Beyond learning objects: Towards learning beans

Fintan Culwin
London South Bank University

Learning objects, in common with other instructional media, are relatively expensive to produce; yet seem to have limited suitability for reuse in different situations. The 'object' aspect of learning objects is borrowed from the domain of object oriented programming. Within this domain there are concepts that extend the notion and utility of object. One of these is the concept of a bean, a bean is a packaged object that is able to semi-automatically install itself into different scenarios. This paper attempts, using an example from the author's own domain, to enumerate a set of design principles that will enhance the possibility of successful reuse of objects. In doing this the learning object will have taken on some bean characteristics and so have started to become a learning bean.

Keywords: Learning object, learning bean, software engineering education, computing education

Introduction

Learning Objects have a degree of comfort associated with them, if only because definitions that include phrases such as “any entity digital or nondigital” (IEEE 2004) logically subsume anything and everything (Wiley 2001). Beyond this there are a plethora of other definitions which differ in scope from those that specify a particular technology (EOE 1998) those that specify a particular instructional sequence (L'Allier 1997) or even a particular mode of interaction (Laleuf & Spalter 2001). Others have attempted to refine the concept of learning object by reclassifying some as instructional objects (Gibbons & Nelson 2002), information objects or content objects (Johnson 2003) and then, in some cases, defining a learning object as an aggregation of such refinements. Thus although there is some degree of comfort associated with the familiarity of the term, upon closer inspection the term has become subject to so many differing interpretations so as to become almost meaningless.

Given the inherent confusion in the meaning of the term learning object it is incumbent upon authors to operationally define the meaning of the term in the context of the ideas that they are presenting. For this paper a learning object is taken to be a meaningfully interactive digital resource that can be web hosted. The qualification of meaningful on interactively is to deny the term to e-book style resources where interactivity is limited to pressing a button to access the next page and/or to start an animation. From the perspective of this paper, such resources do not take best advantage of the capabilities afforded by modern computing and communication capabilities.

One fundamental concept borrowed from the domain of software development, particularly Object Oriented Software Development (OOSD), is that of reusability and interoperability. This conception proposes, in both software and instructional design, that objects can be combined as easily as Lego bricks to produce complex artefacts quickly and economically. This has not proved to be as possible in OOSD as was once thought (Gabriel 2002) nor in instructional design (Friesen 2004). Some authors have borrowed other ideas, such as coupling and coherence, from the general domain of software engineering (Boyle 2003).

Within OOSD the notion of object has been further refined in order to attempt to deliver more of the promises that were once made. One such refinement is that of a bean (specifically a Java bean) where the ability of objects to be reused is enhanced by ensuring they can be located in different situations. This capability is delivered by designing objects according to a defined set of protocols that allow it to be customised in situ; that is whilst it is within the environment where it is to be deployed.

In this paper an attempt is made to start to enumerate the characteristics of situatability as they might apply to highly interactive learning objects. This is accomplished by taking a moderately complex learning object from the author's domain of initial software development education and illustrating how it can be customised in situ to accommodate different learning scenarios. Unfortunately this does mean that the start of this paper will have to explain in some detail a concept taken from initial programming instruction. Fortunately there is a learning object available to assist with assimilating this knowledge (Culwin 2004) which also provides a proof of concept of the ideas presented in this paper.

Once this barrier has been crossed a selection of customisations and enhancements to the basic object will be presented illustrating different scenarios of possible usage. This, together with some other ideas taken from the broad domain of software engineering practise, will lead to a list of design principles that might enhance the deployability and reusabilityof learning objects in general; leading towards the initially unfamiliar and so possibly uncomfortable concept of a learning bean.

A diversion into ‘for loops’

Readers who are familiar with a C style programming language *for loop* will not need to read this section. Those who do not have such familiarity may discover exactly why programming tutors are so concerned to develop and deploy effective learning objects. Those who are especially lucky may be reading this paper online with the learning objects being described embedded within the text, in order to assist with the comprehension of loop construction and operation.

In the initial stages of learning how to program there is a need to learn a control construct that will cause a sequence of actions to be repeated a known number of times. One such construct is known as a *for loop* and in what this paper will denote as its canonical form appears as follows:

```
for ( int index = 0;
          index < 10;
          index++ ) {
    printf( index + “ “);
} /* end if */
```

The effect of executing this code fragment would be to display the integers from 0 to 9 (that is “0 1 2 3 4 5 6 7 8 9") in an output window. The phase: `int index = 0` on the first line will create an integer variable called `index` and set its value to 0. The next line, `index < 10`, is a conditional question that can be read as ‘is the value of index less than 10?’, for which at this stage the answer is yes.

The next line is ignored for the time being and the body of the loop between the `{` and `}` delimiters is executed. In this canonical form it consists of a single output primitive `printf` instruction; which displays the value of `index` (0), followed by a space.

At the end of the loop, noted as such by the `/* end if */` marker, the third line `index++` is executed. The effect of this line is to add 1 to the value of `index`, at this stage taking it from 0 to 1. The condition ‘is the value of index less than 10?’ is asked again and as the answer is still yes the body is executed for a second time again; appending ‘1 ‘ to the ‘0 ‘ already there.

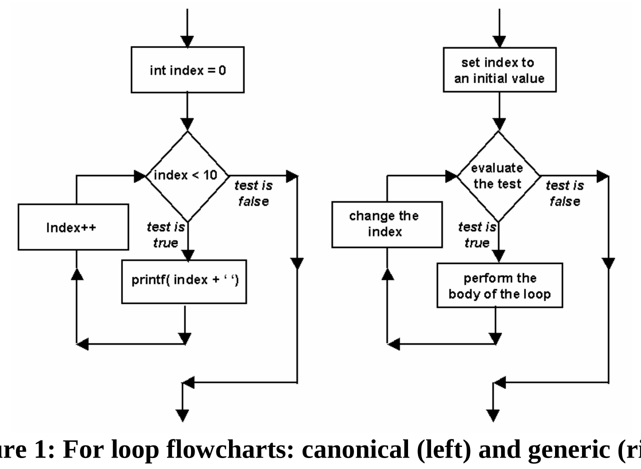


Figure 1: For loop flowcharts: canonical (left) and generic (right)

This pattern of behaviour; incrementing the `index`, checking that it is still less than 10 and outputting its value; continues until the conditional question is no. As soon as this is the case the loop terminates and the program would continue with other instructions.

The execution of a for loop can be illustrated on a flow of control chart, as shown in Figure 1. The precise behaviour of a loop can be controlled by varying the initial value of the index; changing the nature of the condition (‘ is less than’, ‘is less than or equal to’, ‘is greater than’, ‘is greater than or equal to’, ‘is equal to’ and ‘ is not equal to’) as well as the value it is tested against; and the change that is made to the index (‘increment by 1’, ‘ decrement by 1’, ‘ increment by n’ and ‘decrement by n’). Figure 2 gives some examples of variations on the canonical form and the output they would produce.

`for ( int index = 0; index <= 10; index+= 2 ) { printf( index + “ “); } /* end if */`	`for ( int index = 10; index > 0; index--) { printf( index + “ “); } /* end if */`	`for ( int index = 5; index > -5; index-= 2 ) { printf( index + “ “); } /* end if */`	`for ( int index = 10; index < 35; index+= 5 ) { printf( index + “ “); } /* end if */`
0 2 4 6 8 10	10 9 8 7 6 5 4 3 2 1	5 3 1 -1 -3	10 15 20 25 30

Figure 2: Various for loop configurations and their outputs

Practical knowledge of fundamental constructs such as this is essential to successful progression in learning the skills of software development. The for loop is one of many constructs that has to be rapidly and completely assimilated in order for higher level skills to be accommodated. Unfortunately, an increasing proportion of students are either not willing or able to engage effectively with this learning demand.

The politics of for loops

The C programming language is no longer widely used as an initial teaching language, having been superseded by languages such as C++ (c plus plus), Java or C# (c sharp). The canonical form of the for loop in each of these languages is given in Figure 3. Although the differences in the languages may seen trivial to an outsider, particularly with examples as simple as this, there is as almost Swiftian aspect to the ‘language wars’ that can and do erupt in organisations that are considering a change of language.

```
        C++                        Java                              C#
for ( int index = 0;       for ( int index = 0;              for ( int index = 0;
          index <= 10;               index <= 10;                      index <= 10;
          index++ ) {                index++ ) {                       index++ ) {
    cout << index << ' ';      System.out.println( index + ' ');   Console.WriteLine( index + ' ');
} // end if                } // end if                       } // end if
```

Figure 3: Canonical for loops in C++, Java and C#

Apparently even more trivial might be the positioning of the curly brace that delimits the start of the body of the loop. Two conventions are the common use, which are referred to in this paper as K&R and Modern. The layouts are illustrated in Figure 4 and although they have absolutely no effect upon the operation of the loop they are zealously enforced by tutors. At a recent computing learning objects workshop in London one attendee made the very definite statement that they would never use an object being demonstrated as it used the K&R convention (LMU 2004).

```
          K&R style                    Modern Style
for ( int index = 0;            for ( int index = 0;
          index < 10;                     index < 10;
          index++ )                       index++ ) {
{                                   printf( index + “ “);
    printf( index + “ “);       } /* end if */
} /* end if */
```

Figure 4: K&R and modern conventions

Accordingly for a learning object to be successfully redeployed outside the environment it was developed within it must either fortuitously have the appropriate combination of language and layout, or exist in a large number of variants, or be configurable in situ.

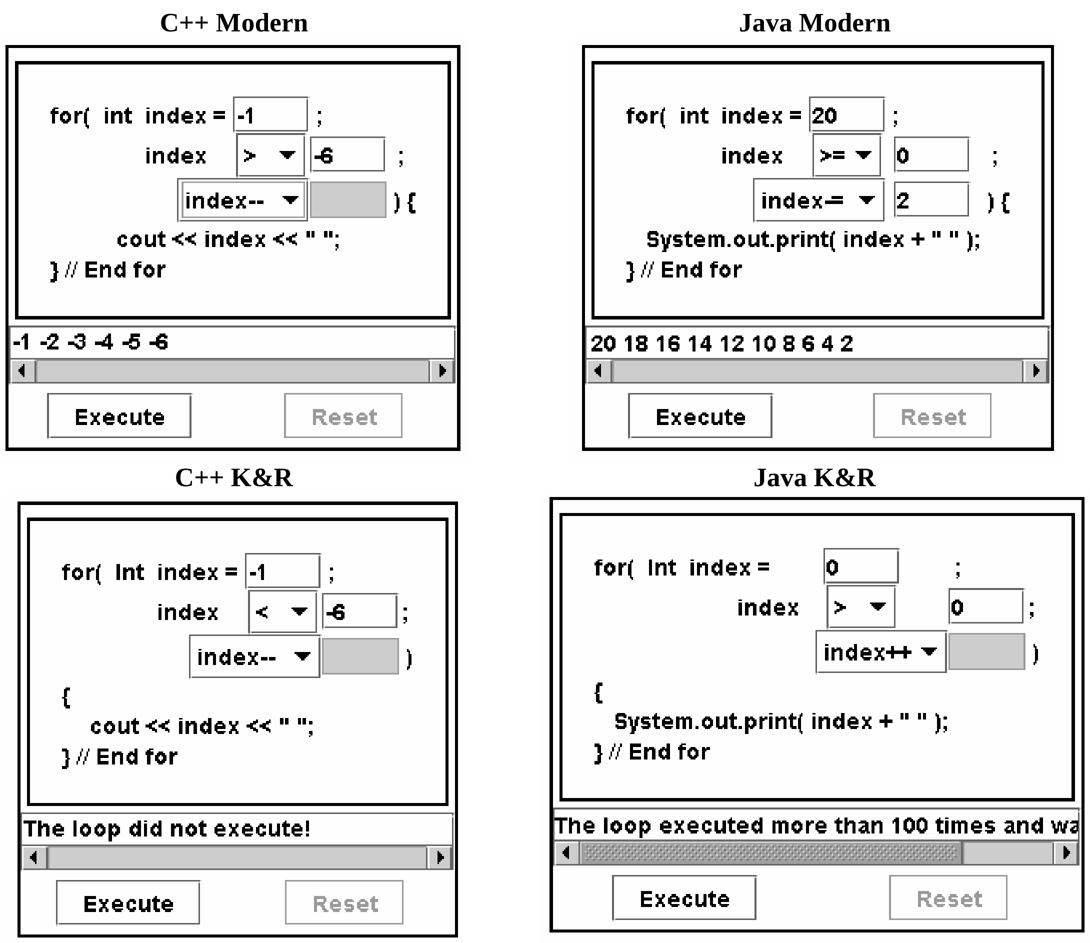
A simple configuration of a for loop learning object

Figure 5 illustrates four instances of a for loop learning object, configured to support both Java and C++, and the K&R and modern layout conventions. The provision of additional languages would be relatively straightforward. The object is written 100% in Java and so is deployable on all browsers under any operating system that have Java support enabled.

The configuration of the applet's language and layout options is accomplished by the use of the standard applet parameter mechanism in the HTML source of the Web page they are mounted upon. The applet will default to Java with a modern layout convention, but the language could be changed to C++ and, independently, the layout to K&R, by the inclusion of the following phrases within the applet tag.

```
<PARAM NAME = "language" VALUE = "cpp" >
<PARAM NAME = "layout" VALUE = "KandR" >
```

The learning object itself supports the rapid exploration of loop configurations. Conventionally exploration of a loop would require its source code to be included within a minimal skeleton program which would then have to be compiled and executed to determine its behaviour; with the edit, compile, execute cycle having to be repeated to explore the effect of changes.



C++ Modern

```
for( int index = -1 ;
         index > -6 ;
               index-- ) {
       cout << index << " ";
} // End for
```

```
-1 -2 -3 -4 -5 -6
```

Java Modern

```
for( int index = 20 ;
         index >= 0 ;
               index-= 2 ) {
   System.out.print( index + " " );
} // End for
```

```
20 18 16 14 12 10 8 6 4 2
```

C++ K&R

```
for( Int index = -1 ;
         index < -6 ;
               index-- )
{
   cout << index << " ";
} // End for
```

```
The loop did not execute!
```

Java K&R

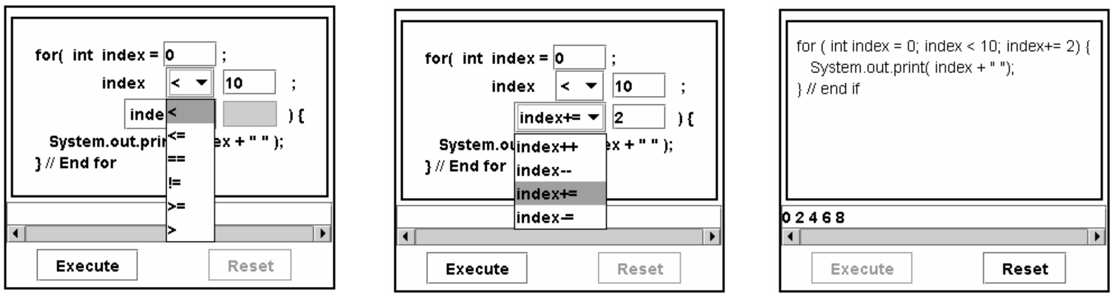
```
for( Int index = 0 ;
         index > 0 ;
               index++ )
{
  System.out.print( index + " " );
} // End for
```

```
The loop executed more than 100 times and wa
```

Figure 5: Simple loop learning object, various configurations and simulated output

The learning object also encourages the safe exploration of pathological configurations. Two pathological configurations are shown at the bottom of Figure 5, the left hand configuration shows a loop that will not execute and the right hand one that will not terminate. (The object currently only notes that the loop body has executed more than 100 times, not that it will never terminate.) The effects of both of these pathological configurations can be confusing when met for the first time outside the sandbox provided by the learning object.

The object is operated as illustrated in Figure 6. The parts of the loop construct that can be varied are the initial value of the index on line 1, entered using a specialised text field that will only allow integer values. The relation for the test on line 2, (‘is less than’ **<** , ‘is less than or equal to’ **<=**, ‘is greater than’ **>**, ‘is greater than or equal to’ **>=**, ‘is equal to’ **==** and ‘is not equal to’ **!=**), which is selected from a pull down menu. The value that the relation compares the index to is also on line 2 and is also entered by a specialised text field. The third line manipulates the index either incrementing it by 1 (`index++`), decrementing it by 1 (`index--`), incrementing by a value other than one (`index+=`) or decrementing by a value other than one (`index-=`). The latter two of these options require the value to be specified using a third specialised text field which becomes available when one of the operations that requires a value is selected from the pull down menu (as illustrated in the Java Modern part of Figure 5).



```
for ( int index = 0; index < 10; index+= 2) {
   System.out.print( index + " ");
} // end if
```

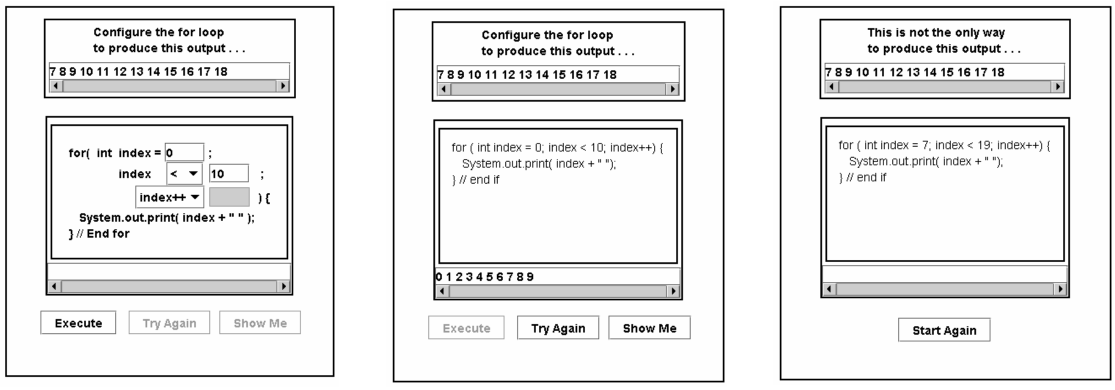
```
0 2 4 6 8
```

Figure 6: Simple loop learning object in operation

The loop structure is initially presented in its canonical form and the *Execute* button is enabled whenever the interface is in a state where the loop could be executed. When pressed the LO transits to the state shown in the right hand diagram of Figure 6, showing the loop as it would appear in source code, and the *Reset* button will return the interface to its initial, canonical, form.

A more complex configuration of a for loop learning object

Figure 7 illustrates the same learning object introduced in the previous section reconfigured into a more complex scenario. This relocation of the essential interactive object places it within an environment where the learner can be challenged to configure a for loop so as to output a pattern of numbers produced by the environment.



```
7 8 9 10 11 12 13 14 15 16 17 18
```

```
for ( int index = 0; index < 10; index++) {
   System.out.print( index + " ");
} // end if
```

```
0 1 2 3 4 5 6 7 8 9
```

```
for ( int index = 7; index < 19; index++) {
   System.out.print( index + " ");
} // end if
```

Figure 7: The simple object in challenge configuration

The precise environment illustrated is one of a series that could be configured to satisfy various requirements. For example the suggestion in this illustration is that the user of the object will have access to a possible solution, via the *Show Me* button as soon as they have executed the loop for the first time. While this may be acceptable in some environment there may be tutors who regard this behaviour as undesirable or unacceptable and may wish for this facility to be omitted altogether or to only be available after a number of tries have been made. For tutors who, for whatever reason, do not want to 'give the answer away', the configuration shown in this illustration would cause them to refuse to use the object. Consequently in order to assure a greater degree of potential reusability learning objects should be configurable to the possibly personal pedagogic preferences of tutors.

A categorisation of reusability

Before exploring other principles that might enhance reusability a categorisation of different classes of reuse will be presented together with indications of how the degree of reuse could be measured. The most obvious meaning of reusability is *deployability*, that is the number of places where an object can be shown to have been accessed from. This will include situations where an object has been replicated as a local copy, and where access to a remote object is embedded within a local resource. For Web hosted resources this aspect of deployability can readily be measured, at least on the public Web, by standard search engines. The number of times an object has been used, as opposed to the number of places where it might be used from, is a little more difficult to measure. In the situation where the object is not replicated as a local copy, the server log of the host where the resource is location will indicate the amount of usage.

Within software development the usual meaning of reusability is not the number of instances of the object that are created and used, but the number of places where instances of the object has been used as a component part of another, more complex, object. For example the numeric input text field as used in the simple for loop object above is an example of software object reuse. When the loop learning object was being designed the existence, capabilities and characteristics of the numeric input text field were known to the designer and three instances of it were specified. When the object was constructed no development work was required for these instances beyond incorporating the instances within the user interface and requesting their contents when required. Even though there are three instances of the numeric component used in the for loop object and potentially many, many instances of the for loop object deployed, this would count as only a single reuse. Measurement of this aspect of reuse seems problematic, unless the object is subject to licensing restrictions. This aspect of learning object reuse will be described as *component* reuse.

One aspect of reuse that seems omitted from many discussions of learning object reuse is that of *individual* reuse. That is an individual learner making use of an object, or a variant of an object, on a number of occasions. Measurement of this aspect of reuse would be possible for usage entirely within managed learning environment but for less structured environments might be possible if the learning object was not locally hosted, as in the discussion of deployability above, or if deployed instances are configured so as to report their usage.

An enumeration of reusability enhancements

Having illustrated a moderately complex exploratory learning object from the author's specialised domain and demonstrated some of the configuration possibilities that might lead to a greater degree of reusability. An attempt will be made in this part of the paper to start to enumerate a set design principles that might lead to objects in general having a greater degree of reusability. Many of these principles are inherently predicated on the ***first principle*** that objects should be designed and built for in situ configuration.

There are other learning objects that address the needs of neophyte software developers. The Codewitz (Codewitz 2004) and London Met. (LMU 2004) projects are producing multimedia animations of programming concepts while others are producing information objects (Gunawardena & Adamchik 2003; Ford 2004). Algorithm visualisations have had a long history within computing education, one favourite topic seems to be sorting techniques with one web site listing over 50 such animations (Brummund 2001). In general these products are static in the sense that they are preprogrammed by the developer and not programmable by the learner. In terms of the for loop structures described earlier in this paper it is as if the only loop that could be explored is the canonical 0 to 9 one. Although this approach may be of some value the expressive nature of programming would seem to require such expressability in its learning objects. This would lead to the ***second principle*** that objects which allow a learner to express themselves are more likely to be reused by an individual learner.

One design possibility that might further facilitate individual reuse might be progressive disclosure. In the for loop example, the simplest possibility might be a loop that always started at 0 and always incremented by 1 terminating at some value specified by the learner. More complex possibilities can then be described by allowing for starting points other than 0, allowing decrementation by 1, other relational tests, etc.; until the full complexity of the loop as described above becomes available. Hence the ***third principle*** is that objects which support progressive disclosure are more likely to be reused by an individual learner.

This principle makes no mention of how the progressive disclosure is to be managed. An instructivist tutor may wish to define a strictly delineated path of disclosure whilst a more exploratory tutor may prefer to make the full complexity available from the outset. This leads to the ***fourth principle*** that objects which are free of pedagogic and other philosophical assumptions are more likely to be reused by tutors. Examples of other philosophical assumptions would include the choice of programming language and layout convention from the for loop example. Where such philosophical baggage is unavoidable then the first principle of configurability should allow the tutor's preferred philosophy to be expressed.

One other aspect of configurability, emphasised for java beans, is internationalisation; commonly known as i18n as there are 18 letters between the initial i and terminal n. One aspect of i18n is the ability of a component to display text in different natural languages, but also extends to other presentational considerations such as the use of dots and commas to format numbers, date formats etc.. For example the calendar date known as the twenty fifth of December 2004 is conventionally presented as 25/12/2004 in the UK, 12/25/2004 in the US and 2004/12/25 in Japan. A bean would be expected to automatically discover the environment it was deployed in and adapt to local conventions. Accordingly the ***fifth principle*** is that objects which support i18n are more likely to be deployed.

The relocation of the simple object as a component part of the more complex object illustrates a further principle that smaller objects are more likely to be reused as components than larger ones. Although the complex object might be contextually reused as a part of on instructional object it is unlikely to be reused as a component part of an exploratory object. This leads to a well known ***sixth principle*** that smaller objects are more likely to be contextually reused than larger ones.

The complex version of the for loop learning object is able to challenge the learner to configure the interface so as to produce a particular output. Not only this, but the precise challenge differs every time the object is reset or revisited. There are several principles here each of which would support individual reuse, but which interact with each other to further enhance the possibilities.

Challenges are made more effective if the object supports meaningful interaction. Objects that support interaction in the form of 'press to continue or start' do not seem to have this property. Objects that support interaction such as 'fill in the blank' or 'drag these images into order' support a greater degree of interaction. Exploratory simulations, such as the for loop example, support interaction that is yet more meaningful. This illustrates the ***seventh principle*** that objects which support meaningful interaction are more likely to be engaging and so are more likely to be individually reused.

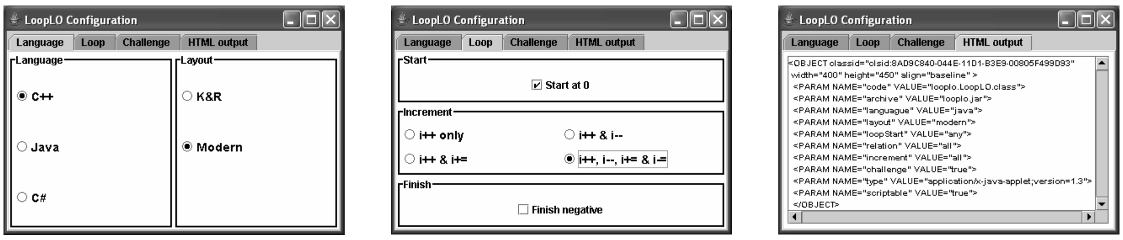
Where a fill in the blank level challenge is presented it may be a hard coded challenge that will never vary or vary only within a relatively small predefined set. Challenges that can be produced upon demand and vary every time they are reset or revisited will be perceived as being more testing and so more effective at consolidating learning and so more likely to be individually reused. Consequently the ***eighth principle*** is that objects which produce different content and/or challenge every time they are encountered are more likely to be individually reused

This capability can be enhanced by having an objective marking capability. In the complex for loop example the output of the challenge and the output produced by the learner's configuration of the for loop can be simply compared to decide if the configuration is effective (but not in this version if it is optimal). This capability provides for a large number of possible configurations of the challenge. Possible patterns of formative usage that might be supported include 'three tries and locked', 'three tries and revealed', 'solve the challenge within n seconds', 'solve as many challenges as possible in n seconds' (possibly leading to a high score board). Summative usage might include 'solve this challenge for n marks' or 'solve this challenge on the first attempt for n marks, second for fewer marks, etc.'. Accordingly the

ninth principle is that objects which have an objective marking capability are more likely to be deployed and individually reused.

The multitude of possibilities presented above will provide a challenge in its own right for the instructional designer. All configuration options should be supported by the same underlying object, a lesson learned from version control in standard software engineering practice. The additional costs involved in supporting one large configurable artefact are generally smaller than the costs involved in supporting a number of preconfigured distinct artefacts. The configuration of the object is therefore best effected in situ by some mechanism of communication from the page it is hosted on. For Java applets this is accomplished by parameters to the applet tag as described above.

A learning object should have a reasonable default behaviour so that it can be used out of the box without further configuration. It should also be supported by documentation in a standard format so that the instructional designer does not first have to learn the structure of the documentation before being able to use the information that it contains. However to be most acceptable it should be accompanied by a configuration utility that will allow the various options to be chosen and will generate the appropriate HTML code for inclusion in situ. A mock up of a possible utility for the for loop learning object is illustrated in Figure 8, and illustrates the ***tenth principle*** that objects which have a configuration utility are more likely to be deployed.



Language: [✓ C++] Java C#

Layout: K&R [✓ Modern]

Start: [✓ Start at 0]

Increment: i++ only; i++ & i--; i++ & i+=; [✓ i++, i--, i+= & i-=]

Finish: ☐ Finish negative

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
 width="400" height="450" align="baseline" >
  <PARAM NAME="code" VALUE="looplo.LoopLO.class">
  <PARAM NAME="archive" VALUE="looplo.jar">
  <PARAM NAME="languague" VALUE="java">
  <PARAM NAME="layout" VALUE="modern">
  <PARAM NAME="loopStart" VALUE="any">
  <PARAM NAME="relation" VALUE="all">
  <PARAM NAME="increment" VALUE="all">
  <PARAM NAME="challenge" VALUE="true">
  <PARAM NAME="type" VALUE="application/x-java-applet;version=1.3">
  <PARAM NAME="scriptable" VALUE="true">
  </OBJECT>
```

Figure 8: Loop learning object configuration utility

The for loop is not the only loop construct available to a developer. Figure 9 illustrates three other constructs that would produce the same output as the canonical for loop. There are situations where one particular construct is preferable, although this is often a political issue. The provision of a loop learning object that can present more than one construct would be more acceptable to tutors who had political options regarding the appropriate construct to use, and so enhance deployability. Otherwise the ability to triangulate knowledge of a for loop by transferring the essential cognitive knowledge to an alternative loop would enhance the possibility of reuse by an individual learner. The ***eleventh principle*** is that objects which triangulate knowledge are more likely to be deployed and individually reused.

while
```
int index = 0;
while( index < 10) {
   cout << index << ' ';
   index++;
} // end while
```

do while
```
int index = 0;
do {
   cout << index << ' ';
   index++;
} while( index < 10)
```

half loops
```
int index = 0;
loop {
  cout << index << ' ';
  exit when ( index++ == 10);
} // end while
```

Figure 9: Alternative canonical loops (C++ style syntax)

One final consideration taken from the domain of software engineering practice is that of open source. Non-proprietary software can be released on an open source license (GNU 1991) that allows other developers to modify and redistribute the artefact on a not for profit basis. The most well known open source project is probably the Linux operating system which had demonstrated the practical and economic feasibility of large scale open source projects. The advantage with respect to learning objects such as this is that developers will be able to further develop it allowing for different possible configurations and remove any bugs that appear, leading to greater opportunities for deployment. This is the ***twelfth principle*** that objects which are free of licensing restrictions are more likely to be deployed and more likely to be reused as components. (The source code for the examples used in this paper is available upon request from the author on an open source license.)

Linked to the open source principle is the open platform principle. Although the Microsoft operating system with internet explorer might seem ubiquitous it is not the only platform and there are significant users of both apple and various Unix environments. Learning objects that are dependant upon proprietary technology may not always be able to be deployed in all environments. Accordingly the ***thirteenth principle*** is that objects which do not use proprietary environments are more likely to be deployed.

A summary of the design principles

1. Objects should be designed and built for in situ configuration.
2. Objects which allow a learner to express themselves are more likely to be individually reused.
3. Objects which support progressive disclosure are more likely to be individually reused.
4. Objects which are free of pedagogic and other philosophical assumptions are more likely to be deployed.
5. Objects which support i18n are more likely to be deployed.
6. Smaller objects are more likely to be reused as components than larger ones.
7. Objects which support meaningful interaction are more likely to be individually reused.
8. Objects which produce different content and/or challenge every time they are encountered are more likely to be individually reused.
9. Objects which have an objective marking capability are more likely to be deployed and individually reused.
10. Objects which have configuration utilities are more likely to be deployed.
11. Objects which triangulate knowledge are more likely to be deployed and individually reused.
12. Objects which are free of licensing restrictions are more likely to be deployed and more likely to be reused as components.
13. Objects which do not use proprietary environments are more likely to be deployed.

Conclusion: towards learning beans

This paper has presented a highly configurable exploratory learning object from the author's domain of initial software development education. This has then been used to enumerate a set of design principles that might enhance different aspects of reusability. It may be that not all of the proposed principles are transferable to objects in other domains. It might also be the case that there are design principles that can be elucidated from a consideration of objects from other domains that are not applicable to this object but which are appropriate for inclusion in the list of design principles.

The java bean white paper (EJB 2000) lists a number of defining characteristics of beans. The most fundamental of which is that beans support properties that determine their in situ appearance and behaviour. Additional characteristics define a mechanism by which these properties can be discovered and how the bean is able to communicate with its environment. The viability and utility of producing a learning object that has properties that determine its appearance, for example the programming language used, and its behaviour, for example if the show me capability, has been demonstrated in the loop object. Accordingly it can be thought of as a proof of concept learning bean.

A mechanism for the automated discovery of properties, which would allow a general learning bean to be manipulated in a tool that would configure and install it, might be feasible if the usefulness of this aspect of beans was generally accepted. Likewise a pattern of behaviour in communicating with the bean's environment would also be possible. Any bean that had the characteristics of being able to produce a challenge and an objective marking capability could be housed in standard environments which control the behavioural aspects of the challenge. That is the mechanism for say, three tries and reveal, is independent of the domain that is being learned and so, given a standard communication protocol, could be reused.

The essential conclusion of this paper, embodied in the first design principle, is that objects which are designed and built for in situ configuration are more likely to be reused than those that are monolithic and non-configurable. Such objects might be better described as learning beans rather than learning objects.

References

Boyle, T. (2003). Design principles for authoring dynamic, reusable learning objects. *Australian Journal of Educational Technology*, 19(1), 46-58. http://www.ascilite.org.au/ajet/ajet19/boyle.html

Brummund, P. (2001). The Complete Collection of Algorithm Animations (CCAA). http://www.cs.hope.edu/~alganim/ccaa/sorting.html [viewed 19 June 2002, verified 2 Oct 2004].

Codewitz (2004). International Project For Better Programming Skills. http://www.codewitz.net/ [verified 2 Oct 2004]

Culwin, F. (2004). Simple Loop Learning Object. http://myweb.lsbu.ac.uk/~fintan/jcf/jcfw.html [verified 2 Oct 2004]

EOE (1998). Educational Object Economy Info Pages. [viewed 19 June 2002, verified 2 Oct 2004] http://www.eoe.org/foundation/info.htm

EJB (2000). Enterprise java bean White paper. http://java.sun.com/products/javabeans/docs/spec.html [viewed 19 June 2002, verified 2 Oct 2004].

Ford, L. (2004). A learning object generator for programming. *Proc. ITiCSE 2004*, Leeds UK 2004.

Friesen, N. (2004). Three objections to learning objects. In McGreal, R. (Ed), *Online Education Using Learning Objects*. London: Routledge/Falmer

Gabriel, R. P. (2002). Objects have failed. OOPSLA Debate November 2002. [viewed 19 June 2002, verified 2 Oct 2004] http://www.dreamsongs.com/ObjectsHaveFailedNarrative.html

GNU (1991). GNU General Public License. http://www.gnu.org/licenses/licenses.html#GPL [viewed 19 June 2002, verified 2 Oct 2004].

Gibbons, A. S., Nelson, J. & Richards, R. (2002). The nature and origin of instructional objects. In D. A. Wiley (Ed), *The Instructional Use of Learning Objects*. AIT Press 2002 ISBN 0-7842-0892-1. http://www.reusability.org/read/chapters/gibbons.doc [viewed 19 June 2002, verified 2 Oct 2004]

Gunawardena, A. & Adamchik, V. (2003). A customized learning objects approach to teaching programming. *ACM SIGCSE Bulletin*, 35(3). http://www-2.cs.cmu.edu/~adamchik/PLO/ITiCSE-03.pdf [verified 2 Oct 2004]

IEEE (2002). IEEE Standard for Learning Object Metadata. http://ltsc.ieee.org/wg12/par1484-12-1.html [viewed 19 June 2002, verified 2 Oct 2004]

Laleuf & Spalter (2001). A component repository for learning objects. *Proceedings of the first ACM/IEEE-CS joint conference on Digital libraries*. ISBN: 1-58113-345-6

L'Allier, J. J. (1997). Frame of Reference: NETg's Map to Its Products, Their Structures and Core Beliefs. http://www.netg.com/research/whitepapers/frameref.asp [viewed 19 June 2002, not found 2 Oct 2004]

Johnson, L. F. (2003). Elusive Vision: Challenges Impeding the Learning Object Economy. http://download.macromedia.com/pub/solutions/downloads/elearning/elusive_vision.pdf [viewed 19 June 2002, verified 2 Oct 2004]

LMU (2004). Learning Objects for Introductory Programming. London Metropolitan University. http://www.londonmet.ac.uk/ltri/learningobjects/examples.htm [viewed 19 June 2002]

LMU (2004). Learning Objects for Computing: Present achievements future prospects. London Metropolitan University. http://www.ics.ltsn.ac.uk/pub/lo/index.html [viewed 19 June 2002].

Wiley, D. A. (2002). Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. In D. A. Wiley (Ed), *The Instructional Use of Learning Objects*. AIT Press 2002 ISBN 0-7842-0892-1. http://www.reusability.org/read/chapters/wiley.doc [viewed 19 June 2002, verified 2 Oct 2004]

Fintan Culwin, fintan@lsbu.ac.uk, Faculty of Business, Computing and Information Management, London South Bank University, Borough Road, London SE1 0AA, UK

Please cite as: Culwin, F. (2004). Beyond learning objects: Towards learning beans. In R. Atkinson, C. McBeath, D. Jonas-Dwyer & R. Phillips (Eds), *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference* (pp. 230-239). Perth, 5-8 December. http://www.ascilite.org.au/conferences/perth04/procs/culwin.html