Organizational Benefits of Object Orientation

Terry L. Wiechmann
Educational Systems Inc.
5 Commonwealth Road
Natick, MA. 01760
508-651-1400

Abstract

This paper will attempt to explain the benefits of Object Orientation by:

- enumerating accepted software quality concepts.
- describing the Object Oriented analysis and design approach to a general problem domain.
- relating the consequences of the Object Oriented approach to the software quality goals.

Because the Object Oriented approach is expanding rapidly into diverse application areas, reference to Object Oriented concepts within this paper will be confined to M style database applications.

Software Quality Issues

Evolutionary changes in nature are only perceptible when observed over a very long period of time. However, the evolution of computer technology has advanced at an incredible rate within our lifetimes. We've all witnessed the progress hardware has made toward the goals of reusability and extendibility. At the same time we are left with the feeling that these goals are unreachable within the software domain. Hardware evolved toward the desired goals of reusability and extendibility more rapidly than software, the result being the Integrated Circuit (IC).

Software, on the other hand, has taken a zig-zag course toward these goals. New approaches to software development seem to promise the same results hardware attained, often failing in and of themselves. Good ideas abounded, but never seemed to be a complete solution, leaving an air of skepticism for anything new and untried.

As a result of this hit and miss approach, we've gained some knowledge.

1. Software must be developed with specific goals in mind.
2. To attain these goals, a specific, well defined approach must be taken; a paradigm is required.

Bertrand Meyer lays a foundation for software quality in his book: Object-Oriented Software Construction.[1] He outlines software quality in terms of internal and external software engineering issues. The diagram below illustrates these issues.

Software Quality Goals

Internal	***External***
Readability	**Reusability**
Modularity	**Extendibility**
Documention	**Correctness**
	Robustness
	Compatibility

These issues are viewed as desirable, software engineering goals. Internal qualities are programming issues and hidden from the user. External qualities affect not only the programmer, but the organization as a whole. Both are important aspects of software engineering quality.

As Meyer correctly states, "...**correctness, robustness, extendibility, reusability and compatibility**. They reflect the most serious difficulties with today's software development practices. Programs too often do not do what they are suppose to do (correctness). They are not well equipped enough to deal with abnormal situations (robustness). They are too difficult to change (extendibility). Their construction does not rely enough on previous efforts (reusability). They do not combine well enough with each other (compatibility)."

Obviously there are many more issues that affect quality. However, it is the five external issues that have the most impact.

Something Old, Something New...

So if these are the goals, do we have to recreate a totally new approach to accomplish them?

No! Like most things that evolve, some approaches die because they can not adapt, others thrive because they solve a part or all of a problem.

Without elaborating upon what constitutes the traditional approach to software development, let me merely state some undesirable aspects:

- The analysis and design of a system is often driven by data attributes, not objects.
- Real world objects are often modeled using some arcane, inconsistent internal representation requiring excessive transformations to produce different views of the object.
- The data, and the code that operates upon the data, are logically separated when they should be treated as a unit.
- Traditional approaches often are not capable of modeling complex, nested objects.

Yes, I'm sure these points could be argued ad infinitum. However, the end result is this: The real world is made up of objects, why not model them as objects?

The Object Oriented approach accomplishes the goals outlined by Meyer because it combines useful, time proven aspects of software engineering with some relatively new concepts. These approaches combine to form a synergy; a paradigm that comes closer to a complete solution to the software engineering problems we face today.

Attaining Quality Goals via the Object Approach

In general, the fowllowing concepts combine to form the Obejct Oriented paradigm:

- Object Identification.
- Object Typing through Abstraction.
- Organizing Levels of Abstraction.
- Polymorphic behavior through Inheritance.

Object Identification

The first, and most important concept is that of an object.

Cox defines an object as follows:

"An **object** is some private data and a set of operations that can access that data. An object is requested to perform one of its operations by sending a message telling the object what to do. The receiver responds to the message by first choosing the operation that implements the message name, executing the operation, and then returning control to the caller."[2]

Several important points can be extracted from this definition:

- All information and behavior contained (physically and logically) in an object is hidden (encapsulated).
- The data structures of the object can only be accessed by the objects methods (code).
- Objects are **not** called and forced to perform an operation, they are sent a message and asked to perform the operation. The receiving object is autonomous and makes its own decisions.

Not mentioned in the definition is object nesting - objects can contain objects. Given a consistent definition and the additional benefit of nesting, modeling very complex objects is simplified.

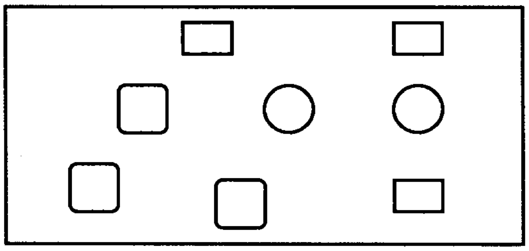
If an organization bases its future on objects, it establishes a consistent, well defined foundation for all of its engineering projects. This fundamental concept has far reaching benefits. Code becomes generalized, consequently there is less of it to support and the learning curve flattens. The quality goal of compatibility is largely attained.

Object Typing through Abstraction

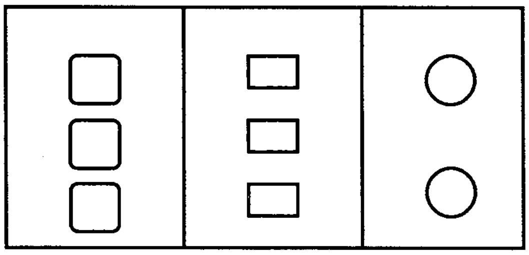
Given a consistent object definition and structure that can be used to model objects in the real world, we are faced with the need to create object types.

Modeling a problem domain using computers involves:

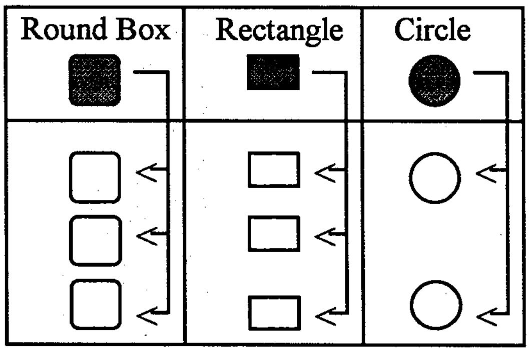
1. Identifying all of the objects within the problem domain.



2. Grouping the objects by common attributes.



3. Creating an object type by storing definitional information and code (methods) that can be used to create the real world object (instance). The creation process establishes an instance-of relationship between the instance and abstract type.

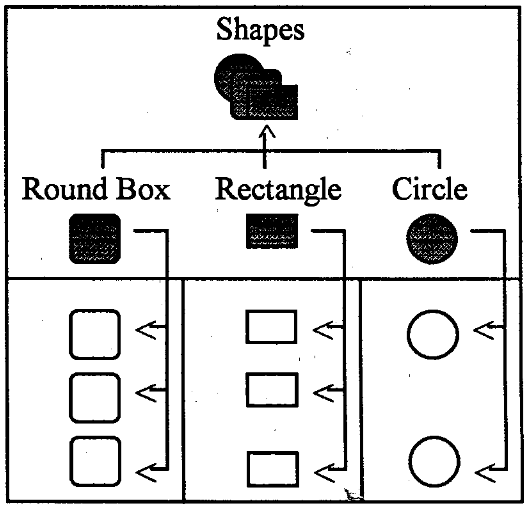


To quote Khoshafian and Abnous: "Abstract data types extend the notion of a data type; they hide the implementation of the user-defined operations associated with the data type. This information hiding capability allows the development of reusable and extensible software components."[3]

From a traditional standpoint, this is what data dictionaries are used for. What is different with the object oriented approach is the code that creates and manipulates the instance is also stored along with the definitional information - not in a separate, monolithic functional structure.

Organizing Levels of Abstraction

The concept of organizing objects by common attributes can be applied to the definitional levels until exhausted. In the example below, the common attributes are extracted to form yet another object type, the Shapes. To illustrate a parent-child relationship known as a kind-of relationship, the first level abstract types are organized under Shapes. Therefore, Rectangle is a kind of Shape.



The example is correct in an elementary sense. However, to the trained analyst, two discrepancies can be noted:

1. The structure is not based on the general case. There are many more geometric shapes than illustrated above.

2. A Round Box is not only a shape, but a specific example of a Circle and a Rectangle. How would this knowledge alter the hierarchy?

Abstracting out levels of commonality to form the object type hierarchy is a classification process. Consequently, these object types are called **Classes.**

Polymorphic Behavior through Inheritance

Abstracting attributes (variables) and behavior (code) into hierarchical layers, from specific (bottom) to general (top), implements the *potential* for polymorphic behavior - the ability to associate different values or behavior to the same name. For example, in the Circle → Shape inheritance path, two different methods with name Area could be stored at the Circle **and** Shapes levels. When a message is sent to an instance of Circle, asking it to calculate its area, the first method with the name Area (at Circle) will be executed.

Adding the concept of inheritance to this hierarchical organization *enables* polymorphism. Anytime a variable or method name is referenced, the name first found in the hierarchical path (from specific to general) is used. It is this mechanism that enables the separation of code and variables. It extends the concept of generalization beyond procedural boundaries by dispersing code and variables along inheritance paths.

Organizing code and variables in this two dimensional structure offers several benefits.

1. Code acting upon data can minimize the use of case statements since it can be specialized for specific objects. Consequently, this feature minimizes the potential for errors, extending the goal of robustness.
2. Errors that are isolated to a specific object tend not to propagate. Consequently, debugging time is minimized, eliminating wasted time and money, thereby extending the goal of robustness.
3. Permits overriding of code and variables, rather than copying code to add or modify functionality. This eliminates redundant code and minimizes support problems, thereby extending the goals of extensibility and reusability.

The Path to Object Orientation

By this time some questions should have arisen. If my organization moves from its traditional system, how do we migrate to an Object Oriented environment? Do we have to re-write everything all at once or can we migrate slowly? What's the cost of such a move and is it worth it?

As with all paradigm shifts, moving from one technology to another affects the bottom line. There are costs associated with moving down the Object Oriented path. Exploring this area in-depth merits another paper. However, let me list some obvious areas.

Hardware Costs - If a shift is made, what use can be made of old technology? Is this an opportunity to replace old equipment or is there a choice?

Software Costs - New software is generally needed to support the Object Oriented paradigm. What language, GUI, database system, or turnkey application are best suited for the organizations needs? Decisions made about software can spell success or failure.

Training Costs - Hardware and Software costs are tangibles. Shifting the mentality of your IS staff is an intangible - it can be the most unpredictable aspect of any transition. This is the area where most attention should be focused. Shifting from traditional analysis, design and programming techniques to the Object Oriented approach requires more unlearning than actual learning of new material.

Clearly, there are many other areas of concern. Given that, it should be sufficient it to say that a carefully planned integration is important to the migration.

Conclusion

Object Orientation adds a new dimension to software engineering, offering an organization's IS staff an intuitive, flexible paradigm to build information systems upon. The concept of an object forms a solid foundation. Data abstraction, organization of abstract types into hierarchies in conjunction inheritance extends software engineering goals beyond the limits of the procedural domain.

---

[1]**OBJECT-ORIENTED SOFTWARE CONSTRUCTION**, Bertrand Meyer, Prentice Hall.

[2]**OBJECT ORIENTED PROGRAMMING, AN EVOLUTIONARY APPROACH**, Addison Wesley, Brad Cox.

[3]**OBJECT ORIENTATION, Concepts, Languages, Databases and User Interfaces**, Wiley Professional Computing, Setrag Koshafian and Razmik Abnous.

REPRINTS ARE AVAILABLE!!

Please call the MTA office at 301-431-4070 for details.