International Review of Research in Open and Distributed Learning

When is a Learning Object not an Object: A first step towards a theory of learning objects

Mike Sosteric et Susan Hesemeier

Volume 3, numéro 2, octobre 2002

URI : https://id.erudit.org/iderudit/1072878ar
DOI : https://doi.org/10.19173/irrodl.v3i2.106

Aller au sommaire du numéro

Éditeur(s)

Athabasca University Press (AU Press)

ISSN

1492-3831 (numérique)

Découvrir la revue

Citer cet article

Sosteric, M. & Hesemeier, S. (2002). When is a Learning Object not an Object: A first step towards a theory of learning objects. *International Review of Research in Open and Distributed Learning*, *3*(2), 1–16. https://doi.org/10.19173/irrodl.v3i2.106

Résumé de l'article

For some, “learning objects" are the “next big thing” in distance education promising smart learning environments, fantastic economies of scale, and the power to tap into expanding educational markets. While learning objects may be revolutionary in the long term, in the short term, definitional problems and conceptual confusion undermine our ability to understand and critically evaluate the emerging field. This article is an attempt to provide an adequate definition of learning objects by (a) jettisoning useless theoretical links hitherto invoked to theorize learning objects, and (b) reducing the definition of learning objects to the bare essentials. The article closes with suggestions for further research and further refinement of the definition of learning objects.

International Review of Research in Open and Distance Learning
Volume 3, Number 2. *ISSN: 1492-3831*

October - 2002

When is a Learning Object not an Object: A first step towards a theory of learning objects

Mike Sosteric and **Susan Hesemeier**
Athabasca University, Canada – Canada's Open University

Abstract

For some, "learning objects[1] " are the "next big thing" in distance education promising smart learning environments, fantastic economies of scale, and the power to tap into expanding educational markets. While learning objects may be revolutionary in the long term, in the short term, definitional problems and conceptual confusion undermine our ability to understand and critically evaluate the emerging field. This article is an attempt to provide an adequate definition of learning objects by: a) jettisoning useless theoretical links hitherto invoked to theorize learning objects; and b) reducing the definition of learning objects to the bare essentials. The article closes with suggestions for further research and further refinement of the definition of learning objects.

Introduction

> *Fill your bowl to the brim and it will spill. Keep sharpening your knife and it will blunt. Chase after money and security and your heart will never unclench. Care about people's approval and you will be their prisoner.*

The advent of the Internet and the expansion of the World Wide Web have created new communication options for our society. New options for leisure activities (surfing the Web, playing games), commerce (Web shopping), and social development (Web activism), to name only a few, have all emerged within the last few years. It is not hyperbole to say we now live in a connected society where access to information has become the defining life characteristic for many of those fortunate enough to enjoy easy access to information technologies. It is not surprising that this enhanced access to information has influenced the way we live our lives. Authors will disagree about the quality and quantity of this influence (Webster, 1997), but few would care to argue that changes have been significant in one aspect or another.

Our concern in this article is with information technology as it is applied to the educational process. But our concern is not with information technology and education in general, but with a specific example of information technology that is being created to bolster the educational system (K-12 and post secondary), known as a *learning object.*

What is a learning object? Good definitions are difficult to find, and it is really the task of this article to begin the process of developing an adequate definition by: a) jettisoning inappropriate theoretical formations; and b) simplifying our definition of learning object.

For the purposes of introduction, however, we can say that a learning object is a digital file used in educational settings to support instruction (from K-12, and all levels of post-secondary instruction). Later in this article, we will discuss how learning objects have special characteristics that distinguish them from more mundane learning resources of the type most educators would be familiar with.

Learning objects have been on the educational agenda for several years now (IEEE, 1998). Even so, the corpus of research on learning objects is less than satisfying. This does not mean that research has not been conducted. Organizations such as the IMS Global Learning Consortium (IMS)[2] and the IEEE[3] have contributed significantly by helping to define indexing (metadata) standards for object search and retrieval. There has also been some commercial (Baron, 2000) and educational work accomplished (Careo, 2000). But there is a vacuum in descriptive, analytical, and critical examinations of learning object technologies.

This article is our entry into what will hopefully be a dynamic and energetic fray. In an attempt to capture their *true nature*, we provide an overview of learning objects. In order to evaluate the usefulness of thinking about learning objects in terms of Computing Science (CS) programming techniques, we start by looking at past attempts to define learning objects, and then continue by looking at the ostensible link between learning objects and Object Oriented Programming theory (OOP for short). We end our examination of OOP theory by concluding that CS OOP theory has *little* to offer in our attempt to define and understand learning objects. Finally, we conclude the article with a short working definition and suggestions as to where future work is needed in the fleshing out of our understanding of learning objects.

What is a Learning Object?

More than a few words have been produced while trying to give a clear picture of what learning objects are all about. Yet confusion is apparent in the literature, as no consistent definition of learning objects seems to exist. A recent article in *Learning Circuits* highlights this difficulty.

> *It may surprise you that no single learning object definition exists within the e-learning industry. Learning objects are different things to different e-learning professionals. In fact, there seems to be as many definitions as there are people to ask* (ASTD, 2002: 3).

Several problems have made defining learning objects difficult. One bothersome difficulty is that existing definitions are far too general to be of any use in identifying, developing, or criticizing learning objects. As an example, consider the ASTD (2002: 3) article cited above. It begins with the following definition of a learning object: “At its most basic level, a learning object is a piece of content that’s smaller than a course.” Friesen (2001) also illustrates this particular problem when he quotes from an IEEE definition of learning objects:

> *The Learning Technology Standards Committee (LTSC) defines an object as “any entity, digital or non-digital, which can be used, re-used or referenced during technology supported learning.” The LTSC provides examples of these objects, including “multimedia content, instructional content, learning objectives, instructional software and software tools, and persons, organizations, or events referenced during technology supported learning.”*

To paraphrase the above definition, a learning object is "anything that can be used during technology supported learning." The definition lumps all digital and non-digital "things" into the learning object category. Obviously, a definition that includes "everything" is not a definition at all. There is nothing in such a definition to suggest how we might distinguish a learning object from more mundane technological support or any other learning resource such as a computer or a keyboard, for example, and there is certainly nothing to assist us if we want to develop a learning object. All that we know from this definition is that learning objects are "something" used in some sort of learning environment.

We realize, of course, that one might say that anything digital *could* be used as a learning object. For example, a picture of a rose (or the actual rose itself) could be used in various scientific disciplines to illustrate biological, chemical, or psychological processes. However, this loose definition is problematic for two reasons. On the one hand, most authors seem to assume that objects are more than mere digital files. As we will see below, most authors like to attribute several special features to learning objects such as reusability, searchability, etc. At the least, our definition needs to include these special features. Of course, including these special features will have the net result of excluding those digital files that do not have the required features.

On the other hand, "things" (and this includes more mundane things used as traditional learning resources) do not become useful in learning environments until they are attached a context to them. Consider this picture http://aloha.netera.ca/uploads/crdc/unt5049b.jpg, which exists inside a Canadian learning repository as a learning object. This image is a piece of multimedia content that can be used during technology-supported learning. However, just looking at the picture linked above teaches us nothing. Are we to learn something about religious devotion, or respect for elders, or multiculturalism, or foreign languages, or the creation of posters? We simply do not know this from casual observation. What would make the above image a "learning object," would be additional information that would allow an instructor or instructional designer (or perhaps even an automated program) to know how to use the object in an educational setting.

In the low-tech world, the instructor normally provides this contextual information, by harvesting objects and putting them onto projection screens, or passing them around to students while engaging in lengthy discourse about them. In fact, instructors provide much more than just contextual information; they interpret objects and creatively reorganize their context, and this requires a vast amount of background information. This is a critical function of instructors and its importance is recognized in the learning object literature. By developing learning object metadata standards that provide the necessary context for the educational resource, the IMS (2000) and the IEEE have helped to provide the necessary infrastructure for contextualized learning objects that has, in the past, been provided almost solely by instructors.

In the literature on learning objects, the importance of context is not in question. The point we are asserting here is that because "context" is so important, it should be made part of the definition. A learning object is not just *any* digital file or any object under the sun. At the least, anything that could be considered a learning object would need associated instructional information. This occurs even with mundane "objects." Images are often placed in textbooks, but the images themselves are always captioned and explanatory material is provided in the text. Of course, in technological settings where the goal is to use these objects in semi-automated instructional systems, the provision of this type of instructional context is critical.

Although authors tend to want to include digital and non-digital content as learning objects, we do not feel this is useful. As they are applied in the real world, learning objects are clearly digital objects. Repositories and standards, and all the work being done on learning objects, refer to digital objects. It makes little sense to include a universe of learning resources when there does not seem to be any real intention to include them in practical work.

A look at the pedagogical intention behind the production of objects is also necessary. Clearly, although many digital objects *could* be construed as learning objects, not all digital files are learning objects. Pornography is one obvious example. Other objects may or may not become learning objects, as pedagogical intent is required for that to happen. For example, a Unix utility program for listing files may be a learning object. But it would only become a learning object if someone decided to use it as one. Intent is necessary, and this brings us to the last component of the learning object definition: associated metadata. We have already seen that files are not useful as learning objects without the provision of context. A rose might be a rose by any other name, but it is not an object unless there is some discourse associated with it.

Armed with this initially simplistic perspective on learning objects, let us now take a first stab at providing a definition:

> *A learning object is a digital file (image, movie, etc.,) intended to be used for pedagogical purposes, which includes, either internally or via association, suggestions on the appropriate context within which to utilize the object.*

There is reasonable clarity in this definition. It usefully limits the universe of learning objects, and it flows from current literature and practice.

If writers in this area stopped at this definition, things would be acceptable. However, writers in this area seem to want to make learning objects "sexier" than they really are. As a result, several attempts have been made to dress up the definition of learning objects. One of the most *counterproductive* approaches has been for theorists to draw on the discipline of computing science and, in particular, object-oriented programming (OOP) for additional theoretical grist.[4] We can see this in the following definition by Quinn (2000):

> *The learning object (LO) model is characterized by the belief that we can create independent chunks of educational content that provide an educational experience for some pedagogical purpose. Drawing on the object-oriented programming (OOP) model, this approach asserts that these chunks are self contained, though they may contain references to other objects; and they may be combined or sequenced to form longer educational interactions. These chunks of educational content may be of any type – interactive, passive – and they may be of any format or media type. A learning object is not necessarily a digital object . . .*
>
> (Quinn, 2000).

Note again the tendency to make anything under the sun a learning object. But putting this aside, our real concern is to assess whether or not we can usefully extract a sensible understanding of what a learning object might be from CS definitions of objects. According to Quinn's definition above, learning objects are: a) self contained; b) modular (i.e., they can be sequenced, combined, etc.); and c) interactive or passive.

The definitional extensions provided by Quinn are less than satisfactory. The problem is that even though Quinn makes a connection to OOP theory, the definition of a learning object is reduced to a thin description of features (i.e., objects can be combined, sequenced, and contain "references") that do not contribute to our ability to understand or visualize learning objects. It almost seems as if Quinn does not really understand what a CS object is all about and can only provide a terminological *gloss*.

We could ignore the inadequate definition of learning objects above if it were the only one. But other authors also provide similarly thin definitions linked to OOP theory. Robson (1999), for example, begins his definition by stating that learning objects are learning resources in an "object-oriented model" and then goes on, like Quinn, to provide terse feature sets for learning objects:

> *Learning resources are objects in an object-oriented model. They have methods and properties. Typically methods include rendering and assessment methods. Typical properties include content and relationships to other resources* (Robson, 1999).

As with Quinn's definition, the problem with Robson's definition is one of depth. His definition may only be useful to someone who has experience with object oriented programming methodology. But without significant background knowledge, we have no way to know what exactly a method is, what the properties are, and what these technical features of learning objects provide in the way of functionality for learning objects. In short, without knowing more about CS's application of object oriented programming, how can we assess the appropriateness of CS theory to our understanding of learning objects?

The answer to that question is that we cannot. And this is a significant problem. Authors toss around theoretical connections to object oriented theory with insufficient theoretical rigor. Although there is nothing wrong with borrowing concepts from object theory to develop our ideas about learning objects, we must do so carefully. We cannot just adopt the concept of "objects," and its related terminology such as "references," "methods," etc., without carefully specifying whether or not a method for a learning object is the same as a method for a code object.

The bottom line here is: whether or not code objects really provide suitable guidance for us in theorizing and creating learning objects? We believe the answer to this question is a resounding "no." We believe most authors will admit this when pressed. As Friesen (2001) notes, not only is there conceptual confusion in the literature and no general agreement on how to map the features of OOP programming objects to learning objects, the fit between the two also seems to be counterintuitive.

What senses of the word "object" are [sic] can be profitably applied to the notion of "educational objects"? The separation [sic] educational object and metadata seems to run counter to the combination of code and data that is said to define software objects (Friesen, 2001).

What to make of this then? We believe we need to jettison object oriented theory altogether and proceed to define learning objects on their own terms. However, recognizing that there may be some resistance to this strategy, in the next section we take a more detailed look at the core concepts of OOP theory to see how well they apply to learning objects. The next section is moderately technical and can be skipped by those

readers so inclined. The conclusion, at the end of the article, simply suggests that we reject the connection to OOP theory when defining learning objects.

The Etiology of Learning Objects

> *"Object-orientation is a new technology based on objects and classes. It presently represents the best methodological framework for software engineering and its pragmatics provides the foundation for a systematic engineering discipline. By providing first class support for the objects and classes of objects of an application domain, the object oriented paradigm precepts better modeling and implementation of systems. Objects provide a canonical focus throughout analysis, design, and implementation by emphasizing the state, behavior, and interaction of objects in its models, providing the desirable property of seamlessness between activities."*
>
> *Robert John Hathaway*

> *"Object-oriented languages and systems are a developing technology. There can be no agreement on the set of features and mechanisms that belong in an object oriented language since the paradigm is far too general to be tied down. We can expect to see new ideas in object-oriented systems for many years to come."*
>
> *Oscar Nierstrasz*[5]

As noted above, the concept of an "object" is taken from CS theory where it has a precise, if evolving, meaning. One of the most succinct definitions of computing objects we have found to date is provided by Conway (2000, p2), who notes: "An object is an access mechanism for data. In most object-oriented languages that means that objects act as containers for data or, at least, containers for pointers to data. But in the more general sense, *anything* that provides access to data – a variable, a subroutine, a file handle – may be thought of as an object."

This is a useful starting point. Objects are containers of data. This does not contradict our definition of learning objects – i.e., pedagogical intent, associated metadata, digital file — nor does it enhance the definition we have set up so far for the learning object. Our definition that learning objects are digital files implies, without needing comment, that these objects would contain data. With this in mind, we will need to delve deeper into OOP theory to see sharper contradictions.

A container of data, digital or otherwise, is somewhat useless unless there is a way to access and manipulate data. And in fact, OOP theory extends the definition of objects to include access methods. In computing science, objects will always be written to provide encapsulated access to the attributes (data) of an object.

The notion of encapsulated access to data basically means that in an OOP program, the only way to read or write an object's data is "through certain subroutines associated with the object" (Conway, 2000: 2). In OOP, subroutines are renamed as *methods* to distinguish them from the more mundane notions of subroutines and functions in unstructured programming. In OOP terminology, we say we access and manipulate object data (attributes) with *methods*. Let us explore these terms in more detail.

Most people with basic programming experience will be familiar with a program subroutine or function. Typically, a subroutine or function will provide some sort of

service to the programmer. The code that provides this service, perhaps like printing a block of information, is moved out of the main program code into a subroutine. This is done for several reasons: it makes code easier to follow, hides program complexity, facilitates code re-use (by encouraging programmers to code general routines), and increases the modularity of programs[6] . In OOP theory, this is called *encapsulation*, and "methods" (the OOP version of "functions") do pretty much the same thing. Interestingly, many of the features that authors on learning objects attribute to OOP theory really belong to general programmer guidelines.

A typical subroutine to print a user's name would look something like this:

```
name($name) {
sprintf("The student's name is %s", $name);
return 1;
}
```

The above example is simple, but it illustrates the functionality of subroutines. It takes a variable (in this case the student's name), and then prints it to some output device in a formatted string. More generally, subroutines take in data (we say we "pass" data to a subroutine), manipulate it, and then output or store it.

OOP methods share all the features of regular subroutines. Methods are designed to allow programmers to re-use code; they also provide encapsulation. However, when individuals program using OOP methodology, they go farther that the encapsulation associated with subroutines because, in addition to wrapping code, they also encapsulate data. This difference is not immediately transparent to the non-programmer.

The same print name function, as a method, might look something like this:

```
name() {
sprintf("The student's name is %s", $this->name);
return 1;
}
```

Notice any difference? Really the only difference is that, unlike the function or subroutine in traditional programming, variables (e.g., the student's name) are not "passed" to the subroutine. To rephrase this point, the programmer does not pass data to the method. Understanding this difference is the key to unlocking the mysteries of OOP programming.

In an OOP program, the program knows which name to print because the method is part of an object and the object itself contains (or encapsulates) the data, as we stated in the definition of the "object" above. The method can access data at will without ever having to be told directly by the programmer what the data are. This "magic," as it may seem, occurs with a lot of up front grunt work. In OOP programming, objects are designed to encapsulate meaningful blocks of data, and then incorporate extra code that allows your object to carry around the relevant data.

Once the object is designed, it can then be used in higher-level code. To do that, however, the object must first be "created" — in other words, data must be gathered and spaces must be created in memory to store that data, and methods for accessing that data. In OOP parlance we say we *construct* the object.

In a program, constructing an object begins with a call to an object constructor. A constructor might look something like this:

$myUser = new User('userid'=>239480);

The above object constructor (signified by the "User" keyword) is simply another "method." The key difference is that the constructor has a specific role or function to play. When programmers write the code for the object, they put all the code they need to create the user object or "data container" inside the User "method" or constructor. Calling the constructor creates the object by gathering data, associating methods, and clearing memory for storage.

In this case the object takes a student ID, finds the associated student, creates data structures to store student data, and returns a user object, which is then stored in the variable myUser for future access. Code in the User constructor might look something like this:

```
$query=qq! SELECT * FROM Users WHERE User_id =? !;
  unless ($user=Bazaar::DB->fetchrow_hashref(
                                -query=>$query,
                                -bind=>$bindid) )   {
        $this->user = Bazaar::DB->fetchrow_hashref(
                                                        -query=>$query,
                                                        -bind=>1)};
```

The code above is a real world example – though stripped down by several hundred lines in order to isolate a single key function and to better illustrate the purpose of a constructor. In the above example, we basically execute a Structured Query Language (SQL) command that connects to a database, and then grabs all the information available for the user whose "User_id" is equal to 239480, and stores this information in a special variable (a reference encapsulated in the object) called "$this->user." By storing information in a variable like this, the data then becomes available to the programmer as part of the object.

Once the user is created, it is possible to manipulate that user object in the code much easier than with traditional functions and subroutines, because it is not necessary to worry about the data associated with the user or the code used to access and manipulate that data. The programmer does not need to pass the user's name in and out, or worry about how to change said name. All data and methods for accessing data are encapsulated inside the well-designed object.

For example, it is possible to:

output the user's name

```
print $user->Username()
```

email the user a message

```
$user->email($message)
```

or delete the user.

$user->delete()

It is possible to even change the properties of the object. In the example below, we change the user's first name from whatever it was before, to "Mike."

$user->username('firstname=>"Mike")

All this is done with incredible ease. Of course, behind these simple one-liners of code there is also considerable program code. But the code examples are used here to show that the OOP method hides that complexity and encapsulates the object properties so that we can easily program with the object. OOP is really a way to reduce the amount of complexity involved in creating large-scale applications.[7]

There is one primary disadvantage of OOP programming — the requirement of a large amount of overhead. To do anything with an object requires more initial planning and coding that with traditional methods. With OOP, one must:

1) Do up-front conceptualization of the object (i.e., determine in advance a reasonable set of properties and methods)

2) Write object constructors

3) Write code to encapsulate and deliver properties

4) Write object methods

All of these tasks begin before one even starts to code the main program routines. All of this extra code requires extra time for the programmer and extra resources from the computer. However, disadvantages associated with OOP programs are far out weighed by their advantages, especially when working on larger projects.

One important advantage is that OOP methodology provides clean code that is easy to understand and sift through for programmers. This is important for larger projects that may see several new programmers enter the project over its lifespan. OOP makes it easier for new programmers to navigate the code logic because the logic is separated from the implementation. For example, if you were a new programmer trying to understand what a particular block of code does, you would see the following:

$user = new User('user_id'=23423); $user->archive() $user->delete()

They would instantly be able to tell what is happening in the code. As new programmers on the project, they do not need to know (or even care) about the several hundred lines of code that are needed to construct the user and then copy the user's data to an archive and delete the user. They can begin modifying program logic immediately. Of course, there may come a time when they may need to debug the archiving function of the program. If that day comes, they will then need to peer into the black box of the archive function and learn the internals of the method. But that is also facilitated by OOP methods. They will know exactly where to start because the program logic is clearly exposed for them. They will not have to deal with the spaghetti-type mess that is common for non-OOP programs.

A second important advantage of using OOP methodology is that the objects and the code are highly *extensible* and extendable. It is easy and quick to add functionality to an OOP

program without adding additional bugs. The way that the objects are constructed gives programmers a convenient "container" in which they may put any additional properties or methods that might become useful to the program at a latter date. For example, if it is necessary to add a user's honorific (e.g., 'Dr.'), the required code and routines can easily be added into the user object. In fact, chances are that there is already a code template in the program that may be used for just that purpose. Typically, functionality can be added to OOP programs in a fraction of the time required for a regularly coded project. And, the bigger the project, the bigger this advantage becomes.

A third benefit of OOP programming is that the code is *reusable*. In programming, reusability means a number of things. First of all, reusability in OOP methodology means that the object code itself can be reused inside the same program (this is called cloning). We have already seen this, and it simply means that more than one copy of an object may be used at the same time. In our code example above, we created multiple user objects, such as we see here:

$user_one = new User('userid'=>239480) $user_two = new User('userid'=>480);

Object reusability also refers to the fact that objects can inherit properties and methods from other objects. Downes (2001) has a solid explication of what this means when he describes the process of moving from simple objects to more complex objects through the inheritance process. Inheriting the methods and properties of other objects in large scale projects[8] saves coding and conceptualization time.[9]

There is one other instance in which OOP code can be seen as reusable. This is in the simple ability to cut and paste large blocks of OOP code and insert it into other programs where it may be modified for other purposes. This "feature" of OOP code comes from the highly modular nature of an OOP program, in which all data is encapsulated and all methods have easily definable purposes. It is thus easy to chop and hack out sections of code for *reuse* in other areas.

To summarize, then, OOP methodology is preferred for larger programming projects because it provides a methodology that makes it easier to understand code logic, easier to debug code when necessary, easier to extend and modify code as project needs evolve, and easier to reuse objects and the code behind these objects.

It does this by providing programmers with a set of conceptual tools that help them organize code and think of their programs in terms of meaningful units. Programmers create objects that provide meaningful containers whereby they can encapsulate data and hide functionality in methods. It is this encapsulation and the methods whereby object data are accessed and manipulated that provide the infrastructure that makes possible the benefits of OOP summarized above.

Learning Objects

Let us recall our Careo example: http://aloha.netera.ca/uploads/crdc/unit60022b.jpg. Some readers might be wondering what OOP methodology has to do with a picture as a "learning object." As was noted above, there is conceptual confusion in the literature and an inability to map the features of OOP programming objects to learning objects dominates the literature (Friesen, 2001).

Friesen (2001) summarizes the thinking in the literature on how computing science objects map to learning objects, when he notes that authors"… most often identify 'modularity', 'interoperability', [and] 'discoverability' as important attributes of educational objects." However, examining these features only emphasizes the theoretical morass. Not only are interoperability and discoverability decidedly not features of OOP programming (as we have seen), there is no general agreement on how each of these items should be conceived or mapped when it comes to learning objects themselves.

To complicate matters further, Friesen points to three more-or-less distinct definitions of the term *modularity* in the literature. Ironically, none of these definitions seem to be at all helpful in theorizing learning objects:

> *Educational objects, as Longmire describes them, must be modular, "free standing, non-sequential, coherent and unitary." Others describe the same idea using slightly different terms. Roschelle, et. al. (1998) states that the object must be adaptable "without the help of the original developers to meet unforeseen needs." According to Ip, et. al., the object must be constructed in such a way that its users "need not worry about the component's inner complexity." The educational object, in other words, should be a "black box" in the sense described in the theory of object-oriented design.*

The above definitions are not helpful. Some parts of the conceptual paradigm, such as the requirement that learning objects be freestanding, non-sequential, coherent, or adaptable, do not map to OOP theory at all. A programming object *may* be some or all of these things, depending upon what the original authors might mean by these definitions. But a programming object also may not be freestanding and non-sequential. Some objects may, for example, be useful only in the restricted context of the program or sub-program for which they were designed. As may be recalled, the real nature of CS objects comes from they way in which they encapsulate code and data. As for the modifiability of an object by other than the original developer, that is a desideratum of *all* program code and is not the exclusive domain of OOP theory.

To be fair, there is a correspondence between the notion of modularity in the concept of a black box and learning object. But here the link to computing science objects actually reduces our understanding by introducing concepts that confuse they layman and require considerable explanation and modification before they can become useful. Friesen points to Berard's explication of objects: "Specifically, the underlying implementations of objects are hidden from those that use the object. In object-oriented systems, it is only the producer (creator, designer, or builder) of an object that knows the details about the internal construction of that object. The consumers (users) of an object are denied knowledge of the inner workings of the object" (Berard quoted in Friesen, 2001).

In terms of the understanding of a CS object, this is exactly right. Here the consumers are other programmers, and it is not necessary for them to understand the inner workings of code. They simply call the black box within the appropriate parameters.

$user->email($message)

However, in the case of the CS object, the notion of the *consumer* is strictly associated with the programmer. The *consumer* of the above method could be the original programmer using her own code object or a third-party programmer who is part of a

development team working on a larger project. The consumer could even be an unknown programmer making use of some more generic library of code. This is easy enough.

Confusion enters the world of the learning object when we search for a *consumer* of said object (as required by CS theory). Here, in looking for an appropriate mirror concept, we end up trying to incorporate even more unnecessary concepts from OOP theory, thus clouding our understanding even further. In the quote from Friesen (2001) an attempt is made to show how applet users as consumers are faced with the *black box* of code design and how these consumers use an applet's *interface* to interact with it:

> *Initially, this principle would seem most appropriate for educational objects, especially for software components. It certainly would seem like a good way to characterize the operation of "executable" educational objects like Java Applets or Flash components. For example, the Java applets collected by the Educational Object Economy (or the EOE, one of the first repositories of educational objects), for example, would seem to conform to this characteristic. Users of an object are denied knowledge of the most detailed, inner workings of the objects. Instead, they must deal with the object via one of two "interfaces" identified by Berard as standard for software objects in general: "The 'public' interface that is open (visible) to everybody", and "the 'parameter' interface" providing the instructor with a limited ability to customize the operation of the object* (Friesen, 2001).

Sadly, the concept of interface does not map outside of the internal workings of the Java applet or OOP theory either. To be sure, programs can have *interfaces*. But they are not the same types of interfaces spoken of in OOP theory when we speak of an object having an interface and there is little meaning in trying to connect the two. In OOP theory, or OOP practice, an object's "interface" means, simply, the methods that are available for manipulating the object's data. For example, a user object might have the following *interface*:

Public $user->rename() Public $user->delete() Public $user->archive() Public $user->copy Public $user->move() Public $user->email()
Private $user->frobnicate() Private $user->dbConnect() Private $user->dbDisconnect

The object interface here is *all* the methods that the programmer created for manipulating data. The interface itself can be divided into *public* and *private* components. The public side of the interface (easily recognized above by the modifier keyword "public") involves those methods to which the application programmer has access. This means that if I am creating a grade book, for example, and want to use an already available user object, I can only use the public methods of that object. The private methods are those that cannot be called outside of the object itself, and to which I cannot have access. Private methods are *internal* to the object.

Typically, private methods are used by the object itself. For example, the user object might contain a public method to rename a user. The method rename might have the following code:

```
Public Rename() {
 $this->dbConnect();
 $newuser=$user->copy();
```

```
$user->delete();
$this->dbDisconnect();}
```

The private methods of dbConnect and dbDisconnect are, basically, internal *utility* functions of the object itself. They are used by the routine to manage a database connection. This is a benefit for the programmer who is using the object and part of the object's encapsulation. A *consumer* of the user object, though, would not have to manage the db connections. The object handles that. In fact, some OOP languages, such as Java, make it impossible to use private methods.

There are good reasons for hiding some methods from the user. However, we will not go into detail in this article. Suffice it to say, that hiding unnecessary functions from programmers reduces bugs and enhances the object's long-term utility to the programmer. By exposing only certain methods, and by guaranteeing that the prototype of these methods never changes, programmers can rest assured that future modifications to the way an object is implemented will not affect the programs into which they have incorporated the object.

Conclusion: To object or not to object

If the above paragraph sounds like so much techno-speak to you, then perhaps we can use that to segue into the main conclusion of this article. Although OOP theory has many interesting concepts wrapped up in a lot of fancy words, the applicability of OOP theory and/or methods to our understanding of learning objects is marginal, at best, and absolutely counterproductive at worst. This can be clearly seen when we consider that: a) few concepts of OOP theory have anything at all to do with learning objects; and b) those that do have marginal applicability actually end up muddying the waters. In simple terms, we end up wasting considerable time and energy trying for force learning objects into an object oriented model.

This is not exactly an original insight. Other authors recognize these difficulties and have even suggested the need to jettison most of the borrowing from OOP theory. Friesen (2001), for example, ultimately reduces the contribution of OOP to only providing the notion of reusability of objects. But even this is not satisfactory, because we really connote something different when we talk about the reusability of learning objects. As well, we do not need OOP theory to define our objects as reusable or modular or platform independent or anything else. In fact, we would probably advance much faster in our understanding of learning objects if we did not use OOP at all. We spend so much time negotiating with the ghosts of OOP theory, that we cut ourselves off from exploring more appropriate theoretical foundations for learning objects. There is a rich literature in developmental psychology, sociology, and even computation that deals directly with issues relevant to our understanding of learning objects, and it is in these resources that we should be looking if we wish to develop our definitions and refine our understanding of learning objects.

So if we jettison OOP theory, where does this leave us in terms of a definition of learning objects? The definition we introduced at the outset is still useful:

> *A learning object is a digital file (image, movie, etc.) intended to be used for pedagogical purposes, which includes, either internally or via association, suggestions on the appropriate context within which to utilize the object.*

This definition is a useful starting point, though it is far from complete. As we have seen, learning object users have considerable expectations about how they will perform. They are viewed as more than mere learning resources or as providers of a host of fancy features that will make them useful pedagogically, economically, and politically. Accordingly, in our wildest dreams and fancies, an object is not an object is not an object. It is much more.

Just how much more is a useful question that we need to explore in more detail. The map for that exploration can be easily laid. We need to ask several questions, including:

> 1. What is the point/purpose of learning objects? Are they here to solve problems in the education system? Are they here to enhance current instruction? Do they form part of a revolutionary front that will transform the provision of face-to-face or distance education?
>
> 2. What features of learning objects will help us realize our objectives (as noted above)? Can simple image files function as objects or must these image files be enhanced in several ways to meet our objectives?
>
> 3. If files need to be enhanced, what technologies will we draw upon to achieve our objectives? Obviously, our choice of technology will need to be guided by a clear set of objectives and an outline of which objects are intended to be met.
>
> 4. What role will standards play? A lot of work has been done to develop standards for meta-data. Given the purposes and features of learning objects, will this work be relevant? Or can we get by with simpler notions of meta-data?
>
> 5. How will we evaluate objects from a practical and/or theoretical standpoint? Although we have not broached the topic at all, as a collective, we have high expectations about how learning objects will perform in the new economy. However, we have no way to evaluate our claims. Nor, does it seem, that we are ashamed to reduce ourselves to polemical justifications (Downes, 2001). Yet, if we are to be taken seriously, we will need to develop evaluative mechanisms. And this brings us to the following question.
>
> 6. What theories can we draw upon to understand learning objects? We need theories, because it is in the development of theories of learning objects that we will find the means to criticize, evaluate and evolve our understanding and use of learning objects. Our theories can be imported from, for example, instructional design, or modified and distilled from eclectic sources. However, we need appropriate theoretical underpinnings.

This last question could also just as easily be captured in the first question above. The process of answering these questions will be iterative. We will have to constantly move back and forth between theory and standards, and actual implementations as we evolve our understanding of learning objects and their applications. We will need to *ground* our theory in implementation details and research on the pedagogical effectiveness of learning objects. That is, we need to make sure we can implement our notions of objects in code and we need to be sure our implementations actually contribute something to the realm of educational theory and practice.

References

ASTD & SmartForce (2002, July). *A Field Guide to Learning Objects. Learning Circuits.* Retrieved October 1, 2002 from: http://www.learningcircuits.org/field_guides

Baron, T. (2000). Learning Object Pioneers. *Learning Circuits* CAREO (2002). Campus Alberta repository of educational objects Retrieved October 19, 2002 from: http://www.careo.org/

Conway, Damian. (20000. *Object Oriented Perl.* Greenwich, CG: Manning

Downes, Stephen (2001). Learning Objects: Resources For Distance Education Worldwide. *International Review of Research in Open and Distance Education* Vol.2(1). Retrieved October 19, 2002 from: www.irrodl.org

Friesen, Norm (2001). What are Educational Objects. *Interactive Learning Environments*. 9(3). Retrieved October 19, 2002 from: http://www.careo.org/documents/objects.html

IEEE Learning Technology Standards Committee (1998). *Learning Object Metadata (LOM)*: Draft Document v 2.1.

IMS Global Learning Consortium (2000). *IMS Learning Resource Meta-data Best Practices and Implementation Guide* v1.1

Ip, A., Fritze, P., and Ji, G. (1997). *Enabling Re-usability of courseware components with Web-based "virtual apparatus."* unpublished document.

Quinn, C. N. (1999). Learning Objects and Instruction Components. *International Forum of Educational Technology & Society.* In R. Robson, *Object-oriented Instructional Design and Web-based Authoring.* Retrieved October 19, 2002 from: http://www.eduworks.com/robby/papers/objectoriented.pdf

Webster, F. (1995). *Theories of the Information society*. New York: Routledge

Endnotes

1. In the literature, the terms "learning object" and "educational object" are used interchangeably. In writing this article, we adopt the term "learning object." However other authors use educational object to refer to the same construct.

2. http://www.imsglobal.org/

3. http://ltsc.ieee.org/

4. The reason for the attempt to connect learning objects to code objects is simple. There is a grammatical affinity between the term 'object' used in 'learning object' and object-oriented programming theory. However, as we will see, grammatical affinity is not sufficient justification for drawing from object-oriented programming theory.

5. http://www.cetus-links.org/oo_infos.html

6. With subroutines, it is possible to create a 1000 line program to, for example, connect to websites, and then reuse that subroutine in multiple parts in the same program, or indeed in different programs altogether. It is even possible to make code more generic and move subroutines into code libraries where they can then be called upon to serve the programmer's whims again and again.

7. OOP hides the complexity of real life applications in other ways as well. For example, with OOP, you can easily manage more than one complex data structure can be managed easily in an intuitive and fluid fashion. It is also possible to have more than one user in your program at the same time. $user_one = new User('userid'=>239480); $user_two = new User('userid'=>480); Because the complexity of these objects is hidden, it becomes easy to juggle multiple data objects without getting variables and references confused. The variables and references are in the object and, once the object is created and debugged, one normally does not have to worry about that complexity again.

8. In practice this feature of OOP is seldom used because in all but the largest projects, it adds unnecessary complexity to programs. If programmers want a student object, they simply code a student object from the start, and do not worry about the added complexity of an object hierarchy.

9. In practice this feature of OOP is seldom used because in all but the largest projects, it adds unnecessary complexity to programs. If programmers want a student object, they simply code a student object from the start, and do not worry about the added complexity of an object hierarchy.