Australasian Journal of Educational Technology

2005, 21(1), 40-59

Object oriented learning objects

Ed Morris
RMIT University

We apply the object oriented software engineering (OOSE) design methodology for software objects (SOs) to learning objects (LOs). OOSE extends and refines design principles for authoring dynamic reusable LOs. Our learning object class (LOC) is a template from which individualised LOs can be dynamically created for, or by, students. The properties of LOCs refine existing LO definitions and design guidelines.

We adapt SO levels of cohesion to LOCs, and illustrate reusability increases when LO lessons are built from LOs like maintainable software systems are built from SOs. We identify facilities required in learning management systems to support object oriented LO lessons that are less predetermined in their sequencing of activities for each student. Our OOSE approach to the design of object oriented LO lessons is independent of, and complementary to, instructional design theory underlying the LO design process, and metadata standards adopted by the IEEE for LO packaging. Our approach produces well structured LOCs with greater reuse potential.

1. Introduction

Our previous research focused on the cost effectiveness (Zuluaga, Morris & Fernandez, 2002) and educational effectiveness (Morris & Zuluaga, 2003) of our online learning approach. This involved both online course development and course delivery phases. We also addressed the deployment, management and scalability of our online courses over a network of learning management system servers (Zuluaga & Morris, 2003).

Each of these mixed mode and 100% online courses utilise mostly textual learning materials, and includes on average four short multimedia supplements such as Java applets, *Flash* animations, voice overs and video clips. We relied more on students interacting with staff during the delivery of an online course than on building interactivity into the online course material during its development. We assumed that these early generation online course materials would soon be replaced as multimedia and learning management systems matured and standardised. So we were not so much concerned with upgrading the original materials over the years,

re-purposing them for new educational programs, or re-packaging them for different media in the future. Nor were we concerned with standardising the packaging of course materials so that they could be re-assembled with materials from other institutions.

Here however, we are concerned with the engineering of e-learning materials for the 'bilities': *interoperability* among different systems connected by the Internet, *accessibility* anytime from another location, *reusability* by other developers to save time and money, *discoverability* in repositories using metadata, *extensibility* of existing courses due to their modular construction, *affordability* due to reduced development costs, and *manageability* by allowing easy changes and updates to small chunks (Computer Education Management Association, 2001).

The concept of 'learning object' (LO) is central to these objectives. A range of definitions of 'learning object' or 'instructional object' exists (Wiley 2001). One that captures a common theme defines learning objects as "small but pedagogically complete segments of instructional content that can be assembled as needed to create larger units of instruction, such as lessons, modules and courses. Learning objects should be stand alone, and be built upon a single learning objective, or a single concept" (Hamel & Ryan-Jones 2002).

Boyle (2002) has proposed LO design principles synthesised from pedagogy and software engineering. From pedagogy, a LO should have a single learning objective. From software engineering, a LO should do one thing and only one thing (strong cohesion), and a LO should have minimal bindings to other LOs (weak coupling).

We expand the above synthesis in section 2 by applying object oriented software engineering (OOSE) design methodology to the design of LOs. We introduce the 'abstraction' of a LO to enable a designer to produce a 'learning object class' (LOC). A LOC is a template from which similar but individualised LOs can be created dynamically during a lesson. (OOSE refers to similar 'objects' being 'instantiated' (created) from a 'class', which encapsulates their shared attributes and activities.) We introduce 'inheritance' so that a designer can evolve a 'child' LOC from its 'parent' LOC, extending and modifying its attributes and activities as desired. The instructional designer can use inheritance to reuse LOCs or re-purpose a lesson by extending and coupling inherently cohesive LOCs. The instructional designer can use instantiation during a lesson to enable student interaction to determine the actual sequence of possible events in a lesson.

In section 3 we illustrate the application of OOSE design methodology to the design of two LOCs. Broadly speaking, one is for the programming

discipline, and the other is for psychology. The first is based on the Java programming language while loop LO of Boyle et al (Boyle, 2003). The second is based on a conflict resolution LO that explains Maddux's five styles of conflict resolution (Rathsack, 2001). These two LOCs from different disciplines demonstrate the general applicability of our approach.

In section 4, we adapt to LOs a scale for grading the cohesion of SOs. We show how to classify a LOs level of cohesion and explain how each of the lower levels further reduces LO reusability. This informs the design of LOs, as we illustrate with the examples from section 3.

Finally, in section 5 we identify facilities required in a learning management system to support object oriented LO lessons. We point out that our OOSE approach to the design of LO lessons is independent of, and complementary to, the instructional design theory underlying the LO design process, and the metadata standards adopted by the IEEE (LTSC, 2003) for LO packaging.

2. Application of object oriented software engineering to learning objects

Software engineering is concerned with the design and implementation of large scale, complex information processing systems that are robust, maintainable, modularly reusable, scalable, and extensible (Pfleeger, 2001). These properties overlap the 'bilities' required of LOs (section 1). This observation underlies the application of software engineering design principles to the design of LOs. Boyle introduced this approach with reference to coupling and cohesion principles for the design of LOs (Boyle, 2002). We extend this approach by applying *object oriented* software engineering (OOSE) design methodology to the design of LOs. OOSE has evolved into a dominant 'branch' in the software engineering 'tree'.

In section 2.1 we show how LOs can be designed with essentially the same techniques used to design software objects (SOs). In section 2.2 we explain how flexible LO lessons can be built from reusable LOs in the same way that maintainable software systems are built from SOs with well-designed interfaces. Our application of OOSE to the design of object oriented LO lessons leads us in section 2.3 to synthesise criteria that define a truly object oriented LO.

2.1 Software objects and learning objects

Object orientation is an approach to software development that organises both the problem and the solution as a collection of discrete 'objects' (Pfleeger, 2001). Each software object (SO) can be based on a physical or

abstract object in the problem space. In the software system solution, the software objects (SOs) collaborate to answer a user's requests.

> Problem: simulate an employee – employer relationship. Solution: Consider an *employee* SO now; the reader can similarly consider an *employer* SO later.

In general the nouns in the problem statement identify the SOs and their attributes. The 'has-a' relationship governs a SO and each of its attributes.

> An employee SO has at least a *name*, a social security (tax) *number* and regular *pay*.

Other attributes of a SO can be discovered by asking "if I am an employee, what should I know?". For instance the problem could indicate that an employment *history* is required.

In general the verbs in the problem statement identify the activities (behaviours, actions, responsibilities, operations) required of a SO.

> An employee SO at the very least *works* and gets paid; the latter possibly comprising 2 activities: *receivePay* and *showPay*.

Other activities of a SO can be discovered by asking "if I am an employee, what should I be able to do?". For instance the problem could indicate that *reportWork* is also required.

The state of a SO at any time is given by the values of its attributes, as determined by the SO's activities. For instance *showPay* should show a higher *pay* value after *receivePay* provides a pay rise.

By analogy with a SO, we consider a LO to have attributes and activities to deliver a single learning objective. Just as the users of a software system (solution) cause interactions between SOs to solve a problem, the students of a 'lesson' can cause interactions between LOs to achieve the lesson's learning outcomes.

> Adapting the above example problem / solution, consider the overall learning objective: understand the employee – employer relationship. Learning outcomes could include knowing the responsibilities of employees in an employment hierarchy. An employee LO can be developed in the same way that an employee SO was developed above. The same attributes and activities can be identified in the same manner as above.

By comparison with a SO, the activities of a LO provide an explanation, not a computation. For instance *receivePay* could explain that pay is in return for work.

In general, a SO is an instance of a software object class (SOC). Abstraction enables SOs with shared attributes and activities to be defined as a single SOC. A SOC acts as a template from which individualised SOs are instantiated (created), as determined by the user's interactions with the software system.

> As a user interacts with a software system that simulates the employee – employer relationship, employees, *bob*, *ted*, *carol* and *alice* could be instantiated. Each SO has a distinct *name*, social security (tax) *number* and *pay*. These SOs could collaborate to perform their *work*.

Analogously, LOs can be created as customised instances of a learning object class (LOC). A LOC is not only a container of learning materials for a single learning objective (attributes), but also a container of operations defined on the materials (activities) that a student interacts with to attain the intended learning outcomes.

> During a lesson, a student could create employees, *bob*, *ted*, *carol* and *alice*. Each LO has at least the distinct attributes identified above. These LOs could collaborate to explain their *work*.

In general, a student initiates a lesson by interacting with a 'driver' that instantiates a LO to service the student's requests. During a LO lesson, other LOs are likely to be instantiated from one or more LOCs. The student interacts with these collaborating LOs to attain their desired learning experience.

Abstraction promotes generality and instantiation provides flexibility and individuality. We assert that LOs can benefit from these qualities, just as SOs do.

The Unified Modelling Language (UML) is a standard for depicting diagrammatically relationships between classes (SOCs), via class diagrams, and between objects (SOs), via object diagrams (Priestley, 1996). Figure 1 shows the Employee SOC in UML and four (4) instance Employee SOs.

Figure 1 could equally show the Employee LOC in UML and four (4) instance Employee LOs.

Access to the attributes and activities encapsulated by a SO is determined by its SOC. In general, attributes of one SO cannot be directly manipulated by another SO. Instead, the SO encapsulating the attribute in question performs the relevant activity in response to a request from another SO. For instance, an employee SO's pay cannot be directly accessed by other SOs; they can only request an employee SO to showPay or receivePay. Not all activities of a SO need be accessible to other SOs. The public activities

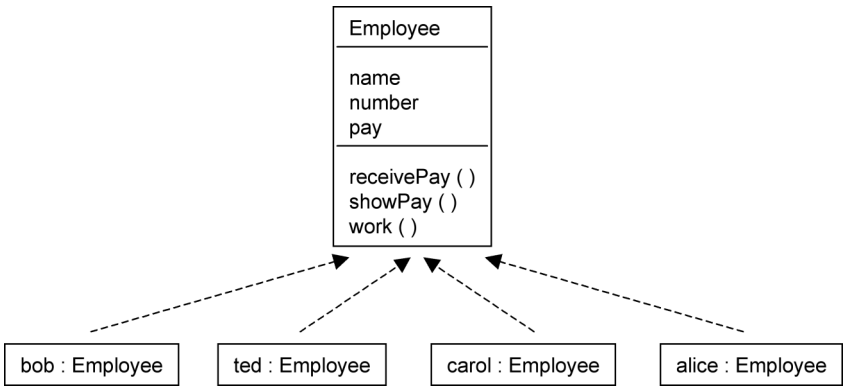

Figure 1: Class Employee and 4 instance objects

supplied by a SOC define an interface that protects the private attributes and activities of its SOs. In effect, SOs request each others' 'services' via the public interface activities supplied by their SOCs. This allows the internals of a SOC to be modified by a programmer without affecting collaborating SOs, provided its interface remains unchanged, eg. receivePay could incorporate a bonus without upsetting any SO that requests showPay.

Encapsulation can be equally applied to LOCs. We assert encapsulation enhances manageability by facilitating updates without requiring changes to collaborating LOCs. Other 'bilities' (section 1) such as reusability and affordability also benefit.

A SOC can be extended into a more specialised SOC by adding further attributes and activities. The 'child' SOC is said to inherit the parent's attributes and activities. The 'is-a' relationship governs a child as a specialised extension of its parent. The child can selectively modify its inherited characteristics too (called polymorphism). For example an Employee SOC could define employment history in terms of career achievements. A junior employee could re-implement its history in terms of final school courses and grades. Inheritance enhances reusability and adaptability of SOCs.

> An *Employee* SOC can be extended to an *AirlineEmployee* SOC on the one hand, and a *HospitalEmployee* SOC on the other hand. An *AirlineEmployee* could add to its inheritance a knowledge of the *travelIndustry* and *airlinePolicies*. A *HospitalEmployee* could add to its inheritance a knowledge of *healthCareIssues* and *hospitalPolicies*. A *Nurse* SOC could further extend the *HospitalEmployee* with knowledge of *patientCare* and the activities to *takeBloodPressure* and *giveInjection*.

Figure 2 depicts the above inheritance hierarchy in a UML class diagram.

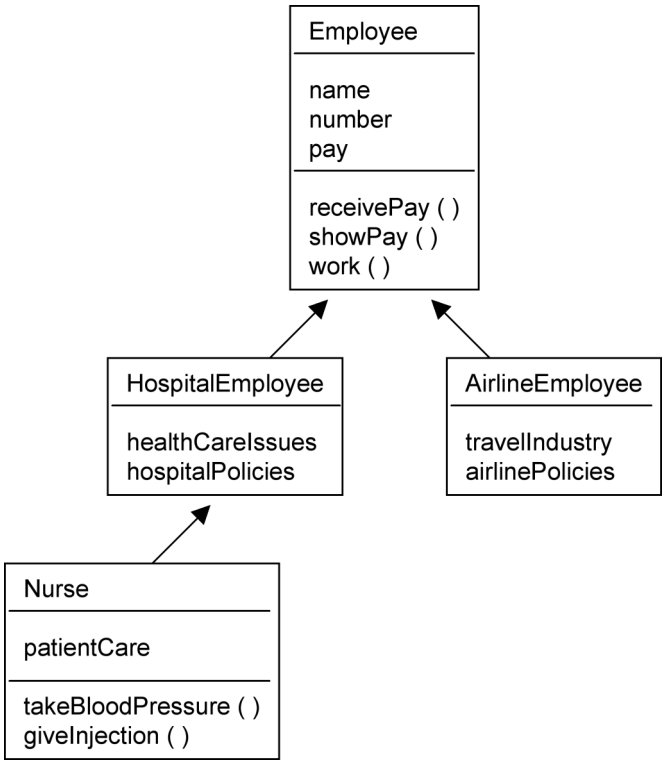

Figure 2: Class Employee and its 'child' classes

Figure 2 could equally depict an Employee LOC inheritance hierarchy in a UML class diagram. We assert that inheritance can enhance reusability and extensibility of LOCs.

2.2 Software systems and learning object lessons

Software systems can be large and complex. Their design can comprise hundreds of SOCs and thousands of interactions of many kinds. Their implementation can amount to millions of lines of code. The software development life cycle (comprising requirements elicitation, analysis, specification, design, implementation, testing and maintenance) can involve numerous teams of professionals of various kinds over many human years. On the other hand, most LOs are designed and implemented by one or two individuals, or a small team, over weeks or months, rather than years. No more than a handful of LOs comprise a typical lesson. So

there is at least an order of magnitude difference in the current scale of software systems design and LO lesson design. However, software systems were originally far smaller. As the underlying hardware improved exponentially, it was still the advent of software engineering design methodologies that facilitated production of larger scale reliable software (Pleeger, 2001). Hopefully, this paper is a contribution toward a LO lesson design methodology that will facilitate the design and implementation of larger scale lessons, courses and educational programs.

An interface in OOSE terms is the boundary around an object that defines which of its attributes and activities are accessible to other objects. An object's attributes remain private by default, but a public 'accessor' activity can be defined in an object to return an attribute to any other object that requests it. An object may also define a public 'mutator' activity to enable an attribute to be altered at the request of other objects. Each interaction between two objects is in the form of a request and an answer. Data can be transferred in both directions – in and out. Such interactions in software systems are generally driven by the user(s). In general objects are dynamically created to provide services in response to user requests.

This SO interaction model is also entirely applicable to a LO lesson. An instructional designer can define the interface of a LOC to provide certain learning activities as services. During a lesson, LOs instantiated from several LOCs can interact to provide a user (student) responsive learning experience.

The Java language also provides a special interface type that can be used to group a number of classes by insisting each class implements all activities specified by the interface. The grouped classes can be considered to have the same 'look and feel'. We think the Java interface type suggests a mechanism for combining LOCs into LO lessons. If a LOC, L, implements interface I along with other LOCs, all these LOCs share a single look and feel. So a user (student) should experience an interactively integrated lesson. If the LOC, L, also implements another interface, say J, then L can integrate with other LOCs that implement interface J. This facilitates reuse of LOC L in a new LO lesson. Multiple interface types can provide different contexts for the one LOC (and its LOs) in different LO lessons. This should aid the 'bilities' listed in section 1, in particular, reusability and extensibility.

2.3 Object oriented learning object criteria

Our above application of OOSE design methodology to the design of LOCs and LO lessons leads us to synthesise the criteria that define a truly object oriented LOC. Our criteria (below) refine the LO definition work of Wiley

and others (Wiley, 2001), and extend LO design guidelines (Hamel, Ryan-Jones, 2002).

1. Each LOC has attributes and activities that meet a single well-defined learning objective and implement measurable learning outcomes in accordance with an instructional theory.
2. Each LOC and its attributes are identified by the nouns in its learning objective and learning outcomes. The verbs identify its activities.
3. Each LOC encapsulates learning activities that are standalone and achievable in a single sitting.
4. Each LOC's attributes and activities contribute packaging metadata (LTSC, 2003), to enhance the 'bilities' listed in section 1 (also see section 4.2).
5. Each LOC in general extends (specialises) its parent LOC's attributes and activities. Inheritance enhances the 'bilities'.
6. Each LOC in general implements a Java-like interface type, which can also be implemented by other LOCs. A single interface for multiple LOCs provides the 'look and feel' for a LO lesson. Implementing multiple interfaces enables a LOC to integrate into a different LO lesson with other LOCs. LOC interfaces enhance the 'bilities'.

Each LO lesson involves student interaction to create LOs on demand and to drive interactions between these LOs to achieve the lesson's learning outcomes. The overall learning activity is necessarily interactive and in general involves (self-)assessment against the learning objectives.

3. Example learning object classes

Below we illustrate our above adaptation of OOSE design methodology to the design of LOCs. The first example is based on the LO developed by Boyle et al (Boyle, 2003) for students to learn how while loops work in the Java programming language. The second example is based on a conflict resolution LO that explains Maddux's five styles of conflict resolution (Rathsack, 2001). We assert that our two LOCs for different disciplines (programming and psychology) demonstrate the general applicability of our OOSE approach to the design of LOCs. Although we write each LOC in Java, the design is our focus, and is language independent. In section 4.3 we demonstrate the advantages of our design over the originals, as measured by the design principles of coupling and cohesion (elaborated in sections 4.1 and 4.2).

3.1 While loop

The Java programming language while-loop LO of Boyle et al (Boyle, 2003) starts by showing the student how a program hammers a nail into wood: while (nail is not flush) hit the nail. Next, the Java code to move a car over a

given distance is displayed, explained, and the student can animate the loop. The student can also step through the code, statement by statement. A second example shows a submarine submerging to a given depth. The code is displayed, explained, animated, and the student can step through. Then the student is asked to build the code from a given set of Java statements to move a horse a given distance. Finally, the student is asked to spot errors in the code to move a lorry a given distance.

Our OOSE design for a while-loop LOC is shown below (Figure 3) as an incomplete class in Java. Class While has one attribute – the loop in question, represented as a string of characters. When an object of class While is instantiated, the loop can be initialised to an input string, or the default generic loop. Class While defines the following activities as operations on this loop – display, explain, animate, step_thru, build, and debug.

```
class While {

   String loop;

   While (String input) {
      // constructor given a loop
      loop = input
   }
   While ( ) {
      // default constructor with generic loop
      loop = “pre-action;” + “while (condition)” + “loop action;” + “post-action”;
   }

   display ( );
   explain ( );
   animate ( );
   step_thru ( );
   build ( );
   debug ( );
}
```

Figure 3: Class While

The student interacts with a Java program that instantiates requisite While objects. For example, if the student follows the sequence intended in Boyle’s LO, the hammer object would be instantiated first. Its code would be displayed, explained and animated. Next, the submarine object would be instantiated, its code displayed, explained, animated and stepped through if desired. Next, the horse object would be instantiated, and the student would build its code. Finally, the lorry object would be instantiated, and the student would debug its code.

Note that another student, perhaps more advanced, could interact with Class While to instantiate While objects (LOs) in another sequence, bypassing some LOs as desired.

3.2 Conflict resolution

Rathsack's conflict resolution LO explains Maddux's five styles for managing conflict. The LO begins by stating that people can disagree, that this can be an opportunity for growth and learning, or it can be detrimental as conflict arises. The ability to manage conflict is important to succeed in one's career and life. The LO then introduces Maddux's matrix depicting 5 styles for managing conflict: avoiding, accommodating, winning/losing, collaborating and compromising. Each style is explained. Then the LO explains that the matrix y-dimension shows increasing assertiveness and the x-dimension shows increasing cooperation, starting from zero at bottom left. This explains the location of each style in the matrix: winning/losing at top left, collaborating at top right, accommodating at bottom right, avoiding at bottom left, and compromise in the centre. The LO then presents 5 conflict scenarios and asks the student to identify the style in use. Finally, the LO explains that Maddux believes no one style is always best for all situations.

Our OOSE design for a conflict resolution LOC is shown in Figure 4 as an incomplete class in Java. Class Conflict has one attribute – the conflict style in question, represented as a string of characters. When an object of class Conflict is instantiated, the style is initialised to an input string. Class Conflict defines the following operations on this style – display, explain, animate, and identify.

The student interacts with a Java program that instantiates requisite Conflict objects. For example, if the student follows the sequence intended in Rathsack's LO, the 'avoiding' object would be instantiated first. This style would be displayed on the matrix (in relation to the other 4 styles), explained, and animated on the matrix (in terms of the x and y dimensions). Next, the 'accommodating' object would be instantiated. Similarly, this style would be displayed on the matrix (in relation to the other 4 styles), explained, and animated on the matrix (in terms of the x and y dimensions). Next, the remaining 3 Conflict style objects would be instantiated. Finally, the Java program that instantiates Conflict objects would ask the student to identify the style in use in a conflict scenario. The Conflict objects could cooperate to randomise this test.

Note that another student, perhaps more advanced, could interact with Class Conflict to instantiate Conflict objects (LOs) in another sequence, bypassing some LOs if desired.

```
class Conflict {

   String style;

   Conflict (String input) {
      // constructor given a style
      style = input
   }

   display ( );
   explain ( );
   animate ( );
   identify ( );
}
```

Figure 4 : Class Conflict

4. Object oriented design principles for learning objects

In a software system, coupling measures how cleanly objects are partitioned. Cohesion measures how closely activities in an object are related. Coupling and cohesion are interdependent measures – the less cohesive an object, the more likely it is coupled with other objects. The more coupled an object is with other objects, the harder it is to alter or upgrade the object in isolation, which lowers maintainability. A strongly coupled object is less reusable without significant maintenance.

In our terminology, the 'objects' referred to above are SOCs, not individually instantiated SOs. Following Boyle (2002), we assert the above applies as much to LOCs in a LO lesson as to SOCs in a software system. Weak coupling between LOCs in a LO lesson promotes the 'bilities' (section 1) in that the maintainability of SOCs is essentially the manageability of LOCs. The more cohesive each LOC is, the less coupling is required when LOCs are reused in a new LO lesson.

Stevens and Myers, (Yourdon & Constantine,1978) devised a table to classify the level of cohesion of a software module (ie. SOC). (Although their work pre-dated OOSE design methodology, it is readily accommodated.) We adapt their 7-level scale to LOCs below. In section 4.2 we show how to classify a LOC's level of cohesion and we explain how each of the lower levels further reduces LOC reusability. We assert that awareness of cohesion levels can improve the design of LOCs, as we illustrate in section 4.3 with the examples from section 3.

4.1 Learning object levels of cohesion

The strongest cohesion is called functional, and the weakest cohesion is called coincidental. Each level in Table 1 is less cohesive that the level above it.

Table 1: Cohesion levels

Cohesion level	Description	Example
Functional (strongest)	Each activity in a LOC contributes to a single learning objective related task or learning outcome.	Learn to calculate net employee salary. This could be one of many tasks an accountant learns. It comprises: getting the gross salary, subtracting legal deductions, computing taxes. Every step contributes to the single purpose outcome of this LOC.
Sequential	The outcome (output) of each activity in a LOC is the input to the next activity in the LOC.	Learn to paint a picture. This LOC could comprise: sketching, painting outlines, coloring shapes, adding texture, signing and dating. Each activity uses the result of the previous activity on the canvas. The picture may be complete, but learning to paint could still be a life-long objective, so the LOC is not functionally complete.
Communi-cational	The activities in a LOC share the same attributes, or inputs and outputs.	Learn to summarise, say, a chapter of a book. This LOC could comprise: reading the chapter, highlighting headings in the chapter, listing key words in the chapter, writing sentences that connect key words in the chapter. Each activity uses the chapter, but not necessarily the result of the previous activity.
Procedural	Control flows from one activity to the next in the LOC, ie. the activities are related solely by their order of execution, which is arbitrary. Data passing in and out of the LOC are unrelated.	Learn to dissect, say, a fish or mouse. This LOC could comprise: cleaning the bench, arranging implements, preparing the specimen, starting experimental notes, using scalpel, recording observations. Each activity leads to the next, but it does not necessarily use the result of any previous activity.
Temporal	The activities in a LOC are related in time only, ie. the activities are executed at about the same time.	Learn to study. This LOC could comprise: turning off the radio and TV, collecting pen, paper and text book, working at one's desk, ignoring phone calls and other distractions, making notes, etc. All these activities occur during study time, but they need not occur in this exact order.

Coincidental (weakest)	The activities in a LOC are unrelated by any of the above.	Learn to tidy up, say, a room. This LOC could comprise: disposing of litter, hanging clothes, finishing a snack, making the bed, vacuuming and so on. Not all these activities need be done (together). The activities are not logically related, nor connected by flow of execution or data.

The reusability issue for each level below functional cohesion is explained in Table 2. Note that the issue at a given level is often in addition to reusability issues at higher levels.

Table 2: Cohesion levels and reusability

Cohesion level	Reusability issue
Sequential	Not as reusable as a functional LOC because the sequencing of its activities cannot be easily altered.
Communic-ational	Either the input or output coupling is generally broader than for the above levels of cohesion. Reuse often needs a subset of this coupling, hence redundant coupling; or a cut down version of the LOC is created, which still duplicates functionality. A communicational LOC can often be split into functional LOCs.
Procedural	Intermediate or partial results are often passed in to or out of a procedural LOC, reducing reusability. It is tempting to combine distinct activities for 'efficiency' or 'convenience', further reducing reusability.
Temporal	Activities in a temporal LOC tend to be related to activities in other LOCs, increasing coupling. Activities in a temporal LOC are often combined because they can occur together. But this compromises reusability in another situation where the activities can occur at different times.
Logical	Broad input coupling is required for a logical LOC to select which activity to perform. The activities are typically combined because they share common parts. Reusability suffers.
Coincidental	A combination of inputs often determines the selected activity. As a result it can be hard to understand a coincidental LOC unless its internal detail is examined. This reduces reusability.

4.2 Determining learning object cohesion

The description or name of a LOC may suffice to determine its level of cohesion, as shown in Table 3.

We note that the presence of any of the above key words could be a valuable indicator for metadata tagging purposes, but further research is required to evaluate reliability.

Table 3: Determining object cohesion by name or description

Cohesion Level	Name or Description
Functional	Simple verb-object phrase.
Sequential	Commas often required.
Communicational	The word 'and' is often present.
Procedural	The word 'or' is often present, or words synonymous with repetition, eg. 'while', 'until'.
Temporal	Time related words apparent, eg. 'start', 'end', 'before', 'after'.
Logical	An 'umbrella' word is present, eg. 'all', 'every', 'total'.
Coincidental	Description or name is meaningless, eg. 'miscellaneous', 'X', 'Z-process'.

In Figure 5 we adapt a decision tree (Page-Jones, 1998) that enables a LOC's level of cohesion to be more accurately determined by asking and answering a few questions, starting at top left.

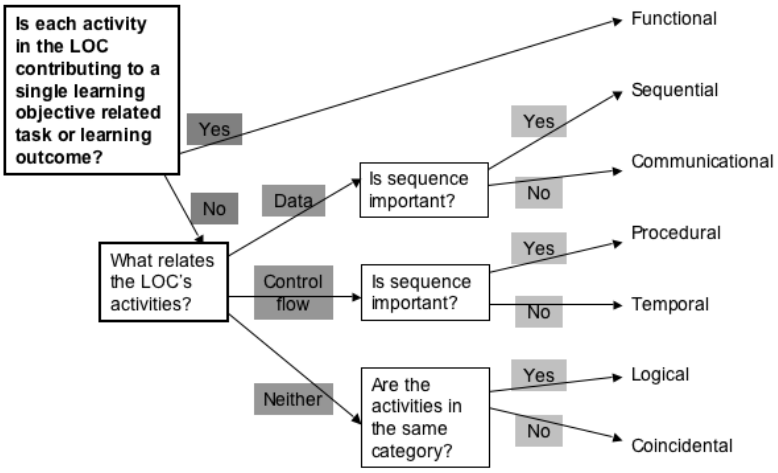

Figure 5: Cohesion decision tree

If all the activities in a LOC share more than one level of cohesion, the LOC has the highest (strongest) of the shared levels of cohesion – chains in parallel rule. If the activities in a LOC exhibit various levels of cohesion, the LOC has the lowest (weakest) level of cohesion – chains in series rule.

4.3 Cohesion of example learning objects

Below we use Figure 5 (above) to establish the cohesion of the examples in section 3.

If we address the first question (top left of Figure 5) for Boyle et al's Java while loop LO, the answer at first appears to be 'yes' – the While LO appears functionally cohesive in that each activity contributes to understanding how while loops work, which is the LO's learning objective. But on closer examination, the activities performed by the While LO do not contribute to one and only one learning objective related task. The LO performs two pairs of similar analysis activities in sequence (hammer and submarine, horse and lorry), followed by one synthesis activity. Data is not passed between the activities, so the sequence is program controlled (ie. chosen by the instructional designer). So the LO exhibits procedural cohesion at best. If we similarly address the OOSE design for a while loop LOC shown in Figure 3, a LO can be instantiated for each of hammer, submarine, horse and lorry. The sequence is determined by the student (via input data), so at least the LOC exhibits communicational cohesion, which is better than procedural cohesion.

The situation is similar for Rathsack's conflict resolution LO and our Conflict LOC shown in Figure 4. If we address the first question (top left of Figure 5) for Rathsack's LO, the answer at first appears to be 'yes' – the LO appears functionally cohesive in that each activity contributes to understanding Maddux's five styles for managing conflict, which is the LO's learning objective. But after closer examination, the activities performed by the Conflict LO do not contribute to one and only one learning objective related task. The LO performs five similar activities in sequence, one for each style. Data is not passed between the activities, so the sequence is program controlled (ie. chosen by the instructional designer). So the LO exhibits procedural cohesion at best. If we similarly address the OOSE design for a Conflict LOC shown in Figure 4, a LO can be instantiated for each of the five styles. The sequence is determined by the student (via input data), so at least the LOC exhibits communicational cohesion, which is better than procedural cohesion.

Since each OOSE designed LOC exhibits stronger cohesion than the original LO, our LOC is likely to require weaker coupling with other LOCs in a new LO lesson, thereby enhancing its reusability.

5. Support for object oriented learning objects

We refer to our OOSE approach to designing LOs and LO lessons (described above) as 'object oriented learning' (OOL). We claim benefits of OOL include the following in addition to enhancing the 'bilities' (section 1). Dynamic instantiation of a LO from its LOC in response to a student's choice enables a LO lesson to not only be more highly interactive but also far less predetermined in its sequence of activities for each student. Also

the instructional designer need not build lessons as a predetermined sequence of LOCs, as the student can be given some choices.

Below we investigate supports available for our OOL approach. We outline how OOL can be accommodated in a LO development project from initial application of an instructional design theory to final implementation in a learning management system.

5.1 Object oriented learning and instructional design theory

It is clear that the design of LOs and LO lessons should be informed by an instructional design theory (Wiley, 2001). After these pedagogical choices are made, an instructional designer can apply our OOL approach, where the focus is on structuring the functionality of LOCs. We assert that OOL is independent of, and complementary to, instructional design theory. Further research is needed to confirm this.

In fact we are presently using OOL to design a LO lesson composed of several LOCs. We intend to use UML during the design process in order to report on its usefulness. But the focus of our study will be on evaluating the effectiveness of our OOL approach on LOC 'bilities' (section 1).

5.2 Object oriented learning and learning management systems

Learning management systems like *Blackboard* and *WebCT* currently facilitate the development of e-learning courses, their collection in a repository, and their delivery to online students at any time over any distance. Learning management systems are also starting to support standardised metadata tagging of LOs to better facilitate the combination of LOs into lessons, courses and educational programs. Our OOL approach contributes to LO metadata tagging as explained in section 4.2. We contend that our approach is automatically accommodated within the pedagogically neutral standards adopted by the IEEE (LTSC, 2003) for LO packaging.

Learning management systems will also need to provide the 'programming language' that enables instructional designers using OOL and students to realise the full potential of the dynamics inherent in the design of LOCs. Our use of Java in section 3 was not only to illustrate the application of OOSE to the design of LOs. Java can also be the programming language used in a learning management system to implement student centred combination of LOCs and the dynamic instantiation of their LOs. However, the power of a general programming language environment such as Java is not necessary for this purpose. Indeed, further research is desirable to produce a complete yet simple programming tool for instructional

designers to use across learning management systems. One avenue to explore is the programmability introduced into the computer aided instruction systems of the past (Gibbons, Richards, 2001). Fortunately, today's graphic user interfaces, more powerful computers, and faster connectivity will make the experience far more friendly for both today's instructional designers and students.

6. Conclusion

We have applied object oriented software engineering (OOSE) design methodology to the design of learning objects (LOs). Our OOL approach extends and refines Boyle's design principles for authoring dynamic reusable LOs (Boyle, 2002) as follows. A prospective LO's attributes and activities are first 'abstracted' into a LO class (LOC). This facilitates dynamic instantiation of an individualised LO for, or by, a student during a lesson; as we illustrated with two example LOCs for different disciplines. We introduced Unified Modelling Language (UML) to illustrate the LOC design process (Figures 1-2). We showed how inheritance and polymorphism further enhance LOC reusability in other lessons.

We explained how our OOL approach can benefit the design of lessons comprising several LOCs. We identified a Java-like interface type as a useful mechanism to assist reuse and re-purposing of LOCs into new LO lessons. This application of OOSE design methodology led us to synthesise the criteria that define a truly object oriented LOC. Our criteria refine the LO definition work of Wiley and others (Wiley, 2001), and extend LO design guidelines (Hamel & Ryan-Jones, 2002).

We adapted to LOCs a scale for grading the cohesion of software modules (Yourdon & Constantine, 1978). We showed how to classify a LOC's level of cohesion, and described how each of the lower levels further reduces reusability. We illustrated with two OOSE designed LOCs that exhibit stronger cohesion than the original LOs. These LOCs are likely to require weaker coupling with other LOCs in a new LO lesson, thereby enhancing reusability. Our adaption of cohesion levels to LOs further extends and refines Boyle's design principles for authoring dynamic reusable LOs (Boyle, 2002).

We explained how our OOL approach is independent of, and complementary to a) the instructional design theory underlying the LO design process, and b) the metadata standards adopted by the IEEE (LTSC, 2003) for LO packaging. Pedagogical decisions can be made by the instructional designer before applying our approach to structuring the functionality of LOCs. Our OOL approach is pedagogy neutral in this

respect. Our approach contributes to LO metadata tagging by identifying attributes and activities for each LOC.

We assert that our OOL approach expands and informs the LO structuring options for instructional designers beyond those offered by general software engineering principles. We believe our approach assists the systematic development of more complex, authentic lessons, composed of dynamically created LOs. We expect further contributions toward an OOL methodology will facilitate the design and implementation of larger scale lessons, courses and educational programs, composed of LOs that increasingly exhibit the 'bilities'.

References

Boyle, T. (2002). Design principles for authoring dynamic, reusable learning objects. In A. Williamson, C. Gunn, A. Young and T. Clear (Eds), *Winds of Change in the Sea of Learning: Proceedings 19th ASCILITE Conference*, pp. 57-64. Auckland, New Zealand: UNITEC. [viewed 26 Nov 2003] http://www.ascilite.org.au/conferences/auckland02/proceedings/papers/028.pdf

Boyle, T. et al. (2003). Learning objects for introductory programming. [viewed 1 Mar 2004] http://www.londonmet.ac.uk/ltri/learningobjects/examples.htm

Computer Education Management Association (2001). Learning Architecture and Learning Objects. [1 Mar 2004] http://learnativity.com/lalo.html

Gibbons, A.S., Nelson, J. & Richards, R. (2001). The nature and origin of instructional objects. In D. Wiley (Ed), *The Instructional Use of Learning Objects.* [1 Mar 2004] http://www.reusability.org/read/

Hamel, C.J. & Ryan-Jones, D. (2002). Designing instruction with learning objects. *International Journal of Educational Technology*, 3(1). [verified 14 Jan 2005] http://www.ed.uiuc.edu/ijet/v3n1/hamel/index.html

LTSC (IEEE Learning Technology Standards Committee) (2003). Standard for Information Technology - Education and Training Systems - Learning Objects and Metadata. http://ltsc.ieee.org/wg12/index.html [viewed1 May 2004]

Morris, E.J.S. & Zuluaga, C.P. (2003). Educational effectiveness of 100% online IT courses. In G. Crisp, D. Thiele (Eds), *Interact: Integrate: Impact: Proceedings 20th ASCILITE Conference*, pp. 353-363. Adelaide: University of Adelaide. http://www.ascilite.org.au/conferences/adelaide03/docs/pdf/353.pdf

Page-Jones, M. (1998). *The Practical Guide to Structured Systems Design*, 2nd ed. Wayland Systems Inc. [viewed1 Mar 2004] http://www.waysys.com/ws_content_bl_pgssd_ch06.html

Pfleeger, S.L. (2001). *Software Engineering Theory and Practice*. Prentice Hall, 2nd ed.

Priestley, M. (1996). *Practical Object-Oriented Design with UML*. McGraw-Hill.

Rathsack, R. (2001). Conflict Resolution Styles. [1 Mar 2004] http://www.wisc-online.com/objects/index.asp?objID=PHR300

Wiley, D. (2001). Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. In D. Wiley (Ed), *The Instructional Use of Learning Objects*. [1 Mar 2004] http://www.reusability.org/read/

Yourdon, E., Constantine, L. (1978). *Stuctured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall.

Zuluaga, C.P. & Morris, E.J.S. (2003). A learning management model for mixed mode delivery using multiple channels (Internet, intranet, CD-ROM, Satellite TV). In G. Crisp, D. Thiele (Eds), *Interact, Integrate, Impact: Proceedings 20th ASCILITE Conference*, pp. 562-568. Adelaide: University of Adelaide. [verified 27 Oct 2004] http://www.adelaide.edu.au/ascilite2003/docs/pdf/562.pdf

Zuluaga, C.P., Morris, E.J.S. & Fernandez, G. (2002). Cost-effective development and delivery of 100% online I.T. courses. In A. Williamson, C. Gunn, A. Young & T. Clear (Eds), *Winds of Change in the Sea of Learning: Proceedings 19th ASCILITE Conference*, pp. 759-766. Auckland, NZ: UNITEC [26 Nov 2003] http://www.ascilite.org.au/conferences/auckland02/proceedings/papers/109.pdf

Ed Morris
School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne, Victoria 3001
http://www.rmit.edu.au/
Email: ted@rmit.edu.au