Applying Object-Oriented Concepts to the Engineering of Complex Software Systems

Georgia E. Nelson
University of Maryland University College
MSWE 603 Systems Engineering
April 19, 2007

1. Introduction

In recent years, the development of software-intensive systems has increased in demand as the popularity of computer-based business solutions has risen. The systems and software engineering community has barely had time to stop and rethink its various development strategies due to the introduction of new technologies seemingly every day. In theory, one would think that engineers would warmly welcome the introduction of these new technologies; however, technology can disrupt current methodologies by changing the way people prefer to do things.

This constant influx of new technology and changing methodologies can create a dynamic development environment. Some professionals in the field argue that software development is harder not only due to this dynamic behavior, but also because of its "invisible" nature (Dorfman & Thayer, 2000, p. 1). Software is complex and hard to understand for the common individual, because it lacks a tangible presence in our world. So, how does one *properly* develop and maintain a *complex* software-intensive system in such a dynamic environment? Furthermore, how can a complex software-intensive system become more *tangible*?

1.1 Structured Development with Object-Oriented Concepts

Starting in the late 1980s, software engineers began looking at a new set of engineering techniques that could help give a natural, less complicated approach to structured software-intensive systems development: the "object-oriented approach" (Wieringa, 1998, p. 459). Ivar Jacobson, a Swedish computer scientist, originally dubbed this development approach "ObjectOry" (Jacobson, 1987, p. 183). Jacobson (1987) described ObjectOry as the structured development of software system components with specialized entities (p. 183). Though the terminology is no longer in use, the underlying concepts behind ObjectOry continue to thrive in today's systems software engineering community.

By definition, a system consists of subsystems and components that work together to fulfill a common objective. Jacobson (1987) referred to these subsystems, as "blocks" that consist of several "components" (p. 184). With object-oriented techniques, engineers are capable of viewing the systems they are developing as a set of communicating, tangible blocks. Engineers deal with each block on an individual basis if requirements change, or if issues appear as development progresses. Theoretically, this would allow engineers to make changes without disrupting the behavior of other blocks in the system.

Despite the fact that software components are considered "abstract concepts", object-oriented techniques make the pieces of a system seem more "tangible" (Korson & McGregor, 1990, p. 48). The ability to visualize and understand the system from an object-oriented perspective paves the way for engineers to better design a system. In turn, it will also be easier to deal with the unexpected changes that inevitably occur as development progresses.

Many systems software engineers have a very close-minded view of object-oriented techniques. They only see these techniques used in high level programming languages, such as C++ or Java. As true as that may be, many of the underlying object-oriented concepts seen in programming can also be applied to the entire engineering process. The premise is that entities, called objects, are the building blocks for components and subsystems. Their messaging capabilities and inheritance concepts make system components reusable and highly maintainable by enhancing modularity. Object-oriented techniques take the development of complex software-intensive systems to a new level by breaking its complexity down to the "way humans think" (Fayad, Tsai, & Fulghum, 1996, p. 109).

2. Objects: The Building Blocks

In an object-oriented system, the components and subsystems are built from the ground-up with objects. Objects are encapsulated entities, which are responsible for fulfilling the specific functionality allocated to them. They also possess the capability to communicate with other objects in the system through messaging and behavioral functions. In software terms, objects are instances of classes, which are the implemented versions of "blocks" in an "object-oriented programming language" (Jacobson, 1987, p. 189). Software engineers can use classes to create instances of objects to manipulate and communicate. The primary purpose of these objects is to work together through a messaging pipeline to meet the needs of the system and its various use cases.

2.1 Objects as Related to Systems Development

One perspective views subsystems as interconnected gears in an old-fashioned clock, each working and communicating their movements keep accurate time. The responsibility of subsystems is to fulfill their assigned requirements determined in the design phase of the development life cycle. Korson and McGregor (1990) argued that the object-oriented approach utilizes the "relationships between objects as a fundamental part of the system architecture" (p. 41). Objects, much like the interconnected gears of our clock, have an assigned purpose, or responsibility. They are built specifically to fulfill their assigned requirements and to work with other objects to ensure that those requirements are met. Engineers can also assemble objects into clusters to form a fully functional subsystem. For example, clustering objects pertaining to network communications would create a communications subsystem responsible for all incoming and outgoing network messages (see Figure 1).

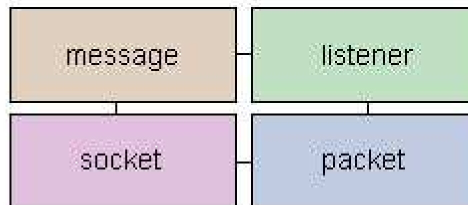

Figure 1: A cluster of network objects forming a communications subsystem

One object-oriented concept that supports modularity and reuse in systems development is encapsulation. Objects achieve encapsulation by placing all of the functionality and data pertaining to it within the object itself (Armstrong, 2006, p. 125). In a software-intensive system, engineers are responsible for distributing design decisions and "control processes" into encapsulated objects (Cockburn, 1999, p. 317). This can lead to the creation of subsystems of clustered objects whose functionality only belongs to them. Not only does this prevent disruptive changes from potentially harming the functionality of other subsystems, but it also can foster reuse and good maintainability practices. Snyder (1991) further promoted "module independence" through encapsulation, because of the "maximal reuse of previously designed" system components it provides to engineers (p. 425).

2.2 The Natural Perspective

So why are object-oriented techniques so approachable? What makes them so much easier to understand? In short, our world consists of many different objects all working together in some form or another. According to Garceau, Jancura, and Kneiss (1993), "objects can identify tangible things," which makes it easier for humans to visualize the system before it has even been built (p. 27). This is especially important when attempting to communicate system designs with individuals who may not be knowledgeable in the terminology used to describe a system. Object-oriented techniques create a "common ground" on which systems engineers and customers can communicate (Garceau, Jancura, & Kneiss, 1993,

p. 27). Oftentimes it is easier to understand something when one can see it as opposed to resorting to one's imagination. An imagination can lead to confusing and somewhat ambiguous interpretations of the system.

Further, the object-oriented approach introduces a new "development model based on the way humans think" (Fayad, Tsai, & Fulghum, 1996, p. 109). It is only natural that humans view their world as a collection of collaborating objects. When developing complex, software-intensive systems, it can oftentimes ease the complexities by designing the system in such a way that everyone involved in its creation (programmers, designers, customers, users, etc.) is able to understand to it.

The human method of approaching a problem is to divide it up and conquer it piece-by-piece. When a need is communicated, a problem of system design is born that must be solved through an engineering process. Functional decomposition involves taking the basic system need and dividing it into smaller pieces of system functionality. With the object-oriented approach, engineers decompose the system's functionality by allocating them to objects. Gilmore and Green theorized that the allocation of system processes into objects was more natural to the "ways of human cognition than functional decomposition" (as cited in Johnson, 2002, p. 13).

3. Communication and Interaction of System Components

System components are, "by nature," always communicating with each other, and need to have interactive behavior to achieve this (Wieringa, 1998, p. 462). Thanks to concepts like encapsulation and abstraction, objects have a firm grip on the data that they are responsible for manipulating. Without the presence of a communications pipeline between them, subsystems and components would be unable to communicate. In other words, objects need to establish working relationships with other objects, through either functions or direct messaging, in order to communicate. For example, when one double-clicks his or her mouse on a document, it signals the computer's operating system to open that file. Figure 2 illustrates a high-level conceptual view of how the mouse, a component of the computer system, signals the software-based operating system component of the system that it has double-clicked on the file it wishes to open.

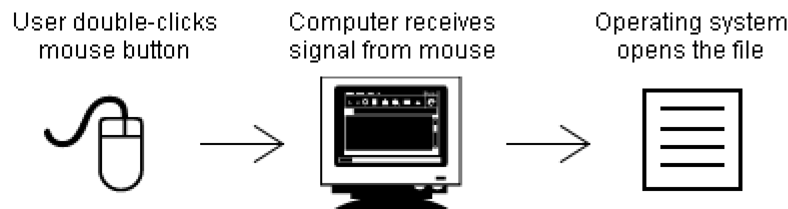

Figure 2: A high-level conceptual illustration of a mouse's double-click message and response

3.1 Messaging

In an object-oriented system, the passing of messages is the primary method used by objects to communicate with each other (Hines, 1995, p. 196). In order to communicate a request, one object must initiate contact by passing a message to another object. The key phrase to note here is "request." Message passing focuses on making requests of other objects, not specific function calls or the passing of data. If other objects simply accessed the attributes and methods of another object, encapsulation would no longer be a valid operation. A highly encapsulated and abstract system environment relies heavily on the passing of messages for subsystem communication.

Cohen (1984) described messages as either being "two-way," "analogous procedure calls," or "one-way," simple "requests that the receiver perform some set of actions" (p. 33). Both types of messages exhibit a cause and effect relationship. In other words, with every action comes an expected reaction. Table 1 describes Cohen's object-oriented message types and the reactions they expect to receive from them. In the case of our mouse, the reaction we can expect from double-clicking on a system file icon is for the operating system to open it. This would be a one-way message.

Table 1: Object-oriented message types (Cohen, 1984, p. 33)

Message Type	Description	Expected Reaction
One-way	A simple message that signals an object with a request	The signaled object should perform the requested action
Two-way	A message, which signals another object and listens for a response	A return response from the signaled object

For systems development, the concept of messaging is a crucial aspect of ensuring that all of the components and modules of the system are capable of working together. Messaging allows fluid communication of data between subsystems. The strength of the interrelationships between objects in a system, otherwise known as "object coupling," is enhanced by the openness of the communications pipeline developed between them (Berard, 1998, p. 65).

3.2 Methods and Object Behavior

As previously mentioned, messaging does not involve the manipulation or passing of data between objects. In order to achieve this, objects access each other's attributes and behaviors through "component functions" otherwise known as methods (Wieringa, 1998, p. 466). Methods may accept parameters, which define what the method will need to perform its task. For example, our computer file will not open unless the operating system received the correct coordinates for the file's location when the mouse double-clicked the screen.

The behavior of an object is governed by its "state," which describes what task the object is currently performing (Armstrong, 2006, p. 125). The object could be idle, functioning, or simply working as a communications conduit to other objects in its cluster. Systems operate under various states at any given time. In order to control the parameters functions can accept, engineers must manage the states in which object can be. Objects also need to reflect those states in order to operate at the proper level of performance. Through methods, system objects may manipulate another object's behavioral state and attributes.

4. Inheritance: Not Just for Geneticists!

It is through genetics that humans understand how they inherit specific traits from their parents. One can look in a mirror and point out features of his or her face that resemble one or both parents. Humans even inherit their quirks and behavioral attributes through careful observation of their parental units. In an object-oriented environment, this is no different. An object may serve as a parent to another object or set of objects, which inherit the basic attributes and behaviors of its parent. In the object-oriented approach, this concept is called inheritance. Inheritance is a "useful mechanism" for component reuse that saves both time and money on project development, but it does have its faults (Snyder, 1986, p. 44).

The limitations of inheritance fall within its conflicts with encapsulation. As discussed earlier, the point behind encapsulation is to keep all of an object's functionality, attributes, and behaviors in the object itself, only allowing access to it through messaging (Armstrong, 2006, p. 125). With inheritance, an object can have children that implement and make use of the parent's functionality and attributes. It is like opening the floodgates to its contents instead of fencing them off into a single system component.

4.1 Applying Inheritance to Systems

Regardless of this conflict, experts have heralded inheritance as being "the most promising concept" available to software systems engineers, because it helps create systems from "reusable parts" rather than by "hand coding" every system "from scratch" (Korson & McGregor, 1990, p. 43). Without reuse, systems would take a significantly longer amount of time to develop. Inheritance can help reduce the amount of redundancy in the system by allowing basic objects to be created whose functionality can be the basis for other objects. Encapsulating basic functionality into a single object from which others may derive, means that its functionality does not need to be duplicated in other system components. This leads to saving development time and money.

Korson and McGregor (1990) further explained that reuse through inheritance allows the developer to "tailor" the child object to his or her needs from the features of the parent without actually altering the attributes and behavior of the parent itself (p. 44). By building objects from other objects through inheritance, developers can create subsystems with a cookie cutter approach. In other words, similar functionality can be applied to several subsystems from a single object or interface, which greatly reduces redundancy in the system's implementation. Inheritance helps form bonds between objects of similar types. Further, it strengthens relationships between system components and allows for a larger amount of system reuse.

4.2 Expanding with Multiple Inheritance

The traditional view of inheritance is simplistic and singular, meaning an object may only inherit from one parent and no other. Naturally, this is not how one may view the world unless all of a sudden we became single-celled organisms. In fact, the more natural view of inheritance would involve two parents. "Multiple inheritance" is the object-oriented concept, which, according to Singh (1995), is a "more natural definition of relationships" between objects (p. 30).

In multiple inheritance, an object may inherit from more than one parent object. This allows for even greater component reuse, by not limiting the system to single inheritance methods. Halbert and O'Brien explained multiple inheritance in his "classic example" of the "TeachingAssistant" object, which was formed by inheriting from the "Teacher" and "Student" objects (as cited in Taivalsaari, 1996, p. 449). Figure 3 illustrates the multiple inheritance relationship of O'Brien's TeachingAssistant object to its parents, the Teacher and Student objects. Note how the TeachingAssistant inherits specific and distinctive behavior from both parents.

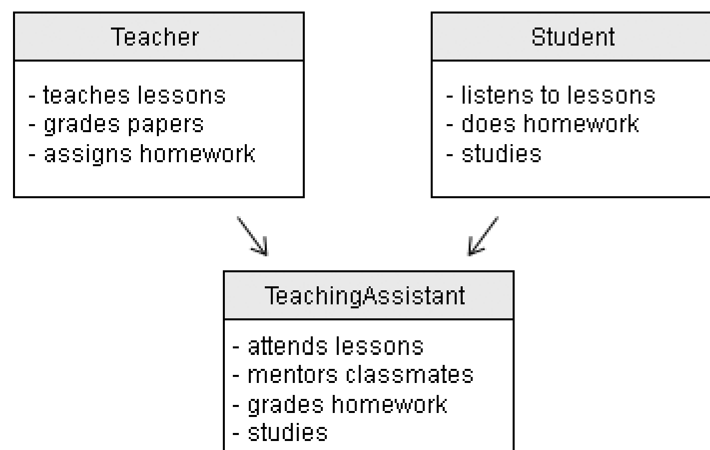

Figure 3: Illustration of the TeachingAssistant's multiple inheritance relationship (as cited in Taivalsaari, 1996, p. 449)

Is multiple inheritance really cracked up to what everyone says it is? Does the concept really add to the reusability of system components? Certainly, it does. However, though powerful, multiple inheritance brings a large amount of "complexity" to the table (Singh, 1995, p. 31). Its complexity stems from the amount of development freedom granted to systems engineers, because they can expand the relationships between objects by inheriting from many instead of one. The more relationships that exist between objects, the more complex the system may become, and a multiple inherited object adds to those relationships with each parent object it inherits from (Singh, 1995, p. 34). This can make the design hard to follow, which will later translate to a software-intensive system that is difficult to maintain.

5. The Unified Modeling Language

Modeling is a concept that allows engineers to visualize and test a system before it is built. Models can be physical, much like a miniature replica of an architect's building design while other models are abstract and convey the general idea behind a system's data flow, decisions, or processes. Since software is such an abstract concept for many people to grasp, modeling has become an important aspect of developing software-intensive systems. Modeling can help communicate the design to all parties involved, especially when trying to look at the system from a high-level perspective (Fowler, 2003, p. xxvi).

In the late 1980s, the "Three Amigos," Grady Booch, Ivar Jacobson, and Jim Rumbaugh, introduced the "Unified Method" of modeling object-oriented concepts (Fowler, 2003, p. 8). This modeling technique, along with several other object-oriented modeling languages from the "late 1980s and early 1990s," later grew into what the industry now knows as the Unified Modeling Language (UML) (Fowler, 2003, p. 2). UML is a method of notation that can be applied to diagramming object-oriented software systems. It includes a variety of notations that capture system concepts through the modeling of its activities, objects, relations, and even communications. Since its creation, UML has been referred to, by some, as being the "defacto industry standard" for the modeling of object-oriented systems (Lykins, Friedenthal, & Meilich, 2001, p. 2).

In many cases, systems engineers on object-oriented projects focus their energy on modeling UML class diagrams. Fowler (2003) described UML class diagrams as diagrams that depict the types of objects in a system as well as their relationships, properties, and functions (p. 35). In order for a system to function, all of its components must communicate with each other in order to fulfill its purpose. One giant step toward understanding the unimplemented system is by mapping these relationships using UML class diagrams. A class diagram serves as a very traditional, "top-down" view of a system (Lykins, Friedenthal, & Meilich, 2001, p. 7). Having a visual perspective on abstract object-oriented concepts can help engineers pinpoint system design flaws before entering implementation. UML class diagrams are also helpful in explaining how the designed system works for those not familiar with the intricate details of software production.

5.1 Tools for Engineers and Designers

Systems software engineering is not an easy task, and oftentimes it can be frustrating, painful, and tedious work. The introduction of "Computer-Aided Software Engineering (CASE)" tools to the industry assisted in suppressing the pain of designing complex software-intensive systems (Fowler, 2003, p. 3). Many of these tools focus their expertise toward the modeling of object-oriented software systems, since they tend to become more complex with the use of advanced concepts like multiple inheritance. The trouble with early UML tools was that they never adhered to a standard. The concepts were there, thanks to the original authors of the modeling technique; however, each tool implemented the methods differently from the rest. It was not until the mid 1990s that the Object Management Group (OMG) stepped in and standardized UML, which helped to bring the tools to order from chaos (Fowler, 2003, p. 8).

Today, there are vast quantities of UML tools available to systems software engineers. The Microsoft Corporation developed one such tool, Visio. Microsoft developed the Visio business-modeling suite to allow its users to "visualize, explore, and communicate complex information" (Microsoft Office Visio 2007 top 10 benefits, para. 1). Additionally, Visio gives engineers a chance to share their models with others through the tool's internet capabilities (Microsoft Office Visio 2007 top 10 benefits, para. 12). Long gone are the days where engineers furiously scribbled models and charts onto chalkboards in balmy conference rooms!

The key feature of UML tools is not just their ability to churn out graphically pleasing diagrams of object relationships or activity diagrams, but also their ability to increase productivity and simplify the complexity of object-oriented software system concepts. Further, UML CASE tools can serve as a means to achieve the methodologies behind modeling, simulating, and documenting the process of object-oriented development techniques as applied to the engineering of complex software-intensive systems.

6. Conclusion

Many industry experts believe that systems software engineering is undergoing a "software crisis," whereby the development of software is becoming increasingly complex and difficult to maintain (Johnson, 2002, p. 11). Like the ocean, requirements are in a constant rocking motion driven by the ebb and the flow of customer needs. Additionally, new technologies and methodologies enter the scene, and shake up the way engineers think about designing their systems. This can make for a very tedious development environment, especially when the developing system is invisible to the human eye.

In order to resolve the ongoing crisis, systems software engineers need to focus on reducing the complexity of software-intensive system development. While selecting appropriate development life cycles, documentation methodologies, and modeling techniques can help achieve this goal, it is not enough to succeed in an uphill battle. Engineers should also look toward new development approaches to bring into the pile of possible solutions.

Object-oriented techniques, though not new to the industry, are oftentimes overlooked for their previously heavy association with programming. Systems software engineers tend to have a limited view of object-oriented concepts. They only apply them to the implementation phase of the development process, and do not stop to think how they could take a different approach and apply those same concepts to the other stages of development.

The object-oriented approach brings much more to the table than simple programming methodologies with concepts, such as encapsulation, messaging, and inheritance. Without encapsulation, objects would have very little control over their attributes and behaviors. Messaging gives systems software engineers more control over the relationships their system components possess by establishing controlled communication with other objects. Without messaging capabilities, objects would not provide the necessary level of cohesion needed between system components. Additionally, the concept of inheritance helps achieve component reusability. Objects would always be duplicating functionality from other objects without the availability of a concept such as inheritance. This can bloat a project and increase its development time.

The object-oriented approach is not merely a set of high-level programming techniques. Instead, it is a collection of concepts, which makes complex software-intensive systems development more tangible and easy to understand for humans. By designing a system as an assembly of objects, engineers can better relate to the system in which they are working. This can reduce the complexity of the system as well as make it less susceptible to harmful effects from being developed in a dynamic environment. The object-oriented approach is not the magical sword that will slay the software crisis dragon; however, it is a step in the right direction toward reducing the complexities of developing software-intensive systems.

References

Armstrong, D.J. (February 2006). The quarks of object-oriented development. *Communications of the ACM*, 49 (2), 123-128. Retrieved March 6, 2007 from ProQuest Computing database.

Berard, E.V. (August 23, 1998). *Basic object-oriented concepts*. Retrieved March 6, 2007 from http://www.toa.com/pub/oobasics/oobasics.htm

Cockburn, A.A.R. (1999). The impact of object-orientation on application development. *IBM Systems Journal*, 38 (2/3), 308-332. Retrieved March 6, 2007, from ProQuest Computing database.

Cohen, A.T. (January 1984). Data abstraction, data encapsulation and object-oriented programming. *ACM SIGPLAN Notices*, 19 (1), 31-35. Retrieved January 23, 2007, from ACM Digital Library.

Dorfman, M. & Thayer, R.H. (2000). *Software engineering*. Los Alamitos, California: IEEE Computer Society Press.

Fayad, M.E., Tsai, W., & Fulghum, M.L. (February 1996). Transition to object-oriented software development. *Communications of the ACM*, 39 (2), 108-121. Retrieved January 23, 2007, from ACM Digital Library.

Fowler, M. (2003). *UML distilled: A brief guide to the standard object modeling language, third edition*. Addison-Wesley Professional: Boston.

Garceau, L.R., Jancura, E.G., & Kneiss, J. (January 1993). Object-oriented analysis and design: A new approach to systems development. *Journal of Systems Management*, 44 (1), 25-34. Retrieved January 23, 2007, from ProQuest Computing database.

Hines, M.L. (1995). Object-oriented software engineering: A conceptual approach. *Proceedings of the 1995 Software Engineering Environment Conferences (SEE'95)*, April 5-7, (pp. 194-206). Kansas City, Missouri, United States: IEEE Computer Society Press.

Jacobson, I. (1987). Object-oriented development in an industrial environment. *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'87)*, October 1-6, (pp. 183-191). Retrieved March 6, 2007, from ACM Digital Library.

Johnson, R.A. (2002). Object-oriented analysis and design: What does the research say? *The Journal of Computer Information Systems*, 42 (3), 11-15. Retrieved March 6, 2007, from ProQuest Computing database.

Korson, T., & McGregor, J.D. (September 1990). Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33 (9), 40-60. Retrieved January 32, 2007, from ACM Digital Library.

Lykins, H., Friedenthal, S., & Meilich, A. (January 15, 2001). *Adapting UML for an object oriented systems engineering method (OOSEM)*. Retrieved February 5, 2007 from http://g.oswego.edu/dl/oosdw3/index.html

Microsoft Office Visio 2007 top 10 benefits. (n.d.). Retrieved March 22, 2007 from http://office.microsoft.com/en-us/visio/HA101650311033.aspx

Singh, G.B. (January 1995). Single versus multiple inheritance in object oriented programming. *ACM SIGPLAN OOPS Messenger*, 6 (1), 30-39. Retrieved March 11, 2007, from ACM Digital Library.

Snyder, A. (1986). Encapsulation and inheritance of object-oriented programming languages. *Conference on Object Oriented Programming Systems Languages and Applications*, September 29 – October 2, (pp. 38-45). Portland, Oregon, United States: ACM Press.

Snyder, C.R. (1991). System engineering impact on software development. *Annual International Conference on Ada*, October 21-25, (pp. 425-431). San Jose, California, United States: ACM Press.

Wieringa, R. (December 1998). A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys (CSUR)*, 30 (4), 459-527. Retrieved January 23, 2007, from ACM Digital Library.

Approved Research Paper Proposal Text

Applicable Topic Area: Area F, Systems Development Strategies

Descriptive Title: Applying Object-Oriented Concepts to the Engineering of Complex Software Systems

Reasoning: Software-intensive systems are becoming more large and complex as technologies advance and customers demand more rich and challenging functionality from engineers. Object-oriented concepts can be directly applied to software in terms of programming, yes, but they may also be applied to the development of systems from the ground-up. I work daily in an environment where new software systems are developed with object-oriented methods, and believe that they are not only easier to maintain, but also easier for the customer to understand when faced with somewhat daunting design documents and diagrams. It is my hope that this paper might spark interest in viewing Object-Oriented concepts as something beyond simple programming.

Outline:

1. Introduction
    - 1.1 Structured Development with Object-Oriented Concepts
2. Objects: The Building Blocks
    - 2.1 Objects as Related to Systems Development
    - 2.2 The Natural Perspective
3. Communication and Interaction of System Components
    - 3.1 Messaging
    - 3.2 Methods and Object Behavior
4. Inheritance: Not Just for Geneticists!
    - 4.1 Applying Inheritance to Systems
    - 4.2 Expanding with Multiple Inheritance
5. The Unified Modeling Language
    - 5.1 Tools for Engineers and Designers
6. Conclusion