between objects. However, this time dependence is the problem encountered when attempting to give a formal characterization of coordination. Without a formal characterization, the construction of an accurate algorithmic model is difficult, if not impossible. Furthermore, some of the most basic issues of constraints deal with time, an issue which is substantially ignored by algorithms.

Algorithms do possess some traits that objects do not possess, although these traits are not generally considered to affect the expressiveness of either algorithms or objects. One of these is composition. We can combine two procedures to yield a composite procedure using the traditional mathematical notion of function composition. We cannot compose objects in a similar manner. The structure between objects resulting from message protocols is inherently different from the internal structure maintained by an object. One cannot be expressed in terms of the other. Just one reason why this is not possible is that separate objects maintain two independent internal states. If the objects are composed, they share the same internal state, at least to some arbitrary degree of granularity.

1.5 Why Interaction Is More Expressive Than Algorithms

Using algorithms, we can model sequential patterns of computation quite easily. Since we could reduce our objects into systems providing only a single operation, which could even be defined as state-independent, we can inherently model such sequential patterns using objects. In this case, our object would serve simply as a wrapper around an algorithm; all factors which distinguish an object from an algorithm would have been removed besides the formalization of the object itself. We can thus say with a high degree of confidence that objects can inherently compute everything that algorithms are capable of computing.

Using objects, we can also model streams of activity from external sources. This is not possible using functions because such operations are not enumerable in a recursive manner, nor are they deterministic. With this known, we can say that objects are more powerful and expressive than algorithms. Objects can be reduced to simple algorithms, providing all of the functionality of algorithms, but algorithms cannot provide for the capabilities afforded to systems through the availability of interactive computation as provided by objects.

With this theoretical notion stated, it is instructive to examine an example which demonstrates the greater expressiveness of objects. We could take a simple problem such as driving a car in a straight line down a street without hitting any objects that cross the street. Using algorithms to solve the problem, we could provide the length of the street as an input along with any predicted interruptions in our travel, such as scheduled train crossings. The algorithm would work fine in a system without other potential inputs (in this case, additional obstructions). Suppose, however, that a child runs out into the road in pursuit of a ball. This clearly was not an event that could have been anticipated; even the unfortunate bounce of the ball which caused it to leave the child's play area was not predictable. The driver in out algorithmic model would not know about the child and would hit it. It would, however, be able to stop for the scheduled train crossing.

Were the problem modeled as an interactive system capable of accepting additional inputs as the operation was being completed, this disaster would have been avoided. The object would have been able to adapt to the unexpected occurrence in a predetermined way (in this case, stopping the car and then resuming travel after the child had safely crossed the street). It is these kinds of systems that are much better modeled by interacting objects than algorithms. The unpredictability of the other component entities of the system cannot be accounted for using an algorithmic model, but the stimulus which these external entities can provide and the state changes which they can incur can be handled quite nicely using interacting objects.

Wegner asserts that interaction is more expressive than algorithms since it provides a weaker abstraction than a Turing machine does. He argues that with this weaker abstraction comes a larger set of potential models due to the decreased number of restrictions placed on our model. This larger set of potential models is of great utility when modeling real world systems. Using interaction, complex empirical models, such as those encountered in physics, economics, and biology, can be modeled in a realistic and feasible manner.

2. Interfaces

Although algorithmic systems rely on interfaces from a technical point of view, the fact that algorithms only support one correct means of invocation make their interfaces simple in comparison to the functionality of the systems that they compose. Interfaces for object-based systems, on the other hand, are complex in nature and play a crucial role in identifying the capabilities of the system.

2.1 What Is An Interface

An interface is a mechanism used to support the specification of behaviors for an object. The interface can be viewed as a contract between the object and its clients guaranteeing that certain actions will be provided upon the request of a client. The ability to separate functionality and implementation of that functionality leads to robust systems as perceived by a dependent client. The client can issue a request for action to an object and be confident that regardless of the methods that the object chooses to employ, an appropriate response will be returned or that an appropriate action will be taken. The internal methodology used by the providing object is not relevant. The object might not even fill the request internally; it may pass it on to another object or work with other objects to fulfill the request.

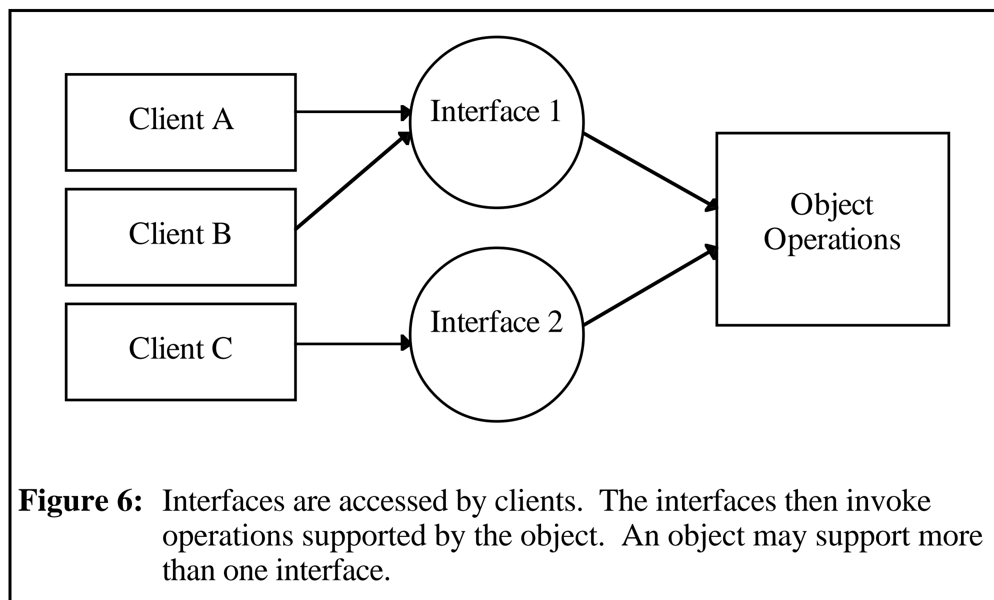


Figure 6: Interfaces are accessed by clients. The interfaces then invoke operations supported by the object. An object may support more than one interface.

An interface reveals the various behaviors that an object supports. The set of functions that an object supports allows the response of the object to external stimuli to be accurately modeled. In some cases, such an analysis is the only means through which a specification of an object can

be derived. If it is known which objects the object being analyzed interacts with, the interface can be important when analyzing the behavior of the object over time. However, the information that can be gathered from the interface alone is not sufficient to do this. The response of the object that can be observed is only the externally visible response. The internal state changes that occur are not observable outside of the object, but they do occur.

2.2 Interfaces And Object Modeling

When constructing an object model, one of the most important aspects to consider is the interfaces that the objects comprising the system will support. Interfaces play an important role in object modeling for two significant reasons.

- They allow for more accurate, understandable, and semantically correct composition of classes than traditional inheritance does. Inheritance of classes typically mandates the inheritance of implementation, which is not always an intuitive concept. Many real world objects perform the same task as similar object but in completely different manners. An example of such objects are planes and cars; both move a person from place to place, but the means through which this is accomplished is very different. Furthermore, additional complications are encountered when dealing with multiple inheritance of classes; these problems are completely avoided when composing interfaces.
- They simplify modeling for the system designer. When designing an object system (or any system, for that matter) it is more convenient to ignore low-level details in favor of concentrating on high-level constructs and concepts. The ability to separate the proposed behavior of an object (its interface) from the means in which that behavior will be provided (its implementation) is invaluable to the speed and accuracy of the design process. Furthermore, since interfaces function as contracts between objects, all that is needed to build such an object system is the interfaces. Cooperating programmers working on a single system can develop concurrently using only the knowledge of the interfaces of the objects with which their own objects will interact. How the cooperating objects perform their assigned tasks is not important. The mere availability of the service is enough upon which to build. Because interfaces can be formally modeled (which is very important when trying to specify system behavior) they are an integral part of a coherent object model.

3. Interaction As a Tool For Object Modeling

Despite the fact that it is the interactive nature of object systems which make them such an expressive tool, the focus of traditional object models has been on the static object model upon which the interaction is built. However, the focus of analysis for most object systems should be on the interactions which occur between objects.

3.1 How They Are Related

In very simple terms, the components of a static object model are the same entities that participate in the various levels of interaction within a system. Namely, objects, classes, and inheritance all combine to provide the interaction among system components that we typically deal with when analyzing interactive systems.

The means through which objects communicate is formalized through the notion of interaction. To fully understand the benefits of interaction, it must be reduced into a model that can be easily formalized and discussed. Most of the commonly used object models provide for at least one view of the system of objects that incorporates the interaction between objects. Some object models even have a view dedicated to this perspective alone; for example, OMT provides a dynamic model that shows inter-object interactions in the form of events and state changes. It certainly is with merit that this done since the utility of a multi-object system clearly lies in the ability of the components to interact; without such interaction, the system would be reduced to a set of coexisting algorithms. It is thus very important for an object model to provide a good perspective on the interaction that occurs between objects within the system.

It is important not to confuse object models with flowcharts or interaction histories with algorithm execution logs. Flowcharts represent operations as the nodes of the graph with shared information represented as the edges connection the nodes. Object diagrams, on the other hand, use the nodes to represent objects and the edges between them to represent the means of invocation of operations local to external objects. An algorithm execution log is a list of time independent instructions that have been executed, but an interaction history lists operations that have occurred in a real or virtual world in a manner showing their time dependent nature.

3.2 Components of the Object Model

As mentioned earlier, three main components participate in the interactive nature of object systems. A carefully constructed object model can use these components to more clearly express the relationships within a system and the way that the system works.

- **Objects.** Objects are the mechanism through which clients can access various shared resources by invoking certain operations local to the provider. The semantics of an object are best defined in terms of a sequence of interactions, which has already been established as the most specific unit of observation that should be applied to an interactive object. The object model can be clarified through the expression of the interaction that occurs between objects since the overall behavior of the system is usually more easily explained in terms of the functions that its component objects provide than in terms of its observable behavior.
- **Classes.** Classes are used to create and maintain both the interface and the implementation of the objects in a system. They add interactive object creation to the model of object interaction provided by objects. Classes are not the optimal entities for inheritance since such composition mixes interface and implementation. Nevertheless, they are clearly an important part of an object model. The designer of an interactive system should emphasize the capability of interactive creation that the class brings to the system when creating the system's object model.
- **Inheritance.** Inheritance allows multiple classes to be interactively composed into classes with additional or more specific functionality. In particular, using inheritance allows a system designer to enhance or specialize an interface. Showing how hierarchies of objects interact often simplifies the object model since there are fewer interactions that must be modeled (since an interaction of a superclass and an external class replaces the interactions between all of the subclasses and the external object).

4. Object Modeling Tool (OMT)

Many techniques for systematically capturing the essence of object models have been developed over time. One of the most popular and successful of such techniques is the Object Modeling Technique devised by Rumbaugh et al.

4.1 Description

The system of object modeling proposed by Rumbaugh et al., known as Object Modeling Tool (OMT) covers system design from analysis to implementation. The most fundamental idea on which the system was contrived is that nouns are better for modeling systems than verbs are. A systems designer following the model goes through three basic phases:

- **Analysis.** The first thing which must be established is the essential aspects of the application being modeled. At this stage, no regard is given to eventual implementation. All that is being done is the factoring out of objects, behaviors, attributes, and the like.
- **Design decisions.** After the analysis stage is completed, the system designer can address the various design decisions that must be made before implementation can begin. It is at this stage of the modeling process that efforts to optimize the implementation should be made. When working in this stage, a system designer should migrate from the application-domain objects identified in the analysis stage to the computer-domain objects, in terms of which the system will actually be implemented.
- **Implementation.** After the preceding two stages are complete (which could require several iterations) actual implementation occurs.

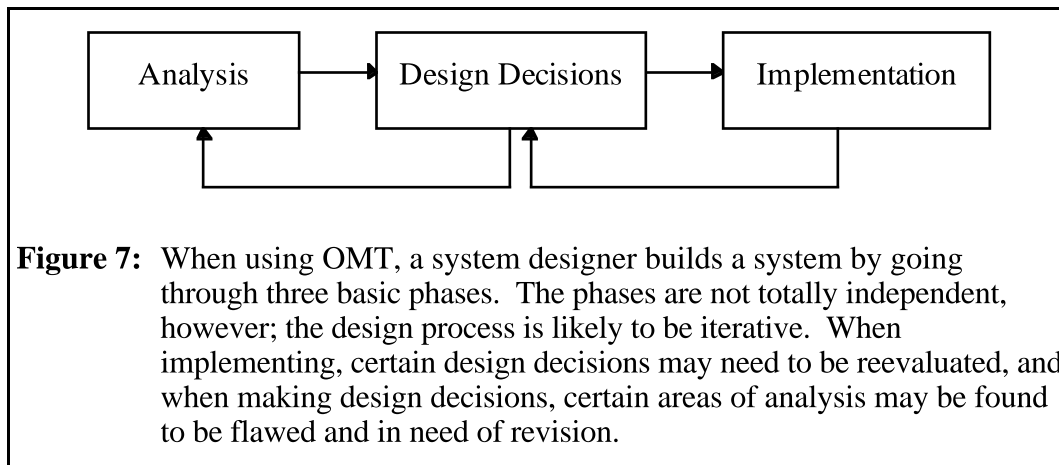


Figure 7: When using OMT, a system designer builds a system by going through three basic phases. The phases are not totally independent, however; the design process is likely to be iterative. When implementing, certain design decisions may need to be reevaluated, and when making design decisions, certain areas of analysis may be found to be flawed and in need of revision.

One of the basic notions behind OMT is that application and computer domain specific objects can be modeled, designed, and even implemented using the same basic technique. OMT exploits this fact to provide seamless integration between the three steps in the system creation process. This excellent design ensures that changes made in one step of the design process need

not be repeated in a later step. The end result of the careful consideration of all of the different facets of design of the system is a robust modeling framework that can handle the most complex problems with ease. Yet, it does not burden beginning modelers with the need to consider excessive levels of detail.

4.2 A Three Level Model

The OMT system yields three different models when it has been completely carried out. Each model is important throughout the design process and grows in complexity and level of detail as development goes on. In particular, implementation detail is gradually added throughout the design process. Rumbaugh maintains that a complete system model requires all three models. At first it would seem like it would be possible to construct a model not requiring all three models for a full system description, but more careful thought reveals that Rumbaugh is correct in his bold statement.

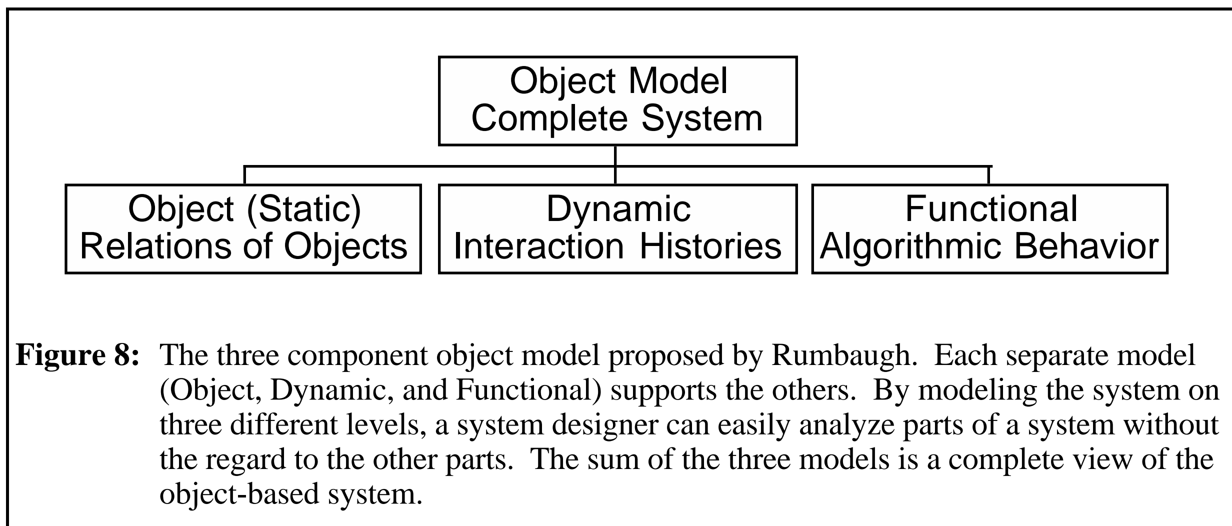


Figure 8: The three component object model proposed by Rumbaugh. Each separate model (Object, Dynamic, and Functional) supports the others. By modeling the system on three different levels, a system designer can easily analyze parts of a system without the regard to the other parts. The sum of the three models is a complete view of the object-based system.

The three models should not be viewed as independent, and changes in one model will almost surely require changes in at least one other model. The three models, in the order in which they should be constructed, are:

- **Object.** The object model describes the objects composing a system and the relationships between them. This model deals with the objects as static entities; therefore inter-object communication can be safely ignored at this stage of design. The OMT system calls for the creation of physical object models, which are essentially graphs in which the nodes represent classes and the edges show relationships among classes. This model is the most important, and great care should

be taken to ensue the accuracy of it before other models are created since any changes in what the objects in the system are will surely change the way in which they interact and manipulate data.

- **Dynamic.** The dynamic model is perhaps the most interesting of the three models. It explains the interaction among the objects composing the system. Another way to view the dynamic model is to say that it describes the effect of the passage of time on the object system; in other words, an interaction history is created. When the dynamic model is constructed, care should be taken to include the mechanisms necessary to control the system. The product of the dynamic model should be a state diagram. The state diagram typically shows the various states that an object can be in as the nodes of a graph and the edges represent changes between states precipitated by various events.
- **Functional.** The functional model is the most algorithmic of the three. It describes the data transformations that occur within the system. When creating the functional model, a data flow diagram should be created. Like the other diagrams, this is a graph. In this case, the nodes should represent processes and the edges should represent data flows.

5. Design Patterns

Just as in almost any system in computer science or the real world, interactive systems have certain design characteristics which recur in multiple models. The availability of a tool to factor out this commonality is required for efficient reuse of the components of object-based systems.

5.1 What is a Design Pattern?

A design pattern is way of describing a solution to a recurring problem. Specifically, within the domain of object modeling, a design pattern is a concise way of expressing a recurring set of relationships between objects. Many of the design patterns dealt with in the most common works on design patterns are patterns that model inter-object communication. The models represented by the various design patterns are customized solutions to different problems involving interacting objects; yet, the patterns are generalized so that they can be reused in a variety of programming and modeling contexts.

While any object can ultimately be reduced to algorithms, design patterns in the traditional sense ignore this reducibility in favor of dealing with a high-level abstraction of object behavior. Algorithms are generally considered to be a component of the less iterative process of implementing a given system after its design. The ability to put off implementational details (such as algorithms) until the very end of the object system construction process enables system designers to easily manipulate portions of their object models using design patterns.

While no pattern can perfectly handle every system into which it is integrated, patterns are designed to be as general and flexible as possible. To better satisfy this desire, patterns themselves do not include implementation details of any kind. Catalogs of design patterns, such as the popular work *Design Patterns* by Gamma, Helm, Johnson, and Vlissides, do address many of the most important implementational issues as a companion reference to the pattern itself. These implementation details should not be viewed as part of the pattern itself, however. Rather, the reader should interpret these implementation notes as hints from individuals that have previously implemented the design pattern. This is similar to reading a book about graphics algorithms and seeing a simple algorithm showing how to ray trace a scene presented in high-level pseudocode. The author may later mention that the data type double should be used instead of float. This is not essential to understanding the algorithm, but it is a helpful note to reduce frustration endured by the implementer.

5.2 Granularity

One of the most difficult aspects of designing an object-based system is identifying the objects. Specifically, determining the granularity of the objects is almost always a major challenge. A large collection of small objects may be a more specific breakdown of a system and may better allow the object model to parallel the real world system being modeled. But the large number of independently weak objects makes programming the system very complex. Since no object is truly useful on its own merits, a high number of inter-object associations need to be maintained. On the other hand, choosing a few large objects may be prohibitive when a systems designer later tries to extend the system. He or she may not be able to access the objects at the required level, and a redesign of the system could occur. Even worse, the interface of the monolithic objects could be contrived to handle the requests that would normally go to the component objects.

Design patterns help address this issue quite nicely. The following are two design patterns that assist a system designer deal with granularity in a manner which provides a good tradeoff between expressiveness and practicality:

- **Facade.** The facade pattern provides a single interface to multiple interfaces comprising the components of a subsystem. By defining a high-level interface, the components of the subsystem actually become easier to use in most situations. The facade pattern directly addresses the problem that arises when creating a high number of small objects. Although the specific objects are easy to deal with independently, getting them to function together as a system is quite difficult. The facade serves as a "mediator" object which provides a simple interface to the general operations that a subsystem is able to perform. An example of a system in which a facade could be used to simplify the system for clients is a fast food restaurant. Instead of a customer (the client) telling each individual worker to prepare certain foods that comprise his order, he places it with a cashier (the facade) which in turn tells the workers (the small and numerous objects) what to prepare. The food order (the result) is then provided through the cashier.

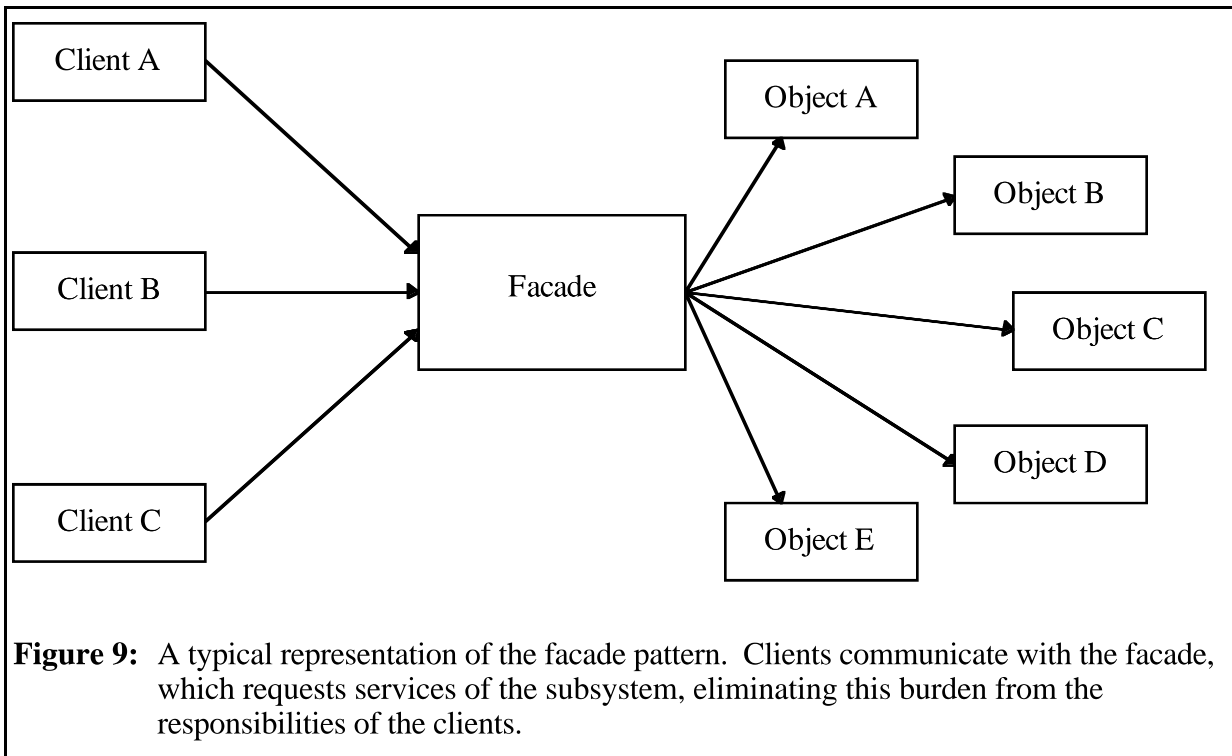


Figure 9: A typical representation of the facade pattern. Clients communicate with the facade, which requests services of the subsystem, eliminating this burden from the responsibilities of the clients.

- **Flyweight.** The flyweight pattern uses a sharing system to allow many small objects with high-granularity to be efficiently used. The flyweight itself is a shared object that can support concurrent access in multiple contexts. When used properly, the flyweight object is indistinguishable from a non-shared instance of the object that is

being represented at the time of access. A major drawback of the pattern is the requirement that object information be extrinsic so that it can be uniformly accessed by the flyweight. A system modeling people at a sporting event could benefit from the use of the flyweight pattern. Each individual needs to be modeled separately since each has his or her own traits (most of which can be perceived by an observer), but managing such a large number of objects would simply be impractical.

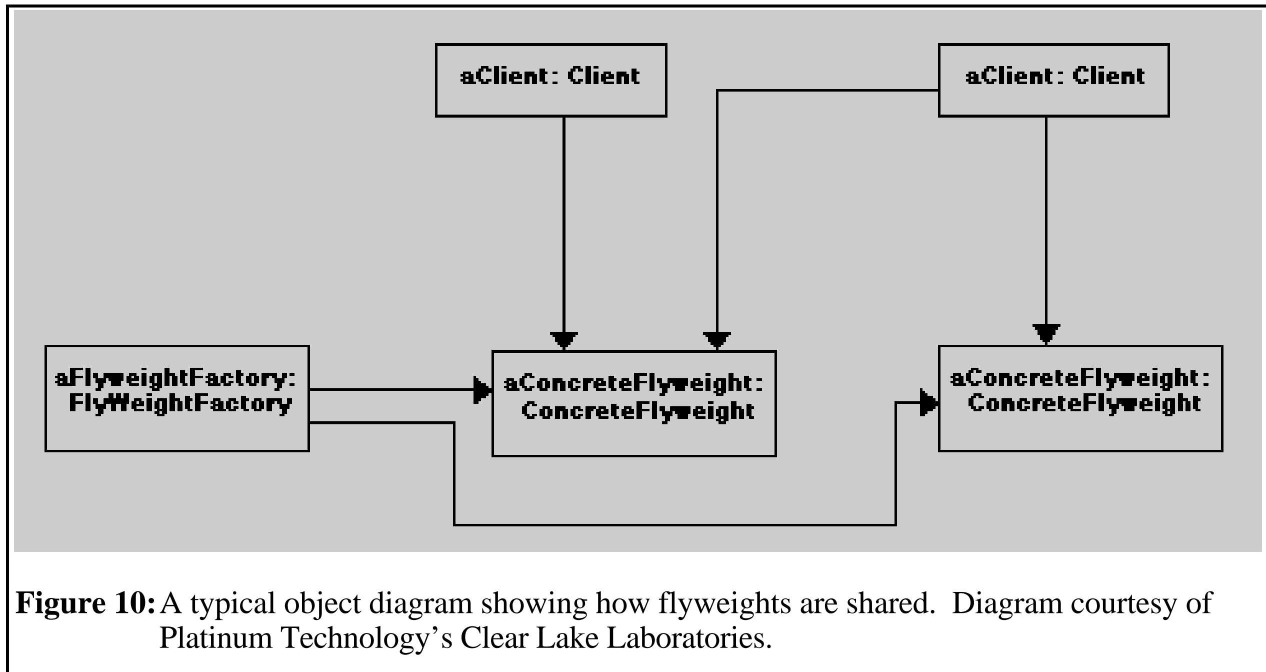


Figure 10: A typical object diagram showing how flyweights are shared. Diagram courtesy of Platinum Technology's Clear Lake Laboratories.

Once a system capable of handling a large number of objects in a clean manner has been devised, the objects must be constructed via a simple mechanism and there must be a convenient way of dealing with them all. Design patterns that address these specific problems have been devised. Specifically, the "factory" and "builder" patterns yield objects which can create other objects. Also, the "command" and "visitor" patterns create subsystems that are capable of performing an operation on another object or group of objects. The availability of these patterns greatly eases the burden of working with small and expressive objects.

5.3 Composition

Design patterns can provide elegant solutions to the problem of object composition. Under tradition systems of composing objects, the compositional nature of the product makes the subsystems difficult to deal with. The need for the client to know whether it is dealing with a composite object or an individual object adds a new level of complexity to the implementation of

the system, and if a system implementer is not careful, it can lead to a perversion of the object model designed. It is easy and even intuitive to try to solve such problems by requesting type information from a host before sending it an operation request. Such methods represent a breakdown in the object model, however, and should be avoided at all costs. In such a scenario, polymorphism is a much more appropriate solution.

Many design patterns have been concocted to solve this exact problem. Most notable is the "composite" as defined by Gamma et al. The composite calls for the creation of a class which supports the functionality of both sub-objects comprising composite objects and the composite object itself. The sub-objects and the composite objects both inherit the interface of this object (and in some cases its implementation also). Now the client can issue operation requests without knowing whether the recipient is a composite object or a sub-object of a system. An important fact to note is that the supported operations of the interface must be meaningful for both the composite and individual objects. For many operations, this is not an issue. For some, such as "add", this can be a tricky issue to deal with cleanly.

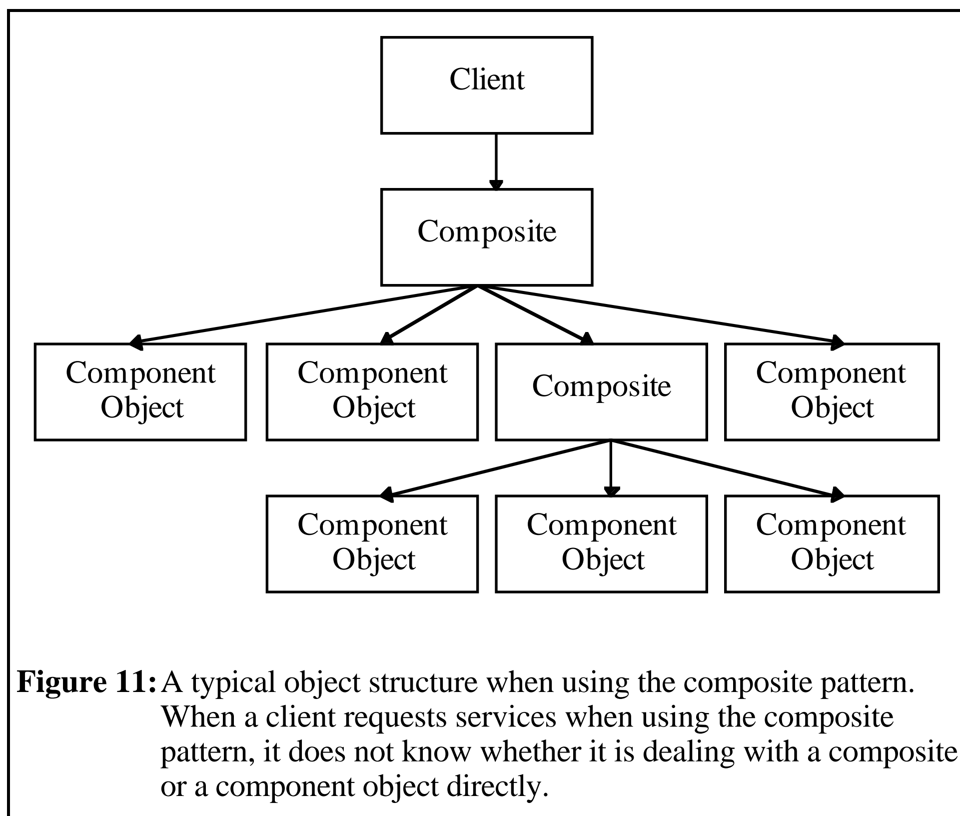


Figure 11: A typical object structure when using the composite pattern. When a client requests services when using the composite pattern, it does not know whether it is dealing with a composite or a component object directly.

The significance of the composite pattern should not be taken lightly. It represents the first documented and reusable means of composing objects as opposed to classes. This does more than just provide a more accurate model; it provides a new means of interactively composing objects at run-time. During the execution of a system, objects can be composed and decomposed upon will without the client ever knowing about the significant structural changes that the serving objects are undergoing. Recent interactive systems have only begun to exploit the possibilities afforded by such interactive composition. As this technology matures, we can expect to see systems that derive functionality that was previously thought to be impossible.

5.4 Specification

To express a design pattern in a complete and robust manner, there are several issues which must become part of its description. This author feels that the following are necessary for the design pattern to be useable yet not excessive. Many of the ideas here are based on the suggestions presented by Gamma et al.

- **Pattern Name.** The pattern should be given a name to make it easier to work with. The name also serves as a reminder of the pattern's function and can streamline communication about the pattern between system designers or system designers and system implementers. It is interesting to look at the pattern name as advertising. When enough professionals begin referring to the same pattern by the same name, it becomes known and can be refined. If each group of researchers refers to the same pattern by their own unique names, the pattern may never be identified as being the same pattern, stifling research, innovation, and refinement of the pattern.
- **Classification.** The classification of the pattern will become increasingly important as the number of recognized and published patterns increases. By classifying patterns by their general goals, catalogs of patterns become more organized. More importantly, suitable patterns can be found more easily by system designers. The classification should not be particularly specific. Good examples of different classifications include creational, structural, and behavioral. These specific classifications fit in well with the three-level object model that OMT uses also.
- **Aliases.** There should be an aliases entry listing other common names for the pattern. This correlates closely to "pattern name" and essentially serves as a pattern cross reference.

- **Problem Solved.** This author has chosen to leave out the "motivation" and "intent" sections of the design pattern description recommended by Gamma et al in favor of a comprehensive treatment of the problem solved. In this section, the creator of the pattern should give a good description of what the design pattern is supposed to do, the underlying rationale as to how it works and why it is a good solution, and what design issues it addresses. Also useful, but not mandatory, is a prototypical example of when the pattern could be applied. If an example is included, care should be taken so that the pattern fits the situation exactly without any modification so the reader does not get a mistaken impression of the pattern's attributes. The rationale section is very important; this is the section that assures that the reader actually understands the pattern that he or she is implementing and so other designers working with patterns can improve upon the model without changing its basic premise unknowingly.
- **Applicability.** The section on applicability should detail factors which could influence whether or not a pattern should be used in a given situation. Ideally, it presents a checklist of criteria which should be met for a pattern to be applicable for a certain object system. The checklist provides a simple and completely objective means of evaluating the suitability of a pattern for application in a given system. Variations on the checklist can also be useful. For example, it is sometimes convenient to say that a pattern should be used unless one of a number of statements about the existing system is true, or that a similar pattern should be used if only some of the criteria are met.
- **Components.** The components of the object system are not necessarily important when viewed as individual objects. Nonetheless, a good list of component objects that comprise a system should be included in the design pattern description. A straightforward listing of the classes that are involved in the design pattern and the role that each plays should suffice for this section. Some of the information here will be duplicated in greater detail in the "structure" section.
- **Structure.** The structure of the design pattern should be expressed both in verbiage and through object diagrams as prescribed by OMT. All levels of the model should be addressed here, with a special emphasis on the object and dynamic models. Generic class and action identifiers should be used when appropriate, but care must be taken that the identifiers are discernible when taken out of context. If an example was provided in the "problem solved" section, it is beneficial to use this problem as an

example when describing the structure. Even if this is done, each object and operation should still be given a generic and domain-specific name.

- **Interactions.** The interactions section should be a list similar to the component list. This time, the list should detail all inter-object communication and the role that these communications play in fulfilling the ultimate goal of the design pattern.
- **Consequences.** The designer of the pattern should continually update the consequences section of the design pattern so it remains a comprehensive list of the known effects of the pattern on the system. Typically, there is no need to list the objective of the pattern as an effect on the system; it should be implied. Things that should be mentioned are any pitfalls that the pattern could cause, tradeoffs that have been made by the designer by choosing the pattern, and specific ways in which other aspects of the system need be constructed.
- **Implementation Notes.** Here, the cataloger of the pattern should list any advice concerning implementation. Depending on the complexity of the pattern and its exact nature, sample code might also be applicable, be it code for a specific language or some form of pseudocode. Specific areas to be addressed might include any language dependent aspects of the pattern, techniques that make the pattern more efficient, and common pitfalls a programmer might encounter in the course of implementing the pattern. Although this section is not truly essential to the pattern catalog, this author feels that for the pattern to be truly useful it should be as easy as possible to implement, and this section certainly aids the realization of that goal.
- **Related Patterns.** Finally, any patterns which are used by the pattern being described should be briefly explained, and the reader should be directed to a source able to provide more information about the pattern. Also, similar patterns which have the same general goals as the pattern being discussed should be mentioned so that the implementer can check these patterns to see if they may be a more exact match to the intended goals.

5.5 Frameworks As Extended Design Patterns

Object-oriented frameworks can be expressed in terms of the fundamental principle of object-oriented programming. Due to the capabilities provided by inheritance, a new class can be implemented by expressing the differences between this class and one that has already been

defined. With object-oriented frameworks, the same principle is applied to whole object systems or sub-systems. Therefore the effort to develop a new object system is a function of the difference in functionality between the system being constructed and functionality already provided by the framework.

In contrast to traditional approaches to software reuse, which are built on the paradigm of libraries containing a large number of small building-blocks, object-oriented frameworks allow the highest common abstraction level between a number of similar systems to be captured in terms of general concepts and structures. The resulting framework is a generic design that can be instantiated for each object system to be constructed.

Frameworks are ideally suited for capturing the common elements in a family of related systems. In this sense, the framework is essentially a large design pattern capturing the essence of one specific kind of object system. The bulk of the functionality can be captured in the framework, which is maintained as a single system. Each resulting system is an instantiation of the framework, and the number of new design elements is determined only by the new components in that system and not by its total complexity.

Frameworks, in conjunction with design patterns, can be a valuable contribution to the evolution of object modeling methods.

6. Conclusion

Interaction is related to the object model in a variety of ways. Despite the work done in the area, this author does not feel that the relationship between the two has been refined enough for a system designer to take full advantage of either tool.

6.1 Design Patterns and Patterns of Interaction

Design patterns are a wonderful tool for modeling interacting objects. Complex interactions that recur in object systems can be classified, published, analyzed, refined, and ultimately perfected. The very nature of interaction makes it difficult to describe, especially at an abstract level. Yet interaction is one of the most fundamental aspect of any modern system and should be the focus of discussion. It is from interaction that object systems derive their fundamental nature, and design patterns allow this to be expressed.

Design patterns are also important for capturing models of interaction since they inherently incorporate the relationship of the interactive behavior of the system with the structural characteristics of the system. The research which has been done that pertains to interaction

substantially neglects the relationship between the static structure of a system and the interaction that it supports.

Patterns represent a breakthrough in communication within the computing profession. Now that a capable vocabulary has been defined, the large body of knowledge of interactive systems can make its way into forums of general discussion where it can be enhanced and refined until interaction can be as freely and easily discussed and implemented as other object oriented concepts such as inheritance already are.

6.2 Relationship between OMT and Interaction

OMT is currently fairly suited for modeling interactive behavior. The dedication of a specialized model to the interaction displayed by a system shows the value that Rumbaugh places on the interactive nature of object systems. OMT does address a number of the issues pertinent to interaction, but it certainly is not as thorough as an ideal system would be. As interaction matures, we can expect to see OMT extended to better incorporate the interactive nature of object systems.

Some of the items treated by OMT that pertain to interaction include:

- **Events.** OMT uses the idea of an event to provide responses from objects upon stimulus from an external source. Events are used to capture the inter-object communication which comprises interaction.
- **States.** States represent the internal status of an object at a given point during the execution of an interactive system. They track the object's attributes and any associations that it may have with other objects. States have a time duration and can affect the result of a received event.

When combined with some advanced techniques, these basic building blocks are capable of capturing the essence of an interactive object system quite nicely.

6 3 A New Perspective On the Object Model

After the object model has been used to find the classes which will compose a system, it should be applied to determine the behavior of the system that each will be responsible for. For this reason, it is helpful to have a preconceived notion of the externally visible behavior of the system before the interactive aspect of it is modeled.

When dividing up behavior between classes, the responsibility to provide services can be broken down into two types of responsibilities which need be addressed. First, it must be

determined which information other objects can request from a given object. In other words, the components of the state of an object that should be publicly accessible must be identified. The success of the object model depends on this step to a large degree. If too much information is available to clients, the encapsulatory nature of the objects has been sacrificed. The second responsibility objects can have to the system is to perform various actions. These actions can affect the object itself (alter its internal state), a subset of the objects comprising the system, or the system as a whole.

The purpose of identifying the responsibilities of objects is to convey a sense of purpose of the class from which it has been instantiated and put the object into its correct place relative to the other members of the system. When the responsibilities of the objects are being identified, the designer should keep in mind the goal of a client-server contract. In this discussion, this contract is said to be fulfilled through an object's interface. The interface should be constructed to represent the list of services that a client object can request of a serving object. As indicated above, the service could be the provision of information concerning the internal state of the serving object or could be to perform some task.

Perhaps the easiest way to identify object responsibilities is by returning to the system specification. The verbs of the specification should form the basis for object responsibilities; once they have been identified, they need only be distributed in a meaningful manner. If the static object model was well formed, it can be of great utility when dividing up responsibilities among objects. The name chosen for a given class probably suggests one or more responsibilities that objects of that class should handle. Furthermore, the inclusion of a class in the static model implies that the system designer saw a need for the object in order to fulfill one or more aspects of the overall system behavior.

When distributing responsibilities between classes, several factors should be considered.

- **Distribution.** System intelligence and functionality should be spread out between classes as much as reasonably possible. The overall system intelligence can be perceived as what the system knows and what it can do. Although many systems naturally fall into a model in which one or two objects are more "intelligent" than the remaining objects, this is not usually the optimum interactive model. By distributing ability and intelligence among a variety of classes, it is possible to produce a resulting system that is more flexible, more easily modified, and more specific.

- **Grouping.** Behavior and its related information should generally be kept together in a single object. Although the nature of object oriented systems allows this rule to be taken lightly, it should not be. It is most logical for classes responsible for certain information to be able to manipulate it in a meaningful way without the assistance of supporting objects. Although some of the power of interacting objects comes from the ability of objects to interact with information maintained by other objects, care should be taken not to abuse the ability. Conversely, if a class requires certain information in order to perform some operation for which it is responsible, it is logical to assign it the responsibility for maintaining the information as well.
- **Sharing.** In general, the responsibility for knowing specific information should not be shared. Sharing information implies a duplication that could lead to inconsistency. It also has a tendency to introduce difficult implementation details. On the other hand, responsibilities should be shared by multiple objects. It can often be observed in object systems (especially large systems) that a certain responsibility seems to composed by several sub-responsibilities. Such compound responsibilities are often best divided among more than one class.

After responsibilities have been distributed, the collaborations between objects must be identified. For purposes of dynamic modeling, a collaboration is simply a request made of one object by another. A collaboration can be explained in terms of the client-server contract by viewing it as the embodiment of the request specified in the interface of the receiving object. A single unit of collaboration flows in one direction only (from the client object to the serving object). Each collaboration should be associated with exactly one responsibility, and should fulfill (or contribute to the fulfillment) of that responsibility. The unidirectional flow of information and the singularity of utility of a single collaboration makes it like an algorithm. In a sense it is accurate to view a collaboration as such; these collaborations are not the unitary building blocks of systems described earlier in this paper, however.

The role of collaboration in an object system is important because the pattern of collaborations within an interactive system reveals how control and information will be shared when the system functions. The identification of inter-object collaborations parallels the identification of inter-object interaction. By finding these paths of interaction, subsystems of the interacting classes can be identified. This often serves as the beginning of a tedious stepwise refinement of the proposed object system, but the result of this refinement is well worth the effort

required. The resulting system will be further abstracted and hence more easily managed and understood.

To determine if an interaction is required between multiple objects, a system designer should begin by determining if the class is capable of fulfilling each responsibility that it was identified as contributing to. If the class is not capable of meeting the demands of the responsibility itself (in other words, its interface cannot be fully guaranteed without assistance), the additional information or functionality needed should be identified. This could be derived from one or more external objects. The needed classes are the ones with which interaction will ultimately occur.

In the course of all of this analysis, it may be discovered that some of the classes are actually too large. Care should be taken to ensure that each class encapsulates a single integrated set of responsibilities. Each class should have a single, overarching purpose. In other words, each class should serve exactly one main function in the system of which it is a part.

After the design has undergone multiple revisions, the original graphs for the three models may need to be redone so that they correspond to the new state of the design. At this point the system needs to be rechecked. Each responsibility should still be mated with a corresponding collaboration, and vice versa. At this stage of iteration it is often useful to walk through the design to ensure that every object is still interacting with the rest of the system in the appropriate manner.

Now that the model is close to finalization, finalized object interfaces can be formed. These interfaces are the means through which the contract to handle certain responsibilities is fulfilled. The process of grouping responsibilities into interfaces should be straightforward if the responsibilities were previously factored into hierarchies as suggested. Classes should continue to support a small and cohesive set of responsibilities throughout this stage of modeling. If the designer notices that this is not happening, the model as constructed up to this point should be evaluated for correctness. Of course, not all responsibilities should be made public. Only those responsibilities which are public should be included in the constructed interfaces.

6.4 Optimizing the Object Model

The various interactions present in the system are most likely numerous at this point and it would be useful if there was a method by which the dynamics of the system could be streamlined to make it easier to understand, maintain, reuse, refine, and extend. To do a good job of analyzing collaborations between objects, an exhaustive description of all the paths along which control and information can flow must be compiled. The collaborations between classes can then be analyzed

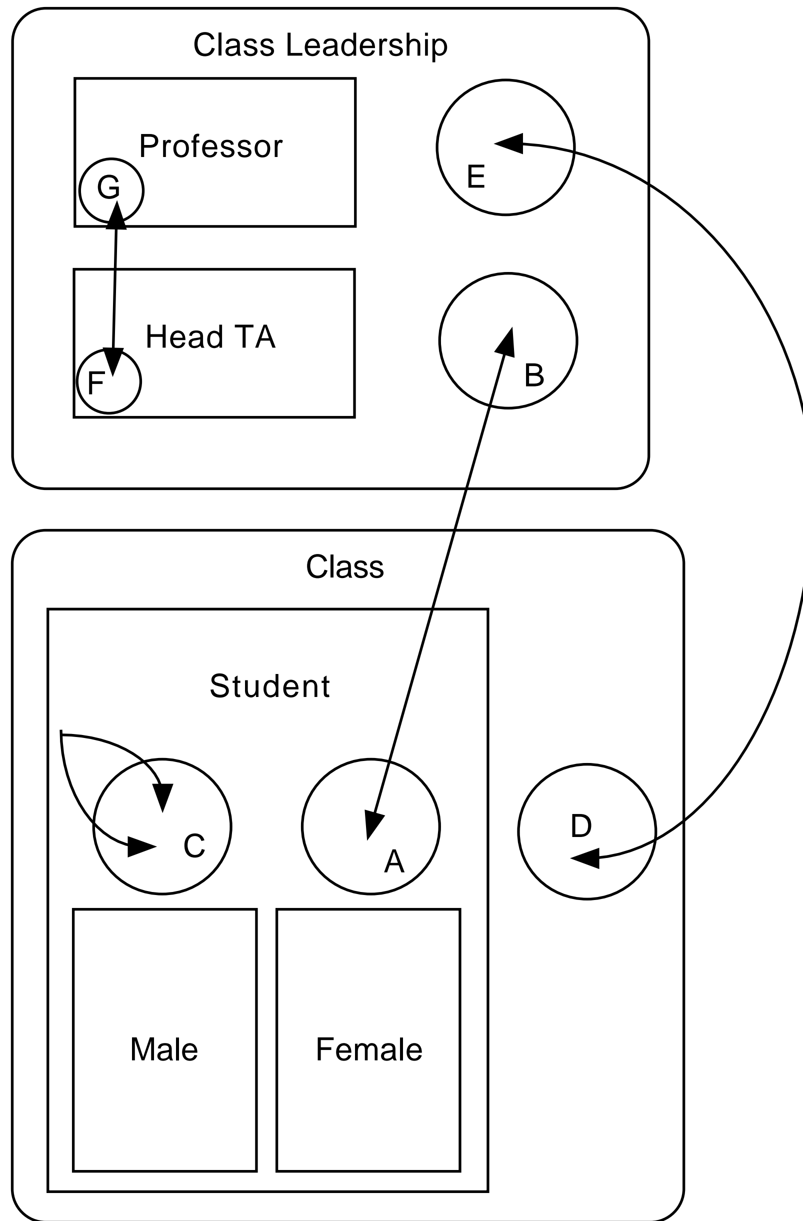
in order to simplify them. This compilation must also provide some information about the static structure of the system as well.

There is not an adequate facility for doing this in OMT at the present time. In fact, few object models provide any means for simplifying this difficult task. This author suggests a new graph, dubbed a “collaboration graph”, as a useful tool for accomplishing this analysis. The collaboration graph is intended for use as a supplement to OMT; it is not intended to replace any portion of it.

A collaboration graph would allow a modeler to examine the collaborations between classes in a graphical form, much as the object, dynamic, and functional graphs allow graphical analysis of a certain aspect of the total system. The ability to graphically analyze the interactions between objects allows a designer to better identify areas of unnecessary complexity. It can also be useful in revealing other design flaws. Unlike the graphs suggested by OMT which contain only one element, collaboration graphs should contain four distinct elements: classes, subsystems, interfaces, and interactions.

The collaboration graph draws from the elements addressed by the traditional OMT graphs. Classes should be shown as rectangles, as they currently are by OMT. Subsystems also need to be shown on this new graph. A rectangle with rounded corners can be used since we will not be showing object instances on the graph. Classes comprising the subsystem should fall within the bounds of these rectangles. Object interfaces are a new component of the object model that need to be graphed. Circles placed within the bounds of class boxes are recommended to represent these interfaces. Although it would be optimal to be able to describe the interface within the symbol used, this may not be practical in many systems. The complete details of an interface need to be presented somewhere in the collaboration diagram, however. Inter-class interactions should be shown using arrows denoting the direction of the flow of information. The arrow should lead from the originating component of the client to the interface of the server being used. Finally, to show inheritance and the availability of the interfaces provided by superclasses, subclasses should be completely contained by the rectangles representing their superclasses.

The proposed collaboration graph is no small chore to construct. The result must be a significant step toward the realization of a more perfect model of interaction to make it worth the effort. The author maintains that the proposed graph can be used to simplify the patterns of interaction used by the system sufficiently to reward the diligent designer who chooses to make an accurate graph.



Interface	Description
A	Students interact with the class leadership
B	The class leadership interacts with students directly
C	Students interact amongst themselves
D	The class as a unit interacts with the leadership
E	The class leadership interacts with the class as a whole
F	The Head TA interacts with the Professor
G	The Professor interacts with the Head TA

Figure 12: The graphical portion of a collaboration graph. In a “real” collaboration graph, the interfaces would be fully explained. Note the representation of inheritance, classes, subsystems, interfaces, and interaction.

Simplification is very important. Without it, interaction paths could flow from nearly any class to any other class, with little justification and no coherent structuring. Such poorly planned flows of information lead to incoherent code, as described by Dijkstra in his famous work "Goto Considered Harmful". Because poorly designed applications are impossible to maintain or sensibly modify, it is essential that the patterns of interaction be as simple as possible without sacrificing functionality of the aggregate behavior of the system. This author proposes the graph not only to aid in visualization of the system, but also as a tool for the successful simplification of the interaction patterns of the system. The proposed graph should not be considered to be the optimal tool for this process. As the reader will soon see, it requires the system designer to work "backwards".

The graph will be simplified in order to simplify the collaborations between objects. Several criteria should be considered when simplifying the graph.

- **Minimize the number of interfaces.** Having one class or subsystem support too many interfaces is a warning that some components of the object system are responsible for too much of the intelligent nature of the system.
- **Each interface should be supported by a single entity.** Only one class (or subsystem) should support a given interface. Any other distribution of the interface can lead to confusion when later analyzing the system; more significantly, it can be downright disastrous when extending the system. If an interface is simply mediating interaction between two classes, it is likely the case that a new path of interaction connecting the two classes directly should be added to the system. Another possibility is that the interface should be broken down into more than one interface.
- **Avoid excessive interaction.** Be sure that interaction is not being used to compensate for incorrect encapsulation of object components. Obviously interaction and encapsulation should be present in all object systems; when to use one as opposed to the other is not always so obvious. A designer should be sure that the object model that has been created provides the correct degree of abstraction.

The graph may not immediately appear to be reducible. Some general methods of simplification that the author has found from analyzing his own object models include:

- **Combining classes whose responsibilities overlap.** Although this is a major design change and can have profound effects on other aspects of the system, it is possible that

in some cases classes can be combined to simplify interaction while retaining the appropriate static model.

- **Build new subsystems.** Sometimes interaction can be simplified by adding a subsystem to a model. Such an intermediating object can centralize interaction between objects. Gamma et al propose several design patterns to accomplish this goal in their book, *Design Patterns*. An understanding of the methods suggested by Gamma et al makes this simplification very easily implemented, and can even increase the overall robustness of the system. Design patterns of special interest to a designer attempting to fulfill this goal include the facade and the mediator.

The proposed graph can be useful for many systems. The reader is encouraged to experiment with the proposed system and modify it to better integrate with his personal design tendencies. After all, as a new concept, the collaboration graph is not as refined and perfected as tools like OMT. If this graph is ever to become an integrated part of what is considered to be a complete object model, it is likely that it will evolve over time from the original proposal presented above.

6.5 Why the Object Model?

The result of the object modeling process is a design based on coherent, carefully chosen, and optimized objects. The responsibilities of each object become messages to which the object will respond by providing the services requested. Interactions handle operations for which an object needs help or more information in order to fulfill its own responsibilities as defined by its interface.

The object model forces a designer to support the basic concepts of object-based technology. Encapsulation of both operations and data is mandated, leading to a well-protected object state. Inheritance is used to incrementally refine the definitions of objects, maximizing their reusability. Because the paths of interaction are mapped out and strictly controlled, the system can be extended without risking unpredictable side-effects. Finally, because the system has been designed with extensibility in mind, functionality can be added to the system with minimal effort.

A strict adherence to the mandates of object modeling (in particular, OMT) ensures that all designs can reap the many benefits of interacting objects. In addition, this author feels that the proposed collaboration graph can be useful for refining object systems.

7. Future Exploration

This paper is by no means a complete analysis of interaction and object models. Future areas of expansion include:

- **Multiple interface models.** This is an area in which a collaboration graph could be incredibly useful since it would make it apparent which classes had too many interfaces and should be split into multiple classes.
- **Expand the discussion of interaction and OMT.** This section may eventually become a standalone section that goes into great detail of how OMT captures interactive properties of systems. Operations, concurrency, and internal activity should be given special treatment. Advanced topics such as entry/exit actions and synchronization could also be added to the discussion.
- **Non-algorithmic nature of design patterns.** A discussion of how design patterns can be used to model the non-algorithmic nature of interacting object systems could be added to better integrate the existing discussion of design patterns with the rest of the text.
- **Comprehensive treatment of coordination.** A discussion of the details of coordination and how it is addressed by the object model would lead to a more meaningful expression of interaction within an object system.
- **Subdivide the OMT dynamic model.** The synthesis of a subdivision of the dynamic model component of OMT may allow for more simplified modeling of interaction and make the information expressed by the model more accessible.
- **Provide more diagrams and examples.** This is especially important in the conclusion and the discussion of OMT.
- **Other perspectives.** Further research into object models and explanations of interaction might yield some useful findings which could enhance the overall completeness of the paper or contribute to my expressed view of the object model or even the proposed collaboration graph.

8. References Consulted

Cook, Steve and John Daniels. Designing Object Systems: Object-Oriented Modeling With Syntropy. New York: Prentice Hall, 1994.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides*. Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Publishing Company, 1995.

Graham, Ian. *Object Oriented Methods.* Reading, Massachusetts: Addison Wesley Publishing Company, 1994.

Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Reading, Massachusetts: Addison Wesley Publishing Company, 1992.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New jersey: Prentice Hall, Inc., 1991.

Stroustrop, Bjarne. *The Design and Evolution of C++*. Reading, Massachusetts: Addison Wesley Publishing Company, 1994.

Wegner, Peter. *Studies In Interactive Computing*. Providence, Rhode Island: Brown University Department of Computer Science, 1996. (Includes *Interactive Foundations of Object-Oriented Programming, The Paradigm Shift From Algorithms to Interaction, Interactive Software Technology,* and *Interactive Foundations of Computing.*