Re-usability and extendability in object-oriented and object-based design
J. Moses

School of Computing and Information Systems, University of Sunderland, Sunderland, SR1 3SD, UK

ABSTRACT

This paper considers the object-oriented paradigm’s ability to produce reusable and extendable software. In so doing it identifies the strengths and weaknesses of the paradigm, with respect to reusability, and compares these with those of a several representative object-based methods. Further, consideration is given to the main method of achieving reusability in the object-oriented paradigm, i.e. the inheritance hierarchy. The impact inheritance can have on the maintainability of the software components produced is considered using an information flow complexity metric.

The paper concludes that although the object-oriented paradigm will assist in producing reusable and extendable software it must be well supported by software tools; and, careful control and management of the development of components for reuse must be undertaken, if a reasonable level of maintainability is also to be achieved.
DIVIDE AND RULE

Using Fenton's definition in [1] of a module as "... a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier". Modularity in software is an attempt to break a problem down into a set of simpler subproblems. Implicit in this approach to problem solving is the need for each subproblem to be cohesive, in the sense that the subproblem should possess an identifiable boundary and its constituent parts should be inter-related. Further, to be useful the subproblem solution needs to be capable of easy use within, or connexion into, the rest of the problem solution. (It is not helpful to find a solution to a solvable subproblem which creates difficulties when attempts are made to connect it into the total problem solution.)

Good modular software design is a natural attempt to apply the divide and rule principle and, implicitly, it should ensure the two software engineering goals of increasing cohesion in a module and decreasing coupling between modules, see [1] for definitions of cohesion and coupling. Hence, modular software, which maximises modular cohesiveness and minimises coupling will tend to assist understandability and consequently improve maintainability of the software modules produced, see Sommerville [2].

Viewed simply, a well designed small cohesive unit of code with minimal coupling should be easier to manage, understand and therefore maintain than a larger one; and, the complete modular system will be easier to maintain, i.e. localised changes in modules will be much easier to effect than those changes required for an equivalent system constructed as one complex amorphous program. Further, cohesive and low coupled modules of software were thought likely to assist reuse. For example, before modularity, Fortran subroutines and functions within the relatively unstructured programs of the 1960's were designed to be reused several times within the same program avoiding duplication of programming effort, see [3]. Later, libraries of subroutines for reuse were developed, e.g. NAG libraries.

In a similar way to that in which a small functionally related mechanically engineered item may find reuse in many more applications than a larger composite item a small unit of code may be more likely to be reused, cf. a particular type of item comprising a nut, washer, bolt and bracket may be part of many different car exhaust systems fitted to many different car makes and models, where as rather fewer exhaust systems will fit more than one model of car. (The more complex and specific the product the less opportunity there generally will be for it to be reused.) Thus, in general a cohesive software module comprising a minimal number of functions and possessing low coupling
is more likely to be publicly reused than a program comprising several functions designed for one system, since some of the functions are likely to be unique to the system.

Hence, modular software construction can be used to produce more easily reusable and maintainable software than software production methods which do not possess the modular concept, e.g. programming techniques of the 1960s. Further, such modules will also tend to assist system extendability because extensions to a system will usually be restricted to changes in a few modules and not propagate through the entire system. However, in order to assist maintainability, reusability and extendability good design principles must be adhered to, e.g. low coupling and high cohesion. Other design criteria to aid good modular software for such quality factors are also used e.g. information hiding, see [2],[4].

Object-oriented, e.g. [4], and Object-based, e.g. [5], methods have extended the notion of modularity to include abstract data typing as the mechanism for building cohesive modules, see [1]; and, Object-oriented methods are often considered superior to other methods for the production of reusable and extendable software, see [4].

Korson and McGregor, in [6], provide details of the common concepts across the range of object-oriented approaches (and also give a comprehensive explanation of each concept). The common concepts employed are the object class, inheritance, dynamic binding and polymorphism. Korson and McGregor also indicate that the reason for the object-oriented paradigm’s outstanding potential to assist the goal of program reuse and extendability is due largely to these concepts (the same concepts which assist reusability also assist extendability).

However, the object-oriented paradigm, its inheritance hierarchy and reuse of such hierarchies engender a set of problems which need to be addressed. Guthrey in [7] provides a critique of object-oriented program development, the criticisms are briefly summarised in the next section.

A CRITIQUE OF SOME OF THE FEATURES COMMON TO OBJECT-ORIENTED PARADIGMS

Although code reuse in a program within an object-oriented hierarchy is obviously achievable, i.e. through object class inheritance and each time an object of a class is instantiated, see [4], it is the hierarchy itself which is
intended to be the major component of reuse and not the object, see Wirfs-
Brock and Johnson [8] which describes research into reusing subsystems and
frameworks both units of object classes and hierarchies.

However, hierarchical reuse and information hiding within a hierarchy can
prevent access to objects within the hierarchy. Thus reuse of an entire hierarchy
can result when only some of the object classes in the hierarchy are needed,
implying some extra, seemingly undesired and unnecessary, code reuse. This
could lead to larger systems and slower execution times than would be
necessary when compared with non-hierarchical reuse, e.g. reusing a subroutine.
This problem might be overcome by careful design for reuse and by the
development of smaller hierarchies. Unfortunately, even to do this could require
breaking the information hiding principle and looking inside the hierarchy at
objects’ state representations to ensure satisfactory matching between publicly
reused hierarchies, which in turn could lead to new code in order to integrate
hierarchies, potentially promoting an increase in code in order to gain the reuse
of some object classes. Hence, hidden internal representation may prevent easy
reuse, increase programming effort and redundant code. This implies that
object-oriented programming can lead to an easy build of code fragments but
a potentially difficult integration. The code redundancy problem has been
addressed by the object-oriented programming (OOP) language Eiffel, see
Meyer [4]: it supplies an optimiser for use in eliminating unneeded code
introduced by inheritance, but it can only do this for a whole system and not
for individual classes.

More support for the view that careful control of the use of object-oriented
programming concepts is necessary comes from Wilde et al. [9]. In [9] the
difficulties in understanding and therefore maintaining object-oriented software
are highlighted. Such difficulties being attributed to inheritance, dynamic
binding and polymorphism. Wilde et al. conclude that tools to help maintainers
to understand a developer’s design are needed and it is essential to plan and
control the use of polymorphism and dynamic binding so that all methods for
a particular message do the same thing.

Hence, to achieve the object-oriented paradigm’s greater potential for
reusability and extendability requires support by software design tools, which
are currently thought inadequate, and requires implementation in an object-
oriented programming language. Therefore, there is still scope for considering
class and object-based methods as valuable approaches to achieving reuse.
OBJECT-BASED METHODS

Development techniques, like JSD [10] and MASCOT 3 [11], have the advantage, over object-oriented techniques, of language independence and are therefore, potentially, applicable to more problem areas. MASCOT 3 has been identified as Object-based in [12], [13] and [14] and has been shown in [12] to possess most of the requirements for an object-oriented development method as enunciated by Ladden [15]. A MASCOT 3 template, see [11] for detailed descriptions, to some extent resembles an object-oriented class, Sutcliffe in [14] did not verify this although it was acknowledged that some inheritance capability is possessed by MASCOT 3. Using Korson and McGregor’s definition of a class from their synopsis of current object-oriented paradigm trends [6] the relationship between a template in MASCOT 3 and a class in the object-oriented paradigm has been shown, see [16].

MASCOT 3’s limited inheritance capability is described in [12], [16] and [17]; and it is hierarchical, since a designed system is composed of subsystems each of which may themselves may be composed of subsystems at lower levels.

Hull et. al., in [18], have stated that they view JSD as having no hierarchical decomposition of design, and that JSD was considered to produce modular components only because it allows processes to hide data and provides state inspection vectors to external processes. However, Sommerville in [2] points out that the JSD implementation phase is a transformation into a set of communicating sequential processes whose state vectors can be inspected by each other, hence these processes cannot be considered as objects in the object-oriented sense. Further, the number of processes tends to be reduced in the implementation phase to make the system more manageable, again something which is an anathema to object-oriented design.

JSD possess no hierarchical design structure, and because of this, cannot represent inheritance. HOOD, [19], at the design stage, should be capable of representing inheritance, however it does not possess as clear and restrictive an interface mechanism as MASCOT 3, i.e. MASCOT 3 provides a lower level of decoupling, see [17].

Hence, for the reasons outlined the author takes the view that, of the object-based techniques considered in this paper, MASCOT 3 comes closest to being able to provide sufficient means of accommodating reuse and extension to be able to compete with the object-oriented design paradigm.
Shepperd [20] and Shepperd and Ince [21] have suggested and validated a modified version of Henry and Kafura's Information Flow Complexity metric [22], which was shown by Henry and Kafura (although criticised later by Shepperd and Ince [21]) to be correlated with program changes. Shepperd's modified version is recommended for use in producing software modules which should prove easier to develop and by implication may also be easier to maintain. The metric can be applied from the architectural design stage onwards in the waterfall model of software development. The metric, named IF4, by Shepperd in [20], is defined for a complete system and IF4_m for the m\textsuperscript{th} software module, and is specified as follows:

\[ IF4 = \sum_{m} IF4_m \]

\[ IF4_m = (\text{unique fan_{in}_m} \times \text{unique fan_{out}_m})^2 \]

where,

\[ \text{unique fan_{in}_m} = \text{total number of local and global unique flows terminating at module m.} \]

\[ \text{unique fan_{out}_m} = \text{total number of local and global unique flows emanating from module m.} \]

\[ \text{unique flows} = \text{total of all local and global information flows with all duplicates removed.} \]

\[ \text{local information flow} = \{ \text{module A invokes module B and passes it a parameter OR module A invokes module B and B returns a parameter to A} \} \]

\[ \text{global information flow} = \{ \text{module A updates a global data structure and module B retrieves from the structure} \} \]
Class Person export
display_name, display_age
feature
  n: string; a: integer;
display_name() is
do
  print_string(n);
end;
display_age() is
do
  print_age(a);
end;
end -- Class Person.

Class Pilot export
display_name, display_age, display_airline, display_license
inherit
  Person redefine display_name, display_age
feature
  airline: string; license: date;
display_name()
do
  special_string_print(n);
end;
display_age()
do
  special_int_print(a);
end;
display_airline() is
do
  print_string(airline);
end;
display_license()
do
  print_date(license);
end;
end -- Class Pilot.

Figure 1. Class definitions for the Person - Pilot hierarchy.
Figure 3 Mascot 3 diagram and Object class descriptors with IFC values
Class Sample_Code
feature
  x:Person; y:Pilot
create
do
  1 x.create
  2 x.name:='fred'
  3 x.age:=21
  4 x.display_name
  5 x.display_age
  6 y.create
  7 y.airline:='BA'
  8 y.license:=31/12/94
  9 y.name:='joe'
 10 y.age:=34
 11 y.display_airline
 12 y.display_date
 13 y.display_age
 14 y.display_name
 15 x:=y -- polymorphic reference, x’ s dynamic type changes
 16 x.display_name -- dynamic binding
 17 x.display_age -- dynamic binding
 18 x.forget
 19 y.forget
end;
end -- Sample_Code

Figure 2. The Sample_Code object definition

access interface ainputpers;
  name(id:int,n:strg);
  age(id:int,a:int);
end.

access interface ainputpilot;
  airline(id:int,a:strg);
  license(id:int,l:date);
  name(id:int,n:strg);
  age(id:int,a:int);
end.

access interface aoutputpers;
  print_name(id:int,n:strg);
  print_age(id:int,a:int);
end.

access interface aoutputpilot;
  print_name(id:int,n:strg);
  print_age(id:int,a:int);
  print_license(id:int,l:date);
  print_airline(id:int,a:strg);
end.

Figure 4 MASCOT 3 access interfaces
THE METRIC AND ITS APPLICATION TO OBJECT-ORIENTED DEVELOPMENT AND MASCOT 3

The MASCOT 3 template access interface provides a mechanism, from the architectural design stage onwards, for identification of information flows. Each access interface associated with a template defines a components interactions with other legally connected components, i.e. those possessing the same access interface name and appropriate reference type (port to window connection).

The complete set of access procedures available for use through each access interface is specified in the definition of an access interface, each access procedure specification includes the names of parameters which are to be passed during communication between instantiated templates (components). These procedures can easily be used to count fan_in and fan_out for system components, without counting duplicate flows (as recommended by Shepperd [20]) and without penalising reuse (a problem identified by Shepperd in [20]).

Figure 1 shows the Eiffel code for the hierarchy Person-Pilot. The class Pilot is a specialisation (of Person), and it inherits the features of the generalisation Person; and class Sample_code uses both Person and Pilot in its definition.

Consider the code in class Sample_code in figure 2, expressed in the Object Oriented language Eiffel [4], the object classes supporting mechanisms (known as class descriptors in Eiffel) must allow object instance of types Person and Pilot to be created at run-time. It follows that each time a new object instance of one of the types is created a message must be sent to the class mechanism to allocate storage for each object, i.e. the identifier of the storage location must be returned to the Sample_code object the calling object (or its class mechanism), see statements 1 and 6 in figure 2. Further, in Eiffel the default values for the created objects are also returned. Therefore, data flows between the class mechanisms for Pilot and Sample_code and Person and Sample_code. In Eiffel it is also possible to allow an object’s data fields to be instantiated individually, statements 3 and 4, each separate field instantiation thus gives rise to a unique data flow.

Returning to the mechanism for Pilot, see figure 3, the Information Flow Complexity metric (IFC) value is 40² for the pilot class descriptor, an object of this type appears to provide four procedures which return parameters (two are inherited form Person); it inputs four individual instantiations (each can be considered as a separate dataflow form Sample_code to Pilot, since each field
can be instantiated separately); further, it outputs, after a create, a default value for the object and its identifier; and, inputs after a forget procedure (used to delete an object) the object identifier. The comparable IFC value from the MASCOT3 design, see figure 3 and 4, which is capable of providing similar, but more limited, functionality as that of the classes defined in figure 1, is 36².

In addition, there are other factors to consider due to the Object-oriented paradigm: redefinition of procedures and functions, polymorphism and dynamic binding. For example, statement 15 in figure 2 allows x to refer to an object of class Pilot here the static type of x is Person but its dynamic type is Pilot. Hence, Person must also be able to access the correct print_name procedure (that in Pilot and not Person), using dynamic binding. Therefore, more information must flow from Person to Pilot: the class mechanism for Person must call up the correct procedure from Pilot, Pilot may then return the parameter to Sample_code. It should be noted that for each procedure redefinition, by a subclass, there will exist an extra indirect dataflow, and redefinition may be undertaken at each level of the hierarchy. The IFC for the object class descriptor for person becomes 15².

Hence, it does appear as if, extrapolating from this example, that the Object-oriented hierarchy could lead to very large values for Shepperd’s metric value over that which a comparable object-based solution might be expected to achieve. Also, dynamic binding and polymorphism are quite difficult concepts to come to terms with and using automatic calculation of Shepperd’s metric during design of an object-hierarchy may be a good guide to the inherent complexities of inheritance in object-oriented systems, and thus to potential difficulties in understanding and maintaining the system.

A further point to note is that systems like Eiffel automatically provide create and forget default functions, although create functions can, and often will, be specified by the software designer/constructor, and that assuming the system is error free in this respect then perhaps the data flows due to forget and any default create procedures could be discounted, if interest is to be centred solely on development effort.

CONCLUSIONS

It must not be forgotten that Shepperd’s version of the metric was found to be highly correlated with development effort and not maintenance, see [20]; and, it may be that different languages require different versions of the metric to accurately predict development and maintenance problems; and that the metric’s threshold value at architectural design which should be used as an indicator of
when to re-design modules should vary from one construction paradigm to another, see [23]. However, using automatic calculation of Shepperd’s metric may assist designers in controlling maintenance difficulties due to the inheritance hierarchy for the object-oriented paradigm.

In addition, experiments are required to establish the relationship between the metric at architectural design and the ensuing development effort and maintainability when different implementation languages are used. Further, experiments are needed to determine suitable threshold values for each language (if indeed there are significant differences between languages), or perhaps different metrics (versions of the modified Shepperd /Henry and Kafura metric) for different languages.

Until the object-oriented paradigm and its programming languages overcome the problems of reuse through inheritance, and until an adequate set of tools are developed for use with OOP languages (Shepperd’s metric may form the basis of a useful tool when automated), there will be scope for considering object-based methods seriously in order to produce reusable code and extendable systems whilst still maintaining a high degree of language independence. In addition, MASCOT 3 will assist the development of maintainable software, because of its high level of decoupling; and, reusable software due to its template and subsystem concepts and, will allow additional reuse due to its inheritance capacity.

REFERENCES


