Owen Durni
Mr. Hardy
AP Eng - Lang and Comp
June 6, 2005

# Why OOP?

Created to aid software design, an object-oriented program contains a collection of individual units, the objects, rather than a mere list of instructions to the computer. Like human cells, objects produced by object-oriented programming, or OOP, "don't know what goes on inside one another," says Cade Metz, an editor for *PC Magazine*, "but they can communicate nevertheless, working together to perform complex tasks" (2). Just as a cell membrane protects the cell, encapsulation protects the vital component of an object: its private data. Inheritance, identical to the process by which cells divide and become more specialized yet still retain their fundamental attributes, is also a key feature of OOP. Furthermore, without abstraction and message-passing, neither the human body nor a program, could possibly function. Nature has even provided a model for the challenging concept of polymorphism; lacking a predetermined form, stem cells only change state when necessary, exactly like a polymorphic method does not bind until a program is run. Certainly, the structure of the human body parallels the design of objects. By embodying the principles of encapsulation, polymorphism, inheritance, and abstraction, object-oriented programming provides significant advantages for implementing complex system structures.

First, when a programmer strictly adheres to all four principles of OOP, he encounters problems most often with encapsulation. According to Richard Mansfield, the author of an article republished online by *DevX*, an IBM associate, the rivalry between encapsulation and inheritance is important (1). Should you make simple classes for stability or abstract classes for flexibility? In theory, a paradox, but in application solved by reasonable compromise. Dan Shusman, manager of Technical Engineering for Inter Systems Corporation, says that this problem, however, has not been solved when designing Relational Database Management Systems because of "...the lack of affinity between the object and relational models" (1). On the other hand, by finding solutions and adopting the principles of OOP, companies obtain the "leverage" to maintain code once rather than fix it in countless locations, says Gary H. Anthes, the author of an article in *Computer World* (2). Despite the problems of encapsulation, preserving it greatly aids in maintaining existing code.

Embodying encapsulation in design allows for frequent code reuse and faster overall system development. Cade Metz, the author of an article in *PC Magazine*, compares OOP to interchangeable parts: "[j]ust as software programmers can use the same software objects repeatedly and even share them... [hardware] manufacturers can easily mass-produce circuits and sell them to other [clients]" (3). This sharing supports the computer's most promising application, simulation, argues John Markoff, the author of a *New York Times* article, because "complex processes can emerge from simple building blocks" (2). These blocks are objects. And with the growth of networks, the Internet, and the World Wide Web, reusing objects has become practical. Encapsulation allows users to exchange data, such as an Excel spreadsheet, and still have both computers know how to interpret it (Metz 2). As the network and the development process evolve, encapsulation will play a larger role in their success.

Often, during the development of encapsulated systems, implementing aggregate objects helps the programming team better understand the task. Like in biology, both data and commands are combined into objects, each with its own state and set of behaviors, contends Stefano Bettelli, author of an article for the *Economist* (2). These objects are more reliable, and easier to debug, or fix. When systems are designed by encapsulating objects, each part of a program is isolated, so developers know exactly where to look when a problem is found (Metz 2). Jeff Ferguson, the author of an article from *Software World*, also points out that the inventors of object-oriented programming languages realize "that [humans] don't express ideas in terms of blocks of code that act on a set of variables; instead, they express ideas in terms of objects" (1). Encapsulation is necessary to create useful and understandable objects, and therefore, is a valuable principle of object-oriented programming.

Secondly, polymorphism is a difficult concept for many of today's new software developers. Because of its complexity, say Ihlsoon Cho and Young-Gual Kim, authors published in the *Journal of Management Information Systems*, "[d]ifficulties in understanding [OOP] and applying a new technology may result in slower recognition of its value, fear of failure, and resistance" (5). In addition, authors of an article published in the *Communications of the ACM,* Boumediene Belkhouche and Cong-cong Xing claim the solution is simple: "...separat[ing] the essence of OOP from [object-oriented languages will] avoid entangling coding and design issues" caused by polymorphism (3). The problem, they argue, is not the structure of OOP, but the way it is taught to students. Although polymorphism is a complex, and difficult concept for new programmers, it is vital for the success of OOP.

While the uses for polymorphism are simplistic today, the fact remains that its future potential is boundless. For example, Bettelli's research has shown that if unitary transformations are represented as objects, it would be "fairly simple to translate programming directives at the classical level into physical control instructions at the quantum level" (2). Even sooner than the emergence of object-based quantum computers will be an improved system of web services. John Fontana, the author of an article in *Network World* says that "[w]eb services are the next evolutionary step in [OOP] for business-to-business e-commerce" (1). Obviously, the future possibilities of polymorphism and its flexibility, reflect OOP's usefulness for both old corporations and new businesses.

Thirdly, inheritance provides the organization necessary for the implementation of complex systems. Not everyone, though, is aware of its advantages yet. Fontana says that "[l]ike all OO approaches, web services require a framework of standards to be interoperable" (3). Anthes agrees that "the hard part about systems is building the interfaces" and organizing the framework between machines. But he also argues that, *now*, "its all integrated" and easy to implement (4). The challenges of inheritance have already been tackled; all that is left is to reap its benefits.

A superb real life example of inheritance is the animal kingdom. John Lewis, the author of *Java Software Solutions*, describes this classification hierarchy as it relates to object inheritance:

> All mammals share certain characteristics: [t]hey are warm-blooded, have hair, and bear live offspring. Horses are a subset of mammals. All horses are mammals and have all of the characteristics of mammals, but they also have unique features that make them different from other mammals.... In software terms, a class called Mammal would have certain variables and methods that describe the state and behavior of mammals. (382-383)

Following Lewis's logic, a class named "Horse" would inherit the "Mammal" class and expand upon its definition. Creating classes through inheritance can apply to *every* complex system, not just this instance. For example, in a card game, one object could represent a card, another the deck, a third the players, and finally one for the game's rules and mechanics. Since the computer is used to simulate environments in the natural world, inheritance applies directly to the creation of efficient systems.

Once the job of establishing a framework is in place, not unlike the animal kingdom itself, inheritance's advantages are obvious. The author of an article for the *Association for Computing Machinery*, Lu Jian says that inherited frameworks "improv[e] software productivity and increas[e] the reusability, extensibility, and portability of software" (2). Eventually, given time, these frameworks will develop into entire software libraries which can then be used to "assemble" programs, rather than write them, says Cho and Kim (2). For instance, Ferguson adds that recently, Unified Modeling Language was developed to aid in providing a platform and language-independent framework for individual unrelated developers (2). As the Internet expands, the vast framework of open-source libraries will continue to make OOP the small and private developer's method of choice because of its reusability.

Not only does the individual programmer seek the benefits of inheritance, but large-scale corporations can also incorporate it's advantages into their projects. "Helping to manage large-scale, complex programming jobs..." says Mansfield, "is the primary value of OOP [because] it's a clerical system with some built-in security features" (1). Lewis aptly states that the programmer, and consequently his employer, can create new classes "faster, easier, and cheaper" with inheritance (382). Code reuse is becoming a vital component in system design, says Richard King, a senior software applications support engineer for Diebold Inc., who reports that "[d]eadlines are so tight now that [designers] would never be able to meet them without aggressively reusing" (Anthes 2). Furthermore, for businesses, the easiest way to increase the productivity of a single programmer is to use object-based programs that allow for code reuse through inheritance (Markoff 1-2). By keeping common features as high as possible in the class hierarchy, efforts needed to maintain an entire project are "minimized" (Lewis et al. 392). King, for example, "has seen a whopping fivefold improvement on the speed of development... by using its 'toolbox of reusable elements'" (Anthes 2). Clearly, code reuse, spurred through class libraries and inheritance, is a key factor in the efficiency of object-oriented development.

Finally, abstraction, the single most important feature of object-oriented programming, is an undeniable asset when designing complex system structures. Ferguson writes that "[a]bstract data types were introduced because in the real world data does not consist of a set of independent variables," but rather it is an aggregation of "sets of related data" (3-4). In addition, abstraction and inheritance blend to create [high-level] parent classifications and interfaces. These classes cannot be instantiated -- meaning that instances of the objects defined cannot be created-- but instead, these kinds of classes serve as models and concepts on which other objects are built (Lewis et. al 399). Abstraction allows the programmer to ignore unnecessary and overly-complicated ideas, in order to focus on a more specific and more specialized design task.

The implications of designing system structures by employing abstraction are invaluable. According to Cho and Kim, "[a]s the need to innovate the organizational software development process [becomes] critical, the role of OO technology [will become] more prominent" (2). Parallel migration, Jian describes, is one of the most practical uses for abstraction: "[a]n OO framework... abstracts parallel issues, and provides programmers with a clear interface... and strategy for code reuse" (3). Furthermore, this recycling of software can help "address chronic software quality and productivity problems," claim Cho and Kim, because previously flawed software can now be developed using a "standardized [set of] quality building blocks" (2). By providing an interface for programmers to easily work with encapsulated data -- no matter how complex -- abstraction simplifies the overall design process.

By simplifying the complex nature of large-scale projects, abstraction allows a programmer to better understand his own role in the larger overall endeavor. Lewis claims that abstraction is essential because "people can [only] manage around seven pieces of information in short-term memory," and abstraction limits the amount of extraneous information that needs to be organized in the mind of a developer (62). Vital components to any successful system will be the creation of standardized blocks that are modular -- or easy to assemble, repair, arrange, and implement. Thus, the most important software quality, from the developer's point of view, is the modularity abstraction provides during design (Cho and Kim 2). These key advantages of abstraction contribute to the overall effectiveness of an object-oriented programmer.

The single greatest feature of abstraction, however, is embedded in the actual object-oriented languages themselves: a familiar syntax. Acting as the link between the computer, the programmer, and the client, abstraction enables software to be explained to nontechnical users in a pseudo-English vocabulary -- rather than a whole stack of zeroes and ones (Ferguson 2). Lewis comments on an everyday example of exactly this kind of feature to which anybody who owns a car can understand: "[y]ou don't need to know how a four-cylinder combustion engine works to drive a car" (62). The details are conveniently abstracted, and only relevant information is provided to the driver: the speedometer, the gauges, the check lights, an accelerator, a brake pedal, and a steering wheel. The greatest achievement of OOP lies in its ability to be freely and easily communicated between software developers and nontechnical customers because it allows for the use of a common vocabulary to *both* write and explain any system.

All of the preceding paragraphs show that structuring complex systems with object-oriented programming provides numerous substantial advantages. Encapsulation allows faster development through code reuse, polymorphism provides flexibility via future adaptivity, inheritance organizes development by emphasizing frameworks, and abstraction fosters understanding with its simplicity. These benefits will become more evident in the future, because according to Cho and Kim, "[a]s OO technology spreads in the software industry, standard [classes, interfaces, and models] can be... accumulated in a class library" (2). Common and repeated tasks like input/output streams, which can become incredibly difficult to design robustly, no longer need to be written for every project. They are already constructed by experts, and usable by anyone. Not only does this make coding more efficient for businesses, but it allows an individual to start with an incredible foundation of existing code to support him. As object-oriented programming continues to develop, and the open-source community continues to grow, future software development will be revolutionized by the successful expansion and absorption of OOP.

## Works Cited

Anthes, Gary H. "Code Reuse Gets Easier." Computer World. 28 July 2003: 24-25. MasterFile Select. EBSCO*host*. Penfield High School Library, Penfield, NY. 08 May 2005 <http://web12/epnet.com>.

Belkhouche, Boumediene, and Cong-cong Xing. "On Pseudo Object-Oriented Programming Considered Harmful." Association for Computing Machinery. Communications of the ACM. Oct. 2003: 115-117. ProQuest. Penfield High School Library, Penfield, NY. 15 May 2005 <http://proquest.com>.

Bettelli, Stefano. "Dream Code." Economist. 5 Apr. 2003: 73-74. MasterFile Select. EBSCO*host*. Penfield High School Library, Penfield, NY. 10 May 2005 <http://web12/epnet.com>.

Cho, Ihlsoon, and Young-Gul Kim. "Critical Factors for the Assimilation of Object-Oriented Programming Languages." Journal of Management Information Systems. Dec. 2001: 125-156. ProQuest. Penfield High School Library, Penfield, NY. 15 May 2005 <http://proquest.com>.

Ferguson, Jeff. "Writing Object-Oriented Code. (Software Intelligence: Technique)." Software World. Mar. 2003: 6-13. Student Resource Center. Thomson Gale. Penfield High School Library, Penfield, NY. 15 May 2005 <http://galenet.galegroup.com/servlet/SRC>.

Fontana, John. "Where Middleware and XML Converge Web Services." Network World. 24 Sept. 2001: 54-56. ProQuest. Penfield High School Library, Penfield, NY. 15 May 2005 <http://proquest.com>.

Jian, Lu, et al. "A Hierarchical Framework for Parallel Seismic Applications." Association for Computing Machinery. Communications of the ACM. Oct. 2000: 55-60. ProQuest. Penfield High School Library, Penfield, NY. 15 May 2005 <http://proquest.com>.

Lewis, John, William Loftus, and Cara Cocking. Java Software Solutions. New York: Addison Wesley, 2004.

Mansfield, Richard. "OOP Is Much Better in Theory Than in Practice." DevX. 15 Feb. 2004. JupiterMedia Corporation. 15 May 2005 <http://devx.com/opinion/Article/26676>.

Markoff, John. "Kristen Nygaard, 75, Who Built the Framework for Modern Computer Languages." The New York Times. 14 Aug. 2002: A21. Custom Newspapers. NOVEL (Infotrac). Penfield High School Library, Penfield, NY. 10 May 2005 <http://web1.infotrac.galegroup.com/itw/infomark>.

## Works Cited

Metz, Cade. "The Perfect Architecture." PC Magazine. 4 Sept. 2001: 186-191. MasterFile Select. EBSCO*host*. Penfield High School Library, Penfield, NY. 10 May 2005 <http://web12/epnet.com>.

Shusman, Dan. "Oscillating Between Objects and Relational: The Impedance Mismatch." Byte.com 16 Feb. 2004: 1-9. MasterFile Selct. EBSCO*host*. Penfield High School Library, Penfield, NY. 10 May 2005 <http://web12/epnet.com>.